Décomposition Automatique des Programmes Parallèles pour l'Optimisation et la Prédiction de Performance

Automatic Decomposition of Parallel Programs for Optimization and Performance Prediction

THÈSE

présentée et soutenue publiquement le 30 Septembre 2016

pour l'obtention du

Doctorat de l'Université de Versailles Saint-Quentin-en-Yvelines et de l'Université de Paris Saclay

(spécialité informatique)

par

Mihail Popov

Composition du jury

Rapporteurs :	Michael O'Boyle	Professeur des Universités, University of Edinburgh
	François Bodin	Professeur des Universités, Université de Rennes et INRIA
Examinateurs :	Christine Eisenbeis	Directrice de recherche, Université Paris-Sud et INRIA
	Alexandra Jimborean	Professeur associé, University of Uppsala
Co-encadrant de thèse :	Pablo de Oliveira Castro	Maître de conférences, Université de Versailles
Directeur de thèse :	William Jalby	Professeur des Universités, Université de Versailles

Résumé:

Dans le domaine du calcul haute performance, de nombreux programmes étalons ou benchmarks sont utilisés pour mesurer l'efficacité des calculateurs, des compilateurs et des optimisations de performance. Les benchmarks de référence regroupent souvent des programmes de calcul issus de l'industrie et peuvent être très longs. Le processus d'étalonnage d'une nouvelle architecture de calcul ou d'une optimisation est donc coûteux.

La plupart des benchmark sont constitués d'un ensemble de noyaux de calcul indépendants. Souvent l'étalonneur n'est intéressé que par un sous-ensemble de ces noyaux, il serait donc intéressant de pouvoir les exécuter séparément. Ainsi, il devient plus facile et rapide d'appliquer des optimisations locales sur les benchmarks. De plus, les benchmarks contiennent de nombreux noyaux de calcul redondants. Certaines opérations, bien que mesurées plusieurs fois, n'apportent pas d'informations supplémentaires sur le système à étudier. En détectant les similarités entre eux et en éliminant les noyaux redondants, on diminue le coût des benchmarks sans perte d'information.

Cette thèse propose une méthode permettant de décomposer automatiquement une application en un ensemble de noyaux de performance, que nous appellons codelets. La méthode proposée permet de rejouer les codelets, de manière isolée, dans différentes conditions expérimentales pour pouvoir étalonner leur performance. Cette thèse étudie dans quelle mesure la décomposition en noyaux permet de diminuer le coût du processus de benchmarking et d'optimisation. Elle évalue aussi l'avantage d'optimisations locales par rapport à une approche globale.

De nombreux travaux ont été réalisés afin d'améliorer le processus de benchmarking. Dans ce domaine, on remarquera l'utilisation de techniques d'apprentissage machine ou d'échantillonnage. L'idée de décomposer les benchmarks en morceaux indépendants n'est pas nouvelle. Ce concept a été appliqué avec succès sur les programmes séquentiels et nous le portons à maturité sur les programmes parallèles.

Évaluer des nouvelles micro-architectures ou la scalabilité est 25× fois plus rapide avec des codelets que avec des exécutions d'applications. Les codelets prédisent le temps d'exécution avec une précision de 94% et permettent de trouver des optimisations locales jusqu'à 1.06× fois plus efficaces que la meilleure approche globale.

Mots Clés: Prédiction de performance, Parallélisme, Compilation, Optimisation, Approche par morceaux, Checkpoint restart

Abstract:

In high performance computing, benchmarks evaluate architectures, compilers and optimizations. Standard benchmarks are mostly issued from the industrial world and may have a very long execution time. So, evaluating a new architecture or an optimization is costly.

Most of the benchmarks are composed of independent kernels. Usually, users are only interested by a small subset of these kernels. To get faster and easier local optimizations, we should find ways to extract kernels as standalone executables. Also, benchmarks have redundant computational kernels. Some calculations do not bring new informations about the system that we want to study, despite that we measure them many times. By detecting similar operations and removing redundant kernels, we can reduce the benchmarking cost without loosing information about the system.

This thesis proposes a method to automatically decompose applications into small kernels called codelets. Each codelet is a standalone executable that can be replayed in different execution contexts to evaluate them. This thesis quantifies how much the decomposition method accelerates optimization and benchmarking processes. It also quantify the benefits of local optimizations over global optimizations.

There are many related works which aim to enhance the benchmarking process. In particular, we note machine learning approaches and sampling techniques. Decomposing applications into independent pieces is not a new idea. It has been successfully applied on sequential codes. In this thesis we extend it to parallel programs.

Evaluating scalability or new micro-architectures is 25× faster with codelets than with full application executions. Codelets predict the execution time with an accuracy of 94% and find local optimizations that outperform the best global optimization up to 1.06×.

Keywords: Performance prediction, Parallelism, Compilation, Optimization, Piecewise approach, Checkpoint restart

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Context

Architecture complexity has increased a lot in the past 20 years with hierarchical cache systems, simultaneous multithreading, out-of-order execution, parallelism and heterogeneity. The conventional way to guide and evaluate these architectural innovations is to study a benchmark suite based on existing programs [1], such as SPEC [2] (Standard Performance Evaluation Corporation) or NAS [3]. Each benchmark suite targets a specific field: NAS targets parallel supercomputers and high performance computing (HPC) while SPEC are for more general computations. Architectural designers have to understand these workloads and tune future systems for them [4, 5].

A first well known issue of this model is that there are new emerging workloads that were never considered before. Computer science evolution opens new domains of computing such as deep learning, gaming, big data analytics, or bioinformatics that bring new workloads with them. Driving the architectural design on a set of existing source code applications may lead to a suboptimal result [6] for the new workloads. Designers achieve good-enough results by adapting, evolving and proposing new benchmark suites that incorporate these new workloads.

Before building new prototypes, architects rely on simulators to evaluate new architectural updates. A second issue is that simulating new architectures with all their complexity is time and resource consuming: an application simulation is easily `10000` [7] times slower than a native execution. This expensive simulation cost limits the number of iterations engineers can perform in a given budget. Different approaches have been proposed by the community to overcome this limitation, such as analytical models [8, 9], machine learning [10, 11], checkpoint-restart [12], or simulation reduction techniques [13].

Figure 1.1 presents the different components involved in the computation process. They are organized in layers and each layer rely on the elements below. The workloads are evolving and so must the rest of the stack.

For instance, the Graphics Processing Units (GPUs) were designed to speedup embarrassingly parallel image processing applications. GPUs are composed of a large number of simple cores that allow them to take advantage of the regular Single Instruction Multiple Data (SIMD) parallelism.

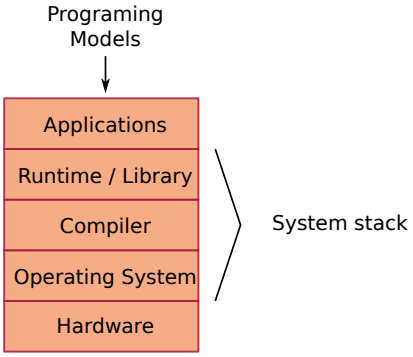


Figure 1.1: There are different layers of abstraction for computing. Each layer rely on the layers below. End users select a programing model and write their applications. These applications take advantage of different runtimes and libraries. Compilers produce assembly executable code from the previous layers. Finally the operating system is in charge to monitor the hardware. It ensures that the hardware correctly executes the assembly.

In the 2000s, the GPUs starts performing tasks that are traditionally handled by the Central Processing Unit (CPU). Using GPUs for CPU tasks is defined as the General Purpose computing on Graphics Processing Units (GPGPUs). GPGPUs guide software developers into updating their workloads by using new programing models and languages (OpenCL or CUDA). This example illustrates how, to achieve fully efficient computations, both the software and the hardware adapt to each other.

Designing multiple elements at the same time by taking feedback from all of them is called *co-design*. Designers have already attempted to co-design both the hardware and the software [14]. They have also combined the architectural design with the compiler design [15].

However, an issue remains: as long as there is no global consensus during early design stages, design decisions must be taken with only limited knowledge of other system components. Achieving such consensus is challenging: for instance right now there is no fully accepted programming model that supports parallel heterogeneous supercomputers.

To partially reduce the gap between the codes and the architectures, both the runtime and the compiler provide standard configurations that achieve good-enough performance across most of the codes and the architectures. Re-engineering the compiler and the runtime to adapt them to the architectures and codes evolution is challenging: modern compilers and runtimes contain millions of lines of code. To validate a new optimization, it must be tested over all possible inputs instead of the few test cases required to publish a paper in a research conference [1].

A common approach to enhance the software system layer is autotuning. Autotuned compilers [16] illustrate this idea: they apply successive compiler

transformations to a program and evaluate them by executing the resulting code. So, autotuners generate many variants of a given kernel or benchmark and each variant targets a specific execution point. Unlike standard configurations, autotuned configurations can take advantage of target-specific optimizations since they are not required to correctly work on a large panel of systems.

Finding the optimal parameters may lead to substantial improvement but is a costly and time consuming process for two reasons:

- there is a huge parameter space to explore. For instance, The LLVM [17] `3.4` compiler provides more than sixty optimization passes. Passes have different impact depending on their order of execution and can be applied multiple times. So, it is not feasible to try all the possible combinations in a reasonable amount of time. Popular approaches [18, 19] minimize this problem by pruning the exploration space and hence reduce the number of points to evaluate,

- each resulting point in the search space must be evaluated. The issue is that for some applications, the evaluation of the optimizations can be extremely expensive since we have to execute the applications.

Reducing the evaluation cost of applications or a set of benchmarks is also a problem that is faced the hardware designers. Both hardware and software designers rely on *benchmark reduction* techniques to accelerate the evaluation. The idea is to avoid the evaluation of redundant parts within the benchmarks or the applications.

Let us consider multiple benchmarks with a similar execution behavior and sharing the same performance bottlenecks. They should be sensitive to the same parameters or to the same optimizations [20]. For instance, increasing cache size will benefit the performance of all the memory bound loops. There is no need to evaluate all of them through the search space: we will only pay more evaluation time without collecting any new informations about the system.

To avoid such scenarios, researchers profile the benchmarks to detect and discard the redundant ones. Different approaches exist to quantify the diversity of a benchmark suite [21]. A popular approach consists into using performance characterization metrics [4]. Benchmarks with similar metrics have a similar execution behavior and hence are considered as redundant.

Reduction strategies use the performance metrics to detect and cluster the redundant benchmarks [22]. Each cluster contains benchmarks that can be replaced by a single representative without diminishing the suite behavior diversity. Evaluating the representatives allows to extrapolate the whole suite behavior without considering redundant benchmarks and hence accelerate the evaluation.

Redundancies appear across applications but can also exist within a fixed application. Applications have different *phases* during their execution. A phase [23] is a set of intervals within a program execution with similar behavior.

Simulators [24] take advantage of these phases to reduce the evaluation cost. By using performance metrics [25] or Basic Bloc Vectors [13] (BBVs) as signatures for each phase, designers cluster them. Representative phases are extracted and used as proxies for the whole benchmarks. To avoid replaying the redundant phases, simulators rely on checkpoint restart or fast forward simulation functionalities to directly target the representative ones.

Unlike simulators, the real hardware do not provide such abilities. Hence, phase reduction cannot be directly applied on real hardware for tuning.

There is an interesting and versatile approach which is *code isolation* [26]. Code isolation extracts regions of code from an application as standalone fragments of code. These fragments can be compiled and replayed independently from the original application on a real hardware. Yet, it is challenging to isolate fragments of code that can faithfully be replayed across multiple execution environments: there are currently no versatile enough code isolator tools with these two properties.

To summarize this introduction, we note that:

- both hardware and software are evolving and understanding them is challenging,
- benchmarking scientific applications and systems is a costly but vital process to validate software and hardware optimizations,
- benchmark reduction techniques detect phase redundancies that cannot be applied on a real hardware to reduce the tuning cost of applications,
- tuning the system stack outperforms the standard execution configuration but requires a prohibitive one-off search,
- no current code isolator is versatile enough to produce proxies for parameter tuning.

In this thesis we propose a new code isolation method that can be used as a fast way to benchmark High Performance computing (HPC) parallel codes. We extract the representative pieces of code and use them as proxies to tune the applications. Each piece of code can map an application phase and so avoid redundant evaluations. The code isolation must have two related properties that we will to demonstrate. First, the fragments of code should be replayable on a large panel of target execution environments. Second, to achieve accurate performance measurements, the replay must be faithful to the original execution.

1.2 Contributions

During this thesis, I participated in the development and the design of Codelet Extractor and REplayer [27] (CERE). The contributions of this thesis strongly rely on CERE.

CERE (`http://benchmark-subsetting.github.io/cere/`) is a collaborative open source framework for code isolation developed at University of Versailles and Exascale Computing Research. It extracts the hotspots (either loops or OpenMP regions) of an application as isolated fragments of code, called *codelets*. Codelets can be modified, compiled, run, and measured independently while remaining faithful to the original execution. In other words, they are versatile and accurate enough to be used as a proxy to evaluate hardware or system designs tradeoffs.

Unlike previous approaches, CERE isolates codes at the compiler Intermediate Representation (IR) level. IR is a good candidate for extraction because it provides a good trade-off between the traditional assembly and source code isolation. Unlike assembly isolation, IR isolation allows to retarget the replay to new Instruction Set Architectures (ISA) or compiler optimizations. It also avoids hazards such as costly support of new languages or replay discrepancies introduced by the compilation front end required by the source isolation.

CERE is evaluated on the SPEC 2006 FP benchmarks: codelets cover 90.9% and accurately replay 66.3% of the execution time. This evaluation was published at the ACM Transactions on Architecture and Code Optimization (TACO) [27].

This thesis has two major and one minor contributions.

Major Contributions

Scalability Prediction Model

Evaluating the strong scalability of OpenMP applications is a costly and time consuming process. It involves executing the whole application multiple times with different number of threads. The first contribution of this thesis is the design and the implementation of a scalability prediction model incorporated into CERE.

CERE maps each parallel region to a codelet. To accelerate scalability prediction, CERE replays codelets while varying the number of threads. Prediction speedup comes from two reasons. First, for codelets called repeatedly and for which the performance does not vary across calls, the number of invocations is reduced. Second, the sequential parts of the programs do not need to be replayed for each different thread configuration.

We evaluate CERE on a C version of the NAS Parallel Benchmarks (NPB) by achieving an average speedup of 25× and a median error of 5%.

This work was published at International Parallel and Distributed Processing Symposium (IPDPS) [28].

Piecewise Holistic Tuning

The second contribution is a piecewise holistic approach to tune compiler optimizations and thread configurations. Codelet autotuning achieves better speedups at a lower tuning cost. With the codelet prediction model, CERE reduces the number of loops or OpenMP regions to be evaluated. Moreover unlike whole-program tuning, CERE customizes the set of best parameters for each specific OpenMP region or loop.

We demonstrate the autotuner capability by obtaining a speedup of `1.08`× over the NAS 3.0 benchmarks. This contribution led to a publication at the International European Conference on Parallel and Distributed Computing (Euro-Par)[29].

Minor Contribution

Finding Metrics for Benchmarks Reduction

Selecting the best architecture for a set of programs and workloads leads to significant performance improvements but requires long running benchmarks. I participated in the elaboration of a method that reduces this benchmarking cost for architecture selection. The method combines code isolation methodology with benchmark reduction techniques.

We extract a set of codelets out of the benchmarks. Then, we identify two causes of redundancy. First, codelets that perform similar operations. Second, codelets called repeatedly. The key idea is to minimize redundancy inside the benchmark suite to speed it up. For each group of similar codelets, only one representative is kept. For codelets called repeatedly and for which the performance does not vary across calls, the number of invocations is reduced.

The third contribution consists into defining distances between the codelets. We gather performance metrics that are used as signatures for each codelet. CERE uses these signatures to cluster the codelets.

This method was evaluated on the NAS Serial (SER) benchmarks, producing a reduced benchmark suite 30 times faster in average than the original suite. The reduced suite predicts the execution time on three target architectures with a median error between 3.9% and 8%. The reduction approach is published at The International Symposium on Code Generation and Optimization (CGO) [30].

1.3 Thesis Outline

Chapter 2 presents the context and the state of the art in the domain of workload characterization. In particular, we detail the different benchmark reduction strategies and tuning heuristics. Chapter 3 presents CERE, the codelet extractor and replayer. Chapter 4 shows CERE reduction strategy to accelerate the benchmarking process. Chapter 5 validates CERE extraction process. Chapter 6 demonstrates how CERE can be used to tune different compiler optimizations, thread configurations or architectures.

Chapter 2

Background

Contents

2.1 Introduction

In high performance computing (HPC), benchmarks evaluate architectures, compilers and optimizations. Since benchmarks may have very long execution times, evaluating a new architecture or an optimization is costly.

Also, we note that benchmarks have redundant computational behaviors [13]. Some calculations do not bring new informations about the system that we want to study, despite that we measure them many times. This thesis, like many related works, aims to enhance the benchmarking process by detecting and reducing these redundant calculations.

This chapter presents the background of this thesis. We start by presenting methods that aim to understand the applications behaviors. Chapter 2.2

shows the different metrics that characterize the behavior of an application. In section 2.3, we explain the statistical approaches used by the current state of the art benchmark reduction techniques themselves presented in section 2.4.

This thesis does not contribute to any novel data processing approaches. It rather takes advantage of them in order to enhance the benchmark tuning process with code isolation. Section 2.5 gives an overview the different code isolation techniques while section 2.6 presents popular tuning strategies.

2.2 Performance Metrics

The purpose of performance metrics is to understand an application behavior on a system. Good metrics are highly dependent on the situation and finding a general representative set of metrics is challenging because of architecture heterogeneity. In other words, we cannot apply the same methodology across all the architectures.

Modern high performance processors have special hardware registers that detect and count performance events. These counters use the *performance monitoring units* [31] (PMU). For example, the Intel microachitecture *Nehalem* provides the counter *FIXC0*. It is used to measure the event *INSTR_RETIRED_ANY* i.e. the number of retired instructions during the execution.

The problem is that the performance events and the counters are not consistent across the different architectures. Let us consider Nehalem which has three levels of cache. During the execution, the PMU can access the number of hits or misses on the third cache level (L3). Unlike Nehalem, Intel *Core2* has only two cache levels: measuring L3 events on a Core2 is impossible. Hence, the architecture disparity prevents from an unified approach to measure the hardware counters across all the systems.

Also, current applications have a large spectrum of performance behaviors and pathologies [32] such as poor vectorization, useless data dependencies, and high register pressure. Sprunt et al. [33] illustrates how hardware performance counters can be used to identify memory access problems and to eliminate some register stalls. Considering the wide range of architectures and applications, many performance metrics have been developed to evaluate the different execution behaviors.

Another point to consider is the profiling cost. Metrics not only provide different informations but are also more or less costly to collect. For instance, some data dependencies can be detected by simply analyzing the assembly. On the other hand, measuring the memory bandwidth requires a full application execution. It is important for a user to select both relevant and affordable metrics for his problem.

In this thesis, we use performance metrics for benchmarking reduction

i.e. to detect if two applications have a similar behavior. We expect that codes with similar performance metrics yield similar micro-architectural behavior. We are looking for metrics with must:

1. correctly discriminate two applications if their behaviors are different. In other words, they must encompass all of the key factors that affect the performance by covering a large enough range of behavioral aspects,

2. be as cheap as possible to collect. We must avoid evaluating correlated metrics since they do not provide new information.

This section gives an overview of different approaches that characterize applications behavior. Performance metrics are divided in two groups: static and dynamic. The first study the code while the second execute it to get the information. Both have different trade-offs, and are used depending on the situation. We compare the different approaches by looking at the information they provide and their collecting cost.

2.2.1 Static Analysis

Static analysis consists in deriving information from a code without executing it. It is traditionally used to find security issues or to debug applications [34] but can also be extended for code profiling [35, 36]. Both source [34, 37] and assembly [35, 32] levels are fit for static analysis. Study of assembly instructions may be difficult to map to source code but has the advantage to look at what is really executed after compilation and link steps.

There are different static metrics that are related to the performance. Producing them usually requires from the static analysis to rely on an machine model. In this thesis, we use Modular Assembly Quality Analyzer and Optimizer (MAQAO) Code Quality Analyzer (CQA) [35, 32].

MAQAO CQA is a loop centric code quality analyzer that provides high level metrics which are data set independent. It relies on a performance model which assumes that all data are resident in the first level cache. Here is a list of some MAQAO CQA static metrics that we select to characterize diverse behaviors:

- **dispatch ports pressure** studies the pressure from micro operands dispatch on execution ports and allows to find which one is the bottleneck,

- **peak performance in L1** helps to find an upper bound of the performance. It is build by assuming no memory related issues, infinite size buffers, and infinite number of iterations,

- **arithmetic intensity** measures the amount of computational instructions over the total number of instructions,

- **vectorization ratio** is the proportion of the vector instructions over the total number of instructions that can be vectorized. This metric indicate for a code the benefits of the vectorization.

Static analysis is relatively cheaper to achieve compared to a full program execution and provides a wide range of analyzes. The drawback is the fact that not all the aspects of the execution behavior can by analyzed. It cannot estimate dynamic factors such as the branch miss predictions or the cache misses that impact the execution time. So, despite its speed, static analysis is not sufficient to fully characterize an application behavior.

2.2.2 Hardware Performance Counters

Unlike static studies, dynamic analysis relies on code execution. It provides information that cannot be statically obtained such as the memory behavior. However, executing the code raises two issues. First, it is costly and time consuming. Second, the results are bound to a particular execution. Workloads executed with different data sets may exhibit different execution behaviors, depending on the data set. Also, due to thread concurrency, parallel programs may have different behaviors across multiple executions: a single region can have different dynamic results across the executions. So, we must profile the application multiple times. The problem is that even with multiple executions, we cannot assert that we fully characterize all the behaviors.

The previously described PMUs provide dynamic data about the CPU resource utilization [31]. To do so, the processors provide the capability to monitor performance events through the monitoring counters. The counters are either with a *general* or with a *fixed* purpose. We saw the Nehalem FIXC0 counter that has the fixed purpose of measuring the number of retired instructions. Developers configure general purpose counters to measure the events that interest them. A large panel of events can be assigned to them, including the architecture memory behavior (cache accesses, cache hits ratios, memory footprints, or the RAM bandwidth), the cycles per instruction (CPI) or the power and energy consumption. We also note that the number of hardware performance counters increases with the more recent micro-architectures.

There are two methods currently used to get the PMU data: sampling and tracing. Tracing measures the total number of events between two probes manually placed by the developer within the code. It allows us to target a specific region of code for profiling but introduces an overhead noise due to the probes.

Sampling records the events from the counters only at specific moments during the execution. There are different approaches to choose when to record a counter. The counter can be measured every N elapsed cycles or every I executed number of instructions. To perform accurate sampling measurements, it is mandatory to select a good frequency for the sampling points. Performing to few measurements may miss some performance events while too many introduce measurements overhead.

There is a limited number of PMU in a CPU and only some fixed purpose PMU can collect specific performance events. So, if developers are interested into too many performance events at the same time, they have to profile the application multiple times. To address this issue, the CPU provides the multiplexing. The PMU measures multiple events during the same run: the counter alternates the measured events. The disadvantage is that alternating the events may introduce noise in the measurements. In this thesis, we use two hardware performance counters tools: Lprof [38], MAQAO dynamic performance profiler and Likwid [39].

A way to overcome some of the respective limitations of the static and the dynamic analysis is to couple them together. The two methods are complementary: static metrics are useful to evaluate the assembly code quality while dynamic metrics cover the memory and data-dependent behavior.

2.2.3 Micro-architecture Independent Metrics

We rely on an architecture model to get the static analysis. We also collect the performance counter metrics with executions on a micro-architecture by looking at its performance monitoring counters. We call these previous metrics *microarchitecture dependent metrics* because they are depend on to the micro-architecture that they target.

With architecture dependent metrics, an applications has different performance metrics depending on the profiled architecture. On the opposite, we have the *microarchitecture independent metrics* [4, 40]. They are inherent to the application we profile. The goal of the architecture independent metrics is to ensure that an application has the same performance signature across all the architectures. We abstract the application behavior from the architecture.

Here is a list of some architecture independent metrics proposed by Hoste et al. [4] and Phansalkar et al. [4]:

- **instruction mix** is the percentage of appearance of various operations performed by the application,
- **instruction level parallelism (ILP)** quantify the instructions independence. It is usually defined as the instructions per cycle (IPC) achievable for an idealized out-of-order processor constrained only by a window size and data dependencies,

- **branch prediction** computes the percentage of forward branches out of the total branch instructions in the dynamic instruction stream,
- **memory foot print** is the amount of memory touched for both instruction and data streams,
- **reuse distance** characterizes the cache behavior of the application. For each memory read, the corresponding 64-byte cache block is determined. For each cache block accessed, the number of unique cache blocks accessed since the last time it was referenced is determined, using a last recently used (LRU) stack. We traditionally gather these accesses into buckets to characterize the application.

The main advantage is that independent metrics may outline some application behaviors that are not observable with dependent metrics. Host et al. [41, 42] compared the profiling diversity of the micro-architecture dependent and independent metrics over 6 benchmark suites. Architecture independent metrics observe different behaviors over 98% of the applications where architecture dependent metrics only observe 57% different behaviors.

The main disadvantage independent metrics is that they are very costly to obtain: despite being faster than simulations, getting these metrics incurs a significant slowdown (between 10 and 200 times slower[4]).

To summarize, there is a wide range of methods to analyze applications. Some are intrusive and more costly but provide more information. Others are lighter but only target specific components. There is no single best method, users must choose or combine different approaches according to their motivation and the price that they are ready to pay.

We do not have to bound to a single approach. For instance, static and dynamic analysis have been combined to characterize hot loops [43, 44]. Marin et al. [9] predict the behavior metrics and execution time of applications also by using a combination of static and dynamic analysis. In particular, they combine static metrics such as instruction mix and dynamic one as histograms of the memory reuse distance. The benefits of this approach is that it models both the instructions execution cost and the memory hierarchy penalty.

In this thesis, we characterize the applications through both static and dynamic metrics. Combining them allows to easily model a diverse range of behaviors. Section 4.3.3 explains how we choose our performance metrics. The next section presents methods that help understanding and manipulating these performance metrics.

2.3 Data Processing

The previous section reviewed methods to characterize an application behavior. They produce performance metrics that describe the different behaviors.

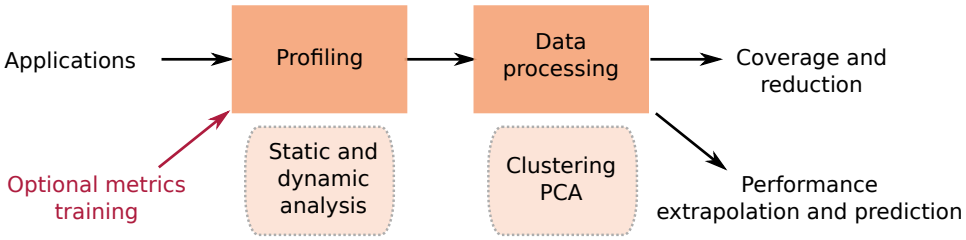


Figure 2.1: Application and system study. The workload coverage and reduction is explained in the following section 2.4.

This section presents approaches that analyze and process these metrics.

Figure 2.1 displays how many state of the art methods [2, 45, 4, 40, 46, 22, 47, 42] study a set of applications on a system, to either predict the execution time or to enhance the benchmarking process. First, they profile the applications to get their static and dynamic metrics. These metrics represent aspects of the applications behavior (see section 2.2). Since there is a wide range of metrics, it is challenging to know which metrics are relevant for which purpose.

There are three commonly used complementary methods to analyze and process the performance metrics:

- *Clustering*: an unsupervised machine learning method which consists into gathering elements that are similar (see section 2.3.1),
- *Principal Component Analysis*: a statistical procedure that converts a set of data into linearly uncorrelated variables called principal components (see section 2.3.2),
- *Evolutionary Algorithms*: a supervised machine learning method which is trained to find the most relevant metrics for the user objective (see section 2.3.3).

Finally, machine learning predictions must be validated. Section 2.3.4 describes how such predictions quality is quantified.

2.3.1 Clustering

Clustering is a statistical method for data processing. It gathers elements that are similar into groups called clusters. Elements in a cluster are more similar to each other than to those in other clusters.

Clustering is used on applications characterization to gather similar applications. Eventually, researchers extend the clustering to a fine granularity to detect and gather phase behaviors within the applications.

To cluster elements, we must define what is the meaning of similarity and how to quantify it. We associate a vector of metrics to each element.

As said earlier, elements are either applications or phases and metrics are usually performance metrics described in section 2.2. To visualize multiple elements, we plot them as points into a multi-dimensional space. Coordinates of the points are defined according to the values of the metric vectors of the points. To define the similarity between two elements, we compute the distance between their respective metric vectors i.e. the distance between their respective points.

In this section, we present two commonly used distances for applications characterization: *Euclidean* [23] and *Manhattan* [48] distances. We use in this thesis the two following clustering methods: *hierarchical clustering with Ward's criterion* and CLustering LARge Applications (*CLARA*). CLARA is an extension of the *K-Medoids* clustering that supports a large number of objects. K-Medoids is related to the *K-Means* clustering but has a better support of outliers. To fully understand CLARA clustering, we present the associated clustering approaches.

Distance Between Vectors

The Euclidean and Manhattan distances between two vectors X and Y with n values are respectively defined as:

1. $Euclidean_{Dist}(X,Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
2. $Manhattan_{Dist}(X,Y) = \sum_{i=1}^{n}|x_i - y_i|$

More intuitively, we define two vectors as single points in a n dimensional space. Euclidean is the straight-line distance between the two points. Manhattan is the sum of the absolute differences of their Cartesian coordinates, or the distance between two points if we follow a parallel path to the dimensional space axes.

For some high dimensional performance vectors, Sherwood et al. [48] indicate in their experiments that Manhattan works better than Euclidean distance. The reason is that a high difference in each dimension has more impact on a Manhattan than on a Euclidean distance. Also, we note that K-Means usually uses the euclidean distance.

Both the two distances and clustering methodologies are currently used for application characterization. There is no consensus about which one is the best one: it depends on the data and the use case.

K-Means

K-means [49] is a simple and fast clustering approach but requires a previously known K number of clusters. It starts by selecting K points in

the parameter space. Different heuristics [50] such as the random partition describe how to select the K elements. K-means performs the following steps:

1. each selected point is defined as the center of a cluster,

2. for each point that is not in a cluster, integrate it to the cluster with the closest centroid,

3. when the K clusters cover all the points, locate their centroids and select the K nearest points to these centroids,

4. repeat step 1,2,3 until the clusters converge: the output clusters do not change between two iterations.

The converged clusters are the output of the K-Means clustering.

K-Medoids

K-Medoids is a similar method as K-Means. The only difference is that in step 3, instead of selecting the centroid of each cluster, the K-Medoids selects the medoid. In fact, K-Medoids selects the K cluster centers of the next iteration directly among the points themselves. However, since computing an average value is easier than a medoid value, K-means requires more computations than K-means.

The most common realization of K-Medoid clustering is the Partitioning Around Medoids (PAM) algorithm. PAM tries to reduce the computation cost of the medoid points search. It tries out all of the points in each cluster as new medoid and selects the ones that lead to lower *within-cluster variance* per cluster. The within-cluster variance of a cluster is defined as the variance of all the metric vectors of elements belonging to this cluster or as Sum of Squares Error (SSE). SSE is the sum of the squared differences between each point and its cluster mean.

CLARA

CLARA extends the k-medoids for large number of points. It works by clustering a sample from the dataset. Then, CLARA assigns all the remaining points from the dataset to the clusters from the sample. The quality of the clustering strongly depends on the sample. Small samples accelerate the clustering but lead to poor clustering efficiency while big samples improve the quality but are more consuming to perform. Section 4.2 presents an example of CLARA clustering over a region execution time across different invocations.

Hierarchical Clustering

Hierarchical clustering [51] is a greedy method. It starts with as many clusters as elements. At each step, the method merges a pair of clusters. The pair is selected to minimize the total within-cluster variance after the merge. Reducing the within-cluster variance forms compact clusters of elements with close metric vectors. The clustering method ends when a single cluster is left. All the successive merges between clusters are recorded in a dendrogram. The final number of clusters K is selected by cutting the dendrogram at different heights. An example of dendrogram is described in section 6.4.

Kaur et al. [52] compare the clustering results of K-means versus hierarchical clustering over query redirections. As a general comparison, they conclude that K-means algorithm is better suited for large dataset while hierarchical is more efficient for small datasets. Hierarchical clustering produces clusters with lower entropy than K-Means but is more costly to perform.

Another advantage of the hierarchical clustering is that it does not require a previously known number of threads. Selecting the optimal number of cluster is important to better understand the data. Too few clusters gather elements that have different properties in the same cluster while too many lead to cluster redundancies.

Elbow Method

A usual method to select the optimal number of the hierarchical clustering is the Elbow method [53]. By ploting the average within-cluster variance versus the number of clusters, we expect that the first clusters significantly reduce the average within-cluster variance. However, at some point, adding clusters stops significantly decreasing the within-cluster variance. This point is the elbow criterion since it minimizes both the number of clusters and the within-cluster variance. The Elbow method cuts the dendrogram tree of the hierarchical clustering at the elbow criterion number of clusters. Section 6.4 illustrates how use the elbow method to detect the optimal number of clusters for architecture selection.

2.3.2 Principal Component Analysis

Principal Component Analysis [54] (PCA) is a statistical procedure that extracts significant information from some data. It reduces the dimensionality of the dataset of possibly correlated variables by converting them into a set of values of uncorrelated variables called principal components. Each principal component is a linear combination of the original variables. This transformation is defined in such a way that the first principal component has the largest possible variance. Each succeeding component in turn has the highest variance possible under the constraint that they are not correlated

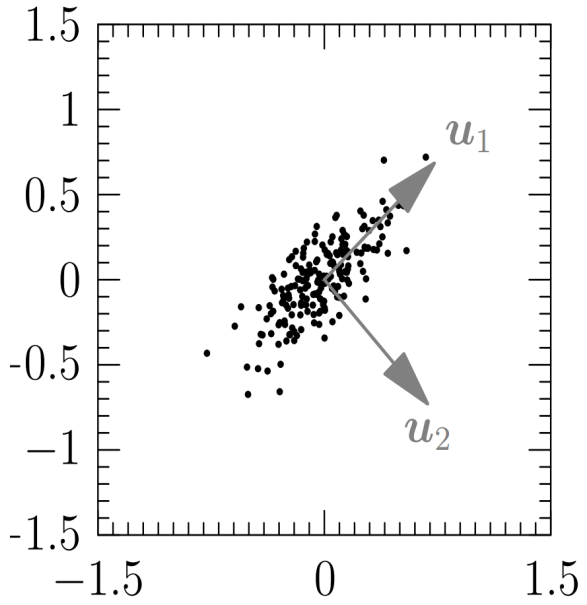


Figure 2.2: An demonstration of PCA applied to a data cloud. `u1` is the first component and `u2` the second. This example was taken from Hyvarinen et al. [55].

to the previous components. So, the method reduces the dimensionality of the data while controlling the amount of information that is lost by keeping the mandatory components.

More formally, PCA converts n vectors $X_1, X_2, ..., X_n$ into m principal components $Y_1, Y_2, ..., Y_m$, such as:

$$Y_i = \sum_{j=1}^{n} a_{ij} X_j, a_{ij} \in \mathbb{R}$$

The transformation has the two following properties:

1. $Var[Y_1] \geq Var[Y_2] \geq ... \geq Var[Y_m]$
2. $\forall i \neq j, Cov[Y_i, Y_j] = 0$

PCA is usually applied to metric performance space exploration [41]. It helps find and remove metrics that are similar. Figure 2.2 illustrates the methodology.

2.3.3 Genetic Algorithms

Predicting the performance from execution parameters [56] or selecting the best compiler optimizations [57] or performance metrics [58] is a challenging

task. In all these cases, we face a huge exploration space where it is too costly to evaluate each point separately. Hence, we use heuristics to guide the exploration space.

A common technique to explore such space are Genetic Algorithms [59] (GAs). A GA is a search heuristic that mimics the process of natural selection.

GA starts by initializing a random population generation. Each individual is a potential solution of the problem (a point in the search space) and has a set of chromosomes named features. These features might be compiler passes for compiler tuning or performance features for characterization. An objective function evaluates the fitness of each individual.

GA is an iterative process producing multiple generations. Fit individuals according to the objective function from the previous generation are selected and mixed to form a new generation. Individuals mixing consists in random recombination of their genome with possible mutations. We expect that the individuals in the new generation have a better fitness than the previous one. We repeat this process until satisfying individuals are found. The search may stop because we do not observe any enhancements in the individuals across the new generations or because we reached a fixed threshold of iterations.

Many parameters such as the crossover probability (chromosomes recombination), the mutation probability or population size tweak GA to enhance it. Moreover, the algorithm can be improved by different techniques such as isolating local optimal populations or using swarm strategies. Despite the diversity of the method, GA is very popular for some computer science problems (see section 2.6).

2.3.4 Training Validation

A last point to consider for machine learning approaches is the validation of the results. GA produces a set of features that can be applied in new scenarios. Similarly, clustering gathers elements that are expected to be similar.

A common way to evaluate the quality of these results is the *cross-validation*. It consists into evaluating the prediction results with the previously trained features on new unseen data sets.

For instance, we trained though GA in section 6.4 a set of performance metrics to cluster similar applications together. The training was performed on the Numerical Recipes (NR) [60] benchmarks. We use the resulting metrics to cluster an unseen benchmark suite, the NAS benchmarks [3]. In section 6.4, we validate how our clustering was able to correctly gather the similar benchmarks.

Cross validation tries to avoids the *overfitting*. A learning approach overfits when the features are too much trained for a specific data set and

so produces poor results on new data sets.

Overfitting usually occurs in two scenarios: the learning process was to long, or to short. Short training is not reliable because it only predicts well for a subset of the problem while long training may reduce the portability of the predictions.

We presented in this section different methods to manipulate performance data. These methods are used by applications reduction techniques to eventually accelerate the benchmarking. The following section presents these reduction techniques.

2.4 Benchmarks Reduction

Researchers use benchmarks to evaluate design trade-offs for architectures or systems. To exhaustively evaluate a new design idea, researchers evaluate the resulting hardware through all the benchmarks of interest. The benchmarks of interest depend on the evaluation target. Usually each benchmark suite targets a specific domain such as HPC and biology or some hardware components. For example the SPEC CPU2006 [2] proposes two versions: the Integer and the Floating Point (FP) Benchmarks. The first aims to evaluate integer applications behavior on desktop computers while the second targets floating points. Assuming that we want to test the gain of a new floating-point unit of a CPU, it makes sense to directly use the relevant benchmark suite, currently the FP benchmarks.

System and hardware evolutions make possible the emergence of new workloads. Designers adapt the benchmark suites to match and represent these newcomers. Their goal is simple yet challenging to solve: first, benchmarks must be diverse enough to cover all the requirements of the domain of interest, and second, they must be as reduced as possible.

An approach to build a benchmark suite consists into selecting relevant industrial applications and incorporating them into the suite. The issue is that it is very hard to quantify how much a benchmark suite is representative of a domain. In particular, proving that a benchmark suite exhaustively represents a domain is almost impossible [5]. Also, industrial applications benchmarking may be limited due to copyright issues.

For such scenarios, designers implement proxy workloads that rely on the same algorithms [1] as the targeted industrial applications. Unfortunately, we cannot ensure that the original and the proxy applications strictly have the same behaviors. Eventually, industrial applications are long to execute and have a lot of redundancies. They are not the best candidates for the benchmark suites because they evaluate multiple times the same components which leads to a high benchmarking cost.

A common way to evaluate a benchmark suite is to study its representation into a *workload space* [40] with performance metrics (such as the ones

presented in section 2.2). We map the benchmarks into a N dimensional space called a workload space, where each dimension is a performance metric and each point is an application. Close points in the workload space represent benchmarks with similar metrics, hence with similar behaviors and targeting the same architectural components. They are redundant and can be removed without any loss in the information retrieved by the benchmark suite.

The challenging part is how to define and process the dimensions of the workload space. Using wrong metrics to techniques for the benchmarking reduction lead to unreliable and non representative benchmark subsets [45]. Experts search these workload dimensions by relying on their knowledge of the architectures but also on machine learning approaches [4] or statistical methods such as PCA [40].

The workload space can be used to guide the design of the benchmark suites. Designers develop the benchmarks to produce diverse performance behaviors i.e. distant points in the workload space. Metrics quantify the program characteristics, allowing a systematic analysis of the benchmarks [5]. However, the workload space is not an homogeneous representation of the requirements of a domain. In other words and for a fixed domain of interest, some portions of the workload space are more important than the others. So, basing the overall design choices on to relevant and irrelevant space portions leads to a suboptimal result.

This section presents the state of the art methods that are used to lower the cost of the benchmarking. These methods are called benchmark reduction techniques. Most of them take advantage of the workload space characterization. They remove overlapping benchmarks or parts within them while still conserving the diversity of the benchmark suite. This is essential to guaranty that the suite is still representative of the domain of interest. First, we focus on methods which remove redundant applications within a benchmark suite. Second, we present methods that avoid inner application redundancies.

2.4.1 Application Subsetting

A simple way to reduce the benchmarking cost is to remove redundant benchmarks. Developers identify redundant benchmarks by ensuring that the suite diversity is not impacted when the benchmark is removed. There are different approaches to quantify a benchmark suite diversity [21].

The most popular method is to use the suite workload space coverage i.e. the space covered by all the benchmarks within the suite. Two benchmark are evaluating similar components if their workload coverages overlap. Developers study the benchmark suites quality and diversity by looking at their workload spaces.

Joshi *et al.* [46] perform an application subsetting across three bench-

mark suites: MediaBench, MiBench, and SPEC CPU2000. To find a subset of representative benchmarks, they measure different microarchitecture independent metrics to characterize each application. Then they apply PCA to both reduce the dimension space and remove correlated metrics. By using K-means over the workload space, they produce 8 clusters out of 22 of the CPU2000 benchmarks. The cluster representatives predict the average ILP or the speedup of the whole suites across different architectures with a maximal error of `9.1%`. Moreover, they also study the SPEC CPU evolution across four successive generations (up to the `CPU2000`), showing that the main difference over the different versions is that both the data locality has become increasingly poor and the number of dynamic instructions significantly higher.

Phansalkar *et al.* [22] compare SPEC CPU2000 and CPU2006. They measure different architecture dependent metrics using the hardware performance counters through PMU and statistical methods. They also process results with PCA and K-means. Unlike architecture independent metrics, PMU can be biased by the idiosyncrasies of a test configuration. To reduce this bias, they measure the metrics across four compilers and ISA. They conclude that only half of the benchmarks are enough to capture most of the information from both the integer and the floating point CPU2006. In particular, 6 integer programs and 8 floating point programs capture the weighted average speedup with an error of 10% and 12% respectively. They also note that benchmarks from the CPU2000 suite that were retained show similar behaviors while the exclusive CPU2006 benchmarks increase the suite diversity.

Vandierendonck and Bosschere [47] analyze SPEC CPU 2000 execution time. They group applications according to their performance bottlenecks with PCA, showing that SPEC CPU 2000 contains redundant benchmarks. The main difference of this approach is the characterization analysis: the study focus on bottlenecks rather than on metrics. Using 14 or 9 benchmarks instead of the whole suite diminish the feedback information respectively by 5% and 10%.

Hoste *et al.* [58, 42, 41] rely on microarchitecture-independent metrics to build a performance database that is used to predict performance of new programs. They compare the diversity of the microarchitecture dependent and independent metrics over 6 benchmark suites. They build two workload spaces, the first based on dependent and the second on independent metrics, and use them to measure the overall distance between the applications. `98%` of the applications have a large distance within the independent space versus only `57%` within the dependent one. Thus, they show that architecture independent metrics such as ILP or reuse distance exhibit more variances than architecture dependent metrics on these benchmarks. Host *et al.* [58] also compare different approaches to process the performance metrics: normalization, PCA, and genetic algorithms. Their results suggest that genetic

algorithms outperform both PCA and normalization by reducing the prediction error of the benchmark subsetting to `0.89%`.

Bienia *et al.* [61] with statistical and machine learning methods study redundancy between SPLASH-2 and PARSEC applications. They use execution-driven simulation with the Pin tool to characterize program's workloads and collect a large set of metrics, similar to the set presented before by Hoste *et al.* [41] and Phansalkar *et al.* [22]. They use PCA to improve the original feature space and hierarchical clustering to find redundancies between applications. They conclude that workloads for programs from SPLASH-2 and PARSEC are fundamentally different. We note that the two suites are both composed of multi-threaded benchmarks and target HPC. SPLASH-2 and PARSEC distinct workload spaces outline how challenging it is to properly characterize a domain of interest.

A major pitfall of these reduction techniques is that each benchmark is represented by a single set of metrics. The workload characterization methodology can miss underlying behaviors not captured by the selected metrics, thus considering distinct benchmarks as similar. The following section shows how such scenarios can appear and how the reduction methods adapt to support them.

2.4.2 Intra Application Subsetting

Programs can have wildly different behaviors during a single execution [13]. In particular, the program execution changes over time in ways that are often structured as sequences of a small number of reoccurring behaviors, and which are called *behavior phases* [25]. Each interval of execution labeled as a phase is expected to yield some distinct execution properties, e.g. performance metrics, compared to the other phases. Popular methods to define such phases include basic block vectors [48], performance metrics [25], or parallel synchronizations [62].

Previously presented reduction methods assume that benchmarks are homogeneous elements: each benchmark is labeled with a single set of metrics. Yet, applications have phases with distinct behaviors. Porting the applications to new systems outlines the suboptimal application grain clustering: new compiler optimizations or architectures impact each phase differently.

A fine grained analysis enhances the overall benchmark reduction process because:

- distinct phases within the same application can be separated in different clusters. As clusters aim to gather code with similar properties, a fine grained analysis enhance the overall clustering quality,

- representative phases can be selected as proxy instead of whole applications for evaluation. Fine grained analysis accelerates the bench-

marking because it avoids both redundancies across the applications but also within the applications.

As simulating the full execution of an industrial benchmark may take weeks [25], a lot of research try to accelerate it. Fine grained analysis is a popular method to reduce the benchmarking time of architecture simulations: simulators evaluate a few representative phases or execution slices [63], instead of the whole applications.

Usually, the same criteria or performance metrics are used to cluster applications and phases within them. Once representative slices are selected, the simulator evaluate them and extrapolate the whole application behavior. For example, Lafage et al. [63] propose a method to find slices of a program that are representative for data cache simulation. It uses hierarchical clustering on two metrics: memory spatial locality and memory temporal locality both accessed through reuse distances.

SimPoint [13, 48] is another popular simulator using phase clustering. It identifies similar program phases by comparing Basic Block Vectors (BBV). Phases are samples of 100 million instructions. Simpoint reduces simulation time by removing repeated phases. BBV are program dependent, therefore SimPoint cannot use representatives of one program to predict another. Hence, the tool do not take advantage on common redundancies between different applications.

Eeckhout et al. [25] extend SimPoint by matching inter-application phases using microarchitecture-independent features. Carlson et al. [62] propose a simulation method that also define phase representativeness with micro-architecture independent information and data signatures. It targets multi-threaded applications phases, by detecting globally synchronizing barriers. These methods take advantage of both redundancies within and across the applications .

Phase reduction has also been applied to multi-threaded simulations. Extending sampling techniques to multi-threaded simulations is difficult because of the threads interactions. Wenisch et al. [64] and Van Biesbrouck et al. [65] both propose techniques to accelerate multi-threaded simulations. They both build their model under the assumption that each thread is independent. Therefore, they do not support explicit threads synchronization.

Perelman [66] applies the SimPoint [48] methodology to parallel applications using instruction-based sampling. However, Carlson et al. [67] and Ardestani et al. [68] both show that instructions are not a good proxy for execution time in multi threaded programs. Instead, they propose a time based sampling method. Carlson et al. [67] provide a methodology for sampling multi-threaded workloads with up to a 5.8 $\times$ simulation time reduction over the NPB and Parsec benchmarks with an average absolute error of 3.5%.

Carlson et al. [62] also propose BarrierPoint, a sampling methodology which detects globally synchronizing barriers in multi-threaded applications.

BarrierPoint estimates total application execution time through detailed simulation of the most representative inter-barrier regions. Regions representativeness is defined with micro-architecture independent information and data signatures. BarrierPoint achieves an average speed up of 24.7 × over the NPB and Parsec benchmarks with an average error of 0.9%.

Also, phase detection can be extended to message-passing applications. Gonzalez et al. [69] propose to use density-based clustering algorithms with hardware performance counters to detect these phases.

To accelerate the tuning or the evaluation of a benchmark suite on real hardware, designers look for redundancies. Once a redundant application is identified, it can easily be removed from the benchmark suite. On the other side, avoiding multiple phases evaluation within an application is more challenging. The application must be rewritten or the system must jump to the region of interest. Simulators handle this issue: they have a checkpoint restart or fast forward simulation functionalities to directly target the phases of interest. But the real hardware do not provide such abilities. Hence, phase reduction cannot be directly applied on benchmarks that are executed on a real system.

An interesting and versatile approach is code isolation [26]. Code isolation finds and extracts regions of code from an application. Through checkpoint restart techniques, these regions ca be replayed without executing the whole application.

It is important to notice that some of the some code isolators can capture and replay the regions on a real hardware. By matching extracted regions with representative phases, code isolation techniques can take advantage of intra application redundancies to accelerate the benchmarking. Also, depending on the isolation technique, it may be possible to retarget the fragments of code to evaluate new configurations. The following section presents the state of the art of the code isolation.

2.5 Code Isolation

Code isolation was originally proposed to quickly debug and tune large applications. Usually, in scientific applications, the *hotspots*, the regions of the application where most of the execution time is spent, represent a small fraction of the total source lines [70]. Code isolation finds and extracts the hotspots. Then, it evaluates them without executing the whole application.

This thesis proposes to use a code isolator technique presented in chapter 3 to extract regions and quickly replay them across different configurations. To understand our methodology in the following chapter, this section presents the benefits, the challenges, and the state of the art methods on code isolation. Also, we perform a detailed comparison between the isolation method used in this thesis and the others in section 3.7.

Outlining and isolating regions of code from the rest of the applications enhances the benchmarking process because:

- the user can concentrate on each code region separately, with a reduced build and run cost,
- regions can be individually modified to evaluate the payoff of new optimizations,
- different code regions may expose different performance bottlenecks, and react differently to optimizations. So, isolating regions can be used to tune performance at a fine-grain level,
- redundant regions evaluation is avoided to accelerate the benchmarking process.

Effective code isolation and replay has benefits but raises several challenges. First, to be practical, isolation must support many programming languages, applications, and optimizations. Second, the extracted code should be replayable on a variety of target architectures, compilers, or thread configurations. Third, to achieve accurate performance measures, the memory working set and cache state must be captured and restored before each replay. Tracing the memory is a complex and costly process which must be tuned to get a good trade-off between capture overhead and accuracy of replay. Fourth, different invocations of the same region of code may have different performance behaviors, which depend on the working set and cache state of each invocation.

Different approaches try to address these challenges in an appropriate way. Related works have considered two moments in the compilation process to isolate the code: assembly isolation and source isolation. Section 2.5.1 presents the benefits and disadvantage of both methods. Section 2.5.2 describes how some code isolators capture and restore the execution context. Finally, section 2.5.3 presents some use cases of code isolation techniques for performance tuning.

2.5.1 Source Versus Assembly

Pieces of code are outlined by the state of the art method at two moments during the compilation: at source level before any optimizations, or at assembly after the back end process. We compare the two approaches in this section.

Simulators have a checkpoint restart functionality to only evaluate specific regions of code: they perform an assembly isolation. Simulation studies [13, 48, 63, 67, 67, 62] extract regions as blocks of assembly instructions that are evaluated without considering the whole application. Using assembly isolation, Simpoint [13] successfully speeds up architecture simulation by sampling a limited number of assembly instructions.

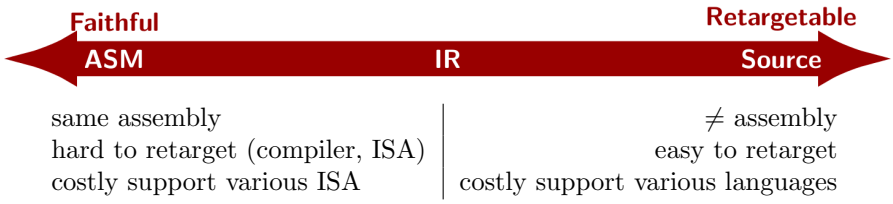


Figure 2.3: Assembly versus source isolation. Source is retargetable but less faithful while assembly is faithful but less retargetable.

Yet assembly isolation is not practical for performance tuning on real hardware because the assembly code cannot be recompiled with different performance flags or easily retargeted to a new architecture. The extraction software is tied to a specific instruction set architecture (ISA). It is also difficult to map assembly regions to source code regions. Compiler optimizations such as loops distribution and fusion respectively split and merge loops. However, this approach is language agnostic and resilient to the compiler effect: what you extract is what is executed.

Source code isolation [71, 26, 72, 73, 74] on the other side is portable. Source code isolation consists into extracting source loops or functions and both recompiling and executing them on different real hardware configurations. Because extraction occurs at source level, before compiler transformations, the performance information gathered during replay can be easily mapped to the source high-level constructs. Kashnikov et al. [75] show the retargetability of source isolation be tuning compiler options.

Unfortunately, source code isolation requires a specific parser and extraction process for each language. Therefore supporting multiple languages is extremely costly because writing a robust extraction pass for complex languages, such as C++, is technically challenging. Finally one must ensure that the source level extraction process does not alter the performance behavior of the original hotspot. Indeed, some of the transformations used during source isolation may hinder compiler optimization passes [71, 73].

Code isolation faces a trade-off between a *faithful* and *retargetable* replay (see Figure 2.3). Retargetable replay allows the extracted code to be executed on new architectures or with different runtime parameters and to be compiled with other optimizations. On the other side, faithful replay is essential to validate the process of using isolated parts as proxies for benchmarking. We consider a faithful replay as an execution which behaves just as the original one: the extraction do not alter the performance.

A common way to evaluate how much a replay faithfully reproduce the original behavior is to study their respective performance metrics (see section 2.2). Depending on the goal, the search requires more or less complex metrics and strict thresholds: studying compiler optimizations may requires

that the replay procures the same assembly as the original one. On the other side, for application debugging, simply reproducing the isolated code semantics is enough. In this thesis, to quantify the similarity between the original and the replay execution, we compute the relative difference between the their respective execution times. Only evaluating the execution time is sufficient to evaluate new optimizations.

Another factor that impacts replay quality is the extraction granularity. Depending on the compilation level (either source or assembly), the granularity of the extraction change. Assembly isolation favors assembly instructions sampled according to their number [66], the elapsed time [67, 68], or contained between successive synchronization points [62]. On the other side, source code isolation [73] target way bigger structure such as loops or functions. A fine grained isolation could accelerate benchmarking processes because more redundancies can be detected and avoided. However, it also complicates the context restoration: since the isolated piece is smaller, it is more sensitive its execution context.

2.5.2 Execution Context

Before replaying an isolated region, the memory state from the original execution must be restored. This ensures that the replay will be equivalent to the original run, even for data dependent branching code. So, to execute isolated regions on real hardware, code isolators must capture and restore the working set for each region. We note that this section does not refer to assembly isolation since simulators already have integrated checkpoit restart strategies.

Capturing the memory working set of the original execution ensures that during the replay, the data accessed are the same as during the original run. However, it is not enough to guarantee that the replay and original run have the same execution time. Indeed, to faithfully capture the performance of the original region it is necessary to warmup the system to match as close as possible the original context. This issue is referred to as the cold start bias [76].

Working Set Capture

Multiple techniques exist to checkpoint the original memory state. A first approach to capture the working set, used by Codelet Finder [74, 71], takes a full snapshot of the original application address space. The application is frozen using the `ptrace` system call, then a helper process dumps the memory contents to disk, and returns the control to the original application. Full memory dumps are large, but have the advantage of perfectly capturing the memory layout, handling pointer aliasing, and preserving the relative alignment and the offsets among data structures. Nevertheless, a full snapshot

of the application memory for each codelet can be prohibitive in terms of memory and replay time.

Code Isolator [26] reduces the memory dumps by analyzing the static data flow of the original application to determine which data structures need to be captured. This method produces small dumps because only the required data are captured, but cannot deal with pointer aliasing. Astex [72] captures the convex hulls of the memory accesses. However it does not preserve the data layout information and does not remap the memory at the same addresses during replay. Therefore pointer based structures such as linked lists are not supported.

Cache Warmup

Usual techniques [76] mitigate cold start bias by modeling the warmup effects during a window of time preceding the region of interest. Multiple heuristics [77, 78] have been proposed to optimally determine the window's size.

Two main approaches have been proposed in the literature for cache state warmup in code isolation. The first approach is to warmup the cache by running a few warmup executions of the region itself [72, 74, 71]. It is an optimistic heuristic that assumes that the codelet working set is hot in the original run. The rationale is that the hotspots forming the regions are loop based and thus can be warmed up by their own previous iterations. This heuristic proves to be efficient in many cases [71].

The second, more accurate approach, warms up the cache by replaying the history of the memory accesses in a simulator [13] or using a warmup routine [26]. These techniques require to trace memory accesses which is costly and incurs significant slowdowns [79].

2.5.3 Performance Tuning with Source Isolation

Code isolation is used for performance tuning. Section 2.4.2 presents how assembly isolation speeds up architecture simulations by avoiding redundant phases. However, unlike source isolation, it cannot be applied to tune real hardware configurations. We describe bellow some of the source isolation techniques

Lee et al. [26] introduce the concept of code isolation for debugging and iterative performance tuning. Their tool, Code Isolator, leverages the Stanford SUIF compiler to outline and generate codelets. They use Code Isolator on a finite element application, LS-DYNA, to quickly evaluate the L1 cache misses of the hotspots.

Petit et al. [72] and Liao et al. [73] both use source code isolation for automatic kernel tuning and specialization. The tool developed by Petit et

al. [80], Astex, uses code isolation to accelerate and facilitate value profiling and code specialization for speculative execution.

Akel et al. [71] evaluate the Codelet Finder (CF) tool, developed by Caps Entreprises [74] and study under which conditions isolated regions preserve the performance characteristics of the original programs. Over the NAS benchmarks, 63.6% of the isolated regions have the same assembly as the original hotspots and 81.6% of the replays have the same execution time performance as the original hotspots.

Source code isolation focus both sequential and parallel codes. While there are many sequential code isolator frameworks, to the best of our knowledge, only Liao et al. [73] has proposed a parallel OpenMP source code isolator. Their approach is based on the ROSE [81] compiler infrastructure. Isolation is achieved through a source to source outliner which extracts tunable kernels out of OpenMP programs. They use the isolator to accelerate computation kernel tuning and show how to find the best parameters for the number of OpenMP threads, the schedule policy, and the chunk sizes. Liao et al. isolate parallel `for` loops at the source level. Outlining a source loop from a parallel region removes the loop from the lexical extent of the parallel region and alters the semantics of the program because the scope of OpenMP data clauses (private, shared, or reduction) is lost. The source outlining approach requires an additional step that repairs the lost scopes.

To conclude this section, there is a wide range of isolation techniques. However it is challenging to get an isolated code which is both faithful and retargetable: source isolation may not be faithful just as assembly isolation is not retargetable.

In this thesis we use a third approach for the isolation: the compiler Intermediate Representation (IR). IR appears to be a good trade-off between them: it is more faithful than source isolation and is more retargetable than assembly isolation. Section 3.2 details this new code isolation approach.

In the following section, we describe some popular tuning strategies. This thesis applies some of these strategies on isolated pieces to enhance the tuning process.

2.6 Tuning Strategies

The current increase of architecture complexity, multiple cores, out-of-order execution, complex memory hierarchies, and non-uniform memory access (NUMA) complicates the performance characterization. Moreover, to switch to multicore, systems are not only complex but also diverse. Current top500 supercomputers are composed of small simple GPU processors, regular desktop processors and hybrid Xen Phi cores. Finding the best architecture is an important problem for high performance computing, data centers, and embedded computers.

Also, achieving full efficiency on a system requires a fine tuning of parameters such as the degree of parallelism, thread placement or compiler optimization. Runtime and compiler standard parameter levels (such as `-O3` compiler flag or `scatter` thread placement) achieve good-enough performance across most of the codes and the architectures. But they cannot take advantage of target-specific optimizations since they must correctly work on a large panel of architectures.

Finding the optimal parameters may lead to substantial improvement but is a costly and time consuming process. For example, compilers such as LLVM [17] `3.4` provide more than sixty passes. Passes have different impact depending on their order of execution and can be executed many times. This leads to a huge exploration space: considering only sequences of 30 passes requires to explore a space over 60^{30} points.

So, to achieve full efficiency, developers must select for their applications the best architecture with the best tuned execution parameters. This process involves running or simulating the applications over the different system configurations. Developers take advantage of the previously described methods to prune the exploration space. They can also rely, in some cases, on benchmark reduction and isolation techniques or on analytical prediction models to accelerate the search.

The most straightforward method is to directly evaluate new architectures or the parameters. Iterative compilation [82] is a well known automated search method for solving the compiler optimization phase ordering problem. The idea is to apply successive compiler transformations to a program and to evaluate them by executing the resulting code. Similar execution driven studies [83, 84] explore the efficiency of different thread placement strategies or frequencies. Smart search algorithms [85, 18] through the parameter space reduce the evaluation cost. Genetic algorithms [57, 19] or adaptive learning [86, 20] accelerate the search by avoiding unnecessary parameters.

There have been many works on iterative compilation. Most of the research try to accelerate the iterative compilation by pruning the exploration space [85, 18, 57, 19, 20] usually using genetic algorithms (see section 2.3.3).

Iterative compilation takes also advantage of benchmark reduction. The problem with some of the traditional fine grained benchmark reduction techniques (see section 2.4.2) is that they operate at assembly level [13]: they cannot be directly used for compiler tuning. Fursin and al. [87] still managed to take advantage the application phases: they evaluate multiple optimizations for a region with a single run by versioning the different iterations of the region. However, they do not use any code isolation techniques so they cannot focus the search which is problematic when a region of interest has a few invocations compared to the others.

2.7 Conclusion

In this chapter, we saw the different concepts required to understand this thesis. In particular, we presented the benchmark reduction strategies, the code isolation, and some tuning strategies. These methods are combined in this thesis to improve the tuning process.

In the next chapter 3 we propose a novel code isolator. Then, chapter 4 presents how this isolator can take advantage of the benchmark reduction strategies to accelerate the benchmarking process.

Chapter 3

CERE: Codelet Extractor and REplayer

Contents

3.1 Introduction

In this chapter, we present **CERE** (Codelet Extractor and REplayer), a code isolation framework based on LLVM. CERE finds and extracts the hotspots of applications as isolated fragments of code, called *codelets*. CERE supports both serial and OpenMP applications: codelets are extracted from hot loops and OpenMP non nested parallel regions. Codelets can be modified, compiled, run, and measured independently from the original applications.

Studying isolated regions instead of whole applications is attractive because the user can concentrate on each codelet separately, with a reduced run cost. Codelets can be individually modified to evaluate the payoff of new compiler optimizations or runtime parameters such as the number of threads

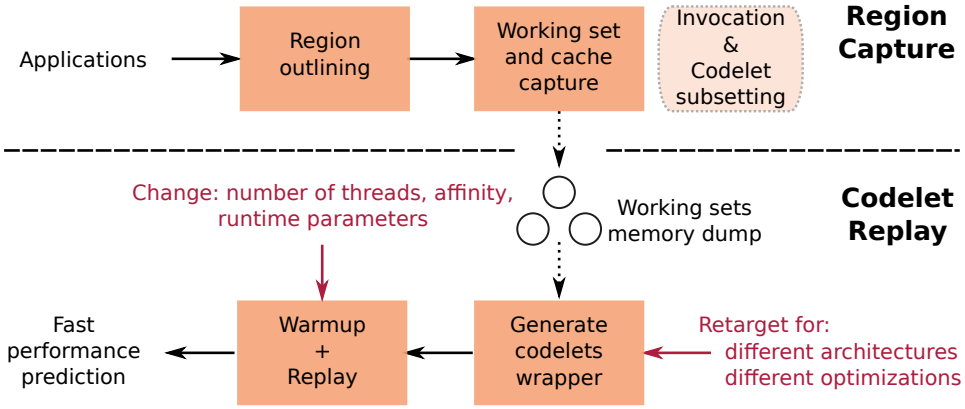


Figure 3.1: CERE usage diagram. Applications are partitioned into a set of codelets, which may be pruned using different criteria. A subset of representative codelet invocations are selected and captured. The codelets can then be replayed with different options and on different targets to do piecewise optimization or performance prediction.

or the thread affinity. Different codelets may expose different performance bottlenecks, and react differently to optimizations. Isolating codelets allows tuning performance at a fine-grain level.

CERE is composed of a compiler instrumenter, a clustering module to find representative invocations and representative codelets, a working set capture mechanism operating at the system memory page granularity, a realistic cache warmup, and a piecewise compiler module. This chapter presents all these components except the clustering approach which is detailed in the next chapter. CERE is made available under the GNU Lesser General Public License (LGPL) version 3 open-source license at: `https://github.com/benchmark-subsetting/cere`.

Unlike previous approaches, CERE isolates codes at the compiler Intermediate Representation (IR) level after Clang front-end translation but before LLVM middle-end optimizations. Therefore CERE is language agnostic and supports many input languages such as C, C++, Fortran, and D. Also, LLVM middle-end and back-end can respectively recompile the codelets with new optimizations and retarget them to new architectures.

Figure 3.1 presents the full CERE pipeline. CERE operates in two main steps: *capture* and *replay*.

During the capture, CERE detects the hotspots and instruments them in the original program with calls to the memory capture library. The execution state is captured at the start of each region of interest.

During the replay, the user selects a particular region to replay. CERE generates a standalone codelet that restores the execution context and jumps to the region of interest. Codelets can be retargeted to evaluate new archi-

tectures, compiler and runtime optimizations.

CERE leverages the LLVM compiler framework to implement the instrumentation and isolation passes. LLVM provides a rich API for manipulating the IR code, which greatly simplifies the process.

Let us consider a benchmark suite. CERE takes as an input the source files of the applications. All the languages supported by the LLVM frontends (C, C++, Fortran, D, etc.) are accepted. The loops or OpenMP regions are outlined and instrumented with profiling probes to identify the applications hotspots. The hot regions are kept and form the full codelet set.

By using reductions strategies, CERE accelerates the benchmarking process of the full suite. CERE prunes the full codelet set and only keep a reduced set of representative codelets. Then, a clustering algorithm analyzes the invocation performance trace of each codelet to find a representative subset of invocations. These strategies are detailed in the next chapter 4. The memory and cache state of each selected codelet invocation is then captured and dumped to the disk.

The output of this process is a set of representative codelets and invocations, which can be redistributed, recompiled and replayed on different systems and architectures. Through a prediction model, the codelet set can be used as a proxy for the benchmark suite to evaluate new optimizations or execution systems. Since the optimization tuning is performed on separate pieces, CERE can detect the best optimizations for each piece.

We note that loops and OpenMP regions follow a common capture and replay process. Yet, due to the parallelism, replaying OpenMP regions requires some additional steps to guaranty a correct codelet execution. Section 3.5.5 presents these steps.

The capture requires executing the application once to get the execution context of all the regions of interest. If the user wants to measure an application on a single architecture with a fixed number of threads, extracting and replaying the codelets does not pay off. It is quicker and more accurate to fully measure the benchmark. But, if the user is interested in comparing the performance of different architectures, thread configurations, or compiler optimizations, the codelet approach is quickly amortized because codelets are only extracted once but replayed many times.

This chapter presents CERE in details. In section 3.2, we justify why we extract the code at the IR instead of the traditional source or assembly isolation. Section 3.3 presents the OpenMP challenges that make multithreaded isolation more complex. In particular, we motivate our choice of why we focus on extracting outer OpenMP parallel regions. Section 3.4 shows how CERE selects good candidate regions for the codelet extraction process. Section 3.5 demonstrates the codelet extraction and replay process. CERE is an isolation approach that allows to tune each piece of code separately. Section 3.6 shows how CERE combines all the best found opti-

mizations in a single binary. Finally, we review and compare CERE to other existing code isolation methods in section 3.7.

3.2 Intermediate Representation Isolation

As discussed in the background in section 2.5.3, CERE (Codelet Extractor and REplayer) targets code isolation at the LLVM [17] compiler IR. IR extraction provides multiple advantages over source or assembly code isolation techniques and is a good trade-off between them. Extracting codelets at the IR level is much simpler than at the source code level which requires parsing complex input languages. It also facilitates the process of instrumenting the code, capturing the memory and outlining the codelet thanks to the powerful integrated flow analysis passes.

Unlike assembly extraction, IR codelets provide many performance tuning opportunities. For instance, the codelet can be replayed using different optimization passes or compiler versions, enabling compiler flag autotuning [75]. By leveraging the available LLVM code generation back-ends, codelets can be replayed on different architectures to facilitate system co-design.

The drawback of the CERE IR extraction is that CERE is tied to the LLVM compiler. However, CERE supports all of LLVM front-ends and back-ends with no extra engineering cost. For example CERE has been tested on all NAS and SPEC 2006 FP programs. While C and C++ benchmarks used the Clang front-end, Fortran programs used the GCC gfortran front-end through the dragonegg plugin [88]. CERE also works on less mainstream languages. For example CERE successfully extracts codelets from D [89] applications compiled with the LLVM D front-end, LDC.

In the following section, we explain why outlining parallel applications is more complicated than serials.

3.3 The Challenge of OpenMP Isolation

Isolating parallel regions of code raises new challenges such as the parallel non determinism of the execution, the support of different thread configurations or the NUMA effects. So, in order to extract regions from OpenMP applications, CERE needs to tackle these challenges.

3.3.1 OpenMP Support

In OpenMP programs, the application concurrency is described through a set of compiler directives and library calls. For instance, a parallel region can be declared using the directive `#pragma omp parallel`. The left part

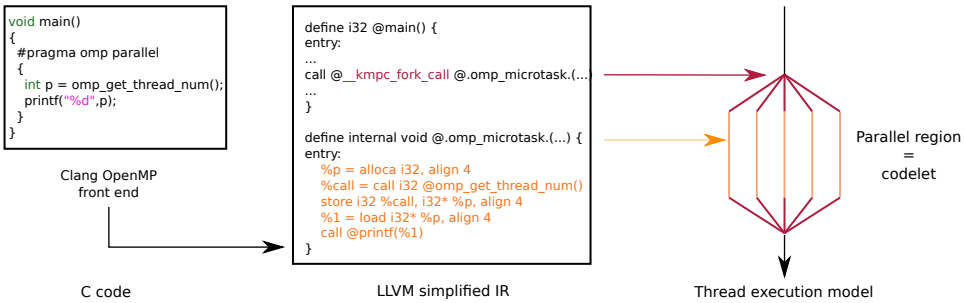


Figure 3.2: Clang outlines each C parallel region as an independent IR function:`omp_microtask`. The call to `kmpc_fork` spawns a pool of threads that runs the outlined microtask.

of figure 3.2 shows a simple C OpenMP program where each thread prints its thread identifier.

In most compilers, including GCC and LLVM, parallel directives are expanded in the front-end before doing any code optimization [90]. Usually, the first step in OpenMP expansion is *outlining* parallel regions. To outline a region the compiler moves the region code inside a separate function. The compiler preserves data dependencies by passing live-in and live-out values through the outlined function arguments. Then the original region is replaced by a call that spawns multiple threads running the outlined function.

The LLVM compiler infrastructure provides a partial support for OpenMP 4.0 [91] since version 3.4. Figure 3.2 shows how LLVM expands a simple OpenMP directive and the IR code it produces. LLVM outlines the region code in a `microtask` function. `kmpc fork`, an OpenMP Runtime library function, spawns a pool of threads. Then, every thread runs the outlined `microtask` function which describes the region parallel work. To define the `kmpc fork` function, we need to link the OpenMP application with the Intel/LLVM OpenMP runtime library [92].

3.3.2 OpenMP Isolation

Parallel codelets should satisfy three important properties. First, each codelet must capture a specific region of code in the original application. Second, it should be possible to change the number of threads and other runtime parameters at replay and it should be possible to replay codelets across different architectures. Third, the set of extracted codelets must faithfully capture the behavior of the original application so that it can be used as a proxy for measuring performance and scalability. In particular, each codelet replay must be deterministic: different replays of the same codelet should execute the same code and have the same performance.

Multi-threaded execution is a well known source of non determinism:

race conditions and synchronization delays between threads may change the order of the operations from one execution to the next. In particular, when multiple threads are running, each one may be executing a different region of code. This makes it difficult to isolate a particular region of code.

Loops are attractive candidates for code isolation. Akel and al. [71] show that they cover more than 90% of the NAS SER benchmarks original execution time. However, extracting loops is incompatible with the OpenMP isolation requirements.

Let us consider a loop in a parallel region executed with multiple threads. The memory context at the beginning of the loop depends on the thread configuration. So, we can only replay such loops at the thread configuration that we used to capture them. In other words, we cannot explore through loop isolation multiple thread configurations with a single capture. Also, there are no synchronizations encapsulating the loop. When multiple threads are executing the loop in the original execution, they do not necessarily start or finish its execution at the same time. However, when we replay the loop with multiple threads, all the threads start at the same time and do not execute further instructions when they finish the loop. For these two reasons, isolated loops in parallel regions may not faithfully reproduce the original behavior.

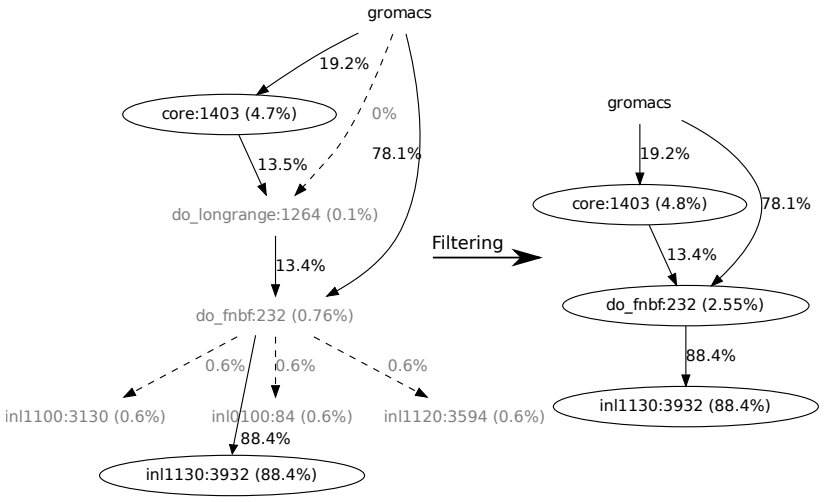
To avoid thread non determinism issues, CERE codelets start at the beginning of a parallel region and finish at the end of the region. The beginning of an OpenMP region is a global synchronization point where all threads positions in the program are known. Capturing codelets at the start of the region also avoids to perform a memory capture for each thread configuration: it enables changing the number of threads at replay. Indeed, the capture happens just before the call to `kmpc fork` that decides how many threads are spawned. The capture of the thread stack and the Thread Local Storage (TLS) are also simplified as they are handled by the `kmpc fork`.

Since OpenMP regions are already outlined, we simply isolate the `kmpc fork` that calls them. We detail the chackpoint restart process bellow in section 3.5.1.

3.4 Application Partitioning

To find interesting regions for performance optimization, CERE concentrates on the application hotspots. CERE supports both serial and OpenMP applications. So, this section presents how CERE selects codelet candidates among loops and OpenMP regions.

In sequential scientific applications, performance is mainly concentrated on loops. Similarly, parallel regions contain all the parallelism of the applications. Therefore, CERE considers all the loops and parallel regions of



codelet	error (%)
do_longrange	2.3
do_fnbf	1.1
inl1130	0.1
inl0100	10.1
inl1120	4.6
inl1100	7.5
core	2.5

Figure 3.3: *(top)* CERE call graph, before and after filtering, for SPEC 2006 gromacs. Each node represents a captured codelet. The percentage inside the node is the codelet's self time. Edges represent calls to other codelets, the edge percentage is the time spent in calls to those nested codelets. *(bot)* Replay percentage error of gromacs codelets using Working Set warmup.

the original program as potential candidates to be extracted as codelets. Then, CERE profiles the candidate regions and keeps the ones significantly contributing to the total execution time.

CERE provides two region level profiler modes. First, a low overhead sampling profiler based on the Google Performance Tools library [93]. Second, an instrumentation profiler, which is slower but more precise. When using sampling, CERE outlines the regions before executing the application; all the outlined regions are profiled using Google's performance toolkit. When using the instrumentation mode, probes to capture the time stamp counter are inserted directly before and after the region.

Despite our efforts to restore the original execution environment through warmup code reinlining, and variable cloning (see sections 3.5.3), the codelet replay sometimes does not match the original region performance. Clearly, those *ill behaved* codelets cannot be used as a performance proxy in bench-

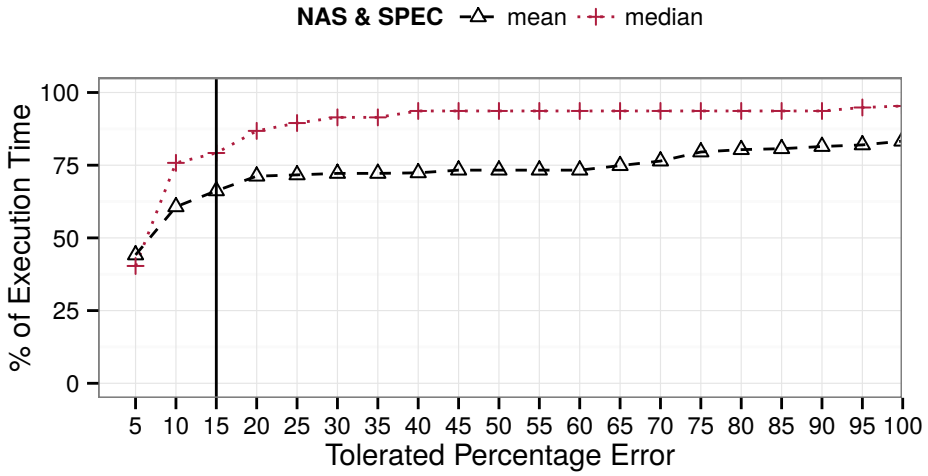


Figure 3.4: Mean and median captured execution time as a function of the tolerated replay error. the NAS and SPEC 2006 FP benchmarks. The mean is lower than the median due to the IO-intensive and short kernel benchmarks described in the validation section 5.3, which skew the distribution.

marking or optimization studies. Therefore CERE runs a sanity check where it replays and profiles each codelet to ensure that only *well behaved* codelets are returned to the user. The tolerated discrepancy threshold can be configured. Its sensitivity is presented on figure 3.4.

After collecting profile data, CERE produces an annotated call graph such as the graph in figure 3.3. This call graph is then pruned by removing the regions contributing for less than 1% to the total execution time. We note that in many experiments, we also remove OpenMP regions contributing for less than 5%. Furthermore, if an ill behave codelet is detected, CERE also removes it from the call graph. When removing a region from the call graph, we propagate its *self time* to its parent codelets. In our example, the time from the three `inl` removed regions is propagated to their caller `do_fnbf`. In the example of figure 3.3, since all the codelets match the original execution time, none would be removed.

Once the removal process is over, CERE extracts all the remaining loops as standalone codelets.

The above selection algorithm extracts all the well behaved codelets whose contribution to the program execution time is over a given threshold. To trade coverage for replay time, for example, when using codelets to accelerate system benchmarking, the user wants the minimal set of codelets that can be quickly replayed while simultaneously capturing the application performance accurately. For this purpose, CERE includes a codelet *selector* that uses integer linear programming to find an optimal codelet set. It is similar to the tuning selection algorithm proposed by [94]. In the example in

figure 3.3 it would drop codelets `do_fnbf` and `core`, losing less than 7.35% coverage but significantly reducing the replay cost.

3.5 Codelet Capture and Replay

This section presents how codelets are extracted and replayed. Table 3.1 demonstrates this process for a codelet. In Step 1, the input program is compiled to LLVM IR by the compiler.

In Step 2, the region to be captured is outlined in a separate function using the CodeExtractor LLVM pass. Codelets are either loops or OpenMP parallel regions. If the region is a loop, CERE outlines the loop body. Else, it outlines the `kmpc_fork` call of the parallel region. The OpenMP support of CERE and LLVM is discussed in section 3.3. Then, CodeExtractor [95] does a flow analysis to detect all the live-in and live-out dependencies of the region to extract. This pass simplifies the codelet extraction process, since it extracts the region code in its own function. The codelet region is outlined in a new function. Finally CodeExtractor inserts a call to the outlined function in the original code. The dependencies are preserved by passing the live-in and live-out values through function arguments. CodeExtractor is also the starting point for the portable memory capture mechanism discussed in section 3.5.2.

Step 3 generates the instrumented binary for memory capture. It inserts special calls to the capture library before and after the outlined region in the original application. The calls are used to trigger the memory and cache warmup state captures, described in sections 3.5.2 and 3.5.3. The instrumented binary execution generates a set of dump files that can be used during replay to restore the memory state and to warmup the caches. The aim is to ensure that the replay context closely mimics the original execution context.

Step 4 is the replay mechanism. It generates a wrapper to directly call the outlined region. This wrapper restores the original execution environment, such as variable cloning, cache and memory restoration. The replay IR code can be compiled with different optimization flags to find the best performance configuration. Or it can be compiled with different back-ends to evaluate the performance on multiple targets. The replay process is detailed in section 3.5.4.

3.5.1 Codelet Checkpoint-Restart Strategy

Traditional checkpoint techniques [96] can save the state of a program at any given point. A full dump of the memory and of the register banks including the program counter allows to restart the program after capture. Yet, this approach requires that the replayed code keeps the same code layout and uses exactly the same registers as during the capture. Traditional

#	Step	A loop output

1	**Front-end:** Transform the C, C++, Fortran, D possibly OpenMP input program into LLVM Intermediate Representation (uses Clang, dragonegg, or LDC).	`original:`
		`%0 = load i32* %i, align 4`
		`%1 = load i32* %s.addr, align 4`
		`%cmp = icmp slt i32 %0, %1`
		`br i1 %cmp, ; loop branch here`
		`label %for.body,`
		`label %for.exitStub ...`

2	**Outline:** Outline either the loop region or the `kmpc_fork` call of the OpenMP region to extract. Flow analysis is used to compute all live-in and live-out values which are passed as arguments. (see section 3.5.1)	`define internal void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr) {`
		`%0 = load i32* %i, align 4`
		`...`
		`ret void }`
		`original:`
		`call void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr)`

3	**Capture:** Insert calls to CERE capture library. Run the instrumented binary to capture the runtime state. (see sections 3.5.2 and 3.5.3)	`define internal void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr) {`
		`call void @start_capture(i32* %i,`
		`i32* %s.addr, i32** %a.addr)`
		`%0 = load i32* %i, align 4`
		`...`
		`call void @end_capture()`
		`ret void }`

4	**Replay:** Generate minimal replay wrapper that calls the outlined region. Compile and run replay possibly with new optimization options or on a different architecture. (see section 3.5.4)	`define i32 @main(i32 %argc, i8** %argv){`
		`; Allocate clone variables`
		`%i = alloca i32`
		`%s.addr = alloca i32`
		`%a.addr = alloca i32*`
		`; Restore arguments and memory`
		`call void @restore(...)`
		`; Call outlined region`
		`call void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr`
		`; Anti-deadcode for live-out values`
		`call void @antideadcode(i32* %i)}`

Table 3.1: Codelet capture and replay main steps.

checkpointing is therefore not suited to test compiler optimizations which may remap registers or change code layout. Also it limits codelet portability to architectures sharing the same Application Binary Interface (ABI) and register layout.

Codelet based piecewise iterative optimization and architecture selection require a portable checkpoint-restart strategy. The outlining pass (Step 2 in Table 3.1) wraps and isolates the region of interest (either the loop body or the `kmpc fork` call) inside a separate function. Because the region now follows a function call, we can guarantee that the accessed data is either in memory or is passed as arguments to the outlined function.

This enables us to simplify the memory capture process: only the memory and arguments to the outlined function must be recorded. Also, the outlined function prototype acts as a clean interface that enables us to recompile and apply transformations to the codelet before replay. Because no assumptions about the register layout are made, codelets are portable across architectures that do not change the memory layout, such as word size and endianness. Our tests have shown, for example, that our codelet replayer allows to recompile changing optimization flags, capturing on -O0 but replaying on -O3, or changing architectures, capturing on Core Duo and replaying on Atom.

Both loop and OpenMP region Codelets portability has been extensively tested and works across six different Intel CPU generations (Atom, Core 2 Duo, Nehalem, Sandy Bridge, Ivy Bridge, and Haswell) running various 64-bit Linux distributions on the NAS and SPEC codelets.

We also tested codelet portability between an Intel Core i3 running 32-bit Linux and an embedded target, an ARM1176JZF-S on a Raspberry Pi Model B+ running 32-bit Linux. This test was conducted on a simple benchmark summing the elements of a large integer array. The capture was performed on the Core i3 system and could be faithfully replayed on the ARM embedded target.

In a second experiment the capture was done on the same Intel Core i3, but this time the system was 64-bit Linux; therefore some of the dumped pages were over the 32 bit address space limit. The replay on the ARM system failed because addresses over 32 bits overflowed. This example illustrates the limits of CERE: portability does not work out of the box for systems with different memory address sizes. Nevertheless, in this case we were able to overcome this limitation by manually remapping the memory dump to fit the 32 bit address space by masking the address' upper bits. After the manual remapping, we were able to replay the benchmark in the ARM1176JZF-S processor.

The outlining process guarantees that codelets captured once, can be distributed and replayed many times on multiple architectures.

To ensure a semantically accurate replay, we need to capture the memory state of each codelet. Also, to faithfully reproduce the performance of the

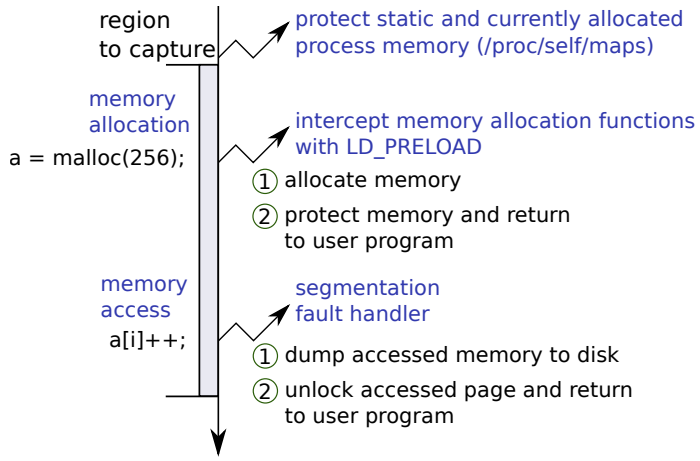


Figure 3.5: The memory dump process operates at page granularity. Each page accessed is dumped by intercepting the first touch using memory protection support.

original execution, codeles must be executed with a similar the cache state as the original execution. In the following sections, we decries how CERE captures both the memory and the cache state of each region.

3.5.2 Capturing the Memory

CERE proposes a page level granularity snapshot. Using the memory protection mechanism we capture the memory pages containing the working set. During replay we remap this set of pages at their original addresses. This ensures that the dump remains small and fast. Furthermore, the replay works even with complex pointer aliasing, because the memory layout is preserved.

CERE captures codelet's working sets by intercepting accesses to the memory pages. Page level capture combines the advantages of the full memory dump in Codelet Finder [71] with the advantages of the data flow capture in Code Isolator [26] or Astex [72]. First, CERE guarantees that all the memory locations accessed by the original program are dumped, including aliases that are not handled with static analysis. The set of captured pages contain the full original working set. Second, because only the touched pages are saved, the memory dump is the smallest page-granularity over-approximation. Therefore, it can be easily stored and distributed.

Figure 3.5 shows the memory dump process. First, all the memory pages of the process are protected and a special segmentation fault handler is set. Each time a protected page is accessed, a segmentation fault occurs and triggers the handler. The handler dumps the touched memory page to disk and unprotects it before continuing the original program execution.

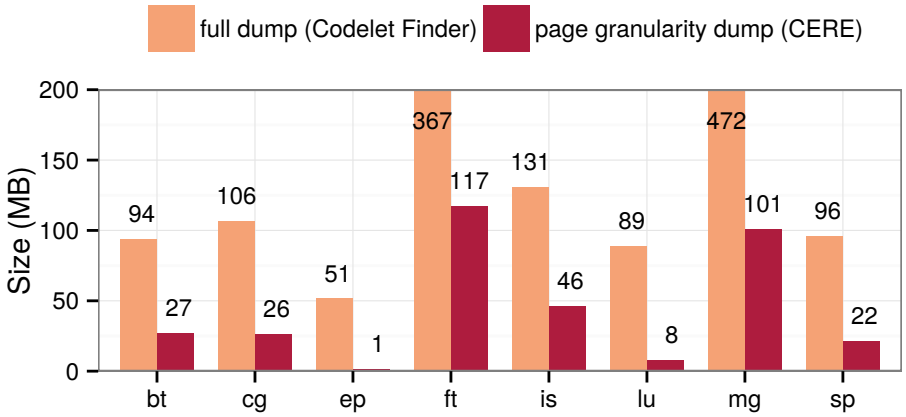


Figure 3.6: Comparison between the page capture and full dump size on NAS.A benchmarks. CERE page granularity dump only contains the pages accessed by a codelet. Therefore it is much smaller than a full memory dump.

It is important to protect all newly allocated memory. If memory is allocated but returned to the user unprotected, the tracer misses the access to the memory segment. We catch all calls to the memory allocation library, such as malloc, realloc, or memalign using the `LD_PRELOAD` mechanism. However, some special memory sections must not be protected, such as the pages containing the code of the tracing library and the segmentation fault handler itself. Therefore CERE carefully avoids protecting its own pages and system specific memory segments.

Figure 3.6 compares the average dump size for the NAS benchmark codelets for two techniques: CERE's page granularity dump and Codelet Finder's full dump. As can be seen, the page granularity dump is 3 to 51 times smaller than a full dump. With this technique CERE extracts light portable codelets from industrial application with large working sets.

3.5.3 Capturing the Cache State

In this section, we address the problem of cache warmup for codelet replay previously discussed in section 3.4. CERE includes three warmup strategies: *Cold*, *Working Set*, and *Page Trace*.

The *Cold* strategy does not do any warmup before executing the codelet. It is therefore inaccurate but has no overhead. It can be used on long codelets for which the cold start bias is negligible.

The *Working Set* strategy prefetches the full working set of the codelet before its execution. It is an optimistic strategy that assumes that the codelet working set was already in cache in the original execution.

The *Page Trace* strategy mitigates cold start bias by replaying a memory

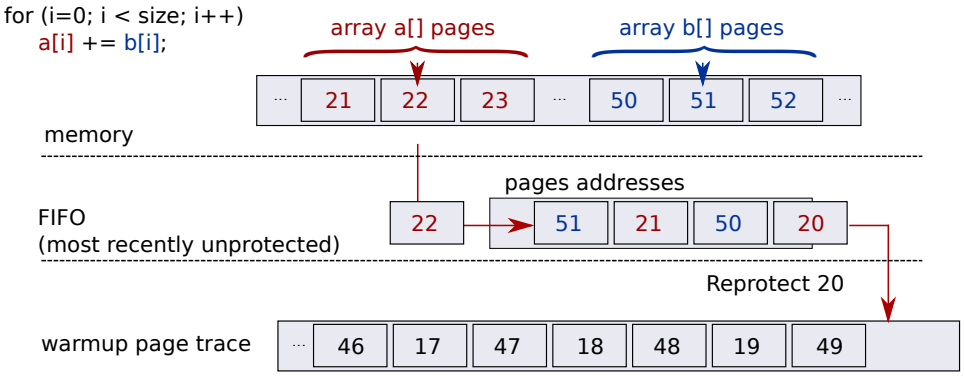


Figure 3.7: Cache page tracer on a simple codelet adding two arrays. Each page access is logged. Recently unprotected pages are kept in a FIFO with N slots (here $N = 4$). Once evicted from the FIFO, the pages are protected again.

trace at a page level granularity. It is less accurate than a full memory trace warmup, but much faster. It provides a good trade-off between cost of codelet capture and replay accuracy. The technique is similar to the page tracing technique in [97].

Our page tracer is implemented on top of the memory dump process described in section 3.5.2: all the memory pages are protected, and a special segmentation fault handler intercepts accesses to memory. The difference is that unlike the memory dump in which only the first touch to a page is important, the page tracer should capture all the memory accesses to a page.

An exact, but costly, technique involves reprotecting each page after each access. Because this page is immediately reprotected, further accesses to the page will provoke a segmentation fault and will be logged by the tracer. The slowdown is too high for our purposes.

To reduce the cost of the technique, we keep the most N recently accessed pages unprotected. The tracer uses a FIFO to track the recently accessed pages. Each time an access to a page is detected, the page is unprotected and added to the FIFO. The oldest page is popped from the FIFO, reprotected, and added to the page access log. Figure 3.7 illustrates this approach on a codelet that adds two arrays together.

If a codelet simultaneously accesses less than N separate memory streams, the FIFO ensures that a page remains unprotected for all the consecutive streamed accesses. Assuming a stride-one access, the page tracer handler is only invoked every 4096 byte (for 4K pages). Therefore we choose N higher than the number of separate memory streams accessed by most loops. Kashnikov et al.[75] show that most application loops use less than 16 simultaneous streams. In our experiments we choose $N = 64$. Nevertheless,

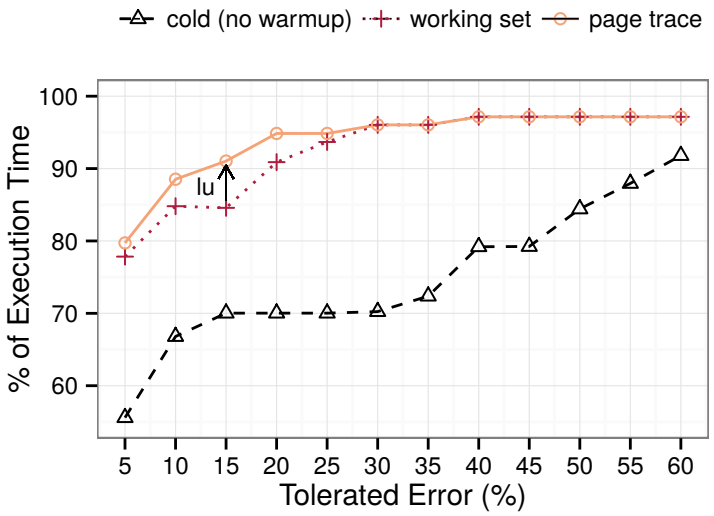


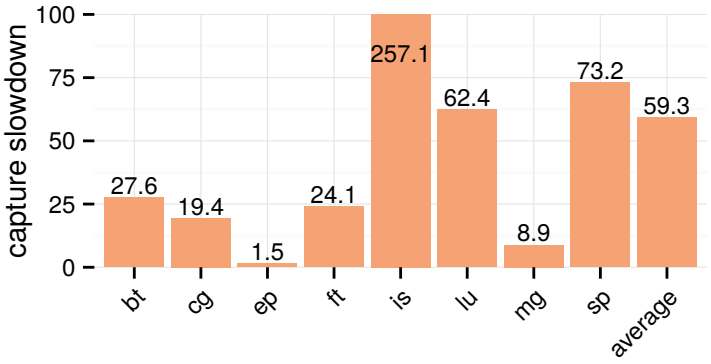
Figure 3.8: Comparison of the three cache warmup techniques included in CERE on NAS codelets. The plot shows the percentage of execution time as a function of the replay error. Page Trace and Working Set warmup achieve the best results. Page Trace is more accurate than Working Set on the LU benchmark.

by keeping the most recent N pages unprotected, our trace is less accurate. In the code of figure 3.7 for instance, each cell of array `a` is accessed twice (it is first read then written to), but the page tracer only sees the first access. When interpreting the trace, one must keep this inaccuracy in mind: the trace presents which pages were accessed but neither how many times nor the precise ordering.

We note that the current version of CERE does not support the page tracer strategy for multi-threaded applications. Since each thread touches its own pages, to faithfully restore the context, we must keep a trace of pages for each threat. OpenMP isolation currently rely on the working set strategy, by replaying the codelet over itself.

Figure 3.8 compares the three warmup techniques implemented in CERE on the NAS benchmarks. The *tolerated error* is the maximum percentage difference between the original execution time and the replayed execution time. The plot shows the percentage of execution time of NAS codelets replayed with an error smaller than the *tolerated error*. For example, if we use the *Cold* strategy, 70% of the execution time can be replayed with an error under 15%.

Figure 3.9 shows the overhead of the capture run for the NAS benchmarks. For each benchmark we compute the slowdown between the original run-time and a full capture run. This measure includes initialization of the capture library, writing the dumped pages and the memory trace logs to



	ATOM 3.25	PIN 1.71	Dyninst 4.0
cg.a	98.82	222.67	896.86
ft.a	44.22	127.64	1054.70
lu.a	80.72	153.46	>>301.4
mg.a	107.69	168.61	989.53
sp.a	67.56	93.04	>>203.66

Figure 3.9: CERE capture overhead. *(top)* CERE capture overhead (Class A). For each plot we measure the slowdown of a full capture run against the original application run. The overhead takes into account the cost of writing the memory dumps and logs to disk and of tracing the memory accesses during the whole execution. *(bot)* Overhead of other memory tracers. We compare to the overhead of other memory tracing tools as reported by [79]. Gao et al. did not measure bt, is, and ep.

	Original	Page Trace	Working Set	Cold
Time (e+11 cycles)	2.34	2.40	2.08	3.13
Error (%)	-	2.56	11.45	25.16

Table 3.2: CERE predicted execution times of the FDTD codelet compiled with `-O2` using the three warmup techniques.

disk for all the codelets in the application. `IS` is particularly slow because one of its codelets accesses memory randomly. This rapidly fills the pages FIFO and slows down the tracer. Figure 3.9 also compares CERE capture cost with the overhead of other memory tracing tools. CERE overhead is similar to ATOM 3.25 overhead and lower than PIN 1.71 and Dyninst 4.0 overhead.

When the user is only interested in a single codelet, CERE includes a single-capture mode which is much faster. The capture library fast-forwards the execution and starts the memory tracer and memory protection when the execution is reaching the zone of interest, but leaving a big-enough window to capture the cache warmup log. Different studies [77, 78] propose multiple techniques to determine the best window size.

We observe that the *Working Set* and *Page Trace* strategies significantly improve the replay accuracy. On the NAS codelets, the *Page Trace* strategy is slightly better than the *Working Set* one. The improvement comes from LU codelets whose irregular accesses are better captured by the *Page Trace* warmup. The next section shows another example where the *Page Trace* warmup is more accurate than the *Working Set* optimistic warmup.

Table 3.2 compares the predicted replay execution times using the three warmup techniques. This codelet is sensible to warmup: *Cold* warmup and *Working Set* warmup are not highly accurate because the replays over and under-estimate memory access costs. In this case, the more realistic *Page Trace* warmup gives the more accurate results. For this reason and except if we mention, all the following experiments were all performed using *Page Trace* warmup.

3.5.4 Replay Codelets

This section describes CERE strategy to preserve the original performance of the codelets. It also presents how codelet replay can be retargeted to new configurations such as recompiling the codelet with different optimizations passes.

Once the memory and cache state are captured, a codelet can be replayed. To replay a codelet, CERE generates the special wrapper shown in Step 4 of Table 3.1. First, it allocates clone variables for the input and output flow dependencies to the outlined region. Second, it restores memory

and cache state. Finally, it calls the outlined region.

When we only execute the codelet, the outlined region results are not used when returning from the call to the codelet. Therefore, LLVM dead code elimination pass is free to fully optimize by removing this call. Clearly that is not our purpose. So, during replay we insert, for each live-out variable, a special `antideadcode` call. It is an empty extern function which forces LLVM to keep the codelet's code, even when using highly aggressive optimization levels such as `-O3`.

Also, we note that the outlining compilation pass dereferences the input and output dependencies. By passing the variables by reference, it is easy to preserve the values modifications during the codelet execution. This is a classic technique in code outliners [26, 73] which has the unfortunate side-effect of disabling many compiler optimizations. In many codelets, dereferencing makes codelet replay slower and therefore unfit to be used as performance proxies of the original code. We solve this problem in three steps. First, we tag each dereferenced pointer with the IR attribute *NoAlias* which informs LLVM that the dereferenced pointer is not aliased. This is known because the extra dereference is created by CERE outliner and used only once during replay. Second, we tag the outlined function itself with the attribute *AlwaysInline* which forces LLVM to reinline the function in the replay wrapper. Third, LLVM alias analysis optimization pass removes the extra layer of dereference. In section 5.3 of our validation experiments, the effect of reinlining and marking cloned variables as NoAlias are measured. These two techniques improve replay accuracy in eleven applications without degrading the other benchmarks

One could think that the outlining step is unnecessary since it is reverted later on by LLVM inliner pass. But as explained in section 3.5.1, the outlining step guarantees that CERE finds a safe checkpoint to capture the context just before a procedure call.

To generate the final replay binary, CERE uses a custom link script, that reserves the virtual memory segments occupied by the working set pages during the memory capture. This step is needed so that CERE can preserve the original memory layout.

Codelets can be replayed with a different configuration as the one used during the capture. CERE uses LLVM to compile the codelets. LLVM includes middle-end optimizations targeting the IR which can be controlled using `opt`, the LLVM optimizer. LLVM also includes back-end optimizations that are architecture specific and are controlled using `llc`, the LLVM static compiler. Finally, the compiler links the binary with the OpenMP library to execute the OpenMP applications.

CERE extracts the codelets at the IR before any middle-end optimizations. Once the replay wrapper is generated without any optimizations, CERE can apply though opt compiler optimizations that we want to evaluate. The compiler optimizations are passed through flags to opt and are

not necessary the same as the one used during the capture. Similarly, llc allows to evaluate back-end optimizations. It also provide the ability to retarget the code and execute it on new architectures. Finally, CERE can also retargets the thread configurations through runtime environment variables.

3.5.5 Parallel Replay

While the previous steps are sufficient to replay serial codelets, two additional steps are required for parallel regions. This section presents the two steps required by parallel codelets to preserve the original performance.

NUMA-Aware Allocation

A replay has to faithfully reproduce the original invocation context. CERE already handles two issues: it restores the memory working set of the region and it warms up the cache to avoid cold-start bias.

CERE uses the same mechanism to warmup loops and OpenMP regions: it relies on a snapshot of the memory at page level granularity. With a memory protection mechanism, the memory pages containing the working set are captured. During replay, pages are remapped to their original addresses. However, this mechanism may be insufficient to faithfully replay the codelets in some cases.

Multi threaded applications raise a new challenge: the placement of the pages across the NUMA nodes. Due to the first touch policy, a page is mapped to the core which first attempts to use it. We must ensure that pages are mapped to the same NUMA nodes as they have been in the original run. The problem is that pages are not necessarily bound to the same NUMA nodes across the different thread affinities. For example, OMP `scatter` maximizes the number of NUMA nodes while `compact` minimizes it.

At replay, CERE serial warmup uses a single thread to remap the pages to their original addresses. Using this strategy on a multi-NUMA nodes architecture bind all the pages to a single NUMA node. The replay pays NUMA latencies that do not appear in the original run and which cause prediction discrepancies.

To solve this issue, we enhance the page capture by saving, for each page, the first thread that touches it. During replay, before replaying the codelet code, each thread touches the pages that it has saved at the capture. Hence, pages are mapped to the NUMA node of the thread which is the first to touch them.

CERE NUMA aware thread exploration has a limitation. Let us consider a NUMA aware capture with N threads. CERE cannot faithfully explore NUMA configurations that have more than N threads. At capture, we track the touched pages for each thread. Capturing with four threads a region allows to get all the pages for these threads. We can remap the pages

to preserve the NUMA placement with two threads since we know which pages were touched by the threads. However, when replaying the region with five threads, we cannot determine which pages will be touched by the fifth thread. We can still replay the code with five threads, but the pages of this thread will not necessary be bound to the right NUMA node.

Lock Support

Unlike sequential applications, parallel codes use synchronizations with locks. OpenMP uses futex (fast userspace mutexes) calls to implement the lock support on Linux. So, to fully support replay of codelets using OpenMP lock primitives, a special *lock capture* step is required.

In Linux, each futex requires a kernel space wait queue. System calls are used to request operations on the wait queue from user space. Our memory capture only saves the user space process memory, therefore it does not preserve the state of the futex wait queue.

To properly support OpenMP locks, we need a special lock capture step that detects all the locks accessed by a codelet. This is achieved by intercepting calls to the lock OpenMP library during capture. Before replaying the codelet, the replay wrapper takes care to properly initialize all the required locks in kernel space.

3.6 Hybrid Compilation

As explained in section 2.6 of the background, exploring the different compiler optimizations for an application is a costly and time consuming process. Moreover, we observe that some applications may have different optimal parameters for different regions of code. For example, compute bound loops and memory bound loops within the same function will not be sensitive to the same compiler optimizations. As regions of code do not benefit from the same parameters, a traditional overall program-evaluation (or *monolithic* evaluation) is not able to achieve the optimal per region optimization.

CERE piecewise tuning finds the best compiler optimizations for each loop and OpenMP region. Unfortunately, LLVM does not provide a mechanism to select compiler optimizations at the function or loop granularity. To compile each region with a different set of optimizations we must extract each region in its own compilation unit. We leverage the `extract` tool included in LLVM which allows to extract an IR function to a separate IR file.

The first step is outlining each region of interest in its own IR function. Before any middle-end optimization is applied, each region is moved to a separate compilation unit using `LLVM extract`. A special pass changes the visibility of symbols used by the extracted region from internal to global so that they are not removed by the compiler. Then, the best compiler sequence

	CERE	Code Isolator	Astex	Codelet Finder
Support				
Language	C(++), Fortran, ...	Fortran	C, Fortran	C(++), Fortran
Extraction	IR	source	source	source
Indirections	yes	no	no	yes
Reduction				
Capture size	reduced	reduced	reduced	full
Temporal	yes	manual	manual	manual
Spatial	yes	no	no	no

Table 3.3: Feature comparison of sequential code isolation tools.

found is applied to each separate IR file and an object file is produced. Finally, all the objects files are linked together producing an hybrid binary. We evaluate the quality of the hybrid binaries in section 6.5.2.

3.7 Related Work

In this section, we compare CERE to related code isolator tools and sampling methods.

3.7.1 Code Isolation

Table 3.3 compares the features of the main serial code isolation tools on multiple criteria. First we compare the supported input languages, the isolation level and the support of indirect memory accesses. Second, we examine whether the tool attempts to reduce the capture size, the number of working sets, or the number of representative pieces to replay. The reduction is explained in the next chapter.

As stated in the background (see section 2.5.3), to the best of our knowledge, only Liao et al. [73] have also published on parallel OpenMP code isolation without simulators. Their approach is based on the ROSE [81] compiler infrastructure. Isolation is achieved through a source to source outliner which extracts tunable kernels out of OpenMP programs.

The source outlining approach requires an additional step that repairs the lost scopes. On the contrary, CERE IR level outlining is simpler because it is done after OpenMP data clauses expansion. Also, unlike CERE which is evaluated on all the NPB, Liao et al. outliner is demonstrated on a single OpenMP `for` loop from the SMG2000 benchmark.

3.7.2 Sampling Simulation

Sampling techniques [13] (i.e. used with assembly isolation) are similar to our work in that they extract representative phases from applications and

allow accurate replay. Nevertheless, all of these sampling techniques must be used in a simulator, whereas our method is more versatile since it produces IR codelets that can be recompiled and run both on simulators and on real hardware.

So, code isolation accuracies of CERE prediction and these methods are not directly comparable since the methods measures accuracy on a functional simulators whereas CERE measures accuracy on real hardware.

Also, another key difference is that current sampling techniques do not allow changing the number of threads at replay. Each thread configuration requires a separate capture. Therefore, unlike CERE, they cannot be easily used to evaluate parallel scalability.

Acknowledgments

CERE is an open source tool developped collaboratively, the full list of contributors can be found at `https://github.com/benchmark-subsetting/cere/blob/master/THANKS`.

3.8 Conclusion

This chapter presents CERE, an LLVM based Codelet Extractor and Replay framework. CERE finds and extracts the hotspots of an application as codelets. Codelets can be modified, compiled, run, and measured independently from the original application. The following chapter presents how the codelets can be reduced to accelerate the benchmarking cost.

Chapter 4

Benchmark Reduction Strategies with Codelets

Contents

4.1 Introduction

Finding the best architecture and compiler optimizations for an application is an important problem for high performance computing, data centers, and embedded computers. Traditionally performance benchmarks are conducted to determine the best architecture and execution options. This process involves running all the benchmark programs in different system configurations. This chapter proposes a method to lower the cost of benchmarking by extracting a subset of representative codelets sufficient to capture the performance characteristics of the original benchmarks or applications.

Related studies [46, 22] identify many similarities or redundancies among different programs in the same benchmark suite. Redundant pieces are stressing the performance of the same architectural components. So profiling similar benchmarks means that we are not only paying the evaluation cost

multiple times but also that we are targeting the same components without getting any new information about the system.

Working at a fine granularity level allows to detect more redundancy. The larger a code fragment is, the harder it is to find a similar redundant fragment in another program. But, when codes are broken into elementary pieces, it is common to find duplicate *computation patterns* [98] such as dot products, array copies or reductions.

CERE takes advantage of those fine granularity similarities to reduce the benchmarking time. In particular, codelets address three sources of redundancy in the benchmarking process:

- **similar computation kernels or regions:** Many benchmark suites share many similar codelets: simple ones, like set-to-zero or memory copy loops, and more complex ones, like Single-precision real Alpha X Plus Y (SAXPY) loops. There is no need to measure multiple copies of the same code. We categorize these redundancies as *spatial* since they concern regions of code with different locations,

- **similar code invocations:** Regions that are repeatedly invoked with the same context in an application lifetime, have the same running time for each invocation. Therefore, a single invocation replay is sufficient to characterize them. In applications where a single region is called thousands of times, measuring only a few invocations achieves significant gains in benchmarking time. We categorize these redundancies as *temporal* since they originated from the same code but at different moments during the execution,

- **sequential regions across different thread configurations:** When varying the number of threads, performance of sequential regions is not strongly impacted. With codelets, scalability can be evaluated by only replaying parallel regions: there is no need to replay multiple times the sequential parts. We consider parts of an application that are not impacted by some transformations (e.g. thread configurations, compiler optimizations, architectures) as *conditional* redundancies. Sequential regions are an example: for scalability studies, the sequential parts can be ignored. However, if the user wants to tune the compiler, he cannot avoid the serial parts.

Taking advantage of the sequential regions with codelets to predict the scalability is easy. CERE measures the sequential time once. Then, it replays the codelets across the different threads configurations. To predict the scalability with a specific number of threads, CERE sums the sequential time with the codelets execution times at the targeted thread configuration. As long as the sequential time remains constant across the different thread configurations, CERE does not need to remeasure it.

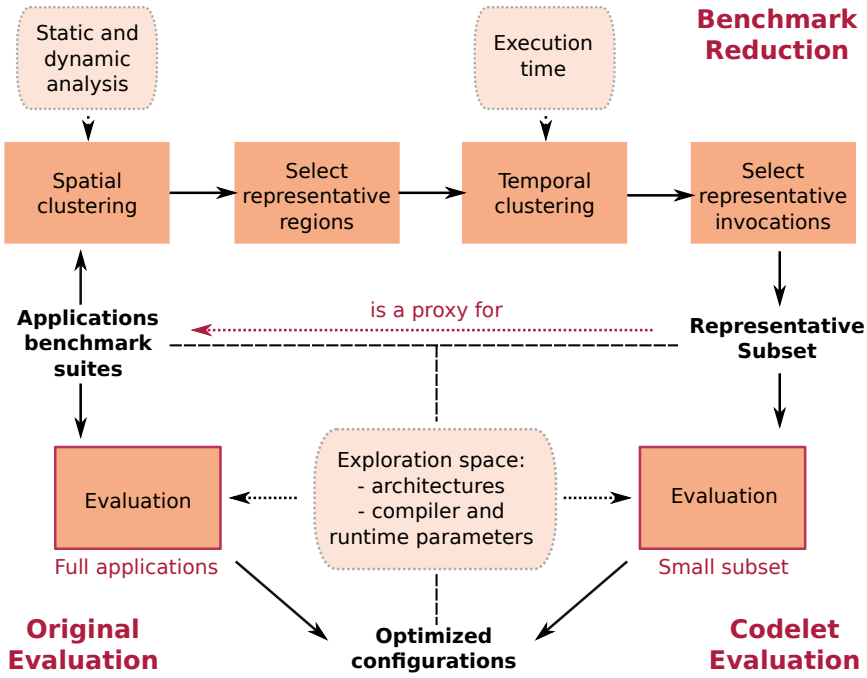


Figure 4.1: CERE benchmark reduction workflow for tuning. With original evaluations, new parameters are directly evaluated over the applications. This is a costly and time consuming process. CERE reduces the benchmarks to a representative subset. The reduction is composed of both a temporal subsetting (see section 4.2) and a spatial subsetting (see section 4.3). Evaluating systems on the subset is faster because we avoid the redundant full application executions. Hence, we quickly amortize the reduction initialization cost because of the huge exploration space.

To address the spatial redundancies, CERE breaks down complex applications into a set of codelets and clusters them. Each cluster contains codelets with similar computation patterns. Since they are similar, we assume that they target and stress the same architectural components. So replaying one codelet per cluster is sufficient to characterize the components that are targeted by the cluster. Determining which regions share common computation patterns is challenging. We handle this issue by comparing performance metrics vectors (see section 2.2).

CERE also addresses temporal redundancies with clustering. Invocations of a region of code that are called with the same working set have the same performance and are grouped together. CERE subsets the invocations of the region. Both the spatial and temporal clustering criteria are discussed in section 4.5.

Combining both spatial with temporal clustering allows CERE to accelerate the benchmarking process. It generates a subset of representative

codelets. The invocations of each of these codelets are clustered and a representative invocation per cluster is extracted. CERE uses these invocations as proxies for the benchmarks. Figure 4.1 illustrates this strategy over an application. Evaluating CERE representatives is faster than full benchmarking as we avoid to replay redundant codelets and invocations. Also, since only redundant pieces are removed, the subsetting preserves the diversity of the benchmarked components.

Invocations within the same cluster share a common execution behavior and should react in the same way to architecture or parameter changes. For example, memory-bound regions will benefit from faster caches, whereas highly vectorized regions will benefit from wider vectors. Therefore, by measuring a single representative invocation per cluster we can extrapolate the performance of all its siblings. We discuss the behavior extrapolation to the cluster siblings in section 4.5.

This chapter presents how the CERE benchmark reduction strategy operates over different use cases. First, we subset a benchmark suite with the temporal reduction in section 4.2. Second, we focus on the spatial reduction to also subset a benchmark suite in section 4.3. Third, section 4.4 illustrates the scalability prediction model. Finally, we discuss the limitations and the future improvements of our methodology in section 4.5.

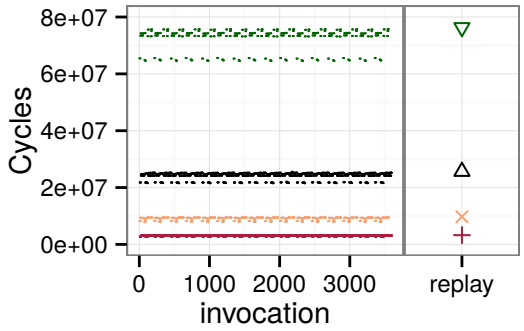
4.2 Temporal Subsetting

4.2.1 Invocations Reduction

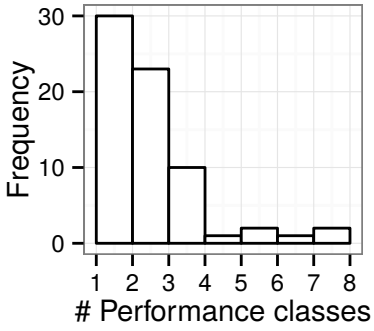
Inside an application, the same region (either loop or OpenMP region) may be called multiple times. While many invocations of the same region have a similar execution time, in some codes, two invocations may have different execution times. This is due to the different working sets or initial conditions.

For example, the codelet `make_ft@shell2.F90:1133` extracted from `tonto` (from the SPEC benchmarks) is one of the steps of a specialized Fast Fourier Transformation. In the original application, this loop is called 3587 times with different workloads. Figure 4.2a shows its execution trace. A cluster analysis of the invocations reveal that they can be sorted into 4 performance behaviors, which are represented with different colors in the figure. Other regions such as `FT_fftxyz_152:58` extracted from `FT`, have a constant workload size but the first invocation is slower because of cache warmup effects (see Figure 4.3).

To accurately replay a codelet, we must capture each different invocation state. When the number of invocations is high, this process becomes costly both in time and space. Fortunately, applications exhibit some regularity; and most of the time the invocations can be reduced to a few representative *performance classes*. Figure 4.2b shows the distribution of the number of



(a) SPEC tonto `make_ft@shell2.F90:1133` execution trace.



(b) Distribution of the number of representative performance classes across all NAS benchmarks.

Figure 4.2: invocation reduction: (a) A clustering analysis of tonto's trace detects four different performance behaviors depending on the workload. The initial 3587 invocations are captured with only four representative replays. (b) Most of the NAS codelets can be captured with less than four representative working sets.

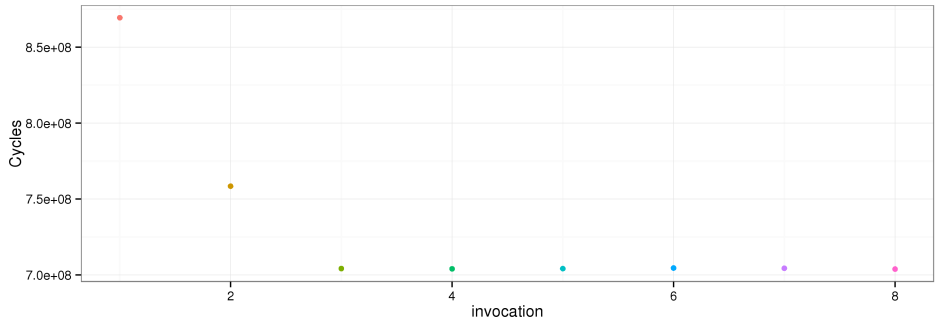


Figure 4.3: The region `fftxyz_152` is executed 8 times during the application lifetime. The first invocation is slower due to the cold cache state.

different classes across all NAS benchmarks. One can note that most of the codelets can be captured with less than 4 representatives. The fact that the performance of a program can be reduced to a small number of representatives has been observed in other domains such as value profiling [99, 80, 100] and iterative compilation [101].

To automatically detect the performance classes and generate a set of representative captures, we use a clustering algorithm. Some of the codelets in our benchmarks have large traces with more than 10^9 invocations and cannot be analyzed using traditional clustering algorithms such as hierarchical clustering or K-means. To be able to efficiently process such large data sets CERE uses CLARA (CLustering LARge Applications) algorithm [102]. CLARA (see section 2.3.1) relies on sampling to reduce the cost of clustering. It extracts a random sample from the original data set and find the cluster medoids. The sampling process is repeated to reduce the bias in the medoid selection. Finally, each point of the original data set is assigned to the nearest medoid's cluster.

Once the performance classes are identified, CERE selects one representative invocation per class. CERE selects the invocation closest to the medoid of the cluster, in other words, the invocation closely matching the median performance of all the invocations inside the cluster. When replaying the benchmark, CERE extrapolates the full region performance by weighting each representative replay time according to the contribution of its performance class in the original execution.

Thanks to invocations reduction, CERE is able to accelerate performance evaluation considerably since only a representative subset of the invocations is replayed: for example, only two out of ten thousand invocations are replayed for codelet `updateTestEv@soplex.c:204` in SPEC 2006 soplex benchmark.

4.2.2 Accelerating System Evaluation

We consider that an execution *system* includes the choice of the compiler optimizations, the runtime parameters (such as thread configurations or frequencies of execution), and the architecture. Benchmarks and applications are widely used to evaluate new systems to improve the performance. This is a costly process because the exploration space is huge and even a single execution may be time consuming.

Temporal reduction identifies invocations of code with similar performance to reduce the benchmarking cost. To detect and remove the redundant invocations, CERE profiles the benchmarks on a reference system and produces a subset of representative invocations.

CERE allows to evaluate new systems by only replaying the representative invocations on them. To correctly predict the execution time of the benchmarks on the new target system, we assume that: *invocations in a*

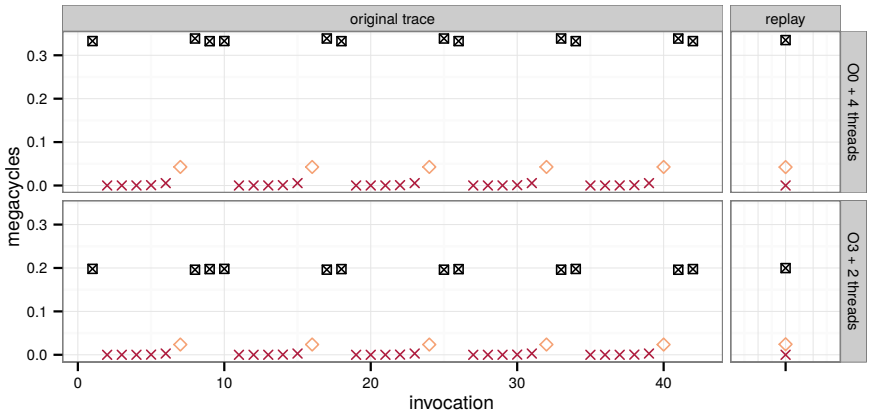


Figure 4.4: MG `resid` invocations execution time on Sandy Bridge over `-O3` and `-O0` with respectively 2 and 4 threads. Each representative invocation predicts its performance class execution time. Tuning compiler optimizations and thread configurations has a similar impact on invocations within the same cluster.

region sharing the same performance class have the same speedup when they are executed on a new system.

More formally, let the execution time of an invocation be t_i^{ref} for the reference system and t_i^{tar} for the target system. For invocation p_i, $s_i = t_i^{ref}/t_i^{tar}$ is the speedup between the reference and the target. The speedup of any invocation from cluster C_k, is close to s_{r_k}, the speedup of the cluster representative,

$$\forall p_i \in C_k, s_i \simeq s_{r_k}.$$

We predict the performance of each original invocation by using the following formula:

$$\forall p_i \in C_k, t_i^{tar} \simeq t_i^{ref} \times \frac{1}{s_{r_k}} = t_i^{ref} \times \frac{t_{r_k}^{tar}}{t_{r_k}^{ref}}.$$

Figure 4.4 illustrates the invocation assumption on MG `resid`, an OpenMP parallel region. `Resid` has 42 invocations grouped in 3 performance classes. We observe that changing the system, currently the compiler optimizations and the thread configurations, has a similar impact on invocations within the same performance class. So, by replaying 3 instead of 42 invocations, CERE predicts the region execution for each parameter to explore.

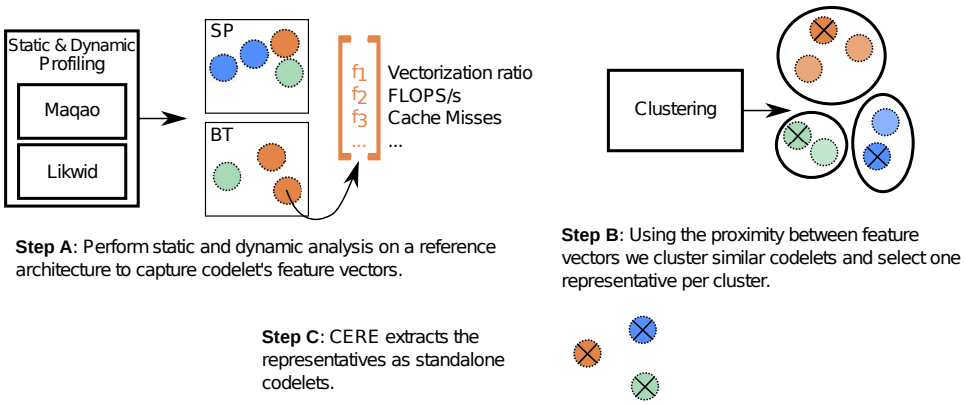


Figure 4.5: CERE spatial benchmark reduction method. Representative codelets are extracted and serve as proxies to evaluate the whole applications.

4.3 Spatial Subsetting

4.3.1 Regions Reduction

A second reduction in the number of replays can be achieved by detecting and exploiting similar and repeated computation patterns. Benchmark suites and applications naturally contain redundant computation patterns across different benchmarks. For instance, two linear algebra solvers, despite using different algorithms, will share common computation patterns such as vector copy loops, dot product computations, or matrix vector multiplications.

The method presented in figure 4.5 detects repeated computation patterns, and keeps only one representative copy of each, reducing a suite of benchmarks to an essential set of micro-benchmarks.

Because only duplicated patterns are removed, the performance diversity of the original benchmarks is preserved.

Different methods and metrics are used to detect the similar computation patterns in codes are described in section 2.4. We select the static metrics because they provide useful information about the codes signature and because they are cheap to access. We also rely on hardware performance counters to complete our analyzes. Performance counters capture the dynamic hazards i.e. the memory behavior that cannot be modeled by the static characterization. We note that other means can be used to achieve this characterization and discuss them in section 4.3.3.

CERE statically analyzes and profiles each region. Static metrics are extracted by the MAQAO Code Quality Analyzer [35, 75] which provides detailed low-level performance metrics. Dynamic metrics are provided by Likwid 3.0 [39] or Lprof [38] which read hardware performance counters.

Likwid dynamic metrics
- Floating point rate in MFLOPS.s^{-1}
- L2 bandwidth in MB.s^{-1}
- L3 miss rate
- Memory bandwidth in MB.s^{-1}
MAQAO static metrics
- Bytes stored per cycle assuming L1 hits
- Data dependencies stalls
- Estimated IPC assuming only L1 hits
- Number of SD instructions
- Pressure in dispatch port P1
- Ratio between ADD+SUB/MUL
- Vectorization ratio for Multiplications (FP +INT)
- Vectorization ratio for Other (FP)
- Vectorization ratio for Other (INT)

Table 4.1: Performance metric set used to cluster regions.

MAQAO and Likwid gather 76 different metrics. Irrelevant metrics add noise that degrades the clustering. Therefore, it is important to wisely select metrics, keeping only those that adequately capture program behavior and improve clustering.

Section 4.3.3 explains how we explore the metric space to find a set of metrics with an optimal clustering quality. Table 4.1 presents the 13 performance metrics used by CERE. CERE profiles each region and produces a metric vector per region that contains the selected metrics. We rely on these vectors to quantify the regions similarities.

CERE groups regions sharing similar metrics vectors into clusters by computing euclidean distances across the vectors. To cluster similar regions, we use the hierarchical clustering with Ward's criterion [51]. The final number of clusters is selected with the Elbow method [53] (see section 2.3.1). Metrics are normalized to have unit variance and to be centered on zero. Normalization ensures that metrics have equal weight when computing a distance between two metric vectors.

We select a representative region for each cluster and extract it as a standalone codelet. A representative must adequately capture the performance metrics of the cluster: as a representative, we choose the codelet closest to the cluster centroid. Using the centroid reduces the approximations caused by the subsetting since the centroid cluster is the closest point to the cluster average values.

Thanks to region reduction, CERE produces a subset of regions out of the initial benchmark suite that preserves its diversity. The next section illustrates how this methodology accelerates the evaluation of new architectures.

4.3.2 Architecture Selection

Like temporal reduction, spatial reduction also accelerates the evaluation of new systems. We assume that *regions in a cluster share the same speedup when they are executed on a new system: they react in the same way to system changes*. So, by measuring a single representative codelet per cluster, we can extrapolate the performance of all its siblings.

In this thesis, we test the spatial reduction only for architecture selection: we reduce the benchmarking time required to evaluate new architecture configurations. First, we present the method over the 28 Numerical Recipes (NR) [60] codelets provided by Noudohouenou [103]. Each NR codelet is composed of a single computation kernel. There is a one-to-one mapping between the NR benchmarks and the NR codelets extracted. Moreover, all the NR codelets are well-behaved i.e. their prediction errors are bellow 15%. NR codes are simple but cover a large spectrum of algorithms.

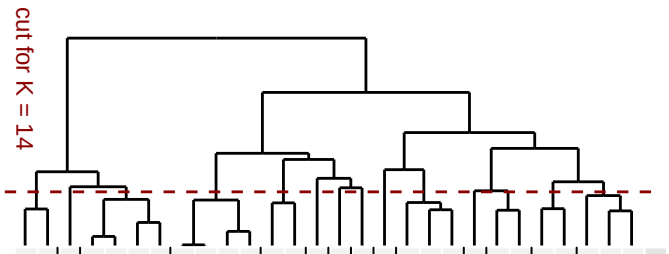
We start by profiling the NR on a reference architecture Nehalem and predict the suite execution time on Atom and Sandy Bridge by replaying the representative codelets. We note that in our experiments, we have never used spatial subsetting alone. It is always combined with temporal reduction since the two methods are orthogonal. Section 6.4 performs a full validation of the reduction methodology by predicting the NAS over Atom, Sandy Bridge, and Core2.

Table 4.2 shows a 14-group clustering built on the reference architecture using the NR codelets. We use the metrics presented in table 4.1 to perform the clustering.

Our goal is to ensure that codelets in the same clusters exhibit similar characteristics and a common behavior. An initial supporting observation is that the vectorization is homogeneous among clusters. We evaluate cluster similarity using two other similarity criteria.

First, we note that many clusters are formed of codelets with similar computation patterns. For example, cluster 10 gathers codelets that divide elements in a vector, cluster 11 codelets that perform a reduced sum, cluster 14 codelets that compute element-wise multiplications on vectors or columns. The two "Dense Matrix x vector product" codelets have been separated because they use different floating point precision.

Second, the *stride* captures the distance between the data points accessed by two successive iterations of a codelet. For example, a stride of one means that the codelet is accessing memory sequentially. A stride of zero means an access to a constant memory location. A Leading Dimension Array (LDA) stride means a row-wise access to a column-wise stored array. If a codelet has two or more types of stride, we separate them with a '&' symbol. Stencil stride means that the kernel uses a five points stencil to access the data. Cluster 14 is composed only of codelets with contiguous access to memory. Cluster 11 contains only (0 & 1) codelets: one contiguous access to sweep



C	Codelet	Computation Pattern	Stride	Vec.	Vec. %	s
1	toeplz_1	DP: 2 simultaneous reductions	0 & 1 & −1	V + S	78	<0.24>
	rstrct_29	DP: MG Laplacian fine to coarse mesh transition	stencil	V + S	83	0.25
	mprove_8	MP: Dense Matrix x vector product	0 & 1	V + S	60	0.15
2	toeplz_4	DP: Vector multiply in asc./desc. order	0 & 1	S	20	0.44
	realft_4	DP: FFT butterfly computation	0 & 2 & −2	S	18	<0.42>
3	toeplz_3	DP: 3 simultaneous reductions	0 & 1 & −1	V	100	0.31
	svbksb_3	SP: Dense Matrix x vector product	0 & 1	V	100	<0.35>
4	lop_13	DP: Laplacian finite difference constant coefficients	stencil	V	100	<0.20>
5	toeplz_2	DP: Vector multiply element wise in asc./desc. order	1 & −1	S	0	<0.36>
	four1_2	MP: First step FFT	4	S	8	0.22
	tridag_2	DP: First order recurrence	−1	S	0	0.44
6	tridag_1	DP: First order recurrence	0 & 1	S	0	<0.32>
7	ludcmp_4	SP: Dot product over lower half square matrix	0 & LDA & 1	V + S	83	<0.45>
8	hqr_15	SP: Addition on the diagonal elements of a matrix	LDA + 1	S	0	<0.39>
9	relax2_26	DP: Red Black Sweeps Laplacian operator	LDA & 0	S	10	<0.12>
10	svdcmp_14	DP: Vector divide element wise	0 & 1	V	100	0.28
	svdcmp_13	DP: Norm + Vector divide	1	V	100	<0.17>
11	hqr_13	DP: Sum of the absolute values of a matrix column	0 & 1	V	100	0.41
	hqr_12_sq	SP: Sum of a square matrix	0 & 1	V	100	<0.46>
	jacobi_5	SP: Sum of the upper half of a square matrix	0 & 1	V	100	0.34
	hqr_12	SP: Sum of the lower half of a square matrix	0 & 1	V	100	0.34
12	svdcmp_11	DP: Multiplying a matrix row by a scalar	LDA	S	0	<0.33>
	elmhes_11	DP: Linear combination of matrix rows	LDA	S	0	0.47
	mprove_9	DP: Substracting a vector with a vector	1	V	100	0.50
	matadd_16	DP: Sum of two square matrices element wise	1	V	100	0.53
13	svdcmp_6	DP: Sum of the absolute values of a matrix row	0 & LDA	V + S	33	<0.30>
14	elmhes_10	DP: Linear combination of matrix columns	1	V	100	0.44
	balanc_3	DP: Vector multiply element wise	1	V	100	<0.47>

Table 4.2: NR clustering with 14 clusters and speedups on Atom. The dendrogram on the left shows the hierarchical clustering of the codelets. The height of a dendrogram node is proportional to the distance between the codelets it joins. The dashed line shows the dendrogram cut that produces 14 clusters. The table on the right gives for each codelet: the cluster number C, the Computation Pattern, the Stride, the Vectorization, and the Speedup on Atom s. The speedup of the selected representative is emphasized with angle brackets.

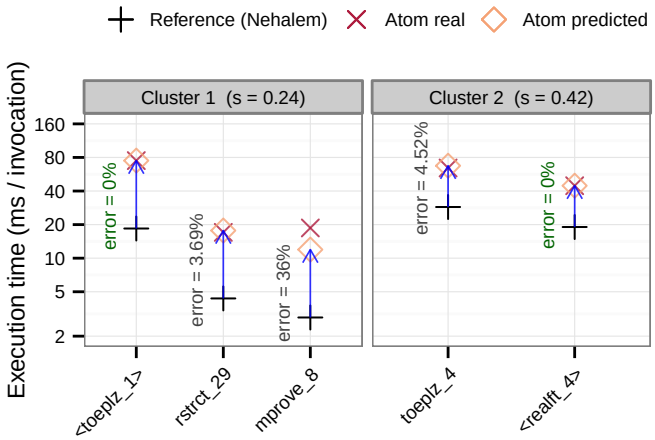


Figure 4.6: Predicted and Real execution times on Atom for clusters 1 and 2. Representatives are enclosed in angle-brackets. They have a 0% prediction error because they are directly measured. The representative speedup is applied to all its siblings to predict their target performance. Because the scale is logarithmic, applying the speedup is depicted by the arrow translation.

the vector and one constant access for the accumulator. Other clusters have more complex stride behaviors.

The clustering is not perfect, but still gathers codelets sharing similar computation patterns, stride accesses, and vectorization.

Our assumption is that codelets with similar metrics have similar speedups on the target architectures. Column s on the table shows Atom speedups. The two codelets in cluster 10 suffer high slowdowns on Atom because they use high-latency division operations. Our metric set captures this pattern and isolates them in their own cluster.

In most of the clusters, speedups are homogeneous. Close codelets in the dendrogram such as in clusters 2, 3, or 14 exhibit close speedups. Yet in some clusters such as 10 or 12 the speedups are distinct. Our dendrogram cut is too rough and a higher number of clusters is needed. In this case, 24 clusters, as recommended by the elbow method, fix the most striking discrepancies. Yet the 24 elbow clustering, though more conservative in terms of prediction, is less interesting to analyze because it has many singleton clusters.

We evaluate the prediction error using the 14 clusters' representatives on Atom and Sandy Bridge. Figure 4.6 details the prediction model for the cluster 1 and 2. As expected, the representatives codelets `toeplz_1` and `realft_4` are perfectly predicted. The prediction is accurate except for `mprove_8`. Table 4.2 dendrogram shows that slightly increasing K puts the offending codelet in a different cluster.

Table 4.3 summarizes the prediction errors for all the NR codelets on

	K = 14		K = 24 elbow	
error	median	average	median	average
Atom	1.8%	12%	0%	1.70%
Sandy Bridge	3.2%	9.30%	0%	0.97%

Table 4.3: Prediction errors on Numerical Recipes with 14 and 24 clusters. NR contain 28 benchmarks. Selecting 24 codelets lead to replay more than half of the benchmarks: this justifies why the median is equal to zero. The fact that the elbow produces a number of clusters near to the total number of benchmarks means that according our performance metrics, the NR are quite diverse.

Atom and Sandy Bridge. The overall accuracy of the prediction is good. Nevertheless, this was expected because as presented in the next section, the NR were used during the metric selection training. The metric set was selected to minimize prediction accuracy. Section 6.4 performs our method on an unseen benchmark set and test architecture to quantify the resilience of the methodology. In the following section, we explain how we select the performance metrics that gather codelets with similar computation patterns.

4.3.3 Clustering Metrics with Genetic Algorithms

Performance metrics must ensure that codelets in the same clusters have similar computation patterns. Since MAQAO and Likwid gather 76 different metrics, it is important to wisely select metrics, keeping only those that adequately represent the program behavior. This section explains how we select our metrics to cluster applications for architecture evaluation.

Evaluating the 2^{76} combinations is too costly. To find a good set of metrics in a reasonable time, we use genetic algorithms [59] (see section 2.3.3). Genetic algorithms (GAs) start with a population of randomly generated individuals. In our case, each individual represents a candidate metric set. This population evolves towards an optimal solution by recombining the best individuals with crossover and mutation operators.

An individual is encoded as a 76 boolean vector. The i^{th} bit is set if and only if metric i is selected. For instance, the vector with all bits set to one corresponds to the individual containing all the 76 metrics. Crossover and mutation operators are provided by the genalg [104] GNU R package.

To evaluate individuals, we consider the average prediction error of NR benchmarks on two architectures: Atom and Sandy Bridge. Best individuals should have a high prediction accuracy on both architectures but with a low number of representatives. To achieve this objective we choose the fitness function: $max(error\_atom, error\_sandybridge) \times K$ where K is the number of clusters.

We perform 100 GA iterations for a population size of 1000 and a mutation probability of 0.01. The Genetic algorithm converges to the optimal metric set presented in Table 4.1 by generation 47.

We intentionally train our metrics on a subset of our configurations. We leave an architecture i.e. Core2 and a benchmark suite i.e. NAS out of the training process. It is fairer to evaluate how our metric set fares on new architectures and new benchmarks. It also allows to avoid the machine learning overfitting presented in section 2.3.3. Section 6.4 validates the performance prediction of the metrics over new untrained systems and applications. Section 4.5 discuss about the relevance of these metrics for a more general usage. In particular, one could try to use these metrics to tune compiler optimizations.

4.4 Conditional Subsetting for Scalability Prediction

We previously presented the spatial and temporal reductions. This section shows the third and last reduction technique that is used by CERE: conditional subsetting. Unlike the previous approaches, conditional subsetting can only be applied when targeting some specific parameters evaluation.

Some system updates do not impact all parts of the applications code. For instance, Amdahl's law [105] stipulates that the execution time $t(n)$ of an application being executed on n threads is,

$$t(n) = T(1) \times (s + \frac{1-s}{n}),$$

where s is the fraction of the algorithm that is strictly serial. In other words, increasing the number of threads will only speedup parallel regions. We do not need to execute the serial parts across the different thread configurations. Serial parts are removed by the conditional subsetting when CERE evaluates scalability.

Traditionally, the scalability of an application is evaluated by plotting the application execution time against the number of parallel threads m. This requires measuring one application run for each thread configuration and so executing m times the serial parts.

Through fast codelet replay, CERE is able to quickly estimate the strong scalability with a single measurement of the serial parts: we only replay the parallel regions.

Our prediction scalability method has two steps. First, *initialization* extracts the codelet set and measures the sequential part of each application, T_{seq}. Second, *prediction* replays the codelets with different number of threads to predict the scalability of the original applications.

Let C be the set of all the parallel regions extracted as codelets from an application. Our model estimates the application execution time with n threads as,

$$t^{predicted}(n) = t_{seq} + \sum_{c \in C} T_c^{tar}(n),$$

where $T_c^{tar}(n)$ is the predicted execution time of the codelet c at the targeted n number of threads. By only measuring the parallel regions execution time, we get the full application execution time.

CERE accelerates the scalability evaluation process because:

1. the sequential part is only measured once during the initialization,
2. codelet replay is faster because the number of invocations of the parallel regions is reduced through temporal reduction.

This method is validated by our experiments in section 5.4.1.

4.5 Discussion

In this section, we discuss the three previously introduced reduction techniques can compare to existing approaches.

4.5.1 Related Works

On of the challenge in code isolation is reducing the replay and codelet capture cost. The replay and capture cost is related to two quantities: the number of codelets and the number of invocations to capture and replay. When codelets or invocations have similar behaviors, it is desirable to only capture and replay a representative subset. A single region in a program may be called thousands of times with different working sets and cache states. Capturing each individual invocation is prohibitive. A first set of studies [71], [26], and [72] allow the user to manually choose which invocations should be captured and replayed.

Previously described approaches for benchmarking reduction can be applied to select the representative subset. Sherwood and al. [48] identify similar computation phases by computing the distance between the region's Basic Block Vectors. Other works [25, 58, 41, 42, 22] use performance feature vectors as a measure of region similarity.

Yang et al. [106] proposes a partial execution framework without code isolation based on relative performance between platforms. They manually insert probes around the kernels of the application. The probes allow to stop the execution during replay after a large enough number of invocations

has been measured. Like CERE, this method reduces the number of invocations of the kernels but it is not automatic. The user must hand-tag the computation kernels with source annotations.

4.5.2 Combine Spatial and Temporal Clustering

In our experiments, we base the clustering of the invocations of a region on a single dimension: the execution time of the invocations. However, it is possible that while two invocations share a common execution time, they have distinct behaviors.

Let us consider two invocations, respectively memory and compute bound, with a similar execution time. Moving these invocations to a new system with more memory, speedups the first invocation but should not impact the second. Since they have the same execution time on the reference system, CERE clusters them in the same performance cluster. So, CERE uniformly predicts the same speedup over the two invocations. Depending on which invocation is selected as representative, CERE will either underestimate the memory bound invocation speedup or overestimate the compute bound invocation speedup.

An approach to avoid such scenarios is to rely on the performance characterization used to cluster regions. In our example, by measuring the hardware performance counters, we can detect which invocation is memory bound or compute bound. It could be interesting to compare the current temporal clustering against the spatial one over the invocations and check the performance prediction quality.

Nevertheless, in our experiments using the time clustering was sufficient most of the time. Here is our hypothesis why a simple execution time clustering is so efficient.

Currently, CERE only clusters invocations from the same region of code. Each invocation is executed with its own working set which explains why invocations of the same region may have different performance. However, since invocations are multiple executions of the same region, they share some common properties i.e. they are all compiled from the same source code. In particular, we expect invocations from the same region to have similar static metrics.

CERE does not cluster invocations only through their execution time, it also takes advantage of the fact that invocations share common static properties. Nine out of our thirteen performance metrics describing applications behavior (see table 4.1) are issued from static analysis which explains why the current temporal clustering is so successful.

As a future work, we can extend the temporal clustering on invocations from different regions i.e. combine spatial and temporal subsetting. In such scenarios, we lose the static properties of the invocations: only using the execution time as clustering criteria will not be enough. So, we will have

use the full set of performance metrics for each invocation.

4.5.3 Enhance Spatial Clustering

Two complementary approaches can be used to enhance the spatial clustering. First, we can use fitter metrics for the clustering. Second, we can improve the statistical process that uses the metrics to cluster the regions.

Architecture Independent Metrics

Despite depending on the reference architecture, the metric set that we used to perform the spatial reduction in section 4.3 is closely related to architecture-independent metrics [58, 42]. For example, codelets can use scalar instructions (S), vector instructions (V), or a mix of both (V + S). We manually analyzed the vectorization of the codelets, *Vec.*, and compared it to the vectorization ratio, *Vec. %*, reported by MAQAO. They are highly correlated. Nevertheless, an improvement of our methodology remains to use these architecture independent metrics to cluster the applications.

Improve the Statistical Process

We rely on a GA to find the relevant metrics. We observe in our final results that there are a lot of strongly correlated metrics. For example, three of the performance metrics describe the vectorization. Also some of the memory behavior metrics are related: the L3 miss rate and the bandwidth.

Correlated metrics carry a similar information about the codes. Similarly, uncorrelated metrics provide different information. The GA selects multiple correlated metrics. This means that the information that they carry is more important for the clustering than the information that is provided by metrics that are not correlated to any other metrics. In other words, the GA artificially gives importance to some information by selecting multiple redundant metrics.

The limitation of this approach is that a metric may be more important than the others but cannot be correctly weighted in the clustering process. An important metric which is not correlated to any other metrics illustrates this idea. A proper way to fix such scenarios is to apply a PCA over the metrics before starting the GA. The GA takes a liner combination of the principal components as inputs to produce the final performance metric set. We believe that this approach can enhance the overall prediction process.

We can also improve the machine learning process by using a more efficient approach. For instance, Artificial Neuronal Networks (ANNs) we successfully applied to predict the execution time of applications [56]. So, we can replace the GA training by Artificial Neuronal Networks.

4.6 Conclusion

This chapter presented the three reduction strategies used by CERE to accelerate the evaluation of new systems. In next chapter, we will validate the codelet extraction process.

Chapter 5

Experimental Validation

Contents

5.1 Introduction

In chapter 3, we presented the CERE codelet extraction and replay methodology. Chapter 4 describes how we can reduce the benchmarking evaluation cost of different systems with CERE.

This chapter presents the experimental setup used in this thesis and validates the codelet methodology. In particular, we check codelet's replay quality on both serial and parallel benchmarks in section 5.3. We use the *benchmark reduction factor* to quantify the acceleration of codelet runs over application runs and the *accuracy* to measure the condifance of the codelets replays. Accuracy is defined as the relative difference between the original and replay execution times of a region. The original time is the time spent inside the loop or the parallel region in the original application. The replay time is the predicted time by the temporal subsetting model which is explained in section 4.2.

We also check that codelets can be used to evaluate the impact of:

- the strong scalability in section 5.4.1,

- the scalability on a new mirco-architecture in section 5.4.2,
- NUMA thread configurations 5.4.3,
- compiler optimizations 5.4.4.

5.2 Experimental Setup

This section describes the experiment setup used during this thesis.

5.2.1 Applications

We performed our experiments on various benchmark suites and an industrial application. Most of the benchmarks that we used focus on High Performance Computing (HPC).

Sequential Benchmarks

As sequential applications, we use the NAS SER [3] `3.0` benchmarks, 28 Numerical Recipes (NR) [60] benchmarks provided by Nodohouenou [103], the SPEC 2006 FP benchmarks, and a Reverse Time Migration [107] (RTM) proto-application. RTM is used in geophysical depth imaging and implements the finite-difference time-domain (FDTD) step used to solve the wave propagation equation in an isotropic medium. A full description and characterization of the FDTD proto-application is provided by [86].

All the benchmarks from NAS and SPEC FP are used in our evaluation, therefore we performed our serial experiments on **55 different applications**. NAS benchmarks were tested on class *A* and *B* data sets. SPEC benchmarks were tested on *ref* data sets.

Parallel Benchmarks

We performed the experiments on a C OpenMP parallel version [108] of the NAS Parallel Benchmarks [109] (NPB). NPB are an established OpenMP benchmark suite and a good target to evaluate OpenMP isolation with CERE. Yet, most of the NPB are written in Fortran and we cannot use dragonegg because it does not support OpenMP directives.

The Omni Compiler Project (OCP) [110] maintains an unofficial C version of NPB 2.3. Unfortunately, it is based on an outdated NPB `2.3` which was released in 1997. In particular, in the NPB `2.3` version the OpenMP parallelism is coarse grained: most of the benchmarks only have one huge parallel region. This makes them a poor choice to evaluate CERE. To overcome this problem, we have updated the OCP unofficial NAS benchmarks so that they mimic the structure of the NPB `3.0` OpenMP official version. This effort involved carefully porting the changes in the official Fortran version

	CERE validation	Spatial subsetting	Temporal subsetting	Architecture selection	Compiler tuning	Thread tuning
NAS SER	yes	yes	yes	yes	yes	no
NAS NPB	yes	no	yes	yes	yes	yes
SPEC FP	yes	no	yes	no	no	no
NR	no	yes	no	yes	no	no
RTM	yes	no	yes	no	yes	no
PARSEC	yes	no	yes	no	no	yes

Table 5.1: Applications use cases.

to the unofficial C version. Our `3.0` C version of NPB is publicly available at `http://benchmark-subsetting.github.io/cNPB`.

The CG benchmark was slightly modified to overcome a bug in the LLVM OpenMP front-end. A global barrier was added after `omp for reduction` clauses. According to the OpenMP specification, this synchronization is implicit and mandatory, but the LLVM OpenMP implementation did not honor it. This issue has since been reported to Intel and corrected [111].

The OpenMP C benchmarks were all run with CLASS *A* datasets. Also, to evaluate more parallel codes, we used Blackscholes, an option pricing with Black-Scholes Partial Differential Equation , from the PARSEC benchmarks [112] . Unlike NAS benchmarks which focus HPC, PARSEC Blackscholes focus finance computing.

Table 5.1 summarizes for all the applications, their usage in this thesis.

5.2.2 Execution Environments

Table 5.2 describes the machines we used in this thesis. They belong to six different Intel CPU generations (Atom, Core2 Duo, Nehalem, Sandy Bridge, Ivy Bridge, and Haswell) and possess quite distinct memory hierarchies. They include different brands that target consumer workstations but also servers and embedded systems. These machines were selected to validate that benchmarking reduction strategy is applicable on significantly different architectures. They also attest that CERE replay process is portable across architectures. Each test machine in this thesis refer to one of these eight architectures.

We use Clang `3.3` and `3.4` to compile the C/C++ applications. Fortran codes were supported through Dragonegg with GCC `4.6` and `4.7` for respectively Clang `3.3` and `3.4`. We also use the Intel compiler `12.1.0` for architecture selection in section 6.4. We compile all OpenMP codes in this thesis with Clang `3.4`. To execute OpenMP applications, we use the open source OpenMP Runtime Library libiomp5 [92].

	Atom	Core2	Nehalem	Xeon Nehalem
CPU	D510	E7500	L5609	E5620
Freq (GHz)	1.66	2.93	1.86	2.40
Sockets	1	1	1	1
Cores per socket	2	2	4	4
Threads per core	2	1	1	2
L1 (KB)	56	64	64	64
L2 (KB)	512	3 MB	256	256
L3 (MB)	-	-	12	12
Ram (GB)	4	4	8	24
	Haswell	Ivy Bridge	Sandy Bridge	Xeon Sandy Bridge
CPU	i7-3770	i7-4770	E3 1240	E5
Freq (GHz)	3.40	3.40	3.30	2.40
Sockets	1	1	1	2
Cores per socket	4	4	4	8
Threads per core	2	2	2	2
L1 (KB)	64	64	64	64
L2 (KB)	256	256	256	256
L3 (MB)	8	8	8	20
Ram (GB)	16	16	6	64

Table 5.2: Test architectures.

5.3 CERE Coverage and Replay Accuracy

The codelet based benchmarking process is only viable if the codelets faithfully capture the original application behavior. This section evaluates the quality of codelet's extraction and replay.

CERE starts by extracting all representative codelets from the test benchmarks. For each codelet, we evaluate the accuracy of its replay. As discussed in section 3.4, we consider that a codelet is accurately replayed if its replay performance is within 15% of the original execution time.

In this section, we validate CERE on the NAS and SPEC benchmarks. CERE includes a report generator that automatically captures the execution traces, selects representative invocations and computes coverage and replay accuracy of a given set of benchmarks. The reports can be visualized in any modern web browser. The user clicks on a captured codelet in the call graph to see its temporal subsetting and replay accuracy statistics. The reports for all NAS and SPEC benchmarks can be viewed at `http://benchmark-subsetting.github.io/cere/`.

5.3.1 Serial Codelets Validation

CERE is evaluated on the NAS 3.0 serial benchmarks and the SPEC 2006 FP benchmarks compiled with `-O3` and LLVM `3.3`. Performance is measured

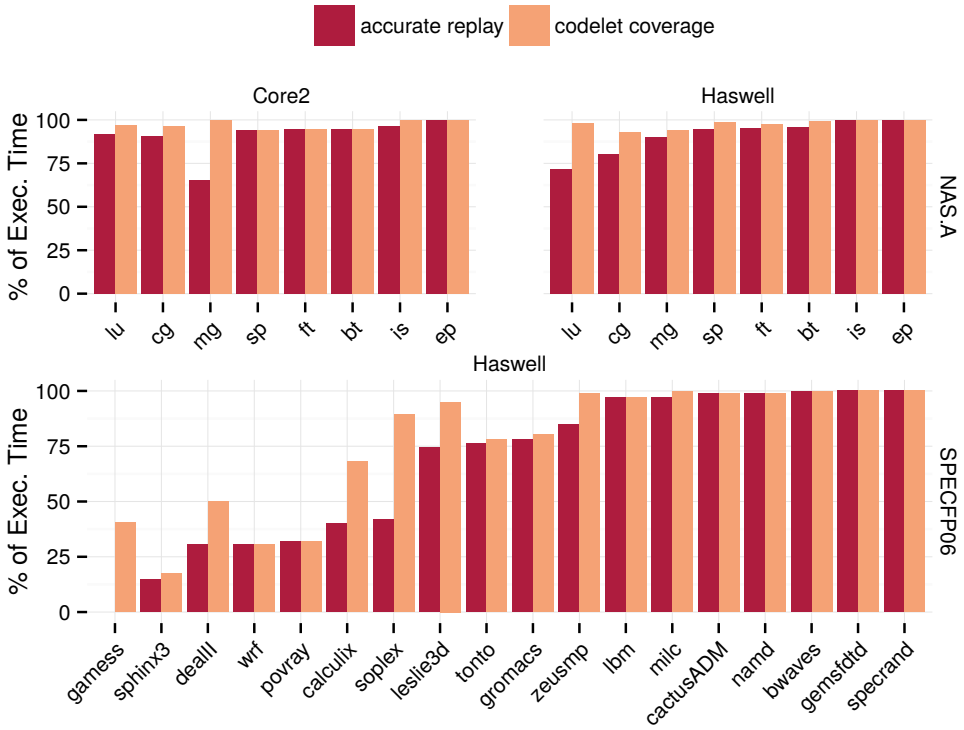


Figure 5.1: Evaluation of CERE on NAS and SPEC FP 2006. The Coverage is the percentage of the execution time captured by codelets. The Accurate Replay is the percentage of execution time replayed with an error less than 15%.

using the Time Stamp Counter (TSC) which provides a precision around 200 cycles. To ensure that the error upper bound due to measurement noise remains approximately 10% for all codelets, we removed codelets whose execution time was less than 2000 cycles per invocation.

Figure 5.1 shows for the NAS and SPEC 2006 FP benchmarks the percentage of execution time captured by codelets and the percentage of execution time that could be accurately replayed. On average, the extracted codelets cover 97.3% of the execution time in NAS and 76.6% in SPEC.

On NAS, both coverage and replay accuracy are very high. MG matching is a bit lower (65.1%) than the other benchmarks because of two borderline codelets with replay errors at 16.8% and 18.5%. With a *tolerated error* of 20%, we would have reached 95% coverage.

NAS codelets were captured and replayed on Haswell and Core2 to show that CERE reliably supports multiple architectures. The small differences in coverage between Haswell and Core2 are due to the changes in contribution of codelets to the execution time, for example CG spends relatively more time on I/Os on Haswell architecture.

SPEC FP results are evaluated on the Haswell architecture. Eleven out of eighteen benchmarks have high coverage and replay accuracy. Here is a list of the problems affecting the seven remaining benchmarks:

- *sphinx3, wrf, povray, and calculix* have low coverage because most of the time is spent in I/O operations. The current version of CERE does not capture codelets performing I/O because the dump does not preserve file descriptors state. However 100% of captured codelets match.
- *gamess and dealII* have low coverage because most of the performance is spent in loops taking less than 2000 cycles, which were not considered.
- *gamess* has low matching because the only remaining codelet, covering 40% of the execution time, is not accurately replayed. It is due to a warmup bug which is being investigated.
- *calculix* has low matching because of a borderline codelet `isortii` which has a replay error of 16% but accounts for 10% of the running time. It is a sort function which is very sensitive to warmup effects.
- *soplex* has low matching because CERE fails, due to a capture bug, to replay its main codelet covering 47.4% of the execution time.

Figure 5.2 shows that the reinlining and NoAlias-tagging performed during the replay compilation pass are beneficial in 11 benchmarks. Overall CERE coverage and accuracy are high in both NAS and SPEC benchmarks, showing that CERE codelets can be efficiently used as performance proxies for many applications.

CERE has higher replay accuracy than the state of the art code isolator tool, Codelet Finder. On NAS, Codelet Finder accurately replays 69% [71] of the execution time, whereas CERE replays 90.9%. On SPEC, Codelet Finder has very low replay accuracy or fails to extract codelets for many benchmarks (the 2013 version of Codelet Finder hangs on gamess, gromacs, cactus, calculix, tonto, specrand, and wrf), whereas CERE accurately replays 66.3% of the SPEC execution time.

5.3.2 OpenMP Codelets Validation

We start the OpenMP codelet validation on a simple case: we capture OpenMP regions on a fixed thread configuration and architecture. Then, we replay the codelets on the same architecture, with varying number of threads, but with the same thread placement affinity. While simple, this process ensures that codelets can be used to measure parallel regions exaction time. Section 6.3 extends this validation by varying the thread affinities in order to tune the thread placement strategy.

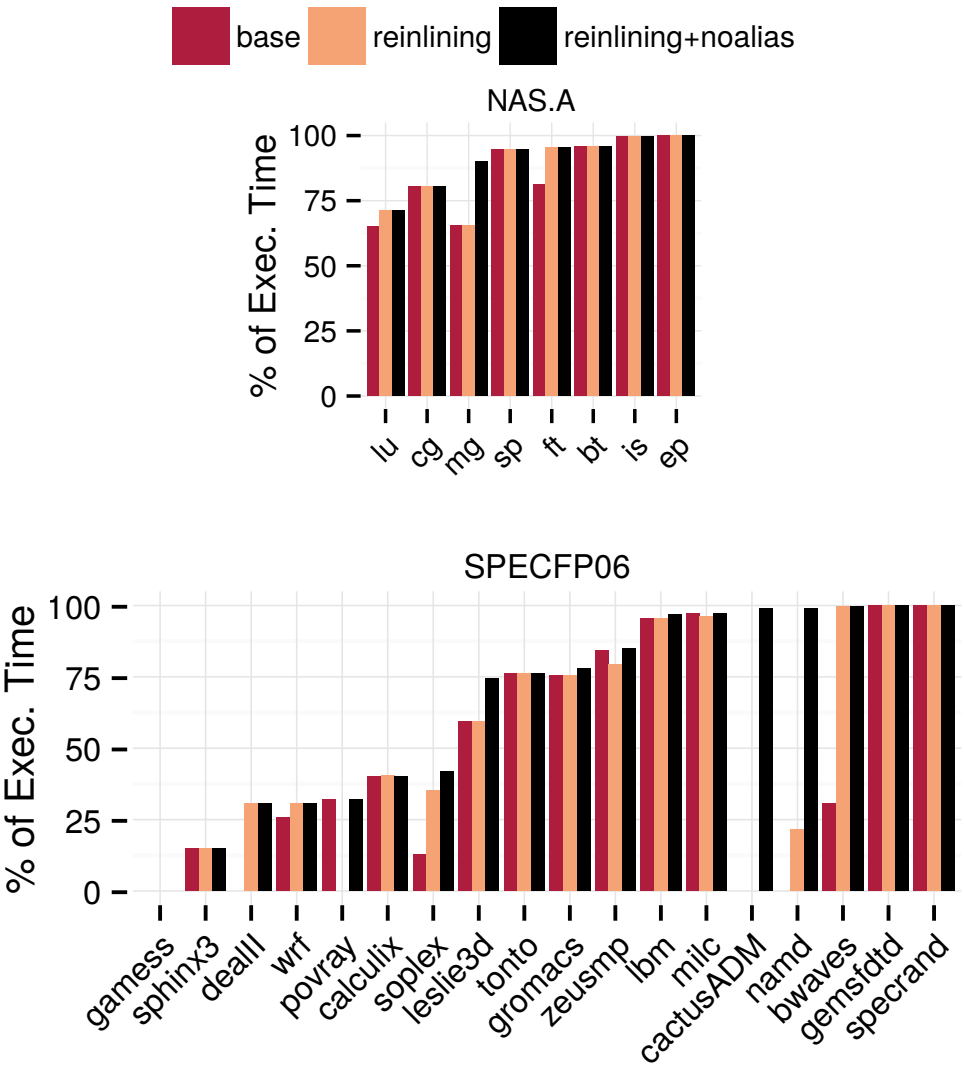


Figure 5.2: Percentage of execution time accurately replayed (error $< 15\%$) on the NAS and SPEC FP benchmarks with different replay configurations. Reinlining and explicitly marking cloned variables as NoAlias improve replay accuracy in eleven benchmarks.

To evaluate OpenMP replay accuracy, we extracted the full set of codelets from all the NAS NPB applications. The codelet set maps exactly to the parallel regions of the original OpenMP application. We reduced this full set, by removing the codelets that represent less than 5% of the original execution time. Originally 59 codelets were captured, and after filtering only 25 were kept. Then we measured the original and codelet execution time for each codelet, and computed the replay accuracy.

During this validation, the CERE page tracing capture presented in section 3.5.2 was not yet supporting OpenMP applications. Instead, to capture the working set of OpenMP applications, CERE used to take a full snapshot of the original application address space. The application is frozen using the `ptrace` system call, then a helper process dumps the memory contents to disk, and returns the control to the original application. Codelet Finder [74, 71] presented in the background uses a similar technique. This method is simple but presents some limitations. In particular, full memory dumps are large, and do not keep a trace of the pages touched during the parallel region execution. So, they are incompatible with the CERE NUMA aware capture and replay methodology described in section 3.5.5.

In these experiments, we ran capture using a single thread run and directly changed the thread number at replay to evaluate the different thread configurations.

Table 5.3 summarizes the replay accuracy over the NPB codelet set measured on Xeon Nehalem using a default `scatter` thread placement. CERE faithfully replays most of the parallel regions: the original and replay performance is close. This test machine has a single socket: there are no NUMA effects across the threads which explains why the NUMA aware replay was not required to faithfully replay the codelets. Also, a single working set was enough to faithfully replay the benchmarks parallel regions except for MG for which we needed to extract multiple working sets to accurately replay its codelet execution time.

Only two codelets are misreplayed: SP `xsolve` and CG `residual norm`. CG `residual norm` error is due to cache state differences between the original and the replay executions. We rely on an optimistic warmup strategy for parallel codes described in section 3.5.3 which is not accurate enough for this benchmark. CG `residual norm` working set is cold in the original run but incorrectly warmed-up during replay. We also suspect warmup issues for SP `xsolve`.

5.4 Codelet Exploration

In the previous section, we validated codelets replay by showing that it is similar to the original execution. This section illustrates how codelets can be used as proxies to evaluate different parameters.

Benchmark Parallel region	**Threads** 1	2	4	8	**weight %**
CG					
conj grad iteration loop	11.18	8.05	0.23	2.66	95.8
conj grad residual norm	12.26	11.27	31.756	3.08	66.4
MG					
resid	0.68	0.03	1.19	0.46	53.5
psinv	1.42	1.07	0.57	1.43	22.8
interp	8.76	8.09	8.18	1.75	07.2
rprj3	1.31	3.15	2.09	7.27	05.8
norm2u3	0.12	0.89	1.53	0.14	05.6
zero3	4.03	4.86	5.46	10.50	05.4
EP					
main	0.01	2.00	0.10	1.36	99.99
IS					
main random generator	0.22	0.50	0.52	3.76	80.6
rank	0.28	0.84	1.64	5.30	39.9
SP					
xsolve	23.27	17.37	10.59	4.37	33.4
zsolve	2.64	1.30	1.86	2.07	30.9
ysolve	8.46	5.55	6.66	5.61	32.7
compute rhs	1.61	1.60	2.1	0.57	26.8
BT					
zsolve	0.14	3.52	2.5	10.80	33.6
ysolve	0.20	3.76	1.82	6.25	32.3
xsolve	0.33	0.26	4.16	11.13	30.5
compute rhs	0.84	0.77	0.08	2.34	11.1
FT					
cffts2	0.38	2.58	3.29	3.20	31.3
cffts3	5.12	4.71	5.06	5.27	30.9
cffts1	1.29	0.99	0.11	0.84	30.7
compute indexmap	3.09	5.20	0.32	3.82	06.6
LU					
ssor iteration	0.03	0.86	0.53	0.03	98.7
rhs	0.89	1.09	0.58	2.01	27.4

The table shows the relative error between the original and codelet execution time for each NPB codelet over different thread configurations. Measures were performed on Xeon Nehalem. The weight % column is the contribution of the region to the total running time (the weight changes across thread configurations, here we consider the maximum).

Table 5.3: Codelet Replay Accuracy

	Core2	Xeon Nehalem	Xeon Sandy Bridge
Accuracy	1.84%	2.9%	7.4%
Reduction factor	25.2	27.4	23.7

Table 5.4: NAS 3.0 C version average prediction accuracy and benchmarking acceleration per architecture.

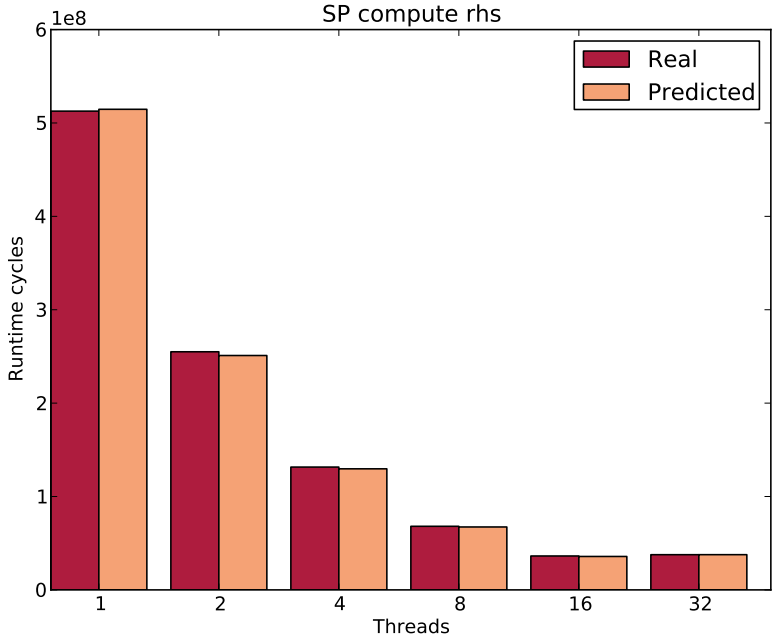


Figure 5.3: Real vs. CERE execution time predictions on Xeon Sandy Bridge for the SP compute rhs codelet

5.4.1 Scalability Exploration

This section describes how CERE predicts strong scalability of applications. These experiments validate over the NAS NPB the prediction model described in section 4.4

Table 5.5 details NPB prediction and benchmarking acceleration over three different architectures. As explained in section 5.3.2, codelets were extracted on a single thread.

Overall CERE scalability prediction is fast and accurate. Table 5.4 summarizes the average accuracy and acceleration over all the NPB. In figure 5.3 we compare the real and predicted execution time on SP `compute rhs` codelet which has the biggest execution time per invocation within SP.

Problems are highlighted in gray on table 5.5 and explained below:

- EP benchmark shows no acceleration at all with our method. EP

TABLE I
SCALABILITY PREDICTION ON NPB

	CORE2				NEHALEM							
	Accuracy		Reduction factor		Accuracy				Reduction factor			
Threads	1	2	1	2	1	2	4	8	1	2	4	8
CG	0.25	1.72	28.77	19.55	10.64	5.38	0.38	22.62	40.41	23.07	12.65	8.07
MG	1.94	1.13	0.55	1.14	0.32	0.74	1.25	0.11	1.81	1.94	2.26	2.73
EP	0.11	0.13	1.0	1.0	0.01	0.19	0.1	1.13	1.0	1.0	1.0	0.99
IS	4.16	4.29	1.01	1.23	0.37	1.63	2.08	4.89	1.12	1.12	1.15	1.2
SP	1.22	2.38	92.82	81.58	8.32	5.69	4.49	5.38	103.02	97.52	94.69	92.74
BT	0.87	4.1	36.48	26.41	0.16	3.1	0.59	7.98	38.33	41.05	44.26	49.18
FT	1.26	3.34	1.76	1.22	0.93	0.75	0.05	0.21	1.93	1.96	2.11	2.32
LU	1.72	1.0	54.44	54.2	0.24	1.0	0.63	0.67	52.91	52.53	51.81	50.25

	SANDY BRIDGE											
	Accuracy						Reduction factor					
Threads	1	2	4	8	16	32	1	2	4	8	16	32
BT	1.51	2.47	2.64	12.99	18.28	15.66	35.83	39.83	43.86	51.13	55.47	46.99
EP	0.1	0.09	5.06	0.31	2.81	0.7	1.0	1.0	1.05	1.0	1.02	0.98
LU	0.01	2.08	1.96	1.95	0.19	2.43	51.86	50.9	50.67	47.7	45.39	20.78
FT	3.09	1.35	0.91	1.3	0.68	2.52	1.9	1.98	2.17	2.49	2.96	2.77
SP	0.78	0.06	0.14	0.05	3.38	0.67	92.19	88.72	85.89	83.01	75.04	49.21
CG	15.15	3.37	9.51	18.77	13.86	26.62	37.28	17.95	9.64	5.29	4.31	4.82
IS	1.53	6.22	24.83	17.14	30.57	27.57	1.12	1.08	1.61	1.61	1.0	0.99
MG	1.07	6.08	4.24	2.3	4.73	54.24	1.88	1.92	2.32	3.01	3.77	1.7

Table 5.5: Overall CERE accurately predicts the scalability on the three architectures. The average prediction error is 4.9%. The prediction is in average 25 × faster with CERE. In this experiment a separate initialization step was performed on each architecture. In Xeon Sandy Bridge the error is higher on IS and with 32 threads. IS misprediction is due to the fact that changing the number of threads changes the memory layout which impacts the sequential regions violating our model assumptions.

sequential part is negligible and its single parallel region is only invoked once. Therefore CERE replay strategy is not faster than the original run.

- IS has low accuracy on Xeon Sandy Bridge. This is because the memory layout in IS depends on the number of threads. A higher number of threads changes the blocking and impacts the execution time of sequential regions as demonstrated in [113]. This violates our base assumption of invariant sequential execution time.

- On Xeon Sandy Bridge, replays running with 16 threads or above show high misprediction errors in many cases. The test machine has 2 NUMA sockets. The CERE warmup and replay strategy that was used in these experiments do not support NUMA behaviors. Thread configurations with high number of threads are more sensitive to NUMA latencies which explains why we have more prediction errors.

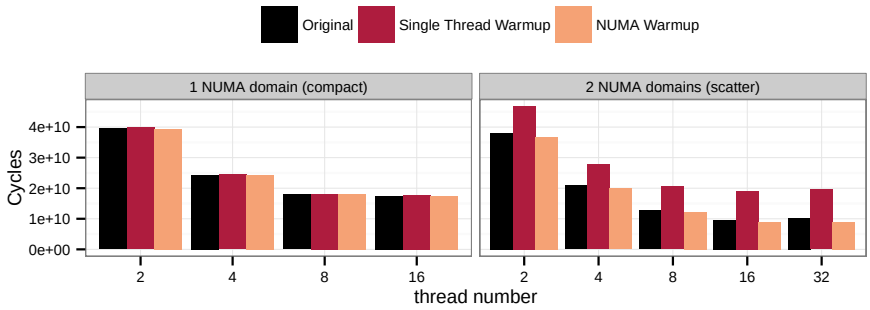


Figure 5.4: Prediction accuracy of a single threaded warmup versus a NUMA aware warmup on BT `xsolve` on Xeon Sandy Bridge. Only a NUMA aware warmup is able to predict this region execution time on a multi NUMA node configuration. We note that the capture was performed with 16 threads.

5.4.2 Cross Architecture Scalability Replay

CERE codelets are portable: they can be extracted on an architecture and replayed on another one. To demonstrate cross-architecture portability, we extracted the NPB codelets on Xeon Nehalem and replayed them on Xeon Sandy Bridge. Table 5.6 summarizes the results. Overall accuracy is high, except for CG replays and high number of threads replays.

As explained in section 3.5.1, codelets replay made no assumptions about the register layout. So codelets are portable across architectures that have the same memory layout. Since Xeon Sandy Bridge and Xeon Nehalem have the same memory layout, we can cross capture and replay our codelets.

As before, the misprediction with many threads is caused by the inability of CERE to correctly predict scalability when there are NUMA behaviors. The following chapter rely on the NUMA aware capture and replay strategy which diminishes these scenarios. CG misprediction is caused by our heuristic warmup which, in this case, proves to be insufficient. Indeed our warmup strategy is optimistic: it assumes that the working set is hot in the original execution, which is not true in this scenario. Perspectives for improving the warmup strategy are discussed in the conclusion.

5.4.3 NUMA Aware Replay

In the previous sections, we observe that without a NUMA aware capture, we cannot faithfully replay the codelets with some `scatter` placement thread configurations. This section illustrates how the CERE NUMA aware capture operates over a parallel region.

Figure 5.4 outlines this problem on a 2-NUMA nodes architecture. Previous CERE warmup uses a single thread to remap the pages to their original addresses: all the pages are bound to a single NUMA node. Replays accurately predict the execution time as long as the affinity binds threads to the

Benchmark Parallel region	**Threads** 1	2	4	8	16	32	**weight %**
CG							
conjgrad iteration loop	7.57	4.44	1.69	1.35	5.44	3.06	95.6
conjgrad residual norm	18.86	15.65	6.99	24.3	42.63	36.7	93.5
MG							
resid	1.34	1.64	1.72	1.59	1.03	63.20	56.0
psinv	20.27	3.13	2.09	37.25	10.06	80.41	23.1
zero3	6.98	3.20	8.12	8.23	3.33	71.06	09.2
interp	7.13	6.55	6.22	2.48	7.50	92.03	07.4
rprj3	7.53	6.50	9.86	11.93	15.38	94.72	05.6
EP							
main	0.04	0.10	0.05	0.03	0.26	0.44	99.99
IS							
main random generator	0.63	1.09	0.67	9.59	29.14	1.98	82.4
rank	0.69	0.42	0.24	0.22	1.49	8.24	42.8
SP							
zsolve	2.54	0.01	6.02	10.4	3.94	4.77	35.1
xsolve	2.80	2.21	4.84	4.25	0.90	9.48	31.9
compute rhs	2.04	1.60	1.30	0.19	1.42	0.53	27.7
ysolve	0.51	0.03	5.23	10.94	1.84	3.72	27.1
BT							
zsolve	0.30	2.00	1.58	12.51	21.08	24.02	33.7
xsolve	0.87	3.97	5.01	12.53	19.78	15.56	33.4
ysolve	0.13	1.75	5.58	16.30	22.39	13.05	33.1
compute rhs	0.19	0.99	1.96	1.43	2.30	3.619	09.5
FT							
cffts2	1.66	0.47	1.57	0.80	0.70	0.91	30.5
cffts3	9.49	7.93	7.83	7.58	7.13	2.10	30.4
cffts1	0.03	0.84	1.5	0.08	1.16	0.34	30.2
compute indexmap	0.80	1.39	2.70	0.30	0.83	8.78	10.1
LU							
ssor iteration	1.44	0.01	2.58	2.27	0.32	0.48	76.8
rhs	0.12	0.54	0.26	0.11	0.45	0.73	26.1

In this experiment, codelets were extracted on Xeon Nehalem and replayed on Xeon Sandy Bridge, enabling quick cross architecture scalability prediction. The table shows the relative error between the original and codelet execution time for each NPB codelet over different threads configurations. The prediction error is low, except for CG residual norm codelet and thread configurations with high number of threads. CG misprediction is caused by our heuristic warmup which, in this case, proves to be insufficient. Misprediction on high number of threads is again due to current used warmup and replay approach which does not support NUMA behaviors. The weight % column is the contribution of the region to the total running time (the weight changes across thread configurations, here we consider the maximum).

Table 5.6: Cross Architecture Replay Accuracy

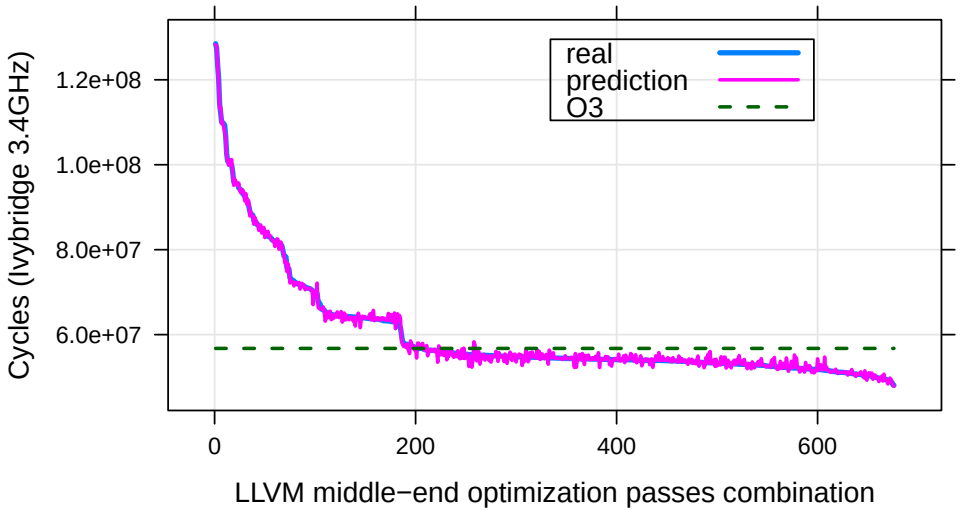


Figure 5.5: SP `ysolve` codelet. 1000 schedules of random passes combinations explored based on O3 passes. We only consider compilation sequences that produce distinct binaries. The passes combinations are ordered according to their real execution time.

same NUMA node. Otherwise, the replay paies NUMA latencies that do not appear in the original run and which cause prediction discrepancies.

The following chapter 6 takes advantage of this approach to tune different thread configurations.

5.4.4 Compiler Exploration

In this section, we test codelets ability to explore compiler sequences through an example.

Here is the experiment setup used to explore different compiler optimization sequences. The compilation search was performed on `LLVM 3.4` using a random pass selection. We use `LLVM opt` and `llc` to change respectively middle-end and back-end optimizations.

Middle-end passes have different impact depending on their order of execution, and can be executed multiple times. `-O3` is a manually tuned sequence composed of 65 ordered passes aiming to provide good performances. In this thesis, random compilation sequences were generated by down-sampling the `-O3` default sequence. Each pass was removed with a 0.7 probability, and the process was repeated four times to explore the impact of pass repetitions. We empirically found that this generation method produces good and diverse candidates.

Back-end passes were selected among `-O0`,`-O1`,`-O2` and `-O3`. We also consider two types of vectorial instructions: AVX and SSE.

Figure 5.5 illustrates the compiler tuning on a region. It presents SP

`ysolve` original execution time versus CERE predicted execution time over 1000 complier sequences. Original compiler sequences are sorted in a descending order. CERE model faithfully predict the different optimizations execution time.

Chapter 6 extends this tuning to all the NAS benchmarks. It also produces the hybrid binaries described in section 3.6.

Acknowledgments

The validations of CERE were done in collaboration with Chadi Akel, Eric Petit, William Jalby, and Pablo de Oliveira Castro. I also would like to thank Florent Conti for his contribution to the scalability prediction model.

5.5 Conclusion

This chapter presented the validation of the codelet approach for benchmarking. It also extends the validation to scalability problems. We note that, predicting the scalability of applications raises NUMA problems. Correctly supporting NUMA behaviors requires a supplementary step in order to faithfully capture and replay the parallel regions. The following chapter uses the page tracing strategy that tackles these NUMA behaviors.

Chapter 6

Holistic Tuning

Contents

6.1 Introduction

There are different approaches to tune parameters or to select the best architecture for an application. For instance, iterative compilation presented in section 2.6 is a well known search method which outperforms default `-O3` compiler optimization level. The idea is to apply successive compiler transformations to a program and to evaluate them by executing the resulting code. Other execution driven studies [83, 84] are used to find the best architectures for a set of programs or to explore the efficiency of different thread placement strategies or frequencies.

A common point of these search studies is that they require many program evaluations and executions. The problem is that executing applications is a costly and time consuming process, especially if we have thousands parameters to evaluate on multiple architectures.

In this chapter, we perform a piecewise holistic exploration to find the best compiler optimizations, thread configurations, and architectures for a set of applications. We rely on CERE: representative loops or OpenMP parallel regions are extracted as codelets. Instead of evaluating parameters or architectures on the whole applications, we separately evaluate them on each codelet.

The codelet based search enhances both the search cost and the search benefits:

- through the reduction strategies presented in chapter 4, codelets minimize the evaluation of the redundancies within the applications during the search process,

- the piecewise evaluation finds the best parameters for each region. CERE combines them in a single *hybrid* binary that outperforms standard overall program *monolithic* tuning.

Similarly to chapter 5 we also use the benchmark reduction factor and the accuracy to quantify the quality of the codelet exploration. We also note that we apply only temporal reduction to accelerate thread and compiler configurations tuning. Architecture selection is currently the only use case where we perform both spatial and temporal reductions.

In Section 6.2, through a motivating example, we present how costly it is to simultaneously explore compiler optimizations and thread configurations. We also demonstrate the benefits of the piecewise tuning. Section 6.3 gives an overview how codelets cane tune thread NUMA configurations. Section 6.4 applies the codelet reduction model to quickly evaluate new architectures. In section 6.5, we use codelets to explore for each region of code, different compiler optimizations. Finally, we discuss the limitation and the future work of the codelet tuning methodology in section 6.6.

6.2 Motivating Example

We demonstrate how CERE operates on SP from the NAS NPB benchamrks over Xeon Sandy Bridge. CERE autotuning achieves a `1.82`× performance speedup over the standard parameters levels. Thanks to the CERE codelet approach, the exploration time is approximately five times cheaper compared to the whole program iterative compilation.

CERE starts by profiling SP and automatically selecting representative OpenMP regions to tune. `Xsolve`, `ysolve`, `zsolve`, and `rhs` are selected and cover 93% of SP execution time. CERE extracts these regions as codelets and tunes them with a holistic exploration across three dimensions: thread number, thread placement, and LLVM compiler passes. Once satisfying parameters are found, CERE produces an hybrid application where each region uses the best found parameters.

We explore the `12` thread configurations presented in section 6.3 and `150` LLVM optimization sequences generated using the random sub-sampling presented in section 5.4.4. Combining them produces an exploration space of `1800` points, which gives an insight of how costly it is to simultaneously tune multiple parameters.

Figure 6.1 shows the performance of two SP parallel regions across this exploration space. We notice that there is a strong interaction between the compiler and the thread parameters as they both significantly impact

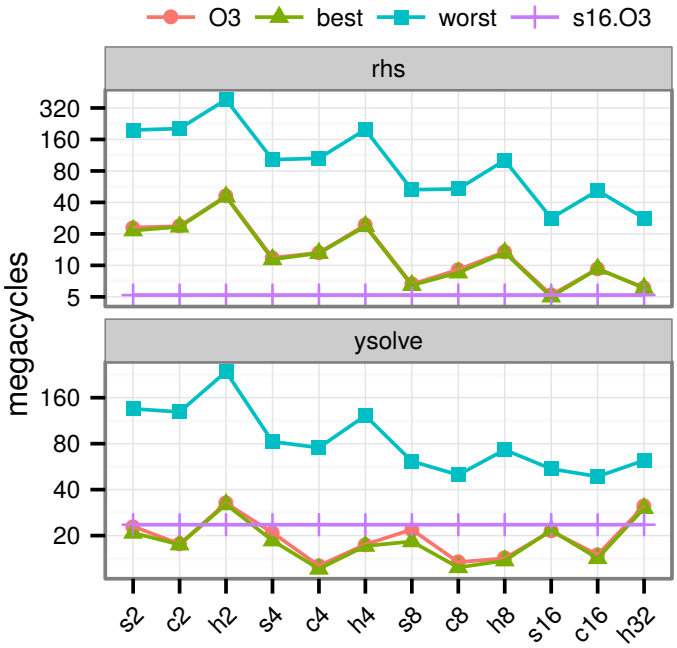


Figure 6.1: Tuning exploration for two SP regions. For each affinity, we plot the best, worst, and `-O3` optimization sequences. Custom optimization beats `-O3` for `s2` (i.e. scatter with 2 threads),`s4`, and `s8` on `ysolve`.

the performances. Moreover, the best parameters are different for the two regions: scatter placement is best for `rhs` while compact benefits `ysolve`.

CERE makes it possible, through codelet replay, to explore each region independently. Also, thanks to the temporal reduction model presented in section 4.2, CERE accelerates the evaluation of thread affinities and compiler optimizations on SP, which are respectively `5.84`× and `4.52`× times faster than a full application evaluation while keeping a low average error of `2.33%`.

Custom parameters outperform the standard `16 threads scatter s16 -O3` on SP. Table 6.1 shows the performance of different thread affinities compiled with `-O3`. The best custom thread affinity `0;1;2;3;4;5;6;7` (single NUMA socket) achieves a speedup of `1.71`× over the standard `16 threads scatter` (two NUMA sockets).

We explored with CERE `350` compiler optimization sequences on the best single NUMA configuration found above. `Xsolve` and `ysolve` work best at the default `-O2 level`, but a custom best sequence is found for `zsolve` and `rhs`. Figure 6.2 shows the performance of each region compiled with the default optimization and the best custom sequences. No single sequence is the best for all regions. CERE hybrid compilation presented in section 3.6 produces a binary where each region is compiled using its best sequence, achieving a speedup that cannot be reproduced using traditional monolithic compilation.

thread affinity		xsolve	ysolve	zsolve	rhs	total
s2	0;8	32.3	23	28.5	23	106.8
c2	0;1	21.4	17.6	18.1	23.7	80.8
h2	0;16	40	32.6	23	46.1	141.7
s4	0;8;1;9	25.9	20.9	26	12.1	84.9
c4	0;1;2;3	15.5	12.7	13.8	13.2	55.2
h4	0;16;1;17	23.8	17.5	16	24.3	81.5
s8	0;8;1;9;2;10;3;11	24.4	21.9	28.6	6.9	81.8
c8	0;1;2;3;4;5;6;7	14.4	13.4	14.3	9.1	51.2
h8	0;16;1;17;2;18;3;19	17.7	14.2	13.9	13.5	59.3
s16	16 scatter	25.1	21.4	35.5	5.3	87.4
c16	16 compact	17	15	15.5	9.7	57.2
h32	32 scatter	36	31.2	38.9	6.4	112.4

Table 6.1: Execution time in megacycles of SP parallel regions across different thread affinities with `-O3` optimization. For n threads, we consider three affinities: scatter s_n, compact c_n, and hyperthread h_n. Executing SP with the c8 affinity provides an overall speedup of `1.71`× over the standard (s16).

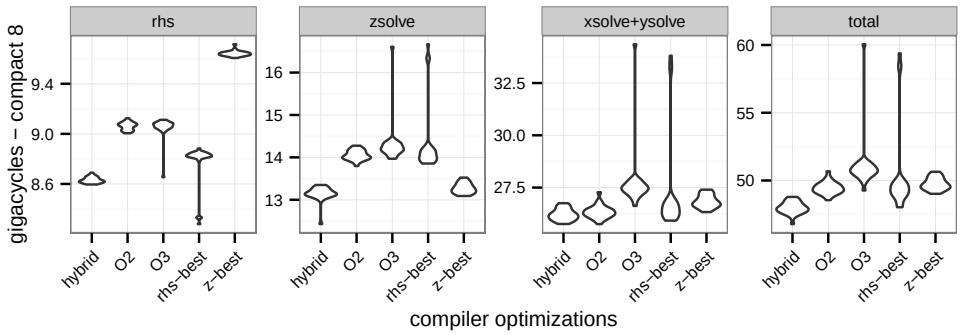


Figure 6.2: Violin plot execution time of SP regions using best NUMA affinity. Measures were performed 31 times to ensure reproducibility. When measuring total execution time, Hybrid outperforms all other optimization levels, since each region uses the best optimization sequence available.

thread affinity	Explicit affinity	# NUMA domains	# physical cores
s2	0;8	2	2
c2	0;1	1	2
h2	0;16	1	1
s4	0;8;1;9	2	4
c4	0;1;2;3	1	4
h4	0;16;1;17	1	2
s8	0;8;1;9;2;10;3;11	2	8
c8	0;1;2;3;4;5;6;7	1	8
h8	0;16;1;17;2;18;3;19	1	4
s16	16 scatter	2	16
c16	16 compact	1	8
h32	32 scatter	2	16

Table 6.2: Thread configurations evaluated on Xeon Sandy Bridge. `s16` maps a single thread to all the physical cores and uses two NUMA domains. It is considered as the default thread configuration for this test machine.

6.3 Thread Configurations

We tuned the thread configurations of the NAS NPB over the Xeon Sandy Bridge. We chose this test machine to explore thread affinities because it has 2 NUMA sockets and each socket has 8 physical (16 hyper-threaded) cores.

Thread configurations were selected to explore different degrees of parallelism, NUMA and hyper-threading effects. Xeon Sandy Bridge has 16 physical cores, so we did not explore configurations beyond 32 threads.

We used the Intel kmp affinity [114] notation to characterize the thread placement. Cores ranked between 0 and 7 reference the physical cores of the first NUMA node while cores between 8 and 15 reference the physical cores of the second NUMA node. Similarly, cores from 16 to 23 and from 24 to 31 reference the hyper-threaded cores of respectively the first and the second NUMA node. Table 6.2 describes the `12` threads configurations combining different number of threads and affinity mappings that we explore. The purpose is to outperform the default `16 threads scatter` configuration.

CERE page memory capture was performed on a `16 threads scatter` run using the NUMA aware capture described in section 3.5.5. Thanks to the temporal reduction model presented in section 4.2, it is possible to quickly evaluate the impact of each thread configuration on only a few datasets.

Table 6.3 evaluates CERE thread affinities replay accuracy and reduction factor over NAS OpenMP. We focused on regions representing more than `5%` of the application execution time. Detailed reports of these regions are presented in table 6.4. On average, a region exploration is `6.55`× faster [1] with codelets than with whole program evaluations. This reduction factor

[1] Unlike in section 5.3.2, we always replay codelets with four warmup invocations. Even if they are called once in the original execution. This choice explains why CERE is four times slower to evaluate EP.

	Compiler passes			Thread affinity		
	#Regions	Accuracy	Reduction factor	#Regions	Accuracy	Reduction factor
BT	3	98.73	79.63	4	95.24	5.28
CG	2	98.65	3.39	2	79.48	1.23
FT	5	98.3	2.6	5	90.71	2.17
IS	3	96.64	1.26	2	94.85	1.04
SP	6	98.78	68.9	4	97.66	20.07
LU	7	95.04	8.49	2	99.00	12.64
EP	1	83.08	0.36	1	99.31	0.25
MG	4	97.22	0.28	4	93.04	0.45
AVG		95.8	20.61		93.66	5.39

Table 6.3: The **accuracy** of the codelet prediction is the relative difference between the original and the replay execution time. The **benchmark reduction factor** or acceleration is the exploration time saved when studying a codelet instead of the whole application. CERE fails to accelerate EP and MG evaluation: EP has a single region with one invocation while MG displays many performance variations.

is achieved due to the CERE temporal reduction: CERE only replays the representative invocations.

As we increase the data sets, the warmup cost overhead becomes smaller compared to the replay execution time. We tested `xsolve` BT with CLASS B data sets and a single warmup invocation to achieve an acceleration of `9.48`×, twice the one achieved in class A, with an accuracy of `98.36%`.

The average CERE prediction accuracy is `93.66%`. It allows the autotuner to outperform the standard scatter `s16` over `EP`, `FT`, `LU`, and `SP` and to perform an average speedup of `1.40`× (see Fig. 6.3). We note that there there is no best thread affinity that wins over all the others: `h32`, `s16`, and `c8` are all optimal on at least two applications.

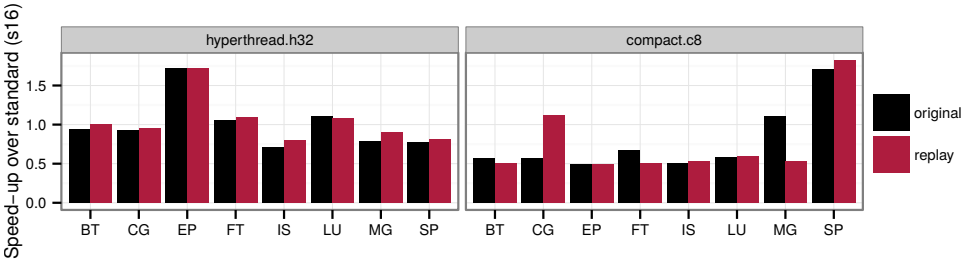


Figure 6.3: Original and CERE predicted speedup for two thread configurations. Replay speedup is the ratio between the replayed target and the replayed standard configuration. CERE accurately predicts the best thread affinities in six out of eight benchmarks. For CG and MG, we miss-predict configurations that use all the physical cores.

Benchmarks	Regions	Invocations	Accuracy	Benchmark reduction factor	Coverage
BT	xsolve	201	92.33	4.96	35.4
	ysolve	201	96.7	4.54	28
	zsolve	201	97.07	3.99	29.7
	rhs	201	95.68	18.71	5.7
CG	conjgrad@405	400	72.71	1.46	71.1
	conjgrad@551	16	95.48	0.8	24.6
FT	cffts1	8	95.54	1.7	21.2
	cffts2	8	94.25	1.68	23.8
	cffts3	8	96.5	1.18	22.9
	evolve	6	82.06	0.94	14
	indexmap	2	79.76	5.59	18.1
IS	main	1	95.2	0.35	71.4
	rank	11	93.77	2.86	27.6
SP	xsolve	401	96.24	20.03	28.3
	ysolve	401	97.11	20.25	27.7
	zsolve	401	98.98	18.95	38.1
	rhs	402	98.37	28.12	5.6
LU	ssor	250	99.51	12.45	81
	rhs	251	96.79	12.73	18.7
EP	main	1	99.31	0.25	99.7
MG	resid	42	93.78	0.52	52.7
	psinv	40	94.32	0.57	16.2
	interp	35	87.06	0.53	8.9
	zero3	39	78.27	0.5	7.4

The **accuracy** of the codelet prediction is based on the relative difference between the original and the replay execution time. The **Benchmark reduction factor** or acceleration is the exploration time saved when studying a codelet instead of the whole application. **Invocations** display the number of times a region is called inside the application. Only regions covering more than `5%` of the application execution time are selected.

Table 6.4: Codelet Exploration of Thread Configurations on Xeon Sandy Bridge

6.4 Architecture Selection

Selecting the best architecture for a set of applications is a costly process which requires benchmarking the applications on the different systems. Given an initial benchmark suite, our method produces a set of reduced benchmarks that can be used in place of the original one to select the best architecture per benchmark.

While we only rely on temporal reduction for compiler and thread tuning, we also take advantage of spatial reduction which is presented in section 4.3 to reduce the evaluation cost of new architectures.

We profile and extract the codelets on a reference architecture, Nehalem, and extrapolate the benchmarks performance on various target architectures. The metric set used to cluster the codelets is trained across Sandy Bridge and Atom over Numerical Recipes (NR) benchmarks. We validate this model by applying it on a new target architecture, Core2, over an unseen benchmark suite, the NAS SER benchmarks using the CLASS B data sets.

Some of the experiments described in this section are prior to CERE release so we extracted the codelets with Codelet Finder (CF) [74, 71]. Unlike CERE, CF does not provide a temporal reduction strategy. So, we manually select a representative invocation for each region of code. It also relies on an optimistic working set warmup approach: it replays the codelet over itself. CF codelets were compiled in these experiments with `icc 12.1.0 -O3 -xsse4.2` except on Atom which does not support `-xsse4.2`

First, we focus on the number of clusters used during the spatial reduction. In particular, we notice that the elbow method selects a clustering that provides a good trade-off between benchmarking reduction factor and prediction accuracy. Then we analyze codelets and applications prediction. Finally, to validate the spatial clustering with CERE, we reproduce the same methodology with CERE codelets.

Reduction Factor Versus Accuracy

Figure 6.4 shows the trade-off between prediction accuracy and benchmarking reduction factor while we increase the number of spatial clusters. As expected the more clusters we add, the lower the median error becomes. On the other hand, the benchmarking reduction becomes less effective because we have more codelets to run on the target architecture. The dashed line in figure 6.4 marks the number of clusters chosen by the elbow method, here 18. If needed, the user can tune the number of clusters depending on what he wants to optimize: faster benchmarking or better prediction accuracy.

The elbow clustering achieves a high reduction factor [2] between 23 and 44, while maintaining a low prediction error between 3.9% and 8%. Unsur-

[2] The reduction factor is the result of both spatial and temporal reductions.

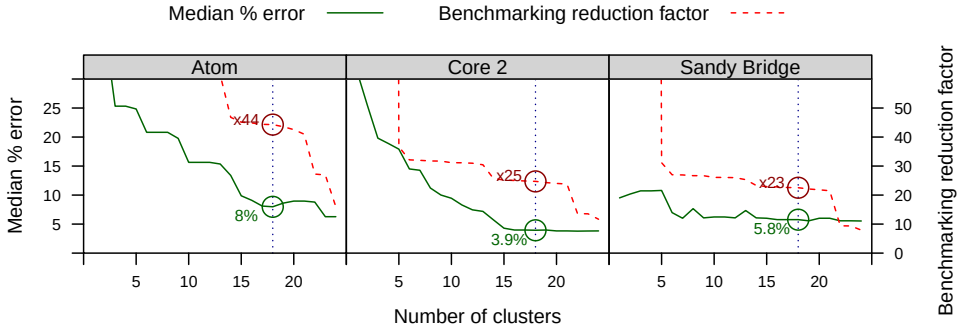


Figure 6.4: Evolution of prediction error and benchmarking reduction factor on NAS codelets as the number of clusters increases. The dotted vertical line marks 18, the number of clusters selected by the elbow method.

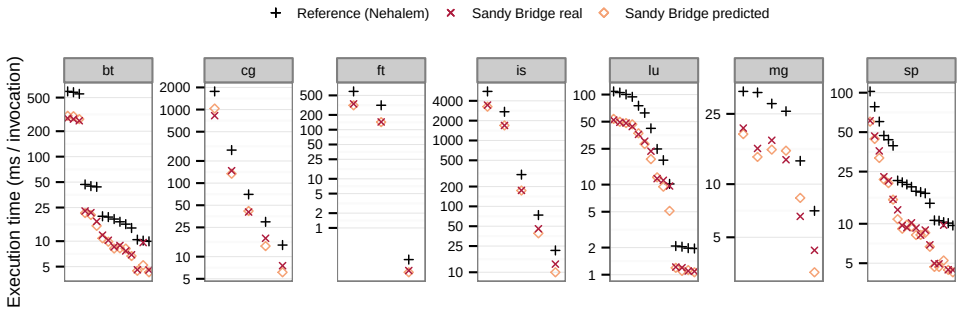


Figure 6.5: Predicted and Real execution times on Sandy Bridge compared to the Nehalem reference execution. Each box presents the codelets extracted from one of the NAS applications. Only three codelets in BT, LU, and SP are mispredicted.

prisingly, Atom, the most different architecture from the reference, has the highest prediction error.

Codelet Performance Prediction

Figure 6.5 shows the predicted and real execution times on Sandy Bridge. The boxes gather the codelets by application. The applications may contain codelets coming from different clusters with different speedups. The execution time on Sandy Bridge is predicted with a median error of 5.8%. The error mainly comes from short-lived codelets (less than 10 ms per invocation) which are more affected by measurement errors such as instrumentation overhead. Codelets are faster on Sandy Bridge than on the reference. It is not surprising as Sandy Bridge frequency is almost twice the reference one. The median prediction error is 8% for Atom and 3.9% for Core2.

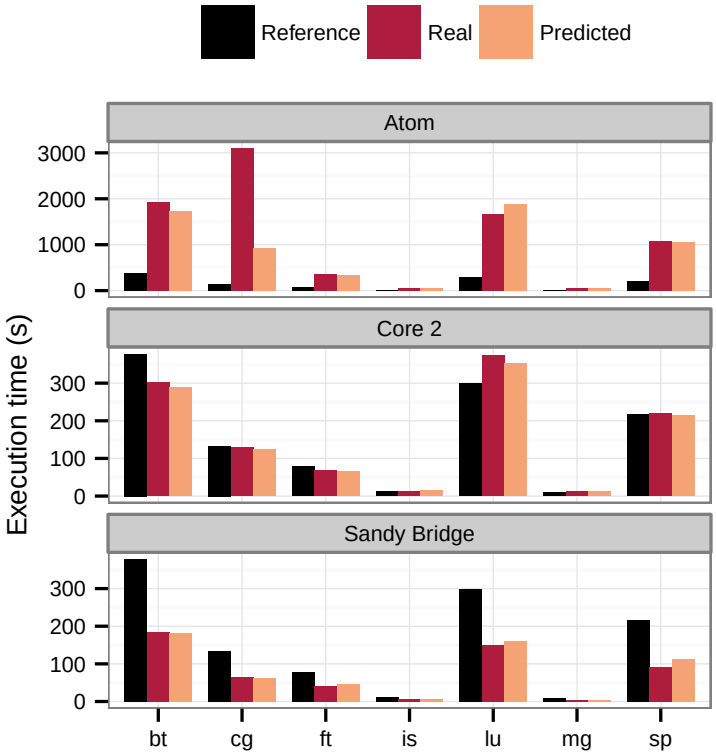


Figure 6.6: Predicted and Real execution times on the target architecture compared to the execution time on the reference architecture.

Application Performance Prediction

Codelet Finder Codelets capture 92% of the execution time of the original NAS applications [71]. Therefore, by aggregating the individual codelet predictions, we accurately predict the original applications performance.

The whole application prediction is done in two steps. First, we estimate the speedup of the part of the application covered by codelets. The application's codelets predictions are aggregated and weighted by their number of invocations. Second, we assume that the speedup of the unknown part of the application is equal to the one of the covered part.

Figure 6.6 shows the application prediction on the three target architectures. Atom is significantly different from the reference: it is an in-order processor, without L3 cache, nor SSE4 vector instructions. Moving to Atom slows down all the benchmarks. The prediction accuracy on Atom is high, except for the Conjugate Gradient (CG) benchmark. CG's huge error is caused by a single codelet representing 95% of its execution time. This codelet is well-behaved on Nehalem and is selected as the representative. Yet, on Atom the extracted microbenchmark is much faster than the original codelet and incurs 1.6 times less cache misses. The microbenchmark is

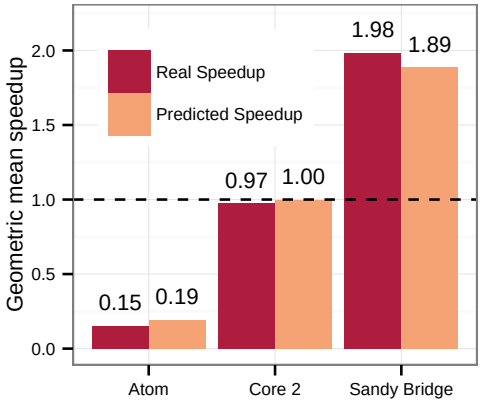


Figure 6.7: Geometric mean speedup per architecture using CF codelets.

not preserving the cache state. This behavior was only observed on Atom.

On Sandy Bridge, all the applications are faster. Sandy Bridge has the fastest frequency and a more modern microarchitecture than the reference. The prediction accurately captures the speedups for all the original applications.

Core2 microarchitecture is older than our reference, yet it has a higher frequency. The performance between both architecture is very close, providing an interesting challenge for system selection. Indeed, the best architecture depends on the application of interest. Some applications are faster, like BT and FT, while other are slower, like LU. Our reduced benchmark set captures this behavior and correctly predicts the trend allowing the user to smartly select the best architecture depending on the application.

In order to evaluate the overall benefits of an architecture, we compute the geometrical mean of the applications speedups. Figure 6.7 shows the predicted and the real speedup for each architecture. The reduced benchmarks accurately predict the expected speedup for each architecture.

Capturing Architectural Changes

To illustrate how our method captures architecture change, we consider two of the 18 clusters. Cluster A contains two codelets `LU/erhs.f:49-57` and `FT/appft.f:45-47`. Both are a triple-nested loop with high latency operations such as division and exponential. They are computation bound. Cluster B also contains two codelets `BT/rhs.f:266-311` and `SP/rhs.f:275-320`. Both are computing a three-point stencil on five planes. Codelets from cluster B are memory bound.

Our metrics correctly separate the two performance patterns: static IPC is high in cluster A whereas memory and cache bandwidths are high in cluster B. The compute bound cluster A is 1.37 times *faster* on Core 2 due to higher clock frequency. On the contrary, the memory bound cluster B is

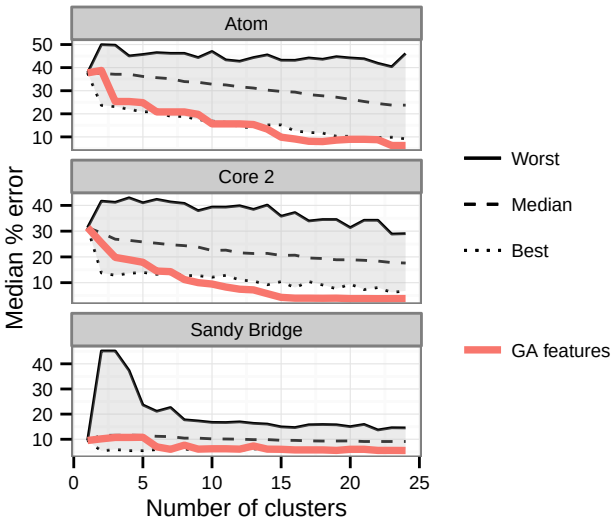


Figure 6.8: Genetic-Algorithm metric clustering compared to random clustering. For each number of clusters, from 2 to 24, 1000 random clusters are evaluated. Clustering with our GA metric set is consistently close or better than the best random clustering (out of 1000).

1.34 times *slower* on Core 2 because the last-level cache is four times smaller than the reference. The clustering correctly separates the two behaviors producing an accurate prediction for both groups.

Evaluation of the Metric-Guided Clustering

Figure 6.8 compares our metric-guided clustering against 1000 random clusterings. We make K, the number of clusters, vary from 1 to 24. For each value of K, we generate 1000 random partitionings into K clusters. We compute the prediction error for each partitioning after applying steps D and E.

The proposed metric-guided clustering is most of the time close or better than the best random clustering. Our choice of metrics and clustering yields competitive results.

Architecture Selection with CERE

We reproduce the same methodology with CERE. CERE profiles and extracts the codelets from the NAS on Nehalem. Then, through the spatial and temporal reduction presented in section 4 it reduces the codelets to a representative subset. Finally, we replay this subset on the target architectures.

Figure 6.9 compares the speedup computed using CERE replays to the real speedup measured by running the full benchmark suite. The perfor-

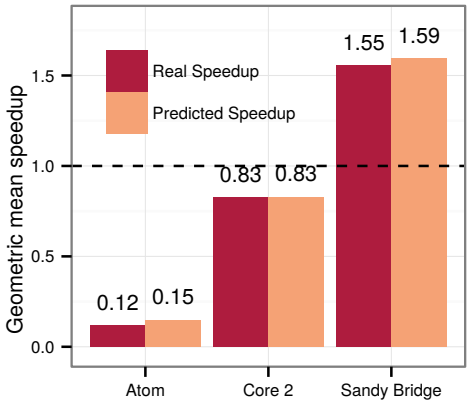


Figure 6.9: NAS geometric mean speedup on three architectures. Baseline is a NAS run on Nehalem compiled with `icc 12.1.0 -O3 -xsse4.2`. The predicted speedup is computed by using the replay performance of eighteen CERE representative codelets using *Working Set* warmup. There is a difference between the speedups of this plot compared to the ones presented in the Figure 6.7 because using CERE requires to compile the target architectures code with LLVM instead of icc.

mance predictions are very close, but CERE replays are 7.3× to 46.6× cheaper than running the full benchmarks.

Table 6.5 details the benchmark reduction cost achieved by only replaying the selected representative codelets. We observe that the *Working Set* warmup is much faster than the *Page Trace* warmup that has the overhead of replaying the memory access history. In this particular experiment, the prediction is the same for both warmup techniques, therefore we recommend to use *Working Set* warmup which is much faster. The benchmark cost reduction and prediction accuracy are comparable to the results achieved using Codelet Finder.

	CERE		Codelet Finder
Warmup mode	Working Set	Page Trace	Working Set
Core 2	× 30.5	× 9.9	× 24.7
Atom	× 46.6	× 10.7	× 44.3
Sandy Bridge	× 18.3	× 7.3	× 22.5

Table 6.5: Benchmarking acceleration by replaying only the representatives. CERE replays are 7.3× to 46.6× faster than running the whole NAS.B suite. CERE benchmark acceleration is comparable to the results achieved with Codelet Finder.

	Original (e+11 cycles)	Replay (e+11 cycles)	Error (%)
-O0	2.78	2.88	3.54
-O1	2.33	2.38	2.12
-O2	2.34	2.40	2.25
-O3	2.32	2.42	4.13

Table 6.6: CERE performance predictions for different Clang optimization levels on the FDTD codelet with LLVM `3.3`.

6.5 Compiler Optimizations

This section presents how codelets are used to accelerate the evaluation of different compiler optimizations. First, we focus on an industrial application that contains a single region. We extract and use the codelet version of the region to find better compilation sequences. Second, apply the piecewise tuning methodology described in section 3.6.

6.5.1 Monolithic Tuning

In the previous section, we showed CERE fast thread evaluation through temporal reduction. Yet, CERE can also be used for quick compiler auto-tuning. We showcase CERE auto-tuning capabilities on a Reverse Time Migration (RTM) [107] proto-application on Ivy Bridge.

The FDTD proto-application is dominated by one Jacobi stencil computation that represents 91.1% of the total running time. In the original application it is called 3 000 000 times. CERE is able to extract the Jacobi stencil codelet. Thanks to the temporal reduction algorithm presented in section 4.2, CERE is able to accurately capture the original 3 000 000 invocations behavior using only two representative captures. These captures were performed with the `-O2` optimization level. After accounting for the replay overhead due to warmup and performing four meta-repetition for accuracy, the replay only takes 0.3 seconds. Compared to the original proto-application run time, 71.1 seconds, the replay is 237 times cheaper.

Table 6.6 show that CERE accurately captures the performance when exploring the Clang `3.3` default optimization levels. We see there is a significant performance improvement between the `-O0` and `-O1` optimization levels. This performance jump is accurately captured by CERE replays.

We extend this study and perform a compiler search of 300 passes over the RTM codelet. Section 5.4.4 describes how we selected these compilation passes.

RTM codelet remains at least `200`× faster to evaluate and finds a compiler optimization `1.11`× faster than `-O3`.

CERE fast replay could also be used to evaluate changes in performance

	Original (e+11 cycles)	Replay (e+11 cycles)	Error (%)
LLVM 3.3	2.34	2.40	2.25
LLVM 3.4	2.03	1.92	5.23

Table 6.7: Evaluation of `3.3` and `3.4` LLVM versions on the FDTD codelet using *-O2*.

	compilation passes
opt	-targetlibinfo -no-aa -tbaa -basicaa -globalopt -basiccg -prune-eh -inline-cost -always-inline -simplify-libcalls -lazy-value-info -jump-threading -correlated-propagation -reassociate -domtree -loops -loop-simplify -licm -lazy-value-info -domtree -memdep -gvn -O1
llc	-O1 using SSE instructions

Original (e+11 cycles)	Replay (e+11 cycles)	Error (%)
1.55	1.60	3.31

Table 6.8: LLVM 3.3 best compilation sequence for RTM on Ivy Bridge.

across LLVM compiler versions. The idea is that a codelet captured in LLVM 3.3, can be replayed with a later version of the compiler. Currently in CERE, for this to work the IR must be compatible between the capture and replay LLVM versions. Backwards compatibility of the IR is generally possible between minor LLVM version, but not a strong guarantee of the LLVM project. We captured the FDTD codelet using LLVM 3.3 and replayed it using both LLVM 3.3 and LLVM 3.4. Table 6.7 shows that CERE replay accurately predicts the real execution times achieved when compiling the FDTD proto-application with the two compiler versions.

Finally, table 6.8 presents a custom compiler optimization sequence that we accidentally found for LLVM `3.3`. Executing RTM over LLVM 3.4 default `-O3` achieves a speedup of `1.31`× and which is observed by the RTM codelet. We ensure the validity of the resulting code through the RTM numerical validation. We were not able to translate this compilation sequence to LLVM `3.4` since there are no direct equivalence of the compilation pass `simplify-libcalls` in LLVM `3.4`. We also note the impact of the vectorial instructions on the performance: using AVX instead of SSE with the compilation sequence slowdowns the code with a factor of `0.86`×. The RTM codelet faithfully reproduces this behavior.

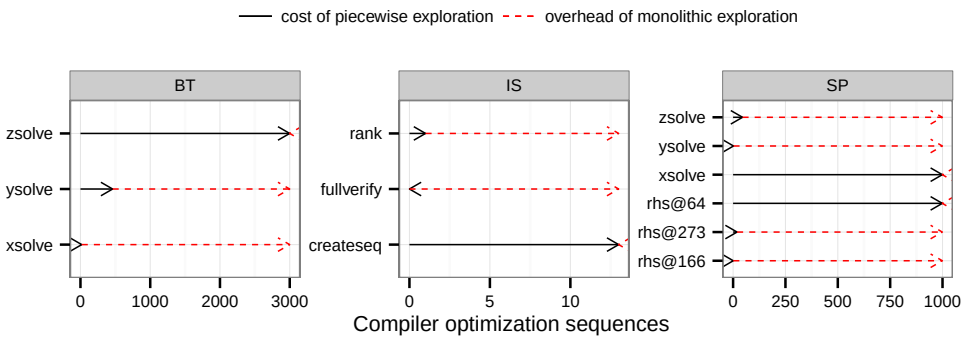


Figure 6.10: Compiler sequences required to get a speedup over `1.04`× per region. CERE evaluates the sequences in the same order for all the regions. Exploring regions separately is cheaper because we stop tuning a region as soon as the speedup is reached.

6.5.2 Piecewise Tuning

Regions within an application may not be sensitive to the same compiler optimizations. SP `rhs` and `zsolve` regions from the motivating example in section 6.2 illustrates this idea as they have different best found compiler optimizations. So, instead of evaluating compiler optimizations on the whole application, we separately evaluate them on each codelet. Unlike monolithic approaches, CERE enables tuning each codelet independently: each region is optimized with the best found compiler sequences.

Until now, we always tune a single parameter which is already challenging and time consuming. In this section, we show a motivating example that demonstrates how costly it is to simultaneously tune multiple parameters. CERE piecewise holistic approach both accelerates the motivating example tuning and outperforms its benefits over standard monolithic approaches. Then, we evaluate the hybrid binaries.

This section details how CERE takes advantage of such scenarios. First, we present the hybrid compilation, and second, we validate the piecewise approach over the NAS SER.

Table 6.9 presents CERE predictions accuracy and reduction factor through compiler optimizations with `3000` compiler sequences for `BT`, `500` for `MG` and `1000` for the others NAS SER. The average CERE prediction accuracy and acceleration for a region is `95.8%` and `20.61`×.

Figure 6.10 presents the number of explored compiler sequences required to find a sequence achieving a speedup over `1.04`× (empirically defined) per region. Unlike monolithic approaches which must continue exploration until all regions are optimized, codelets can stop the search over a region once a satisfying speedup is found and focus the exploration on other regions. Here, CERE evaluates `BT ysolve 461` times instead of `3000` times. Each evaluation is on average `99` times cheaper than a full application run due to

Benchmarks	Regions	Invocations	Accuracy	Benchmark reduction factor	Coverage
BT	xsolve	201	98.65	102.17	25.2
	ysolve	201	98.44	99.54	27.3
	zsolve	201	98.13	98.55	27.1
CG	conjgrad@491	16	99.07	3.02	88
	conjgrad@607	16	94.44	7.65	9.6
FT	appft	1	94.2	4.48	5
	fft3d@152	8	98.18	3.84	27.6
	fft3d@137	8	99.37	4.91	26.8
	fft3d@112	8	97.85	4.2	30.6
	evolve	6	98.94	5.84	7.8
IS	rank	11	91.64	3.91	21.6
	createseq	1	97.94	0.38	60.0
	fullverify	1	97.92	1.24	15.2
SP	xsolve	401	99.71	161.67	11.2
	ysolve	401	99.02	149.29	16.3
	zsolve	401	99.32	152.25	17.4
	rhs@273	402	98.36	154.97	10.8
	rhs@64	402	96.02	159.07	8.5
	rhs@166	402	95.6	159.81	8.9
LU	buts	15500	93.23	8.75	18.2
	jacu	15500	94.02	37.07	14.4
	blts	15500	94.91	8.75	17.8
	jacld	15500	93.45	8.67	15.2
	rhs@166	251	95.22	164.73	8.0
	rhs@64	251	94.97	162.35	8.2
	rhs@273	251	97.26	170.11	6.3
EP	main	1	82.8	0.24	98.4
MG	interp	35	97.37	1.55	8.0
	rprj3	35	96	0.3	6.0
	resid	42	96.5	0.29	48.3
	psinv	18	97.09	0.47	22.0

The **accuracy** of the codelet prediction is based on the relative difference between the original and the replay execution time. The **Benchmark reduction factor** or acceleration is the exploration time saved when studying a codelet instead of the whole application. **Invocations** display the number of times a region is called inside the application. Only regions covering more than 5% of the application execution time are selected.

Table 6.9: Codelet Exploration of Compiler Passes on Ivy Bridge

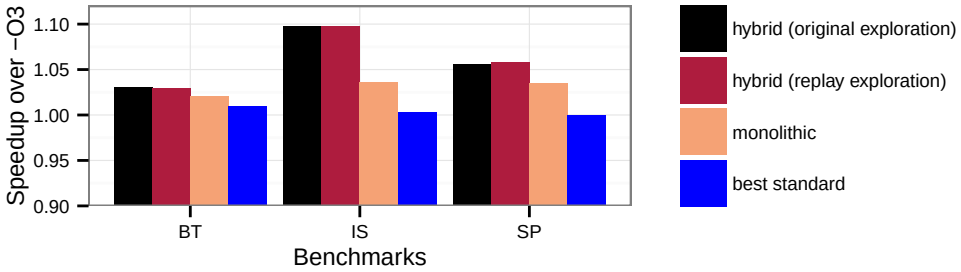


Figure 6.11: Speedups over `-O3`. We only observe speedups from the iterative search over `BT`, `SP`, and `IS`. Best standard is the more efficient default optimization (either `-O1`, `-O2`, or `-O3`). Monolithic is best whole program sequence optimization. Hybrids are build upon optimizations found either with codelets or with original application runs.

the codelet invocations clustering.

The focus of this thesis is not on the compiler flag selection, that is why a naive random compiler pass search was used. Nevertheless, CERE results could be improved with more sophisticated techniques for passes selection such as genetic algorithms [18] which would also benefit from the piecewise approach.

The piecewise search also allows to emphasis the parameter search on a specific region. IS illustrates this idea: it only times a sorting algorithm included in a region which represents `22%` of the application execution time. Through a codelet, CERE extracts the sorting region and can tune it without executing the rest of the application.

CERE outperforms the standard `-O3` over `BT`, `SP`, and `IS` with an average speedup of `1.06`× (see Fig. 6.11). `IS` random generator and sorting algorithm do not benefit from the same optimizations which explains the significant difference between the hybrid and the monolithic approach. Hybrid binaries based on original or replay explorations have the same performances which ensure that we do not miss any optimizations through the codelets.

We make the simplifying assumption that optimizing a region does not affect other regions. This is not always true: due to memory effects, it is possible to have performance interactions between neighbors. We find a compilation sequence which gives a speedup of `x1.08`× over LU `jacu`. Unfortunately, optimizing `jacu` has the side effect of slowing down by `0.92`× the neighboring region `jacld`.

To stress the CERE prediction accuracy model, we performed a simultaneous search of 1000 compiler sequences across the thread affinities on LU `ssor`. CERE predicted region execution time with a mean accuracy of 99% across parameters.

6.6 Discussion

Previous studies [115, 101] already consider a fine grained benchmarking at loops or functions level. The main limitation was the large increase of the design space. We tackle this problem by reducing the search evaluation. Nevertheless, using piecewise tuning raises some questions that we detail bellow.

Like us, Kulkarni and al. [115] propose a piecewise search at the function level granularity. They propose a per-function compilation using the VPO compiler framework. Yet, they do not use any extraction mechanism during the search: exploring two functions within the same file requires to execute the program many times.

Data Sensitivity of Piecewise Tuning

Up to now, all the iterative compiler studies that we presented find the best optimizations through multiple runs of the same data set. A problem of the iterative compilation is to determine the resilience of the optimizations found across other unseen data sets.

Chen and al. [101] perform an iterative compilation over 32 programs. For 14 out of the 32 programs, they achieve 85% of the program-optimal performance using a single data set for training. Also to reach 99% performance on 14 of their benchmarks, they need to use between 154 and 856 data sets. Adding more data sets in the training process avoids the overfitting of the iterative compilation. overfitting is explained in section 2.3.3.

Understanding if a more fine grained optimization increases the chances of overfitting should be considered. Fine grained optimization increases the performance benefits. Chen and al. [101] show that increasing the number of trained data sets increases the overall performance on multiple applications. The fine grained optimization increases the overall performance. So, maybe the the piecewise tuning may require additional data sets to handle the tuning benefits.

Region Dependency Checker

We assume in this thesis that we can apply compiler optimizations to each region separately without impacting the others. Even if we often take advantage of this assumption to produce hybrids, LU proves that it is wrong. Understating if two regions can be separately optimized without impacting each other is a challenging task.

Lets consider two regions A and B and the architectural state e.g. cache state branch predictor state of our test machine. We execute A, N instructions, and then B. We have two leads to determine if two regions are related.

If N, the number of instructions separating the regions is a small number, it is likely that the architectural state is still strongly impacted by A when we reach B. Ans so, changing A impacts the execution context of B. To avoid such scenarios, a first insight is to ensure that N is big enough. This number can be defined according to the machine test cache sizes (we present a similar data size adapting method in section in section 4.5.3).

However, even if two regions have no architectural impact on each other, they can still use the same data sets. For instance, changing A data locality through an optimization will impact B when it reuses this data. To ensure that A and B are independent, they must operate on distinct data sets. We can statically analyze the data sets or dynamically profile the regions: we must execute A and B with different data sets and observe the correlation of their respective performance. If their performance are strongly correlated, we can expect that they use a common data sets.

Acknowledgments

These experiments are the result of a collaboration with Chadi Akel, William Jalby, Yuriy Kashnikov, and Pablo de Oliveira Castro. I also would like to thank Asma Farjallah for providing the RTM proto-application.

6.7 Conclusion

Tuning a system for a set of applications is a costly process which requires benchmarking the applications on the different configurations. We propose to reduce the tuning cost by extracting a set of representative codelets which captures the performance characteristics of the original applications.

Thanks to CERE reduction model, evaluating new architectures is `30`× faster. Similarly, finding the best compiler optimization or thread configuration for a region of code is `30`× faster. These accelerations make costly iterative search techniques such as the iterative compilation accessible in reasonable amount of time. For instance, executing the RTM prot-application codelet only takes 0.3 seconds instead of 71.1 seconds and is reliable to tune diverse compiler optimizations.

CERE not only accelerates the parameter space search but also enables an hybrid compilation which outperforms traditional monolithic compiler tuning. In particular, We achieved `1.11`× speedup over the NAS SER benchmarks.

Chapter 7

Conclusion

Optimizing computer systems is complex and involve three main components working together: the applications, the software stack, and the hardware. Hardware is designed to improve the performance of the current applications while developers write codes that try to take advantage of the hardware. Meanwhile, the system stack tries to reduce the gap between them. However, since hardware and application are both complex and diverse, achieving optimal performance is challenging.

The system stack provides standard configurations that achieve good-enough performance across most of the codes and the existing architectures. A common approach to enhance the performance is system autotuning. It consists into applying successive transformations and evaluating the resulting codes. The huge exploration space combined with the evaluation time required to measure each configuration limits application tuning.

The main limitation of parameter tuning approaches is their high exploration cost. Yet, applications have phases with redundant behaviors. So, current tuning approaches waste resources by evaluating multiple times the same redundant behaviors.

The first contribution of this thesis is a method that takes advantage of these phases to accelerate the tuning process. Instead of running applications, we explore each configuration through codelets. Codelets are pieces of code extracted from the applications hotspots that can be executed as standalone programs. To extract and replay these codelets, we collaboratively design and implement Codelet Extractor and REplayer (CERE), an open source framework presented in chapter 3.

To tune a set of applications, CERE extracts the codelets once. Chapter 4 shows how CERE removes the redundancies by keeping a subset of representative codelets and use them as proxies to quickly evaluate the different configurations. Using codelets as proxies for autotuning requires that codelets faithfully reproduce the application behavior with the exploring parameters. In particular, we show how through a new NUMA ownership strategy, we can explore thread configurations on NUMA architectures.

Codelet acceleration makes affordable a simultaneous exploration of compiler optimizations and thread configurations. We validate in chapters 5 and 6 how CERE faithfully predicts applications execution across compiler optimizations, thread configurations, and architectures. In particular, we predicted the NAS benchmarks execution time with an average accuracy of

94.5%. We also evaluate new micro-architectures or thread scalability on average 27× faster than with whole program executions.

Regions of code are not sensitive to the same optimizations. So exploring each region separately allows to optimize regions at a fine granularity. A second contribution of this thesis is a piecewise holistic tuning approach. In chapter 6, we demonstrate the fine granularity tuning benefits over standard approaches. By tuning regions with their best compiler sequence, piecewise optimization provides speedups up to 1.06× over the best overall configuration.

7.1 Publications

Parts of the work presented in this thesis were published in the following proceedings and journals:

- **Piecewise Holistic Autotuning of Compiler and Runtime Parameters.** Mihail Popov, Chadi Akel, William Jalby, and Pablo de Oliveira Castro. In Euro-Par 2016 Parallel Processing - 22nd International Conference, Lecture Notes in Computer Science. Springer, 2016 (to appear).

- **PCERE: Fine-grained Parallel Benchmark Decomposition for Scalability Prediction.** Mihail Popov, Chadi Akel, Florent Conti, William Jalby, and Pablo de Oliveira Castro. In Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, pages 1151–1160. IEEE, 2015.

- **CERE: LLVM Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization.** Pablo de Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. ACM Transactions on Architecture and Code Optimization (TACO), 12(1):6, 2015.

- **Fine-grained Benchmark Subsetting for System Selection.** Pablo de Oliveira Castro, Yuriy Kashnikov, Chadi Akel, Mihail Popov, and William Jalby. In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, pages 132:132–132:142, New York, NY, USA, 2014. ACM.

7.2 Perspectives

The methodology proposed in this thesis can be improved. It also provides new research clues and challenges for future works.

Better Exploration Strategy

This thesis tunes compiler passes through a naive random pass search. CERE tuning methodology is transverse to the traditional space exploration techniques [18, 19] which focus on how to guide the user over the parameter space. We could improve the piecewise tuning by guiding the optimizations that we explore. Improving the exploration strategy accelerates the tuning by making it quickly reaching better configurations.

Also, instead of targeting the whole optimization space, we can focus on a small subspace. Purini and al. [116] find, through LLVM iterative compilation runs, good general sets of compilation sequences that should work well on any given program. They can quickly tune new applications by directly searching passes within the good set instead of exploring the whole optimization space. Codelets could serve as proxies to quickly find and test these optimal sequences.

Unifying Temporal and Spatial Reduction

As discussed in section 4.5, the codelet subsetting can be improved by combining both spatial and temporal reductions. Related studies [42, 22, 48] detect redundant computation patterns or phases in codes with performance metrics. They take advantage of these redundancies to accelerate the process.

Currently, CERE only clusters invocations from the same region of code. An easy way to accelerate the tuning is to cluster invocations that are not necessarily from the same region. To detect these similar invocations, we can use the same performance metrics as the one used for the spatial clustering presented in section 4.3. Improving the reduction strategy may enhance the codelets prediction accuracy but also accelerate the tuning process.

OpenMP Codelet Prediction Accuracy

To use codelets as proxies for tuning, they must faithfully reproduce the performance of the original regions. So, it is necessary to warmup the system to match as close as possible the original context [76].

In this thesis, we use two warmup strategies: a realistic page tracing [97] and an optimistic working set [72] approaches. The page tracing strategy is more accurate because it replays a memory trace at a page level granularity.

CERE current version only replays OpenMP regions with the optimistic strategy. Despite our efforts, CERE model still miss-predicts some applications execution time in multi threaded environments.

To tackle this issue, we can use the realistic strategy for the OpenMP regions. We have to keep a trace of the pages touched by each thread. At replay, each thread has touch its recored pages.

Extend Hybridization to Thread configurations or Architectures

In this thesis, we only use hybridization for compiler optimization. Nevertheless, we can apply this method on other parameters such as thread configurations.

SP from the motivating example in section 6.2 illustrates this idea. SP regions `rhs` and `xsolve` do not benefit from the same thread placements. With codelets, we can determine the best thread configuration for each region, and change the affinities during the execution.

Similarly, we can detect the best architecture for each piece of code. CERE can be useful to detect the best core for each region of code on heterogeneous architectures.

However, unlike most of the compiler optimizations, changing the thread configurations raises an execution time overhead [83] that needs to be compensated. Similarly, migrating data to an other core also raises an overheard. So, to apply a piecewise tuning for architectures or thread configurations we need to ensure that the overhead of moving to a new system is compensated by the speedup it provides.

Use Codelets to Guide Architecture Design

Current benchmark suites are guiding the evolutions of the future hardware [1]. CERE automatically extracts codelets and reduces the benchmark suites while preserving their diversities. We could replace the original benchmark suites with codelets [30]. Codelet simulation is faster because of the reduction strategy and will allow designers to evaluate more architectural trade-offs.

Codelets can also be applied to find or evaluate new *dwarfs* [1]. Berkeley Dwarfs are algorithmic methods that capture patterns of computation and communication which are representative of the real world applications. Dwarfs purpose is to represent the application requirements without overlooking specific optimizations for some hardware platforms. CERE similar computation patterns detection can be used to ease new dwarfs integration or to detect similar dwarfs.

Energy Prediction

We only use codelets to tune applications execution time. Prediction can be extended to other performance metrics such as energy consumption or memory behaviors. Similarly to Host et al. [19], we could apply a Pareto frontier with a fine grained compiler tuning over execution time and energy consumption.

The possible issue is that we may have to train a new set of metrics to handle the energy.

Spatial Reduction for Compilers or Threads

We use Genetic Algorithms [59] (GAs) to train the spatial reduction metrics for architecture selection. Architecture selection metrics gather together codes with similar computation patterns. An architectural change impacts in the same way codes with the same computation patterns.

We suppose that compiler optimization or thread configuration changes may also have the same impact on codes with the same computation patterns. Fursin et al. [20] take advantage of this idea for compiler optimizations: they cluster applications and expect that applications in the same cluster are sensitive to the same compiler optimizations. We can extend this idea, by clustering codelets instead of applications.

Reducing the granularity should improve the detection of the computation patterns and enhance the overall clustering quality [98]. Let us consider an application with two small regions, respectively memory and compute bound. An overall characterization study may conclude that the application is balanced because of the regions compensation. A fine grained study on the other side is able to detect the distinct behaviors.

We can already evaluate this approach with our methodology. Cavazos et al. [117] use hardware performance metrics to chose the best compiler optimization. It is interesting to see if the performance metrics that we use for architecture selection are also relevant to gather codes that are impacted in the same way by compiler optimizations or thread configurations.

Bibliography

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[2] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[3] D. Bailey *et al.*, "The NAS parallel benchmarks summary and preliminary results," in *Proceedings of the conference on Supercomputing*. ACM/IEEE, 1991, pp. 158–165.

[4] K. Hoste, "Analysis, estimation and optimization of computer system performance using machine learning." Ph.D. dissertation, Ghent University, 2010.

[5] C. Bienia and K. Li, *Benchmarking modern multiprocessors*. Princeton University New York, 2011.

[6] J. J. Yi, H. Vandierendonck, L. Eeckhout, and D. J. Lilja, "The exigency of benchmark and compiler drift: designing tomorrow's processors with yesterday's tools," in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 75–86.

[7] K. C. Barr, "Summarizing multiprocessor program execution with versatile, microarchitecture-independent snapshots," Ph.D. dissertation, Massachusetts Institute of Technology, 2006.

[8] S. Hong and H. Kim, "An integrated gpu power and performance model," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 280–289.

[9] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1. ACM, 2004, pp. 2–13.

[10] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. O'Boyle, G. Fursin, and O. Temam, "Automatic performance model construction for the fast software exploration of new hardware designs," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, pp. 24–34.

[11] E. Petit, P. de Oliveira Castro, T. Menour, B. Krammer, and W. Jalby, "Computing-kernels performance prediction using dataflow analysis and microbenchmarking," in *International Workshop on Compilers for Parallel Computers*, 2012.

[12] C. Haine, O. Aumage, E. Petit, and D. Barthou, "Exploring and evaluating array layout restructuration for SIMDization," in *Proceedings of the 27th international conference on Languages and Compilers for Parallel Computing*, ser. LCPC'14, (to appear) 2014.

[13] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on.* IEEE, 2001, pp. 3–14.

[14] G. De Michell and R. K. Gupta, "Hardware/software co-design," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349–365, 1997.

[15] C. Dubach, T. M. Jones, and M. F. O'Boyle, "Exploring and predicting the architecture/optimising compiler co-design space," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems.* ACM, 2008, pp. 31–40.

[16] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing.* IEEE Press, 2008, p. 4.

[17] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International symposium on Code Generation and Optimization.* IEEE, 2004, pp. 75–86.

[18] S. R. Ladd, "Acovea: Analysis of compiler options via evolutionary algorithm," 2007.

[19] K. Hoste and L. Eeckhout, "Cole: compiler optimization level exploration," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization.* ACM, 2008, pp. 165–174.

[20] G. Fursin *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *International Journal of Parallel Programming*, vol. 39, no. 3, pp. 296–327, 2011.

[21] J. Y. Joshua, R. Sendag, L. Eeckhout, A. Joshi, D. J. Lilja, and L. K. John, "Evaluating benchmark subsetting approaches," in *2006 IEEE International Symposium on Workload Characterization*. IEEE, 2006, pp. 93–104.

[22] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 412–423.

[23] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[24] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1. ACM, 2003, pp. 318–319.

[25] L. Eeckhout, J. Sampson, and B. Calder, "Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE, 2005, pp. 2–12.

[26] Y.-J. Lee and M. Hall, "A code isolator: Isolating code fragments from large programs," in *Languages and Compilers for High Performance Computing*. Springer, 2005, pp. 164–178.

[27] P. de Oliveira Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, "CERE: LLVM Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization," *Transactions on Architecture and Code Optimization*, vol. 12, no. 1, p. 6, 2015.

[28] M. Popov, C. Akel, F. Conti, W. Jalby, and P. de Oliveira Castro, "Pcere: Fine-grained parallel benchmark decomposition for scalability prediction," in *International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1151–1160.

[29] M. Popov, C. Akel, W. Jalby, and P. d. O. Castro, "Piecewise holistic autotuning of compiler and runtime parameters," in *Euro-Par 2016 Parallel Processing - 22nd International Conference*, ser. Lecture Notes in Computer Science. Springer, 2016 (to appear).

[30] P. de Oliveira Castro, Y. Kashnikov, C. Akel, M. Popov, and W. Jalby, "Fine-grained Benchmark Subsetting for System Selection," in *International symposium on Code Generation and Optimization*. ACM, 2014, pp. 132–142.

[31] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, "Using papi for hardware performance monitoring on linux systems," in *Proc. Conf. on Linux Clusters: The HPC Revolution*, 2001, pp. 25–27.

[32] A. S. Charif-Rubial, E. Oseret, J. Noudohouenou, W. Jalby, and G. Lartigue, "Cqa: A code quality analyzer tool at binary level," in *High Performance Computing (HiPC), 2014 21st International Conference on*. IEEE, 2014, pp. 1–10.

[33] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, no. 4, pp. 64–71, 2002.

[34] S. C. Johnson, *Lint, a C program checker*. Citeseer, 1977.

[35] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, W. Jalby *et al.*, "Maqao: Modular assembler quality analyzer and optimizer for itanium 2," in *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, 2005.

[36] "Intel Architecture Code Analyzer." [Online]. Available: https://software.intel.com/en-us/articles/intel-architecture-code-analyzer/

[37] "Clang Static Analyzer." [Online]. Available: http://clang-analyzer.llvm.org/

[38] "Lprof ," http://www.vi-hps.org/upload/material/tw21/MAQAO.pdf.

[39] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010, pp. 207–216.

[40] A. S. Phansalkar, *Measuring program similarity for efficient benchmarking and performance analysis of computer systems*. ProQuest, 2007.

[41] K. Hoste and L. Eeckhout, "Comparing benchmarks using key microarchitecture-independent characteristics," in *Workload Characterization, 2006 IEEE International Symposium on*. IEEE, 2006, pp. 83–92.

[42] ——, "Microarchitecture-independent workload characterization," *Micro, IEEE*, vol. 27, no. 3, pp. 63–72, 2007.

[43] V. Palomares, "Combining static and dynamic approaches to model loop performance in hpc," Ph.D. dissertation, Université de Versailles-Saint Quentin en Yvelines, 2015.

[44] Z. Bendifallah, "Generalization of the decremental performance analysis to differential analysis," Ph.D. dissertation, Université de Versailles-Saint Quentin en Yvelines, 2015.

[45] H. Vandierendonck and K. De Bosschere, "Experiments with subsetting benchmark suites," in *Workload Characterization, 2004. WWC-7. 2004 IEEE International Workshop on.* IEEE, 2004, pp. 55–62.

[46] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John, "Measuring benchmark similarity using inherent program characteristics," *Computers, IEEE Transactions on*, vol. 55, no. 6, pp. 769–782, 2006.

[47] H. Vandierendonck and K. De Bosschere, "Many benchmarks stress the same bottlenecks," in *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, 2004.

[48] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5. ACM, 2002, pp. 45–57.

[49] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA., 1967, pp. 281–297.

[50] G. Hamerly and C. Elkan, "Alternatives to the k-means algorithm that find better clusterings," in *Proceedings of the eleventh international conference on Information and knowledge management.* ACM, 2002, pp. 600–607.

[51] J. H. Ward, "Hierarchical grouping to optimize an objective function," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236–244, 1963. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/01621459.1963.10500845

[52] M. Kaur and U. Kaur, "Comparison between k-mean and hierarchical algorithm using query redirection," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 7, 2013.

[53] R. Thorndike, "Who belongs in the family?" *Psychometrika*, vol. 18, no. 4, pp. 267–276, 1953. [Online]. Available: http://dx.doi.org/10.1007/BF02289263

[54] I. Jolliffe, *Principal component analysis.* Wiley Online Library, 2002.

[55] A. Hyvärinen, J. Hurri, and P. O. Hoyer, *Natural Image Statistics: A Probabilistic Approach to Early Computational Vision.* Springer Science & Business Media, 2009, vol. 39.

[56] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *European Conference on Parallel Processing*. Springer, 2005, pp. 196–205.

[57] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *SIGPLAN Notices*, vol. 34, no. 7. ACM, 1999, pp. 1–9.

[58] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 2006, pp. 114–122.

[59] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[60] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical recipes: The art of scientific computing*. Cambridge university press, 1986.

[61] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 47–56.

[62] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, "Barrierpoint: Sampled simulation of multi-threaded applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[63] T. Lafage and A. Seznec, "Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream," in *Workload characterization of emerging computer applications*. Springer, 2001, pp. 145–163.

[64] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[65] M. Van Biesbrouck, T. Sherwood, and B. Calder, "A co-phase matrix to guide simultaneous multithreading simulation," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE, 2004, pp. 45–56.

[66] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong, "Detecting phases in parallel applications on shared mem-

ory architectures," in *International Parallel and Distributed Processing Symposium*. IEEE, 2006, p. 10.

[67] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 2–12.

[68] E. K. Ardestani and J. Renau, "Esesc: A fast multicore simulator using time-based sampling," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 448–459.

[69] J. Gonzalez, J. Gimenez, and J. Labarta, "Automatic detection of parallel applications computation phases," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–11.

[70] D. E. Knuth, "An empirical study of Fortran programs," *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, 1971.

[71] C. Akel, Y. Kashnikov, P. de Oliveira Castro, and W. Jalby, "Is Source-code Isolation Viable for Performance Characterization?" in *International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*. IEEE Computer Society, 2013.

[72] E. Petit, G. Papaure, F. Bodin *et al.*, "Astex: a hot path based thread extractor for distributed memory system on a chip," in *Proceedings of Compilers for Parallel Computers workshop (CPC2006)*, 2006.

[73] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas, "Effective source-to-source outlining to support whole program empirical optimization," in *Languages and Compilers for Parallel Computing*. Springer, 2010, pp. 308–322.

[74] CAPS entreprises. (CAPS) Codelet finder. [Online]. Available: http://www.caps-entreprise.com/

[75] Y. Kashnikov, P. de Oliveira Castro, E. Oseret, and W. Jalby, "Evaluating architecture and compiler design through static loop analysis," in *High Performance Computing and Simulation (HPCS)*. IEEE, 2013, pp. 535–544.

[76] R. E. Kessler, M. D. Hill, and D. A. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches," *Transactions on Computers*, vol. 43, no. 6, pp. 664–675, 1994.

[77] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in *Computer Design: VLSI in Computers and Processors, 1996. ICCD'96. Proceedings., 1996 IEEE International Conference on.* IEEE, 1996, pp. 468–477.

[78] J. W. Haskins Jr and K. Skadron, "Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation," in *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on.* IEEE, 2003, pp. 195–203.

[79] X. Gao, M. Laurenzano, B. Simon, and A. Snavely, "Reducing overheads for acquiring dynamic memory traces," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International.* IEEE, 2005, pp. 46–55.

[80] E. Petit and F. Bodin, "Code-Partitioning for a Concise Characterization of Programs for Decoupled Code Tuning," Mar. 2010. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00460897

[81] D. Quinlan and C. Liao, "The rose source-to-source compiler infrastructure," in *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*, vol. 2011, 2011, p. 1.

[82] T. Kisuki, P. M. Knijnenburg, M. F. O'Boyle, F. Bodin, and H. A. Wijshoff, "A feasibility study in iterative compilation," in *High Performance Computing.* Springer, 1999, pp. 121–132.

[83] A. Mazouz, S.-A.-A. Touati, and D. Barthou, "Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures," in *High Performance Computing and Simulation (HPCS).* IEEE, 2011, pp. 273–279.

[84] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio: making dvs practical for complex hpc applications," in *Proceedings of the conference on Supercomputing.* ACM/IEEE, 2009, pp. 460–469.

[85] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on.* IEEE, 2003, pp. 204–215.

[86] P. de Oliveira Castro, E. Petit, A. Farjallah, and W. Jalby, "Adaptive sampling for performance characterization of application kernels," *Concurrency and Computation: Practice and Experience*, 2013.

[87] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam, "Quick and practical run-time evaluation of multiple program optimizations," *T. HiPEAC*, vol. 1, pp. 34–53, 2007.

[88] D. Sands, "Reimplementing llvm-gcc as a gcc plugin," in *Third Annual LLVM Developers' Meeting*, 2009.

[89] A. Alexandrescu, *The D Programming Language*. Pearson Education, 2010.

[90] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "Openuh: An optimizing, portable openmp compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, 2007.

[91] A. B. . A. Bataev, "Towards OpenMP Support in LLVM," in *2013 European LLVM Conference*, 2013.

[92] "LLVM OpenMP runtime," https://www.openmprtl.org/.

[93] "Google Performance Tools," gperftools v2.2.1. [Online]. Available: http://code.google.com/p/gperftools

[94] Z. Pan and R. Eigenmann, "Fast, automatic, procedure-level performance tuning," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 2006, pp. 173–181.

[95] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergström, "Kernelgen–the design and implementation of a next generation compiler platform for accelerating numerical models on gpus," Technical Report 2013/02, University of Lugano, July 2013. http://www. old. inf. usi. ch/file/pub/75/tech_report2013. pdf, Tech. Rep., 2013.

[96] J. Duell, "The design and implementation of Berkeley lab's linux checkpoint/restart," *Lawrence Berkeley National Laboratory*, 2005.

[97] A. N. Burton and P. H. Kelly, "Performance prediction of paging workloads using lightweight tracing," *Future Generation Computer Systems*, vol. 22, no. 7, pp. 784–793, 2006.

[98] M. Arenaz, J. Touriño, and R. Doallo, "Xark: An extensible framework for automatic recognition of computational kernels," *Transactions on Programming Languages and Systems*, vol. 30, no. 6, p. 32, 2008.

[99] M. A. Khan, H.-P. Charles, and D. Barthou, "An effective automated approach to specialization of code," in *Languages and Compilers for Parallel Computing*. Springer, 2008, pp. 308–322.

[100] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on.* IEEE, 1997, pp. 259–269.

[101] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu, "Evaluating iterative optimization across 1000 data sets," in *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10)*, Toronto, Canada, 2010.

[102] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis.* John Wiley & Sons, 2009, vol. 344.

[103] J. Noudohouenou, "Performance prediction based on codelet driven application characterization," Ph.D. dissertation, 2013.

[104] E. Willighagen, "GNU R package 'genalg'," 2013. [Online]. Available: http://cran.r-project.org/web/packages/genalg/

[105] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference.* ACM, 1967, pp. 483–485.

[106] L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Proceedings of the conference on Supercomputing.* ACM/IEEE, 2005, pp. 40–40.

[107] E. Baysal, "Reverse time migration," *Geophysics*, vol. 48, no. 11, p. 1514, Nov. 1983.

[108] M. Popov, "NAS 3.0 C OpenMP," http://benchmark-subsetting.github.io/cNPB.

[109] D. H. Bailey, *Nas parallel benchmarks.* Springer, 2011.

[110] "Omni Compiler Project Benchmarks." [Online]. Available: http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/

[111] "LLVM implicit barrier issue report," https://github.com/clang-omp/clang/issues/52, accessed: 2015-01-01.

[112] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[113] S. N. Natarajan, B. Swamy, A. Seznec *et al.*, "Modeling multi-threaded programs execution time in the many-core era," 2013.

[114] Intel, "Reference Guide for the Intel(R) C++ Compiler 15.0," https://software.intel.com/en-us/node/522691.

[115] P. A. Kulkarni, M. R. Jantz, and D. B. Whalley, "Improving both the performance benefits and speed of optimization phase sequence searches," in *SIGPLAN Notices*, vol. 45, no. 4. ACM, 2010, pp. 95–104.

[116] S. Purini and L. Jain, "Finding good optimization sequences covering program space," *Transactions on Architecture and Code Optimization*, vol. 9, no. 4, p. 56, 2013.

[117] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 2007, pp. 185–197.