Unifying hardware and software benchmarking: a resource-agnostic model

Davide Morelli

University of Pisa
Computer Science Department

June 2015

Ph.D. thesis

Abstract

Lilja (2005) states that “In the field of computer science and engineering there is surprisingly little agreement on how to measure something as fundamental as the performance of a computer system.”. The field lacks of the most fundamental element for sharing measures and results: an appropriate metric to express performance.

Since the introduction of laptops and mobile devices, there has been a strong research focus towards the energy efficiency of hardware. Many papers, both from academia and industrial research labs, focus on methods and ideas to lower power consumption in order to lengthen the battery life of portable device components. Much less effort has been spent on defining the responsibility of software in the overall computational system energy consumption. Some attempts have been made to describe the energy behaviour of software, but none of them abstract from the physical machine where the measurements were taken. In our opinion this is a strong drawback because results can not be generalized.

In this work we attempt to bridge the gap between characterization and prediction, of both hardware and software, of performance and energy, in a single unified model. We propose a model designed to be as simple as possible, generic enough to be abstract from the specific resource being described or predicted (applying to both time, memory and energy), but also concrete and practical, allowing useful and precise performance and energy predictions. The model applies to the broadest set of resource possible. We focus mainly on time and memory (hence bridging hardware benchmarking and classical algorithms time complexity), and energy consumption. To ensure a wide applicability of the model in real world scenario, the model is completely black-box, it does not require any information about the source code of the program, and only relies on external metrics, like completion time, energy consumption, or performance counters.

Extending the benchmarking model, we define the notion of *experimental computational complexity*, as the characterization of how the resource usage

changes as the input size grows.

Finally, we define a high-level energy model capable of characterizing the power consumption of computers and clusters, in terms of the usage of resources as defined by our benchmarking model.

We tested our model in four experiments:

Expressiveness we show the close relationship between energy and classical theoretical complexity, also showing that our experimental computational complexity is expressive enough to capture interesting behaviour of programs simply analysing their resource usage.

Performance prediction we use the large database of performance measures available on the CPU SPEC website to train our model and predict the performance of the CPU SPEC suite on randomly selected computers.

Energy profiling we tested our model to characterize and predict the power usage of a cluster running OpenFOAM, changing the number of active nodes and cores.

Scheduling on heterogeneous systems applying our performance prediction model to features of programs extracted at runtime, we predict the device where is most convenient to execute the programs, in an heterogeneous system.

Acknowledgements

I want to thank my supervisor Antonio Cisternino, for the constant insight, for ensuring I was following the right path in my research, and for his friendship.

The advice from Andrea Canciani has been extremely important to assess the methodology in general, and to develop the energy model in particular.

Leonardo Bartoloni has helped me understanding the probabilistic framework underlying a model based on regression analysis.

The last experiment reported in this thesis has been developed with Gabriele Cocco, using his Ph.D. research we created a scheduling methodology for heterogeneous platforms.

I am grateful for the support and constant affect I received from my family: my parents, my grandparents, and my sister, who supported and encouraged me throughout this journey.

I wish to thank my wife for sharing her life with me, constantly pushing me to pursue my dreams.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Benchmarking: art or science

1.1 Introduction

Benchmarking is the analysis of the performance of *computer systems*. A *system* is "any collection of hardware, software and firmware" (Jain, 1991). Performance is measured using *metrics*. A *metric* is "the criteria used to evaluate the performance of the system components" (Jain, 1991). Performance analysis is therefore "a combination of *measurement*, *interpretation*, and *communication* of a computer system's *speed* or *size*" (Lilja, 2005). However, today's approach to *measurement* and *interpretation* is not systematic, it changes from researcher to researcher, sometimes with little apparent common ground.

Communication of results is a crucial aspect, that in our opinion has determined the lack of development of a mathematically sound benchmarking model. One of the main purposes of measuring the performance of a computer is to let the readers of the benchmark results compare that computer with alternatives (Lilja, 2005). Therefore, the performance metric needs to be simple to read. Traditionally this has led to choosing single numbers as performance measures (Mashey, 2004). In this work we demonstrate the limits of any approach based on a single dimension metric, that inherently leads to a large characterization error.

There is surprisingly little consensus on the basic elements of computer system performance analysis: the performance metric. Several performance metrics have been used, yet the debate on which metric should be used is still not settled. This confusion is well represented by the "megahertz myth" [1], a widely spread misconception about the performance of Apple comput-

[1] `http://www.theguardian.com/technology/2002/feb/28/onlinesupplement3`

ers compared to IBM PC computers. The origin of the myth resides in the difference between the RISC (Apple) and the CISC (PC) architectures, that results in a huge difference in the clock rates. However, clock rate is misleading, as the effective performance of a computer depends on the effective time needed to complete a task. Using clock rate as a performance metric is improper, because of the profound difference in how Reduced Instruction Set Computing architectures (RISC) and Complex Instruction Set Computing architectures (CISC) work.

Several benchmark suites have been been used to measure different aspects of the computational systems. Some based on reduced version of scientific software (e.g. *Livermore Fortran kernels*, *NAS kernels*, *LINPACK*), some based on synthetic programs (e.g. *Whetstone* and *Dhrystone*), other focusing on specific aspects of hardware (e.g. I/O benchmarks).

Given the abundance of performance metrics and benchmarks available, computer manufacturers can usually find a particular combination that will show that their system outperforms the competitors. Sometimes explicitly exploiting deficiencies of benchmarks [2], if not simply cheating [3].

In section 4.1 we analyse the most used metrics and benchmarks.

Measuring performance of a system using standardized benchmark suites is risky. Computer manufacturers are known to tune their systems to perform well on the reduced set of programs contained in the benchmarks suites. The actual performance of the system in real applications could not reflect the performance scores achieved with benchmark suites.

Moreover, there is not a clear mathematical set for the practice of analysing performance. The "war of the means" (Mashey, 2004) is a clear example of this lack of strict mathematical foundation in the field: the improper use of arithmetic or geometric mean to summarize data can lead to misleading results.

[2] "ATI cheating benchmarks and degrading game quality, says NVIDIA", available at `http://www.pcauthority.com.au/Feature/232215,ati-cheating-benchmarks-and-degrading-game-quality-says-nvidia.aspx`, last accessed 06/06/2015

[3] "Futuremark confirms nVidia is cheating in benchmark", available at `http://www.geek.com/games/futuremark-confirms-nvidia-is-cheating-in-benchmark-553361/`, last accessed 06/06/2015; "Nvidia Points Finger at AMD's Image Quality Cheat", available at `http://www.tomshardware.com/news/radeon-catalyst-image-quality-benchmark,11701.html`, last accessed 06/06/2015; "Intel graphics drivers employ questionable 3DMark Vantage optimizations", available at `http://techreport.com/review/17732/intel-graphics-drivers-employ-questionable-3dmark-vantage-optimizations`, last accessed 06/06/2015

The analysis of the performance of hardware and the analysis of performance of software have always been developed as separated models. Inarguably, they concur in equal measure to the overall performance of computational systems. We believe that only an approach that takes them into account at the same time can capture their relationship.

1.2 Benchmarking is pre-science

Kuhn (2012) proposes a model for scientific evolution. In his view, scientific progress in a field can be described by cyclic occurrences of three phases:

pre-science : when the field does not have a widely accepted and clear paradigm that explains the phenomena. One or several paradigms could be already present and discussed, but none is dominating the field yet;

normal science : when a paradigm has emerged over the competitors and is dominating the field. In this phase the scientific pursue does not challenge the dominating paradigm, and performs incremental research, extending the reach of the model, and clarifying its relationship with other phenomena;

revolution : when evidence from experiments challenges the dominating paradigm. The scientific community fragments and recognises the crisis. New paradigms are proposed.

We believe benchmarking to be in the *pre-science* phase, as it lacks of an accurate, simple, with broad scope, unifying paradigm. Lilja (2005) states that "In the field of computer science and engineering [...] there is surprisingly little agreement on how to measure something as fundamental as the performance of a computer system.". The field lacks of the most fundamental element for sharing measures and results: an appropriate metric to express performance.

Kuhn (1977) defines 5 criteria that should be used to determine the choice of a theory:

Accuracy the theory needs to explain and predict phenomena with empirically adequate experimentation and observation.

Consistency the theory needs to be consistent with itself, as well as with the other accepted theories

Scope the consequences of the theory should go beyond the immediate scope of the laws and observations it is designed to explain.

Simplicity the theory should offer a simple explanation of phenomena, similar to Occam's razor

Fruitfulness the theory should "disclose new phenomena or previously unnoted relationships among those already known" Kuhn (1977).

Colyvan (2001) lists similar requisites for a good scientific theory: is elegant; is as simple as possible; explains the observations; allows predictions that can falsify the model; is *fruitful* in disclosing potential for future work.

In our opinion, the benchmarking practice does not comply to all of those requisites. Characterisation models usually have a very narrow scope, focusing on particular details. This might produce accurate models, but their utility will not go beyond the setting that they describe. As we will see in chapter 4, in many cases the models are very detailed, crafted around the specific hardware that is being modelled, making it difficult for other researchers to apply the same approach to different settings (*scope* criterion). In other cases complex models with non-linear projection on high dimensional feature space are used to characterize performance (e.g. Support Vector Regression), without reporting the improvement over simpler linear approaches. This goes against the *simplicity* criterion

Popper empathises the "falsifiability" of a theory as a requisite for that theory to be science (Popper, 2014). Lilja (2005) lists "repeatability" and "easiness of measurement" as two of the criteria that a good analysis should have. For a model to be falsifiable, it needs to be high level enough to be replicable on different hardware than the one used to define it. To avoid over-fitting, its scope should not be too narrow. Interestingly, there is no general setting for hardware and software characterization in literature, with every model focusing only on particular aspects of computing systems.

For all this reasons, despite being a practice with decades of history, in our opinion benchmarking should be considered more *art* than *science*. Our work is an attempt to identify the issues in the field and to propose a mathematically solid abstract model, capable of ensuring a common ground to the benchmarking activities. Our hope is that it might help future researchers to finally find a paradigm capable of being accepted as what Kuhn would call "normal science".

Chapter 2

Lack of a unified model

In science, a *model* is a simplified version of reality, therefore inherently false, but still useful (Box and Draper, 1987). Models are false in the sense that, being a simplification of reality, they necessarily omit some detail and offer a *surrogate* of the phenomenon analysed. The very process of simplification is the interesting aspect of model generation. The choice of the abstraction level, hence which aspect of reality should be included in the model and which should be excluded, is the choice that gives the model the potential do be useful. Useful models help understand phenomena, reducing their complexity to simpler terms that can be interpreted and manipulated.

The activities involved in modelling a phenomenon are:

- the recognition of a structure in the phenomenon. Structure emerges from patterns in observations;
- the interaction of several entities in a single system. The identification of the actors involved in the phenomenon is fundamental for the isolation of the factors to be modelled;
- the generation of a model. Is the formalization of the principles and the formal systems that explain reality. Models contain assumptions, that justify the proposed synthesis of the phenomenon;
- the validation of the model. Falsifiability is a requisite of any theory to be considered scientific (Popper, 2014). Therefore, the model should therefore be easily repeatable and easy to measure (Lilja, 2005).

As discussed in the previous chapter, several authors Colyvan (2001), Kuhn (1977), and Hawking (2011) agree that good theories should: offer elegant models; be be as *simple* as possible, because simple explanations should be preferred over complex ones; they should be consistent with the observations, showing *explanatory power*; and they should be able to create accurate predictions of the modelled phenomena. Kuhn (1977) and Colyvan (2001) also add that a good theory should be *fruitful*, i.e. have the potential to be seminal for future work.

It is our opinion that the current state of the benchmarking field lacks of a convincing model that embodies these characteristics. Researchers and commercial players have focused on characterizing very specific aspects of computational systems, generating a large amount of models that well describe particular aspects, but failing to offer an holistic description of reality. The strong inter-dependency of hardware and software has not been exploited by models that usually focus only on one of those two entities. No model offers characterization of both aspects at the same time. When the performance of a system is analysed, specific models for the considered *computational resource* are produced. We could not find models capable of describing the general and abstract behaviour of software running on hardware, with respect to generic *resource consumption*, including completion time, energy consumption, or performance counters.

2.1 Time and energy

Completion time and energy consumption are closely related. Energy is equal to the completion time multiplied by the average power absorption: $E = PT$. They are proportional, but the information about completion time is not enough to derive energy, and vice versa. The power absorption of a computing system could change depending on the nature of the computation, e.g. the level of active cores in a parallel algorithm. Because of their close relationship, it makes sense to create a benchmarking model that allows us to use both quantities to characterize machines and programs.

Several models have been proposed to predict the power consumption of hardware, the energy consumption of programs, or the completion time of programs, always separately. Because each of these models focus on a different aspect, they are profoundly different. The information contained in the relationship between time and energy is never exploited.

In this work we attempt to design a benchmarking approach that uses several different metrics at the same time, and is capable of characterizing and predicting both completion time and energy consumption in a single model.

2.2 Hardware and Software

Computing systems are made by hardware and software. The performance of a system obviously depends on both factors. Traditionally they have been characterized separately.

Hardware benchmarking is usually done using standard suites like SPEC. They try to assign a score to the evaluated system. Necessarily, the performance description of the hardware is limited to the tasks included in the benchmark suite. When using the same hardware to execute a different program, interactions between instructions and architecture, not present in the benchmarks used to characterize the hardware, could arise. Therefore, the performance analysis could not be accurate when changing the workload.

Software characterization models usually profile the execution by instrumentation or simulation. They usually assume a specific hardware, and they model the time needed to complete every operation on that specific hardware. The model of the software created for a specific architecture does not apply to different architectures.

Imagine a program that makes use of a particular memory access pattern, that has good performance on the hardware being used to profile the program. Imagine now running the same program on a different machine, with a faster CPU, but with a different memory topology. The second machine might generally be considered better than the first, but the program might have unexpected poor performance on it, because of the not favourable memory access pattern. Profiling the program only on the first machine, this issue could not reveal itself in the characterization phase, but only when the model is applied to a different machine. In general, unless the model is designed to allow measures coming from multiple machines, unexpected behaviour, like the one previously described example, could occur. The same is true when characterizing a machine: if the benchmark suite used to analyse it does not contain a relevant software behaviour, it will be missed in the analysis, that at least in some cases will not be representative of the effective performance of the machine.

More generally, the combination of particular hardware features with particular combinations of instructions will reveal interesting behaviour of programs, that will have unexpected consequences in real programs. We believe that designing the benchmarking model to characterize both software and hardware simultaneously will use the peculiar traits of programs to create good characterizations, instead of suffering from them.

Surprisingly, even though is obvious that a model designed to characterize both hardware and software simultaneously would be able to exploit the interaction between them, we only found a small number of models designed to characterize both aspects of computation simultaneously (the most relevant are from Kuperberg et al. (2008), Kuperberg et al. (2008), and Saavedra and Smith (1996)). The most promising research in this direction is from Saavedra and Smith (1996), where a machine and a program are characterized in the same model. We decided to design a simple approach where hardware characterization is the dual of software characterization, allowing an arbitrary number of programs and machines to be used, to ensure that the model will capture interesting patterns that may be visible only when the same program is measured on different computers, or when the computer is tested with different programs.

2.3 Computational energy models

Energy consumption has become of primary importance in IT computing. In High Performance Computing (HPC), the goal of achieving maximum performance has traditionally led to neglect energy efficiency, but in the recent years initiatives like the Green500 [1] witness that the performance-at-any-cost paradigm is no longer feasible (Hemmert, 2010; Kindratenko and Trancoso, 2011). Part of the problem is related to hardware architecture optimization and dealt by manufacturers; however, the runtime behaviour of executed programs can help reducing the overall energy consumption (Yao et al., 1995; Shin and Choi, 1999). Moreover, the existence of time slots in which the completion time of a computation can be considered equivalent allows for interesting optimization problems in which energy consumption can be adjusted in that time slot. For example, typical Computer Fluid Dynamics jobs require long time to complete and often a variance of hours in the completion time may be acceptable due to organizational procedures (i.e. week ends, night time etc.).

Resource accounting is a major challenge for the management of cloud environments (Sekar and Maniatis, 2011; Lindner et al., 2010), especially in the billing model that is currently most diffused: pay-as-you-go, where the customers are charged for the resources actually used by their processes. Energy is one of the most important expenses in a cloud environment, therefore measuring the energy consumed by each client is particularly valuable (Buyya et al., 2010; Kim et al., 2014, 2011).

In literature we can find several attempts to model the energy consumption of tasks, in some cases parallel tasks. However, not many models at-

[1] `http://www.green500.org`

tempt to characterize parallel programs running at the same time on the same machine, or cluster.

In this work we propose a simple high level energy model for concurrent parallel tasks, running on the same computational environment, designed to describe tasks running on clusters. With our model the job scheduler can retain the ability to precisely estimate the energy consumption of every single job. Moreover, the instantaneous power prediction can be used to limit the power usage of a cluster or of a datacenter, to avoid overloading the power distribution system.

Chapter 3

Contributions of this work

3.1 Contributions

Our contributions to the field are:

- The formalization of a simple and high level approach to benchmarking.
  - The proposed model can characterize and predict generic resource consumption of *programs* running on *computational environments.*
  - The model is unified: it characterizes hardware and software using the same approach; it can accommodate experiments and measures coming from different architectures in the same model; it can accommodate measure from heterogeneous *computational resources*, e.g. completion time, energy consumption, performance counters.
  - We show how the characterization of the *resource* of interest (e.g. completion time, or energy), or the characterization of the *program* of interest, can be interpreted algebraically. Standard notions like norm and cosine distance, applied to those characterizations, are expressive tools that contain useful information.
  - We postulate the existence of *computational patterns*, ideal programs that can be used as building blocks to describe the behaviour of real *programs.*
  - We describe 3 algorithms that can be used to create characterizations and predictions, the simplex solver, the linear regression solver, and the non negative matrix factorization solver.

  - We show the algebraic relationship between the solving algorithms, the computational patterns, and the characterization.

- We extend our model to create the notion of *experimental computational complexity* of software. We discuss its relationship with theoretical time complexity. We show why any approach that uses a single dimensional metric to characterize software will inherently have a large error

- We define a simple energy model, that supports concurrent parallel tasks, and a large range of architectures, from single core to large clusters.

3.2 Plan of work

In this part we give an overview of the benchmarking practice. We state that the community is still missing a common metric to indicate performance, the basic tool to characterize hardware. Analysing the state of art, we argue that a high level of fragmentation is present: hardware and software are obviously closely related, but they are modelled separately; completion time, energy, and the other computational resources necessary for a program to perform its task are undoubtedly related to each other, but they are usually modelled with different models. Different and incompatible models are usually built for different architectures. Software characterization models are usually built only on a specific architecture. The state of art reveals that benchmarking, performance analysis, and energy characterization, are conducted as separate research fields, without a unified systematic methodology. Energy and completion time are closely related. However, with the increasing degree of parallelism of modern architectures, thy are not necessarily interchangeable. Completion time is not a representative metric for highly parallel tasks. Total CPU time is more adequate for multi-core architectures, but is not a feasible approach with heterogeneous architectures, e.g. CPU and GPU.

In the second part we present our benchmarking model, a unified approach that characterizes and predicts generic resources consumption (e.g. completion time, energy, performance counters) of programs. The model is unified because it can be used to describe both hardware and software, and the consumption of different resources. We also introduce the concept of *experimental computational complexity* that defines, empirically, the behaviour of programs as the input size grows. Moreover, we define an energetic model that describes the power consumption of concurrent parallel programs.

In the third part we report the outcome of a set of experiments we designed to validate our model: we test the expressiveness of our *experimental computational complexity* using micro-benchmarks; we validate the accuracy of predicting completion time of a representative set of programs (the CPU SPEC suite), building the model on a set of architectures, and testing it on a different set of architectures, to show that the model can be applied to different architectures; we predict the power consumption of OpenFOAM concurrent parallel tasks running on a small cluster to validate our energy model; we predict the best device on an heterogeneous machine where to run an OpenCL kernel, using our prediction model.

In the last part we draw conclusions on our research, discuss the implications, and indicate possible future work.

Chapter 4

State of the art: characterization and prediction of performance and energy

In this chapter we will present the state of the art in the field.

Performance characterization of hardware is a practice with a long tradition. In section 4.1 we outline the characterization approaches. We then continue listing the most used metrics, discussing the limitations of each one. We also provide a short overview of the more used benchmark suites.

In section 4.2 we describe the state of the art of software measurement and characterization. We list the most important approaches, with particular attention to the work that inspired our research.

In section 4.3 we describe the state of the art in energy characterization of computing systems, listing the measurement tools, and the approaches (profiling, simulation, black-box). We focus on energy models, capable of characterize computing systems energetic behaviour, and the use of energy as a possible approach to describe the overall computational effort needed to complete a task.

We close this chapter describing the state of the art in scheduling on heterogeneous architectures. This topic is interesting because it requires the creation of models of performance of a given task on different devices, which is related to our research goal.

4.1 Hardware benchmarking

The evaluation of the performance of hardware involves techniques that can be reduced to three categories: *analytical modelling*, *simulation*, and *measurement* (Jain, 1991; Lilja, 2005).

Analytical modelling is a simplification of reality, therefore usually has low accuracy. However, it can convey useful information about the analysed hardware.

Measuring the performance of machines can produce the most accurate description of their performance, because is using the actual hardware. However, the performance score is not necessarily representative of the effective performance with real-world programs. Because of the difficulty involved with experimentation, and because it needs access to the actual hardware, it is not always feasible. Also, Measuring requires a large amount of time and resources, to buy the computer and the measuring system, to prepare the experiment, to run the experiment (possibly multiple times to ensure statistical validity), and to process the measurements.

Simulation is a trade-off of the *analytical modelling* and the *measuring* approaches. Creating simulation requires creating an emulator of the hardware, a process similar to creating an *analytical model*. The results therefore reflect the limitations of the *analytical model* (simplification of reality). However, the simulation could reveal unexpected behaviour, not modelled by an *analytical* approach. Running simulations can require a large time. Creating an emulator is usually less expensive than acquiring the hardware and running experiments.

No approach comes without problems, and often the combination of several approaches, if feasible, is to be preferred (Jain, 1991).

4.1.1 Metrics

Lilja (2005) states that good metrics should conform to the *linearity*, *reliability*, *repeatability*, *easiness of measurement*, *consistency*, and *independence*:

Linearity : performance changes should be linearly proportional to changes in the metric. People think linearly, using non linear metrics makes them less intuitive;

Reliability : better scores should correspond to actual better performance. This might seem obvious, but the MIPS metric is notoriously not reliable: system A could score higher MIPS than system B, but take more time to complete tasks;

Repeatability : it should be possible to repeat the analysis, obtaining the same score. This is a basic principle of science;

Easiness of measurement : related to the previous criteria, it should be possible to repeat analysis. Like in the scientific method is important to be able to repeat experiments;

Consistency : the unit of a metric should not change as we change system, or comparing alternative systems would be meaningless;

Independence : the metric should be independent from the interests of hardware manufacturers.

Jain (1991) lists the following categories for performance metrics, depending on the nature of the workload used as benchmark, and the aspect of the system that we are interested in measuring: time; rate; resource utilization; error rate; time to failure. For our research, metrics that fall in the "resource utilization" and "completion time" categories are interesting. The number of metrics that have been proposed throughout the years is large. Some try to capture specific aspects of the performance of computing systems (e.g. I/O or connectivity). We will now list the most used performance metrics that attempt to describe the overall performance of a system.

Execution time is the simplest measure of performance that can be defined. It is simply the time needed by a defined task to complete. Measuring completion time on a computer requires executing instructions that start, stop, and log the elapsed time. Those instructions have an overhead and the result will therefore not be precise. For this reason, this approach is only suitable for long-running tasks, where the overhead of the measurement is small compared to the execution time.

Clock rate is simply the CPU operating frequency. It was a popular performance metric before the "megahertz myth" showed is limits. As discussed in section 1.1, "clock speed" is a misleading metric, because it might not reflect the effective performance of the machine. An architecture might have slower clock rate, but be able to perform complex operations in a smaller number of CPU cycles than different architecture, with faster clock.

MIPS is an acronym for Millions of Instructions Per Second, it measures the amount of computation performed in a second. *MIPS* was, and still is, a very diffused metric to express performance. However, it suffers the same limitations of *clock rate*, because the number of instructions performed in a second does not necessarily reflect the amount of useful work performed in the same amount of time (the difference between RISC and CISC). As an example, imagine a system a where each instructions performs complex operations, and system b where the same operation requires 3 time more instructions, because each instruction performs simple operations. The *MIPS* of b might be two times larger than the *MIPS* of a, but a would still be able to perform more work than b in the same amount of time.

MFLOPS measures the number of floating-point arithmetic operations that the system can perform in a second. It improves *MIPS*, because ensures that the performance scores between different system are comparable. However, the concept of "floating-point arithmetic operation" is vague. There are different kind of operations, with different complexity (sum, division, transcendental, trigonometric). Therefore, *MFLOPS* suffers of similar problems as *MIPS*. Moreover, ignoring every non-floating-point instruction is dangerous, because every real-world computations require other instructions. Imagine a machine with an extremely fast Floating Point Unit (FPU), but with extremely slow memory and little cache; most real-world programs make use of memory, and this machine will have poor performance, but a high *MFLOPS* score.

QUIPS adopts a different approach and measures the *quality* of the solution that a system can provide in a limited amount of time, instead of the *quantity* of operations. This approach is reasonable thinking about fields like weather forecasting, or other numerical problems where the precision of the solution is subject to a tolerance level, arbitrarily set to limit the number of iterations. *QUIPS* has limited scope, as it is only appropriate to describe the performance relative to problems where the quality of the solution is measurable as a continuous value.

Speedup is the ratio between the completion time of a workload on a reference machine and the completion time of the same workload on the reference machine. If the *speedup* is larger than 1, then the measured machine has better performance than the reference machine, because it requires less time to complete the workload.

SPEC is a performance metric that refers to the *SPEC CPU* benchmark suite. The completion time of the programs present in the suite are normalized by the completion times on a reference (fixed) machine; then the geometric mean of the ratios is used a single dimensional metric. The *SPEC* metric suffers from the same problem as any other metric that expresses performance as a single number. We will discuss why this approach can not accurately describe the complex interactions of computing systems in chapter 7.6.

4.1.2 Benchmarks

There is an interminable list of benchmarks used to assess the performance of computing systems. Some attempt to capture the overall performance, other try to capture the behaviour on specific tasks (e.g. database, MPI communications, or I/O). In this section we report a small list with the benchmarks that we have found interesting for our work.

Patterson and Hennessy (2008) describe the problem of choosing the program to evaluate hardware and groups benchmarks in four categories: real programs, kernels, toy benchmarks, synthetic benchmarks. They also presents the main benchmark suites and points out the main problems: a single program can not be representative for all the possible workloads. Moreover, the currently used performance metrics (e.g. MIPS, MFLOPS) are not consistent nor useful.

LINPACK (Dongarra et al., 1979) is probably the most successful benchmark for longevity and adoption. Originally designed to assist the users of the LINPACK package, it solves a system of linear equations, using the BLAS library. The performance metric is MFLOPS and completion time.

System Performance Evaluation Cooperative (SPEC) is a consortium of hardware manufacturer. It is the first attempt to provide a standardized methodology to experimentation and reporting in the Benchmarking field. SPEC has added programs to the suite (originally consisting of only 4 programs) to reflect the increased complexity in the hardware industry. They have produced different benchmark suites for different aspects of computation. The most known is the *CPUSPEC* suite (Henning, 2000), but other versions (e.g. for web servers, for Java VM) were released. The chosen performance metric is a ratio (called SPECratio) between the execution time on the profiled system and a reference system.

LINPACK and *SPECCPU* are the most used benchmark suites for evaluating performances of hardware. Salapura et al. (2005) use *LINPACK* to evaluate hardware. Phansalkar et al. (2005); Phansalkar (2007) analyse the similarity of programs in *SPECCPU* showing its redundancy.

Rivoire et al. (2007a) propose JouleSort, a benchmark for energy efficiency of hardware, and lists the existing energy efficiency benchmarks and metrics (Rivoire et al., 2007b).

Performance Evaluation of Cost-Effective Transformations (*PERFECT*) is composed of 13 complete applications (not synthetic or kernels), selected as representative high-performance computations (Berry et al., 1989). It reported total elapsed time and CPU time.

Livermore Fortran kernels were published in 1986 (McMahon, 1986), consisting of 24 "do" loops, focusing on scientific computation, written in Fortran. They have evolved over time, and have been ported to C. They report the performance as arithmetic, harmonic and geometric means of the MFLOPS of each program in the suite. They became popular because the programs are kernels of real scientific computations. They were somewhat representative of the performance of the actual corresponding scientific computations, but require much smaller time to measure.

The Numerical Aerodynamic Simulation (*NAS*) kernels are representative of fluid dynamic scientific programs (Bailey et al., 1991). They contain complex numeric operations, but the performance is measured in MFLOPS.

Whetstone and *Dhrystone* are not used any more, but used to be very popular. *Whetstone* (Curnow and Wichmann, 1976) was a synthetic program that measured floating point performance, reporting results in number of Whetstone interpreter instructions per second (*MWhips*); *Dhrystone* (Weicker, 1984) measured integer performance, reported as number of Dhrystone operations per second.

The Embedded Microprocessor Benchmark Consortium (EEMBC) analyses the performance of hardware and software running on embedded systems. In 2009 they released the *CoreMark* (Gal-On and Levy, 2012), which performs several algorithms, like sorting or matrix manipulation, iteratively. It produces a single number score. EEMBC also released *BrowsingBench*, to measure browsing performance, and *AndEBench* to measure performance of Android platform.

Other benchmarks, that have been used in the past (Price, 1989; Jain, 1991), include:

- During the 1980s, the Digital Review magazine created a benchmark to stress floating-point performance, reporting results as the geometric mean of all tests. Results are also reported normalized on various systems. This benchmark suite has been criticized for using an instruction mix not representative of real-world programming flows.

- The *Dodoc* benchmark uses Monte Carlo method to simulate operations within a nuclear reactor. Results are reported as ratio between CPU time needed to complete the task and an arbitrary reference.

- Simulation Program with Integrated Circuit Emphasis (*SPICE*) from the University of California at Berkeley, stressing integer and floating-point performance.

- *Stanford Integer* and *Stanford Floating Point* suites contain small real-world programs. The performance metric is completion time.

There are also simple programs that are used in benchmark suites, to test particular aspects of computation. For example, the *SIEVE* kernel is based on the Eratosthenes' sieve algorithm to find all the prime numbers below a certain number. It is used to compare microprocessors. Another example is the *Ackermann's Function* kernel, used to analyse the efficiency of procedure-calling.

4.2 Software characterization

The approaches used to characterize software usually fall in one of the following categories:

Simulation : the program is run on a simulated computing system (Mukherjee et al., 2002). A virtual machine that models a particular architecture is used as the host to run the program. All the relevant software events are traced during execution and can be analysed in details. For example it is possible to study the cache-miss rate of a program with a particular memory topology. Simulation tools usually allow the researcher to specify several aspect of the simulated architecture. One of the most used simulation package is *SimplesScalar* (Burger and Austin, 1997), a uniprocessor performance simulation tool. The community has expanded the original tool with multi-threaded support. Other tools include *Rsim* (Hughes et al., 2002), used to simulate shared memory multiprocessors, and *Asim* (Nellans et al., 2004), that extends *SimpleScalar* with a finer grain support.

Profiling . Tracing interesting events during the execution of a program (such as number of memory read, or conditional jumps), is a useful technique used to analyse and profile software. Developers use tools like "gprof" (Graham et al., 1982) or "valgrind" (Nethercote and Seward, 2007) as a daily practice, to study difficult bugs, or simply to analyse the performance of their code. Patel and Rajwat (2013) survey embedded software profiling tools, evaluating profiling frameworks like SnoopP (Shannon and Chow, 2004), Airwolf (Tong and Khalid, 2008), *DPOP* (Shenoy et al., 2010), *DAProf* (Nair and Lysecky, 2008), and others. A limit of profiling is the overhead associated with the sampling procedure, and the behaviour modification induced by the instrumentation necessary to sample the code. This overhead can be contained reducing the sampling, generating reports at regular intervals instead of continuously (Metz and Lencevicius, 2004). However, real-time systems rely on precise events timing, and the behaviour modification could produce non representative reports. Sevitsky et al. (2001) and Ammons et al. (2004) also propose interesting profiling tools.

Black-box . Approaches like *profiling* require instrumentation of code, altering its code with extraneous calls to the profiling framework, introducing overhead. Moreover, there are cases in which instrumenting closed source programs, running on real-time systems, is not viable. Consider for example a scheduler acquiring information about running processes to decide the resource allocation; instrumentation would require an excessive amount of overhead. *Black-box* measurement approaches do not require any change to the executable, and minimal modification of the runtime. Typically only performance counters need to be periodically collected. This still induces an overhead in the computation, but much smaller than with *profiling*.

Saavedra and Smith (1996) propose an approach that was particularly inspiring for our work: a model was proposed to characterize both hardware and software, the overall resource cost is modelled as a linear decomposition of simple components. Vijaykrishnan et al. (2000) also propose an energy characterization model for both hardware and software. Our model lies in the same category, but works on a coarser grain level, and allows the simultaneous use of multiple devices and programs.

Metrics for software similarity are very interesting because allow us to predict the behaviour or programs using measurements of similar programs and allow their characterization.

Yamamoto et al. (2005) propose a metric of similarity based on source code analysis. For the scope of our research we are interested in methods that do not require access to the source code, because we want to be able to characterize software as a black box.

Bonebakker (2007) uses performance counters to characterize software, a technique that is becoming a standard de facto (Curtis-Maury et al., 2006; Phansalkar, 2007; Eeckhout et al., 2002; Duesterwald et al., 2003). Benchmarks are analysed, PCA is used to reduce the solution space and clustering techniques are used to identify families of programs. Eeckhout et al. (2002) have a similar approach, using statistical data analysis techniques such as principal components analysis (PCA) and cluster analysis to efficiently explore the workload space in order to solve the problem of finding a representative workload (the right benchmark with the right input dataset). The idea of defining the similarity of programs to predict the energy usage of a target program is becoming largely accepted (Phansalkar, 2007; Chang et al., 2003; Duesterwald et al., 2003).

Other two key concepts are the idea that the environment where the program is run must be taken into account (Chang et al., 2003) and the need of finding a model capable of offering results that do not change if the same program is run on a different hardware: Sherwood et al. (2001) characterize software with a model consistent with the change of architecture, they have a high level approach (not instruction level), but they do not focus on energy consumption (Sherwood et al., 2002).

Completion time is the right metric to analyse performances of programs (Hennessy et al., 2003). However, completion time clearly heavily depends on both hardware and software. Characterizing a program with its completion time on a certain machine does not say much about its performance in general. For multi-threaded programs Instructions Per Cycle (IPC) or Cycles Per Instruction (CPI) are used. However, as discussed in section 4.1.1, they are poor performance metrics. Alameldeen and Wood (2006) conclude that total execution time should be used.

Computer science traditionally described the complexity of algorithms with the Big-O notation, focusing on asymptotic behaviour. However, programs running on real-world systems need to comply to the physical limitations of hardware, and an asymptotic description of their expected behaviour is often not enough to predict their performance. *Empirical Computational Complexity* attempts to fill the gap between the elegance of the theoretical time-complexity approach, with detailed estimates provided by measures from experiments, and characterizations provided by profilers. In chapter 7 we introduce the notion of *experimental computational complexity*, based on the characterization that our benchmarking model automatically extracts from the experimental data. Similar approaches can be found in the work of Goldsmith et al. (2007). Martin (2001) stated that there is a need for an “energy complexity” description of programs. We also use *experimental* *computational complexity* to investigate on the relationship between time complexity and energy consumption.

4.3 Energy characterization

This section presents the most relevant research approaches on:

1. tools and devices used to measure the energy consumption;
2. the high level approach: simulation, profiling, black-box;
3. the models for characterization of energy consumption and estimation of hardware and software;
4. the relationship between energy consumption and parallel algorithms, virtual machines and time complexity;
5. proposals of policies for the energy management.

4.3.1 Power measurement approaches

In this section we discuss the state of the art in measuring the energy consumed by software. We think that the the use of expansive devices, uneasy to use by non technical users, reduces the reproducibility of the experiments. We also found that too often the measurements are taken aiming at finding the energy consumption of very small events (assembler instructions), losing the ability to see interactions between hardware and software (e.g. patterns of memory access).

The measurement tools

Tiwari et al. (1994) are the first authors to present a model to estimate the energy needed to complete a sequence of instructions, where the energetic cost of single assembler instructions are found empirically. This approach has inspired most of the research that followed, in the sense that most researchers tried to decompose the energetic cost of programs down to the minimum components: assembler instructions (Steinke et al., 2001). However, Tiwari finds that the energetic behaviour of a single instruction varies largely because of interaction with other instructions. He takes into account very simple programs (few instructions repeated many times) and the interaction of instructions in the pipeline of a super-scalar architecture but we believe that many other phenomena occur in real programs.

In many cases the chosen measurement tools were professional expansive digital multimeters (Russell and Jacome, 1998; Flinn and Satyanarayanan, 1999; Bircher and John, 2008; Seo et al., 2009). Russell and Jacome (1998) measure the energy consumption of 32 bit RISC using the same approach as Tiwari: repeating a single assembler instruction many times. Flinn and Satyanarayanan (1999) also use a multimeter to measure energy, it also implements a system monitor to profile programs. The measurements are precise but the testing set is difficult to build because of the cost of the measuring tools and the difficulty of handling them.

In most of the approaches the sampling is time driven, dividing the experiment in equally small time frames (usually in the scale of milliseconds). Chang et al. (2003) propose an interesting shift: the sampling is energy driven. Every time an energy quanta is consumed the measurement system will take a sample. This approach is particularly good to measure idle times, because the act of measuring is less intrusive (if the sampling is time driven more interrupts will be produced and measurements will be altered). But this approach requires non standard measurement devices and the operating system has to be modified to respond to interrupts received every time an energy quanta has been consumed.

More recently, Dutta et al. (2008) proposed hardware modifications that could give energy measurements usable by the operating system, and Bircher and John (2008) propose a measurement system very precise but difficult to be replicated. Both proposed approaches require engineering skills and allow to identify the energy consumption of single hardware components.

The high level approach: profiling, simulation, black-box

In many cases the measured program is analysed and profiled. Flinn and Satyanarayanan (1999) measure the energy consumption with a digital multimeter while a system process monitors the programs. Data from static analysis of source code are gathered and matched with the energy measurements to find the cost of single instructions.

Another common approach is simulating the execution of algorithms on modified virtual machines or power level performance simulators (Brooks et al., 2000; Vijaykrishnan et al., 2000)). Usually the simulators are coupled with a power model (with the energetic cost of every instruction) and this gives estimates of the energy that the program would actually consume on a real hardware. This approach is particularly interesting for hardware producers because is easy to have a quick idea of the possible power savings just editing the power model, simulating the change in a component of the hardware. But this approach is only feasible for simple architectures, where a cycle accurate simulator is possible and for relatively small programs.

The approach often referred to as black-box measures the software as a whole without trying to break down the energy consumption of instructions. This approach is the simplest to implement, does not need modifications to the operating system, can work with a simple ammeter. Rivoire (2008) describes the procedure and in Rivoire et al. (2007a) uses this method to evaluate sorting algorithms. The measuring approach is as simple as possible: the current is measured in AC from the wall outlet. Other works falling in this category are Sinha and Chandrakasan (2001) and Vijaykrishnan et al. (2000).

4.3.2 Energy consumption of virtual machines

Energy consumption of virtual machines is a research field rapidly growing, mostly for the hype about *cloud computing*. The most investigated issue is the management of clusters of virtualized applications, by means of energy aware management models (Raghavendra et al., 2008; Beloglazov and Buyya, 2010a,b; Lefvre and Orgerie, 2010; Kim et al., 2014) and tools (Dhiman et al., 2010); cost models of live migration (Liu et al., 2011) and consolidation (Beloglazov and Buyya, 2010a; Srikantaiah et al., 2008); disk usage techniques (Ye et al., 2010). There are plenty of models for various aspects of cloud computing, nonetheless no model has been proposed yet to characterize software running in a virtualized environment. Also, there is no simple tool available to measure the energy efficiency of hardware and hypervisors running in a cloud environment, because the available specialized benchmarks are either too complex (SPECvirt [1]), incomplete (VMmark [2] is unable to measure guest systems with more than one virtual processor) or not freely available (Intel's vConsolidate).

4.3.3 Energy consumption related to parallelism

Few are the models that describe the energy efficiency of parallel computations. Cassidy and Andreou (2011) extend Amdahl's law, proposing a function for finding the optimal level of parallelism minimizing the energy consumption while preserving performances.

Amdahl is also used as a basis for energetic models of parallel computations (Cho and Melhem, 2008; Woo and Lee, 2008; Cho and Melhem, 2010)

[1] http://www.spec.org/virt_sc2010/
[2] http://www.vmware.com/products/vmmark/

in our opinion without proposing a simple and expressive model. We also think that energy consumption could be used as an expressive metric of an algorithm's parallelism.

4.3.4 Energy consumption related to time complexity

The relationship between energy consumption and time complexity is mostly yet to be studied. Martin et al. (2011) are the only authors we could find proposing the idea of energetic complexity of software. They also define metrics (similar to the well known Θ) and note that the widely used $E \times t$ (energy multiplied by time) metric is misleading because CPU power usage and CPU frequency are tied by a quadratic ratio, so the best metric is $E \times t^2$.

4.3.5 Energy management policies

Policies have been proposed (at many levels: firmware, operating system, etc.) to reduce the energy consumption of hardware and software. This area gained much attention when the laptops and the smartphones became ubiquitous, since then saving battery life has been a primary concern. Lebeck et al. (2000) study the energetic cost of policies of Page Allocation, we see how the best results are obtained if we take into account both hardware and software simultaneously. Neugebauer and Mcauley (2001) use energy as a scheduling resource in the Nemesis OS. Zeng et al. (2002) also propose a scheduling policy that aims at saving energy.

A more recent research topic is the tuning of processor speed by using DVFS. Isci et al. (2006), Rangan et al. (2009) and Bircher and John (2008) propose models of energy usage and attempt to improve the policy used to adjust the processor speed in order to save energy not losing performances.

4.3.6 Coarse grain energy models

Most of the research around energy consumption modelling and power aware scheduling is based on instruction-level power models (Li and John, 2003; Tiwari et al., 1994; Brooks et al., 2000); Dynamic Voltage Scaling (Mishra et al., 2003; Yang et al., 2005; Chen and Kuo, 2007); CMOS logic (Mudge, 2001). Already in 2004, Bianchini and Rajamony (2004) reported these approaches as "current state of art", and stated that modelling overall energy consumption and peak power would be the future challenges for server systems. But while copious amount of literature can be found on low-level energy models, not the same can be said for models about the upper layers of the software stack.

Gu et al. (2014) review methods for power consumption measurement systems. The need for high-level energy model is evident when trying to describe the energy consumption of clouds, at the server level. Feng et al. (2005) and Kim et al. (2011) showed that energy consumption can successfully be characterized and predicted, even with simple and high-level energy models (at the system level), complying with the recommendations of Bianchini and Rajamony (2004). Kim et al. (2011),Wang et al. (2011), Ma et al. (2009) and Zhang et al. (2011) use statistical approaches to model and predict the energy consumption of programs.

Several energy model have been proposed to characterize the energy consumption of tasks (Goiri et al., 2010; Kim et al., 2011), also parallel tasks have been modelled (Garg et al., 2009; Li, 2012; Wang et al., 2010). Most of the research focuses on Dynamic Voltage Scaling (DVS) of the CPU. However, not many models attempt to describe the power and energy consumption of concurrent parallel tasks, running on the same *computational environment*.

In this work we present an energy model that has the same level of abstraction as Feng et al. (2005) and Wang et al. (2011), but it models the system power consumption in a different way. The most abstract form of our model is compatible with one of the models reported in Gu et al. (2014), we factor out some parts in a refinement in order to be able to apply statistical analysis and automatically characterize the computational environment. Our model is also a generalization of Kim et al. (2011), they specialize the general energy formula to describe energy consumption in terms of performance counters, focusing on virtual machines (VMs) running on a single machine. We start from the same high-level energy formula, but we focus on a higher level, modelling the energy consumption of a cluster, in terms of number of active machines and cores, still achieving accurate results.

Our work is closely related to Kim et al. (2011). They propose a simple high level energy model. Like us, they use the coefficients of linear regression to characterize servers, and use the coefficients to predict energy consumption. They apply this model to several virtual machines (VMs) executing concurrently on a single server, to separate the energy consumption of each VM. We extend their approach to model the effect of scaling the model on more physical machines. Our model has similar accuracy, but a coarser grain, in particular we do not rely on performance counters, but only on time measurements. This difference makes our model easier to be adapted to systems with heterogeneous components (GPU, Xeon Phi, systems with dedicated co-processors, etc.). With our approach there is no need to develop ad-hoc probes, we only need the utilization time, whereas Kim et al. (2011) needs some measure of the kind and number of operations performed by each component. Moreover, collecting performance counters is not always easy, and adds a computational load, especially when this has to be done for each core or for each component in the case of heterogeneous systems.

Another approach similar to our is SPAN (Wang et al., 2011), where the energy consumption of tasks is predicted using the number of instructions per cycle (IPC). However, the scope of the models is different: our work uses completion time whereas SPAN relies on performance counters; we model the parallel execution of multiple tasks running on the same machine, whereas SPAN only models a single active task; we model a multi-node architecture whereas SPAN focuses on a multi-core architecture.

An important consequence of the choice of relying on the number of operations is that, as can be seen in the Evaluation section of Kim et al. (2011), when the number of operations does not describe well the program, other counters are needed, such as the number of memory accesses. This led the authors to conclude that "The first model is a simple model that calculates the amount of energy consumption by multiplying the processor time with the average power consumption of the processor, [...]. Due to the oversimplification, this model still shows poor accuracy.", whereas we show that only relying on processor time, with an appropriate model, accurate predictions are possible.

Other authors attempted to model and predict power consumption using statistical approaches. Feng et al. (2005) propose a high level power model, with different levels of granularity, system, node, and component level. They model different components, and model parallel jobs, but they do not take into account concurrent tasks. Moreover, the profiling phase is complex and it requires manipulation of the computational environment. We think that, to be adopted in a real world scenario, the profiling procedure should involve the least possible effort.

Ma et al. (2009) also use a statistical model trained to predict GPU power. Support vector regression (SVR) is used and compared to the predictions obtained using least square regression, with similar results. SVR involves augmenting the dimensionality of the input domain and usually a consequent mapping to non-linear spaces using kernel tricks. This allows the model to capture non-linear behaviours, but it makes it very hard, if not impossible, to interpret the model, that will have to be accepted or rejected only looking at the fitting of test data. This approach is prone to over-fitting.

Zhang et al. (2011) use a different approach: random forest. The model has an $R^2 = 0.89$, meaning that roughly 90% of the information has been described by the model. The median absolute error is reported to be 0.043, without further description of the error distribution.

Bircher and John (2012) model power consumption of subsystems (CPU, memory, IO, disk and GPU) with ad-hoc linear models for each subsystem. Their goal is to show that models can be created in a training phase using performance counters, and that an estimate the system power consumption can be carried out at runtime without the need for power sensing hardware. During the training phase, power measurements require employing resistors connected in series with the power source on each subsystem. The relative prediction error in each subsystem is usually less than 0.1, but the paper does not report the total error, that from an estimate should be less than 0.2. They use 13 performance counters.

4.4 Scheduling on heterogeneous architectures

For our research, scheduling on heterogeneous architectures is an interesting topic, because selecting the best device for a program requires characterization of both hardware and software, and to carry out predictions of their performance.

Scheduling on systems exposing multiple devices has been quite a studied problem. In this section we present and discuss the most recent and relevant works in this research area and in particular those focusing on scheduling on CPU-GPU heterogeneous systems, trying to underline the major differences with our approach.

Best device prediction is a classification problem, hence popular classification approaches, such as SVM (Support Vector Machines), have been successfully used. Wen (2014) uses Support Vector Machines, with a Gaussian Kernel, to predict whether an algorithm would run faster on the GPU or on the CPU. An important limitation of this approach is that it is very hard (if not impossible) to interpret the model created by an SVM, which has to be accepted or rejected only looking at the prediction results and might suffer from over-fitting. We tried to apply SVM to our data (using the same kernel described by Wen (2014)) and could not replicate their best device prediction accuracy. In contrast, as we show in the experimental part, using a simple method such as linear regression it is possible to analyse the regression coefficients, that will usually have an intuitive interpretation and it is possible to verify them; e.g. the coefficient assigned to the *number of instructions* feature should be in the same order of magnitude as the processor frequency, or the coefficient assigned to the *number of memory accesses* should be compatible with the memory bandwidth.

Other statistical approaches have been attempted. Iverson et al. (1999) uses K-Nearest Neighbour to predict completion time on the basis of code and input similarities. Huang et al. (2010) apply Sparse Polynomial Regression to a set of automatically selected features for completion time estimation. We think that linear regression should be favoured, because it is an easier approach, it offers an understandable model and it is suitable to progressive refinements. As stated by Huang et al. (2010), some non-linear aspects may be impossible to model using linear methods. Nonetheless, linear regression is a feasible approach if the relation between the dependent variable and the explanatory variables (e.g. completion time and code features) is linear, which shifts the problem to meaningful feature selection. For example, whereas the size of the input matrices (n) has not a linear relation with the completion time in matrix multiplication (n^3 for sequential implementations), the completion time is linear on the number of operations/memory accesses. The problem is therefore to select the appropriate features to consider in order to predict the completion time.

Part II

Benchmarking Model

In chapter 5 we define a high level approach to a generic and unified of computational systems characterization model. Our model creates an analytic representation, called *surrogate*, of the entity we are interested in modelling (a *target program* or a *target resource*), from measurements of *resource* consumption of *benchmarks* running on known hardware. The *target program* is expressed in terms of the *benchmarks*, the *target resource* in terms of the other *resources*. In this chapter we introduce the concept of *computational pattern*.

In chapter 6 we discuss the different algorithms that can be used to extract *surrogates* and to create predictions. Linear Regression is a simple model that has an intuitive interpretation of the *surrogate*, yet offering good predictive capabilities, as show in chapters 11 and 12. Linear Regression assumes that the *target program* can be expressed as a linear combination of *benchmarks* (or the *target resource* in terms of *resources*). This can be explained using *computational patterns*.

In chapter 7 we introduce the concept of *experimental complexity* as an empirical method to describe how the resource consumption of an algorithm changes as the input size grows. In this chapter we also describe how to use our model to predict bottlenecks. In section 7.5 we discuss the compositionality of the model. We show how it can be explained in terms of *computational patterns*. In section 7.6 we discuss the algebraic reasons why a single number can not be a representative *surrogate* for a *program* or a *computational environment* performance. This explains why simple uni-dimensional metrics like FLOPS fail to describe performances.

In chapter 8 we introduce an energetic model, a set of equations that show the relation between completion time, instant power and energy consumption. This model is capable of describing both sequential and parallel concurrent computations.

Chapter 5

High level model

In this chapter we provide a general and high level description of our black box, regression based approach. The approach is black box in the sense that does not require any knowledge of the source code of the program. It is easy to apply because it does not require to change any aspect of the hardware, it can be used with standard metrics such as performance counters.

Our approach can be seen as a Machine Learning task, more specifically Supervised Learning, that infers properties from training data. The training data consists of a set of measures for a specified set of programs, here called benchmarks. The measures need to be taken on a specified set of resources. A regression model is created on the training data set, and the regression coefficients are used as a characterization of the entity subject of interest, usually a particular program, or a particular resource, such as completion time or energy. The regression coefficient can also be used to create prediction of resource usage.

The model we propose has several qualities required by a good scientific model:

- it can explain past observations;
- it can predict future observations;
- it is computationally cheap to create a characterization of the target entity (program or resource), and to create predictions;
- the quality of the characterization and predictions can be assessed, therefore refused;
- the model is generic and applies to different phenomena;

- the model is as simple as possible;

The model provides a surrogate of the program (or of the computational resource) we want to describe. To be able to compute the surrogate using the data measured with experiments, we need a solver. The simplest solver is linear regression.

In the rest of this chapter:

- we define all the important terms used in the model, such as *program*, *resource*, and *solver*, that creates characterizations and predictions;
- we define the general setting for the model, that can be implemented using different *solvers*;
- we introduce the concept of *computational patterns*;
- we discuss some of the algebraic properties of the model;

5.1 Definitions

In this section we provide the definitions for the terms used in the rest of the work.

5.1.1 Program

Definition 1 (Program)**.** *A program is a particular and defined sequence of instructions. Programs is defined by a software and some input data.*

We will use p or p_i to indicate generic programs. If known, we will use the program's name, e.g. povray or gcc.

The same *program* can be run on different micro architectures, even if it will generate different low level sequence of processor instructions, it will still be considered the same *program.* When called to process different input sizes, because the sequence of high level instructions will considerably change, it will be considered a different program.

5.1.2 Computational Environment

Definition 2 (Computational Environment)**.** *A computational environment is a computational system that can execute programs.*

We will use upper-case letters to indicate computational environments, e.g. A or B.

Examples of *computational environments* are embedded computers, smart phones, PCs with different micro-architectures, clusters. We consider part of the *computational environment* the hardware as well as the operating system and all the software running on the machine at the same time as the *program* being measured.

5.1.3 Measure of resource consumption

Definition 3 (Resource). *A resource is a finite asset of the computational environment that is used by programs to run.*

The energy or the time used by a computer to complete a *program* are examples of *resources*. The same resource on different *computational environments* are considered different *resources*: e.g. completion time on computer A and completion time on computer B are different *resources*. Therefore a *resource* also always refers to a *computational environment*.

We use r or r_i to indicate generic *resources*.

Definition 4 (Measure). *A measure is a positive real number that describes the quantity of resource used by a certain program to run on a certain computational environment.*

We indicate the *measure* of the consumption of the *resource* r to execute the program p, we write $\mu_r(p)$, or $\mu(p)$ if clear from the context. In the experimental we will indicate measures of completion time with t and measures of energy consumption with e.

A *measure* always refers to both a *program* and a *resource* (therefore a *computational environment*), i.e. a *measure* quantifies the usage of a particular *resource* on a particular *computational environment* by a *program*.

Not every *resource* can be used in our model, it needs to provide measures that respect the mathematical definition of measure, i.e. they need to have the following properties:

- Non-negativity:

$$\forall x \quad \mu(x) \geq 0 \tag{5.1}$$

Non negativity

all measures must be non negative real values

- Null empty set:

$$\mu(\varnothing) = 0 \tag{5.2}$$

Null empty set

the resource consumption of running an empty program must be equal to zero

- Countable additivity:

$$\forall x_i \quad \mu(\cup x_i) = \sum \mu(x_i) \tag{5.3}$$

Countable additivity

if two programs p_1 and p_2 do not share a computational resource, then the measure of its consumption when executing p_1 and p_2 must be the sum of the resource consumption of executing them individually: $\mu(p_1 + p_2) = \mu(p_1) + \mu(p_2)$. It's often difficult to ensure that programs do not share resources, in such case countable subadditivity is sufficient

$$\forall x_i \quad \mu(\cup x_i) \leq \sum \mu(x_i) \tag{5.4}$$

Countable sub additivity

Examples of valid resources are processor time, completion time, memory allocations, energy. Examples of invalid resources are % processor time (it may decrease), active memory (memory could be deallocated), power (instant power could decrease). Usually invalid resources can be made valid combining them with time.

An interesting example is average Power, which is not a valid *resource*. Let's consider a program p_3 that is composed of 2 *programs* p_1 and p_2 executed sequentially: $p_3 = \{p_1; p_2\}$. If P_{p_1} the average power during the execution of p_1 is higher than P_{p_2} the average power during the execution of p_2, then P_{p_3} the average power during the execution of p_3 will be $P_{p_2} \leq P_{p_3} \leq P_{p_1}$. This violates countable sub additivity (the power used by a part of a *program* is higher than the power used by the whole *program*). On the other hand, because both average power and completion time are always positive quantities, countable sub additivity holds for energy: $E_{p_3} = E_{p_1} + E_{p_2} = P_{p_1}T_{p_1} + P_{p_2}T_{p_2}$, $E_{p_3} \geq E_{p_1}$, $E_{p_3} \geq E_{p_2}$ (where T_{p_1} and T_{p_2} are completion times for p_1 and p_2).

Countable sub additivity does not hold if a resource acts a bottleneck for the computation. In section 7.4 we discuss the details of this phenomenon and how this can be used.

5.1.4 Computational Pattern

Definition 5 (Computational Pattern)**.** *A computational pattern is an ideal program the exhibits a peculiar resource consumption.*

Examples of *computational patterns* are: a *program* made in its entirety by floating point operations; or a *program* that triggers a cache miss at every instruction. *Computational patterns* are usually ideal *programs*, real *programs* can not consist only of a single *computational pattern*. At most synthetic *benchmarks* can approximate particular *computational patterns*. Some *computational pattern* could be reasonably be guessed (in some case even designed), but in general they are unknown, and may arise when new micro-architectures are created: a novel micro-architecture could expose a peculiar resource usage when used by a certain sequence of instructions.

Our model assumes that all *programs* can ideally be decomposed in sequences of *computational patterns*. The *computational patterns* form a basis of the resource consumption space (because they are orthogonal with respect to resource consumption). Any program, including both the *benchmark* and the *target program*, can be written as a linear combination of the *computational patterns*. If every *computational pattern* used by the *target program* is contained at least in one of the *benchmarks*, and if the *benchmarks* are not linearly dependent, we can operate a change of basis and express the *target program* as a linear combination of the *benchmarks*. If the *target program* contains *computational patterns* that are not contained in any *benchmark*, then the change of basis will lose information.

5.1.5 Solver

Definition 6 (Benchmark)**.** *A benchmark is a program used to predict the measure of the target resource and the target program.*

Definition 7 (target resource)**.** *The target resource is the resource that we want to predict for the target program.*

We use r_t to indicate the *target resource*, or the name of the resource if it is known.

Definition 8 (target program)**.** *The target program is the program that we want to model in terms of the benchmarks, whose target resource we are interested in predicting.*

we use p_t to indicate the *target program*, or the name of the *program* it is known.

Definition 9 (target measure)**.** *The target measure is the measure of the target resource and the target program that we want to predict.*

Definition 10 (Solver)**.** *A solver is an algorithm that, given a set of measures of the benchmarks and the target program, creates a surrogate for it.*

Definition 11 (Surrogate)**.** *A surrogate is an analytic representation of the target program (or the target resource). The surrogate is the output of the solver.*

The actual nature of the surrogate depends on the solver, it can be a simple structure as a vector (for linear models), or a couple of matrices (when using the Non Negative Matrix Factorization solver), or have a more complex structure (when using a Support Vector Regression based solver).

If we assume that programs are linear combinations of *computational patterns*, then the surrogate is surrogate is a linear combination of benchmarks used.

The surrogate does not depend on the *target resource* or *computational environment*

5.2 The model

In this section we present an abstract description of the model, where a surrogate for the interesting aspect is created using measures the resource consumption of benchmarks. The model will be implemented in section 5.2.1 to obtain a *surrogate* that characterizes a *target program*, and in 5.2.2 to obtain a *surrogate* that characterizes a *target resource.*

We define a matrix $\mathbf{X}$ containing the measures of *resource* consumption of the *benchmarks.* To define $\mathbf{X}$ we proceed as follows:

- we decide the set of *resources* we want to use to build our model
- we decide the set of *programs* we will use as *benchmarks*
- we measure all the *resources* for all the *benchmarks*
- we build the matrix $\mathbf{X}$ using the measures. The way the measures are arranged in the matrix depends on the actual implementation of the model, as described in section 5.2.2 and 5.2.1

We then define a vector $\mathbf{y}$ with the measures of the aspect we are interested in modelling, either the *target program* or the *target resource.*

The first goal of the model is to create a *surrogate* of the aspect we are modelling. Analysing its surrogate we will be able to characterize the *target program* or the *target resource*, gaining a better understanding of its behaviour.

To build a surrogate β we will use some function f that maps $\mathbf{X}$ and $\mathbf{y}$ into β, as shown in equation 5.5. The function f depends on the *solver* we choose.

$$\beta = f(\mathbf{X}, \mathbf{y}) \tag{5.5}$$

solver

The model can usually also predict the *target resource* consumption for the *target program*.

Firstly, we define a new vector $\mathbf{x}$ with a new set of measures. As we will explain in section 5.2.1, if the *surrogate* β describes the *target program*, then $\mathbf{x}$ contains the measures of the *target resource* for the *benchmarks*. In section 5.2.2 we will show that if the *surrogate* β describes the *target resource*, then $\mathbf{x}$ contains the measures of the *target program* for the other resources.

p is a prediction of the *target resource* usage for the *target program*.

The *solver* defines a function g is a function that maps β and $\mathbf{x}$ to p, as shown in equation 5.6

$$p = g(\beta, \mathbf{x}) \tag{5.6}$$

predictor

f , β and g depend on the *solver*. In chapter 6 we will explore several options, most of which can create both a surrogate β and a prediction p. We will see that even the simplest approach (linear regression) still performs remarkably well, and offers *surrogates* easy to interpret as well as precise predictions.

5.2.1 Characterizing a target program

In this paragraph we describe one implementation of the abstract model to obtain a *surrogate* of the *target program* capable of characterizing it, and to be able to predict its *target resource* consumption.

The *surrogate* created with this approach characterizes the *target program* in terms of the *benchmarks*. If the behaviour of the *benchmark* is known, this will provide useful information about the behaviour of the *target program.*

As described in the previous section, the matrix $\mathbf{X}$ contains all the measures of the consumption of all the chosen *resources* of all the chosen *benchmarks.* We organize $\mathbf{X}$ in the following way:

- all the measures relative to the same *resource* lie in the same row.
- all the measures relative to the same *benchmark* lie in the same column.

Equation 5.7 shows the matrix $\mathbf{X}$, with m *resources* and n *programs.*

$$\mathbf{X} = \begin{pmatrix} \mu_{r_1}(p_1) & \mu_{r_1}(p_2) & \cdots & \mu_{r_1}(p_n) \\ \mu_{r_2}(p_1) & \mu_{r_2}(p_2) & \cdots & \mu_{r_2}(p_n) \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{r_m}(p_1) & \mu_{r_m}(p_2) & \cdots & \mu_{r_m}(p_n) \end{pmatrix} \tag{5.7}$$

The vector $\mathbf{y}$ contains the measures relative to the *target program*, organized as a column of $\mathbf{X}$, as shown in equation 5.8.

$$\mathbf{y} = \begin{pmatrix} \mu_{r_1}(p_t) \\ \mu_{r_2}(p_t) \\ \vdots \\ \mu_{r_m}(p_t) \end{pmatrix} \tag{5.8}$$

The *surrogate* characterizes the *target program.* The *surrogate* depends on the *solver* used. Simple *solvers* create *surrogates* that can be easily interpreted. For example, the *surrogate* created by Linear Regression, described in section 6.2, expresses the *target program* as a linear combination of *benchmarks.*

If the *solver* provides a *predictor* (the g function in equation 5.6), then the *surrogate* can be used to predict the consumption of the *target resource* by the *target program.* To obtain a prediction we need to build a the $\mathbf{x}$ vector, by measuring all the *benchmarks* on the *target resource.* The measures are organized in the same way as the rows of $\mathbf{X}$, as shown in equation 5.9.

$$\mathbf{x} = \begin{pmatrix} \mu_{r_t}(p_1) & \mu_{r_t}(p_2) & \cdots & \mu_{r_t}(p_n) \end{pmatrix} \tag{5.9}$$

Applying equation 5.6 to the g function provided by the *solver*, the *surrogate* β, and the vector $\mathbf{x}$, we obtain a prediction of the measure of the consumption of the *target resource* by the *target program.*

5.2.2 Characterizing the target resource

The abstract model can also be implemented to characterize the *target resource*.

The *surrogate* created with this approach characterizes the *target resource* in terms of the other *resources*. For example it could describe the completion time on a particular machine in terms of the performance counters.

The matrix $\mathbf{X}$ is organized as the transpose of the matrix described in the previous section:

- all the measures relative to the same *resource* lie in the same column.
- all the measures relative to the same *benchmark* lie in the same row.

Equation 5.10 shows the matrix $\mathbf{X}$, with m *resources* and n *programs*.

$$\mathbf{X} = \begin{pmatrix} \mu_{r_1}(p_1) & \mu_{r_2}(p_1) & \cdots & \mu_{r_n}(p_1) \\ \mu_{r_1}(p_2) & \mu_{r_2}(p_2) & \cdots & \mu_{r_n}(p_2) \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{r_1}(p_m) & \mu_{r_2}(p_m) & \cdots & \mu_{r_n}(p_m) \end{pmatrix} \tag{5.10}$$

The vector $\mathbf{y}$ contains the measures relative to the *target resource*, organized as a column of $\mathbf{X}$, as shown in equation 5.11.

$$\mathbf{y} = \begin{pmatrix} \mu_{r_t}(p_1) \\ \mu_{r_t}(p_2) \\ \vdots \\ \mu_{r_t}(p_m) \end{pmatrix} \tag{5.11}$$

The *surrogate* characterizes the *target resource*. The *surrogate* depends on the *solver* used. For example, the *surrogate* created by Linear Regression, described in section 6.2, expresses the *target resource* as a linear combination of the other *resources*. In the 13 chapter we show how to describe completion time as a linear combination of performance counters.

If the *solver* provides a *predictor* (the g function in equation 5.6), then the *surrogate* can be used to predict the consumption of the *target resource* by the *target program*.

To obtain a prediction we need to build a the $\mathbf{x}$ vector, by measuring all the *resources* (except the *target resource*) on the *target program*. The measures are organized in the same way as the rows of $\mathbf{X}$, as shown in equation 5.12.

$$\mathbf{y} = \begin{pmatrix} \mu_{r_1}(p_t) & \mu_{r_2}(p_t) & \cdots & \mu_{r_n}(p_t) \end{pmatrix} \tag{5.12}$$

Applying equation 5.6 to the g function provided by the *solver*, the *surrogate* β, and the vector $\mathbf{x}$, we obtain a prediction of the measure of the consumption of the *target resource* by the *target program*.

The output of the *predictor* is the same as in the implementation that characterizes *programs* instead of *resources*. In both implementations we predict the *target resource* usage of the *target program*.

5.2.3 A unified HW and SW model

In the previous sections we have shown how the same abstract model (described in section 5.2) can be implemented to characterize either *programs* (described in section 5.2.1) or *resources* (described in section 5.2.2). Both implementations build their *surrogates* starting form the same data (the matrix $\mathbf{X}$), simply disposing measures in a different way (the matrix $\mathbf{X}$ described in 5.2.2 is the transpose of the matrix $\mathbf{X}$ described in 5.2.1).

It is worth noticing that the model used to characterize hardware is the dual of the model used to characterize software. To see how closely related the two models are, consider the following:

- Regarding the model used to characterize and predict the *target resource*:
  - be $\mathbf{X}_{\mathrm{HW}}$ the matrix containing the measures organized with each row representing a *benchmark* and each column representing a *resource*;
  - be $\mathbf{y}_{\mathrm{HW}}$ the vector with measures of the *target resource* for the *benchmarks*;
  - be $\mathbf{x}_{\mathrm{HW}}$ the vector with measures of the *resources* for the *target program*;
  - the *surrogate* β_{HW} is the characterization of the *target resource* and is given by $\beta_{\mathrm{HW}} = f(\mathbf{X}_{\mathrm{HW}}, \mathbf{y}_{\mathrm{HW}})$;
  - the prediction of the *target resource* consumption of the *target program* is given by $p = g(\beta_{\mathrm{HW}}, \mathbf{x}_{\mathrm{HW}})$.

- Regarding the model used to characterize and predict the *target program*:
  - be $\mathbf{X}_{\text{SW}}$ the matrix containing the measures organized with each row representing a *resources* and each column representing a *benchmark*;
  - be $\mathbf{y}_{\text{SW}}$ the vector with measures of the *target program* for the *resources*;
  - be $\mathbf{x}_{\text{SW}}$ the vector with measures of the *benchmarks* for the *target resource*;
  - the *surrogate* β_{SW} is the characterization of the *target program* and is given by $\beta_{\text{SW}} = f(\mathbf{X}_{\text{SW}}, \mathbf{y}_{\text{SW}})$;
  - the prediction of the *target resource* consumption of the *target program* is given by $p = g(\beta_{\text{HW}}, \mathbf{x}_{\text{SW}})$.
- Note that:
  - $\mathbf{X}_{\text{HW}} = \mathbf{X}_{\text{SW}}^{\top}$;
  - $\mathbf{y}_{\text{HW}} = \mathbf{x}_{\text{SW}}^{\top}$;
  - $\mathbf{x}_{\text{HW}} = \mathbf{y}_{\text{SW}}^{\top}$.

In the next chapter we will see that with the *solvers* we propose, if $\mathbf{X}_{\text{HW}}$ and $\mathbf{X}_{\text{SW}}$ are square matrices, $p = \mathbf{x}_{\text{HW}}\beta_{\text{HW}} = \mathbf{x}_{\text{SW}}\beta_{\text{SW}}$.

Our model is capable of describing both hardware and software characteristics, in a unified Hardware and Software approach. As discussed in chapter 2, this is a desirable feature of benchmarking, because the performance of a *program* heavily depends on the *computational environment* where it is executed, and the performance of a *computational environment* depends on the software it runs. Hardware and Software are intimately tied, and a model that does not take this fact into account, will inevitably build *surrogate* that depend on the *computational environment* used to build the model, hiding important information. Our approach allows us to identify the contributions of Hardware and Software separately.

5.2.4 A resource agnostic model

The model is built using measures from different *resources*. As discussed in chapter 2 the approaches in literature usually focus on a single *resource* (e.g. completion time, or energy consumption), and usually a single *computational environment* is used, i.e. models and experiments are limited to a single computer. This makes the training phase difficult, because statistical models are best trained if a large amount of data is available. Moreover, the model can only contain the information present in the training data, but an algorithm could have different behaviours on different *computational environments.*

In our model, it is not only allowed, but even desirable to use heterogeneous *resources* (from different classes of *resources*, such as completion time, performance counters and energy consumption). Moreover, it is desirable to have measures from different *computational environments*, especially if the *benchmarks* and the *target program* have a different behaviour on the different *computational environments.* Our approach is therefore capable of capturing subtle behaviours that are usually difficult to model, simply because the data used to build it contains information about those behaviours that only emerge when a *program* is run on different architectures, or different aspects of the *computational environment* are measured.

It is worth noticing that the model does not require the *resources* to necessarily come from physical machines. It is possible to combine our approach with a simulation tool, to trace the execution of *programs* running in an emulated operating system. This approach is useful when is difficult to run experiments on physical machines, or when the number of available measured *resources* is not large enough. For example, it is possible to simulate the *program* counting cache-misses, running on architectures with different cache size and layout, to characterize the pattern of memory access. *Measures* coming from physical *resources* and simulated resources can be combined in the same model. Obviously physical *resources* will be more reliable and representative of actual performance.

The same model can be used to predict every admissible *resource.* In our work we focus on completion time and energy consumption, but as long as the *resource* meets the requirements described in definition 3 it can be used to build the model, and can be the *target resource* of a model. Our model is therefore resource agnostic.

5.3 Role of computational patterns

This section explains the role of *computational pattern* in the algebraic structure of our model. Examples of *computational patterns* are: a certain amount of floating point multiplications, or a certain pattern of memory access. Assuming to be able to measure the *computational patterns* on a large enough number of *resources*, they will be linearly independent, because by definition 5 each *computational pattern* expresses a different *resource consumption* behaviour, forming a basis of the *resource consumption space.* The *resources* involved in the peculiar behaviour of the considered *computational patterns* need to be measured to be able to distinguish them.

As an example, consider two *computational patterns*: pattern a captures integer sums, pattern b captures integer multiplications. Imagine now measuring those *computational patterns* only on architectures where the ratio between the number of CPU cycles necessary to perform integer multiplication and integer sums is the same. The *computational pattern* will be collinear in the *resource consumption space.* However, as soon as we measure them on a different architecture, where this ratio changes, the patterns a and b will become distinguishable. This is the why our model not only allows, but encourages the usage of measures from different architectures. Their usage in the same model will let *computational patterns* emerge.

However, as stated in Definition 5, a *computational pattern* is an ideal concept. Real programs are composed of *computational patterns*, but in general a *computational pattern* can not be a real *program.* A *program* can not consist of a single *computational pattern*, as every *program* to run has to perform a several operations, e.g. loading instructions from memory, and can not exclusively be composed of a single *computational pattern.* Therefore *computational patterns* can not be measured directly, we can only measure real *program*, that are composed of several *computational patterns.* Ideally, synthetic *benchmarks* can be designed to approximate particular *computational patterns.*

As already stated, our model assumes that all *programs* can ideally be decomposed in sequences of *computational patterns.* Therefore, if the matrix **W** contains the measures of the *resource* consumption of each *computational pattern* (the rows are *resources*, the columns are *computational patterns*), and the matrix **H** contains the *benchmarks* expressed as linear combinations of *computational patterns* (the rows are *computational patterns*, the columns are *programs*), then the matrix of the measures of the *resource* consumption of the *programs* **X** can be expressed as the matrix multiplication of **W** and **H**, as shown in equation 5.13.

$$\mathbf{X} = \mathbf{W}\mathbf{H} \tag{5.13}$$

Benchmarks as compositions of patterns

If two of the measured *resources* are very similar, the corresponding rows of **W** will be very similar, making **W** not full rank. As stated, *computational*

Table 5.1: Composition of programs in terms of computational patterns

program	a	b	c	d
p_1	1	1	0	0
p_2	0	0	1	1
p_3	1	0	1	0
p_4	0	1	0	2

Table 5.2: Measures of the computational patterns

system	a	b	c	d
μ_1	2	1	1	1
μ_3	1	2	1	1
μ_3	1	1	2	1
μ_4	1	1	1	2

patterns are linearly independent, in the sense that each one expresses a different *resource* consumption behaviour. The *resources* to be used (the rows of **X** and **W**) should be chosen to reveal the different behaviour of *computational patterns*, i.e. the computational patterns form a basis of **W**.

In the following example we have 4 programs p_1, p_2, p_3 and p_4, composed of linear combinations of 4 *computational patterns* a, b, c and d, table 5.1 shows the composition of each *program* in terms of how many instances of each *compositional pattern*. This table is the transpose of the **H** matrix.

Let's assume we can measure the usage of 4 *resources* of *computational patterns*, where they show different usage behaviour: on μ_1 they all cost the same, on μ_2, μ_3, μ_4 and μ_5 all *compositional pattern* cost the same except one, that costs twice as much, as show in table 5.2. This table is equivalent to the **W** matrix.

The 4 *programs* will therefore have the costs shown in table 5.3. This table is the **X** matrix.

The matrix **W** is a function that maps vectors that express *programs* as linear combinations of *computational patterns* (the columns of **H**) into vectors that contain the *resource* consumption of *programs* (the columns of **X**).

Table 5.3: Measures of the programs

system	p_1	p_2	p_3	p_4
μ_1	3	2	3	3
μ_2	3	2	2	4
μ_3	2	3	3	3
μ_4	2	3	2	5

5.4 Algebraic characterization of X

In this section we discuss the conditions that **X** should satisfy, and we present a few algebraic tools that can be used to explore the information contained in **X**.

5.4.1 X needs to be full rank

Depending on the *solver* used, there might be different conditions over **X**, e.g. if Linear Regression is used as a *solver* it should not have more columns than rows, as it would result in an under-determined system, as discussed in section 6.2.

As a general rule, the matrix **X** should be full rank. If the rank is not full, it means that some columns of **X** can be expressed as a linear combination of the other columns. In other words, the information contained in a subset of the columns is enough to explain the whole matrix.

If the model is used to characterize a *target resource*, then the columns represent *resources*. If the rank of **X** is not full, it means that some *resources* can be expressed as a linear combination of the other *resources*. E.g. measuring the same phenomenon using two different metrics.

If the model is used to characterize a *target program*, then the columns of **X** represent *benchmarks*. If the rank of **X** is not full, then some *benchmarks* can be expressed as a linear combination of the other *benchmarks*. E.g. two benchmarks are in fact the same program, only using a different input size, and the computation does not change behaviour with a different input.

5.4.2 Norm

The norm of the columns of **X** can be useful in several situations, for example to obtain a simple uni-dimensional measure of the cost of a *program*, or to quickly compare two related *resources*, e.g. the energy consumption on two *computational environments*.

Different norms can be used, depending on the context where they are applied. In this section the most common norm is discussed: the Euclidean Norm.

Norm of a program

If the model is used to characterize a *target program*, the columns of $\mathbf{X}$ represent *benchmarks*. The norm of the j^{th} *program* is simply the norm of the j^{th} column of $\mathbf{X}$, as shown in equation 5.14.

$$\|\mathbf{p_j}\| = \sqrt{\sum_{i=1}^{m}(\mu_{r_i}(p_j))^2} \tag{5.14}$$

Norm of a program

Similarly, if the model is built to characterize a *resource*, the columns of $\mathbf{X}$ represent *resources*, but the norm of a *program* can be calculated using the corresponding row of $\mathbf{X}$.

The norm of a *program* can be used to get a quick idea of a *program*'s cost. Consider three programs p_1, p_2 and p_3, and three resources r_1, r_2, and r_3, the completion time on three different machines. As shown in table 5.4, p_1 is slightly faster than p_2 and p_3 on the first two machines, but is two times slower than p_2 and p_3 on the last machine. Asking "which program is faster" is not well posed and does not have an answer. Instead, asking "which program is usually faster" can be answered: two out of three times p_1 is faster than both p_2 and p_3, but this answer is misleading, because in the single case where p_1 is slower than p_2 and p_3 the difference is considerable. Using the norm it can be seen that p_1 has the larger norm ($\|p_1\| = 2.91$, $\|p_2\| = 2.54$, and $\|p_3\| = 2.51$), providing useful information about their respective performances.

Norm of a resource

If the model is used to characterize a *resource*, the columns of $\mathbf{X}$ represent *resources*. The norm of the j^{th} *resource* is simply the norm of the j^{th} column of $\mathbf{X}$, as shown in equation 5.15.

Table 5.4: Example of *resource* consumption of two *programs*

	p_1	p_2	p_3
r_1	1.1	1.2	1.3
r_2	1.8	2.0	1.9
r_3	2.0	1.0	1.0

$$\|\mathbf{r_j}\| = \sqrt{\sum_{i=1}^{m} \mu_{r_j}(p_i)^2} \tag{5.15}$$

Norm of a resource

Similarly, if the model is built to characterize a *program*, the columns of **X** represent *programs*, but the norm of a *resource* can be calculated using the corresponding row of **X**.

The norm can be used to compare two *resources*. Consider the example reported in table 5.4, imagine the resources to be completion times on three different machines. It is usually interesting to know "which machine is the fastest", performance bench marking attempts to address this issue (e.g. the SPEC CPU suite). However, "fastest" is not well defined, as it depends on the program. The third machine has the lowest completion time for two out of three *programs*, and the first machine has the lowest completion time only for p_1. Therefore, it might seem reasonable to conclude that the third machine is the "fastest". However, considering the norms of the *resources* ($\|r_1\| = 2.08, \|r_2\| = 3.29, \|r_3\| = 2.45$), is clear that the first machine has the best overall performance.

5.4.3 Cosine similarity

Cosine similarity measures the cosine of the angle between two vectors. Cosine similarity is close to 0 when they are orthogonal (least similar), and is close to 1 when they are very close, independently of their module (most similar).

Cosine similarity can be derived using the euclidean dot product, as shown in equation 5.16.

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= \|\mathbf{a}\| \|\mathbf{b}\| \cos\theta \\ \cos\theta &= \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} \end{aligned} \tag{5.16}$$

Cosine similarity

Programs similarity

Cosine similarity can be applied to *programs* as shown in equation 5.17, where $\mathbf{p_i}$ is the vector with all the measures of the i^{th} program for all considered *resources* (the i^{th} column of the $\mathbf{X}$ matrix, when it is built to characterize a *target program*, or the i^{th} row of the $\mathbf{X}$ matrix, when it is built to characterize a *target resource*).

$$\cos\theta_{\mathbf{p_1},\mathbf{p_2}} = \frac{\mathbf{p_1} \cdot \mathbf{p_2}}{\|\mathbf{p_1}\| \|\mathbf{p_2}\|} \tag{5.17}$$

Programs similarity

If two programs have cosine similarity close to 1, they show a similar behaviour on the considered *resources*, i.e. they may use a different amount of *resources*, but in a similar way. If they have similarity close to 0, then they have a very different behaviour with respect to the considered *resources*.

The *benchmarks* used to build the model should have a low similarity. If they have high similarity they are not capturing different aspects of computation.

If programs are too similar, $\mathbf{X}$ will not have full effective rank. Some of the singular values of $\mathbf{X}$ will be small, even if not zero. A good set of *benchmarks* will therefore only include *programs* that have low similarity between each other.

Resources similarity

Cosine similarity can also be applied to *resources* to quantify how similarly two *resources* are used by the *benchmarks*. Equation 5.18 shows how to apply equation 5.16 to *resources*. r_i is the vector containing the measures of all the *benchmark* for the i^{th} *resource* (the i^{th} column of the $\mathbf{X}$ matrix, when it is built to characterize a *target resource*, or the i^{th} row of the $\mathbf{X}$ matrix, when it is built to characterize a *target program*).

$$\cos\theta_{\mathbf{r_1},\mathbf{r_2}} = \frac{\mathbf{r_1} \cdot \mathbf{r_2}}{\|\mathbf{r_1}\|\|\mathbf{r_2}\|} \tag{5.18}$$

Resources similarity

If two *resources* measure the same quantity, they will have similarity equal to 1, even if they are using different scales. For example the completion times on two machines with similar CPUs, same architecture, but with different clock rate, will be different, but the similarity will be close to 1. A similarity near to zero is achieved only if the *resources* have very different behaviour with the selected *benchmarks.*

5.5 Conclusion

In this chapter we have introduced our benchmarking model, from a conceptual perspective. Our model unifies hardware and software characterization in a single approach. The model does not focus exclusively on a particular *resource*, but allows the simultaneous usage of heterogeneous resources, such as completion time, energy consumption, performance counters.

We introduced the notion of *computational patterns*, as ideal sequences of instructions that expose interesting behaviour of programs on computing systems. They can not be directly measured, but they justify the assumption that *programs* can be explained as linear combinations of *computational patterns.*

Chapter 6

Solvers

In this chapter we present a few of the possible algorithms that can be used to create *surrogates* and *predictors*.

Simplex is discussed in section 6.1. This algorithm explains the *target program* as a non negative composition of the *benchmarks* (or the *target resource* as a non negative composition of *resources*), and could be used in case the *benchmarks* are close enough to pure *computational patterns*.

Linear regression is discussed in section 6.2. Linear regression is the simplest *solver*. Because of its simplicity it produces intuitive *surrogates*, that can be used to characterize the *target program* or the *target resource*. Nonetheless, it has good predictive capabilities. Most of the experimental section uses Linear Regression as *solver*.

Non-Negative Matrix Factorization is discussed in section 6.3. This algorithm offers a possible decomposition of *programs* in terms of *computational patterns*.

6.1 Simplex

Under the assumption that *computational patterns* can be fully captured by micro-benchmarks, and that each *computational pattern* maps to a different micro-benchmark (i.e. every micro-benchmark captures a different *computational pattern*), we can assume the *surrogate* β to be a vector of non negative numbers, such that multiplying $\mathbf{X}$ by β we obtain $\mathbf{y}$, plus some model error, as shown in equation 6.1.

$$\mathbf{X}\beta = \mathbf{y} + \epsilon \tag{6.1}$$

Simplex

The *surrogate* needs to contain non negative values, because they represent the amount of *computational patterns* present in the *target program*, there can be zeros, but not negative numbers, or the model would be inconsistent (a negative quantity has no justification in this model). Non integer values are allowed assuming that the *benchmarks* used in the basis $\mathbf{X}$ can contain several instances of the corresponding *computational patterns*.

We are interested in finding the vector $\mathbf{x}$ that minimizes the norm of the error ϵ. As error we use the Manhattan norm (defined in equation 6.2). Finding the β that minimizes the Manhattan norm of ϵ can be transformed into a linear programming problem.

$$\|\epsilon\| = \sum_{i}^{n} |\epsilon_i| \tag{6.2}$$

Manhattan norm of ϵ

6.1.1 The simplex algorithm

The simplex algorithm is a popular algorithm (Murty, 1983) that does not directly use simplices, but it operates on simplicial cones (the corners of the feasible region). The simplex algorithm solves linear programming (LP) problems expressed in the canonical form:

$$\begin{aligned} &\text{maximize} && \mathbf{c}^\top \mathbf{w} && (6.3)\\ &\text{subject to} && \mathbf{A}\mathbf{w} = \mathbf{b} && (6.4)\\ &\text{and} && \mathbf{w} \geq 0 && (6.5) \end{aligned}$$

LP canonical form

A solution of the LP problem consists of a $\mathbf{w}$ vector that satisfies the constraints expressed by inequalities 6.4 and 6.5 and maximizes the linear objective function 6.3.

The canonical form is also called the *primal* problem, and can be converted in the *dual* problem, as shown in equations 6.6, 6.7, and 6.8.

$$
\begin{aligned}
&\text{minimize} && \mathbf{b}^\top \mathbf{z} && (6.6)\\
&\text{subject to} && \mathbf{A}^\top \mathbf{z} = \mathbf{c} && (6.7)\\
&\text{and} && \mathbf{z} \geq 0 && (6.8)
\end{aligned}
$$

LP dual form

The strong duality theorem states that if there is an optimal solution $\hat{\mathbf{w}}$, then the dual has also a an optimal solution $\hat{\mathbf{z}}$, and $\mathbf{c}^\top \hat{\mathbf{w}} = \mathbf{b}^\top \hat{\mathbf{z}}$.

The objective function we want to minimize is the Manhattan norm of ϵ, the sum of the absolute values of the elements of the vector. However, the absolute value is not a linear function, but it can be made linear introducing 2 slack variables for each ϵ_i as shown in equation 6.10.

$$
\begin{aligned}
& && \epsilon = \mathbf{X}\beta - \mathbf{y} && (6.9)\\
&\text{minimize} && \min \sum_i^n |\epsilon| = \min \sum_i^n (z_i + z_{n+i}) && (6.10)\\
&\text{subject to} && z_i \geq \epsilon_i \quad \forall i && (6.11)\\
& && z_{n+i} \geq -\epsilon_i \quad \forall i && (6.12)\\
& && \beta_i \geq 0 \quad \forall i && (6.13)
\end{aligned}
$$

LP dual form

We can now express our problem in the dual form: equation 6.9 shows how to define ϵ in terms of the matrix $\mathbf{X}$ (the measures of *resource consumption* of the *benchmarks*) and vector $\mathbf{y}$ (the corresponding measures of the *target program*), as defined in section 5.2; equations 6.11, 6.12, and 6.13 show the constraints.

The problem in dual form can be transformed in the canonical form, and solved using the simplex algorithms.

6.1.2 Geometric interpretation

The constraints specify a convex polytope, possibly unbounded, over which the objective function is maximized. If the model is used to characterize a *target program*, the columns of the matrix $\mathbf{X}$ contain measures of the *benchmarks*. If the columns are linearly independent, the *benchmarks* are vectors in the *resource consumption vector space*, and they define the vertices of the convex polytope that defines the feasible region. If the vector that

Table 6.1: Fictional example for the Simplex *solver*

program	resource 1	resource 2
a	1.0	0.1
b	0.1	1.0
c	0.5	0.5
d	0.8	0.0

identifies the *target program* in the *resource consumption vector space* lies inside the convex polytope, an optimal solution β with no error can be found. If the *target program* lies outside the convex polytope, the solution will have some error, and will lie on a face or on an edge of the convex polytope.

Figure 6.1 shows an example, where *programs* a and b are used as basis (the matrix $\mathbf{X}$). The points in the graph represent *programs* in the *resource consumption vector space*. Table 6.1 shows the *resource* consumption of the four *programs*. The grey region is the convex polytope that identifies the area of the space that can be expressed as a linear combination of *programs* a and b (the basis). The *program* c lies within the convex polytope, it can therefore be expressed as a positive linear combination of the *programs* of the basis. The *program* d lies outside the convex polytope, it can not be expressed as a positive linear combination of the *programs* of the basis, and its *surrogate* will contain some model error.

From the previous example is clear that collinearity in the basis should be avoided. If a *program* in the basis can be expressed as a linear combination of the other *programs*, it lies inside the convex polytope, so it can be removed without loss of expressiveness.

If a program lies outside the convex polytope, it should be used as a program of the *basis*, eventually removing other *programs* from the basis, because it improves the expressiveness, allowing a larger set of programs to be expressed without error. When this occurs, it means that the *programs* used in the basis were not able to capture some *computational pattern* that is captured by the new *program*.

The example can be extended to higher dimensional *resource consumption vector spaces*. If the number n of *programs* used in the basis is lower than the number m of *resources*, the polytope will be a subspace. For example, if $n = m - 1$ the polytope will lie on an hyperplane that splits the space in 2 halves. Unless all the points lie on the hyperplane, more specifically in the polytope, the model will contain some error. If a point is outside the polytope and is not collinear with the basis, the only way to produce a

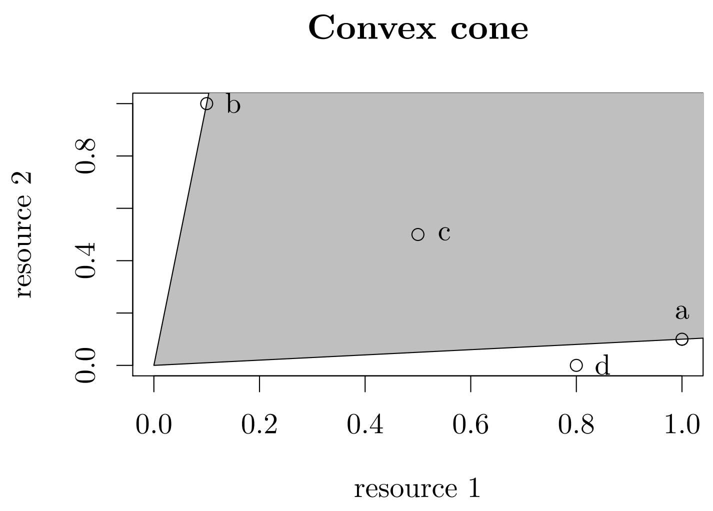


Figure 6.1: Convex polytope of solutions in the *resource consumption vector space*

Table 6.2: Data for the fictional example for the Simplex *solver*

program	instructions	cache-misses	completion-time
a	100000	100	993.72
b	20000	1000	1132.13
c	30000	500	932.69

surrogate without error is to extend the basis with the new program.

In section 10.1.1 we present a simple experiment where *simplex* is used to characterize a *program*. The *simplex* can also be used to characterize a *target resource*. In this case the columns of $\mathbf{X}$ contain the *resource* consumption, the rows of $\mathbf{X}$ the *benchmarks*, and the $\mathbf{y}$ vector contains the known measures for the *target resource*. The *surrogate* will express the *target resource* as a positive linear combination of the *resources*. Consider an fictional example with the measures shown in table 6.2. We measure the number of instructions, the number of cache-misses, and the completion time, on three programs a, b, and c. We want to characterize completion time using instructions and cache-misses.

The *simplex solver* characterizes completion time as shown in equation

6.14. The *solver* found the same coefficients we used to create the data in table 6.2: time measured in millisecond, every instruction takes in average 10^{-2}msec and every cache miss takes 1msec. We also added some multiplicative Gaussian noise with mean 1 and standard deviation 0.5 on the $\mu_{\text{instructions}}$ coefficient, to simulate the effect of non measured differences in the instruction costs on the completion time.

$$\mu_{\text{time}} = \mu_{\text{instructions}} 10^{-2} + \mu_{\text{cache-misses}} \tag{6.14}$$

Example *surrogate*

Figure 6.2 show the *resources* in the *programs vector space*. The axes are the *programs*, the *resources* are points in the space. Completion time and cache-misses are close to the origin because they are several order of magnitude smaller than instructions; for the same reason they are also very far from instructions. Completion time lies near to the plane identified by the *resources* in the basis (instructions and cache-misses), close to the cone. The model error is therefore small. Analysing the cosine similarity between the *resources* we see that the *resources* used in the basis have low similarity (see equation 6.15). Completion time is very similar to cache-misses (see equation 6.16). Their vectors in the *programs vector space* are close. This is expected, because the coefficient relative to cache-misses used to generate completion time is 3 orders of magnitude larger than the coefficient relative to instructions.

$$\frac{\mu_{\text{instructions}} \cdot \mu_{\text{cache-misses}}}{\|\mu_{\text{instructions}}\| \|\mu_{\text{cache-misses}}\|} = 0.3771 \tag{6.15}$$

$$\frac{\mu_{\text{completion-time}} \cdot \mu_{\text{cache-misses}}}{\|\mu_{\text{completion-time}}\| \|\mu_{\text{cache-misses}}\|} = 0.8537 \tag{6.16}$$

Cosine similarity between *resources*

More details about the properties of the convex polytopes that span from positive linear dependence can be found in Davis (1954).

6.1.3 Limits

The simplex algorithm is a powerful tool, used to solve most of the linear programming problems. When used as a *solver*, it offers a simple and straightforward interpretation of the *surrogate*. However, it is based on a strong assumption, that it is possible to create and measure *benchmarks* that

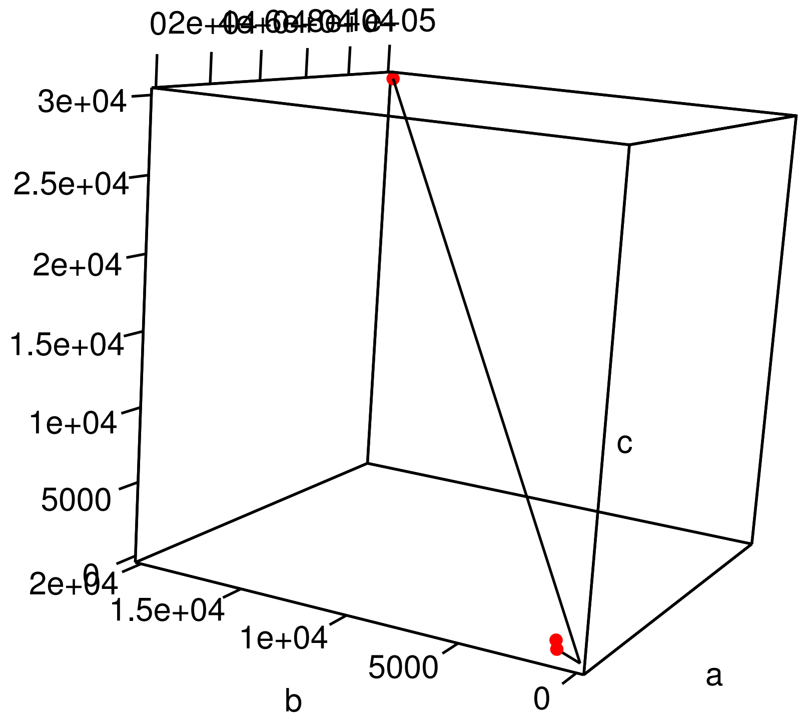


Figure 6.2: *Resources* in the *Programs Vector Space*

perfectly embody *computational patterns*. As discussed in section 5.3, this is generally not true. This approach is therefore sound only in controlled experiments, where is reasonable to assume that the interesting *computational patterns* are captured by the *benchmarks* in the basis. With real world *programs*, the *computational patterns* involved will be a very large number, possibly larger than the number of measured *resources*. The basis would probably need to be extremely large, leading to a high risk of over-fitting, and unreadable *surrogates*.

In next section we relax the non negative constraints on the *surrogates*, allowing the *solver* to find solutions outside the polytope (but still on its same subspace).

6.2 Linear regression

In the previous section we explored the use of *simplex* as a *solver*, under the assumption that it is possible to create and measure *benchmarks* that perfectly embody *computational patterns*. As discussed in 5.3, this is a very optimistic assumption. In this section we remove this assumption. Differently than with the *simplex solver*, we do not consider the *benchmarks* that form the matrix $\mathbf{X}$ equivalent to *computational patterns*. However, both the *benchmarks* and the *target program* are still considered formed by linear combinations of *computational patterns*.

The non-negativity constraints over the elements of the *surrogate* β is removed, which means that the *surrogates* do not have to be inside the polytope identified by the *benchmarks*. In the example we provided for the *simplex solver* (6.1)), where a and b are the *benchmarks*, not only c, but also d can be expressed without error.

6.2.1 Example

To understand why negative values in the *surrogate* are admissible, and what this means in terms of *computational patterns*, consider the following example.

We measure 3 *programs*:

- *program a* performs a CPU intensive task: multiplication of two small matrices. This task also uses a large amount of CPU, and a small amount of memory;
- *program b* performs a memory intensive task: sum of two large matrices. This task uses a large amount of memory, and a small amount of

CPU;

- *program d* performs a CPU intensive tasks: calculates π. This task does not use any memory, only CPU.

The measured *resources* are:

- *resource 1*: number of arithmetic instructions
- *resource 2*: number of cache misses

Consider the following (oversimplified) *computational patterns*:

- *computational pattern 1*: CPU usage
- *computational pattern 2*: memory usage

Program d is composed exclusively of the *computational pattern 1*, without any usage of *computational pattern 2*. *Program a* and *program b* are composed of both patterns, in different measures.

Imagine using *program a* and *program b* as the *benchmarks* to form the basis $\mathbf{X}$, and to use *program d* as our *target program*. The three *programs* in the *resource usage space* could look like figure 6.1 (the example in the *simplex solver* section). *Program d* is outside the convex cone. Therefore it can not be expressed as a positive linear combination of *program a* and *program b*. However, it can be expressed by the following surrogate: $\beta = (0.81, -0.8)$.

The *surrogate* can be interpreted as follows: "*program d* can be decomposed as 0.81 instances of *program a*, subtracting 0.08 instances of *program b*". If we consider what this means in terms of the *computational patterns*, it translates to: "*program d* can be characterized as composed by 0.81 times the *computational patterns* present in *program a*, subtracting 0.08 times the *computational patterns* present in *program b*". This interpretation expresses the fact that *program d* is closer to the *computational pattern 1* than both *program a* and *program b*. Therefore, we have to cancel the memory usage of *program a*, using *program b*, to describe *program d* in terms of *a* and *b*.

Similar considerations apply when expressing a *target resource* in terms of other *resources*.

Allowing negative values in the *surrogate* is therefore consistent with the assumption that all *programs* are composed of *computational patterns*, and linear regression can be used to find the *surrogate* of the *target program* (or of the *target resource*).

6.2.2 Linear regression

Over the years, linear modelling has been largely employed to describe or estimate the dependency of certain phenomena from a set of known variables in most of the scientific research fields (Blanco-Fernndez et al., 2013; Roundy and Frank, 2004; Srinivasan and Bellur, 2014; Farahnakian et al., 2013; Isobe et al., 1990). The linear model presented in this work is an evolution of Morelli and Cisternino (2014), where we have shown that linear regression can be used to predict the energy consumption and the completion time of real world algorithms. The purpose of linear regression is to model the relationship between a *dependent variable*, also called *measured variable* or *regressand*, and a set of *independent variables*, also known as *explanatory variables* or *regressors*.

Linear regression assumes a linear relationship between the dependent variable and the explanatory variables. This does not imply that only linear behaviour can be explained: as long as the independent variables expose the same kind of non-linearity that characterizes the dependent variable, linear regression represents a suitable method. For many algorithms the completion time is non-linear with respect to the size of the input, but can be linear with respect to certain features. For example, matrix multiplication completion time is non-linear with respect to the size of the input but is linear with respect to the number of arithmetic operations. Using the number of arithmetic operations number as an explanatory variable, linear regression is able to build a model capable of expressing completion time as the dependent variable.

Linear regression is defined by formula 6.17

$$\mathbf{y} = \mathbf{X}\beta + \epsilon \tag{6.17}$$

Linear regression

Where:

- $\mathbf{y}$ is the dependent variable
- $\mathbf{X}$ is the matrix of the explanatory variables
- β is the vector that contains the regression coefficients
- ϵ is the error term (fitting residuals)

6.2.3 Algebraic characterization of the solver

In this section is shown how to use Linear Regression to create a *solver* for our model, and the algebraic structure and properties are discussed.

The *Solver* equation 5.5 immediately adapts to the *Linear Regression* 6.17. If $\mathbf{X}$ has linearly independent columns, and has at least as many rows as columns ($m \geq n$), then the Moore-Penrose pseudo-inverse $\mathbf{X}^{+}$ of $\mathbf{X}$ can be used to estimate β. The pseudo-inverse can be calculated using equation 6.18.

$$\mathbf{X}^{+} = (\mathbf{X}^{\top}\mathbf{X})^{-1}\mathbf{X}^{\top} \tag{6.18}$$

Left inverse

$\mathbf{X}^{+}$ can be estimated using the singular value decomposition (SVD) as shown in equation 6.19, where $\mathbf{\Sigma}^{+}$ is calculated by taking the reciprocal of each non-zero element on the diagonal, and replacing the others with zeros. To ensure numerical stability diagonal elements close to zero (in the same order of magnitude as the numerical precision used) will also be replaced with zeros.

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\top} \tag{6.19}$$

$$\mathbf{X}^{+} = \mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{\top} \tag{6.20}$$

SVD estimation of the pseudoinverse

$\mathbf{X}^{+}$ can be used to estimate β, as show in equation 6.21.

$$\hat{\beta} = \mathbf{X}^{+}\mathbf{y} \tag{6.21}$$

Estimation of the surrogate

Is therefore straightforward to map equation 6.21 to the f function from the solver equation 5.5.

The function g of the predictor equation 5.6 is simply a vector dot product between the vector with measures of the *benchmarks* on the *resources* and the β surrogate, as shown in equation 6.22.

$$p = \mathbf{x}\beta \tag{6.22}$$

Predictor for linear regression

Table 6.3: Fictional example for the Linear Regression *solver*

	p_1	p_2	p_3	p_T
CPU	1.0	0.1	0.1	1.0
Mem	0.1	1.0	0.5	2.0
I/O	0.0	0.0	1.0	3.0
Energy	10.1	2.0	2.5	$\mu_E(p_T)$

Extracting a surrogate and predicting measures is easy and computationally inexpensive.

As an example, consider table 6.3, that contains fictional measures of three *programs*. The measured *resources* are some metric of CPU, memory, and I/O. Program p_1 is CPU bound, p_2 memory bound, and p_3 is I/O bound. The *target program* is uses both CPU, memory and I/O. The *target resource* is Energy, the last row of X.

Consider now building the model to characterize p_T. The resulting $\mathbf{X}$ and $\mathbf{y}$ vectors are shown in equation 6.23. The *surrogate* of p_T (equation 6.25) shows that the program can be explained with a prevalence of p_3 (the I/O bound *program*), but both the CPU and memory bound programs are present. The measures of the Energy consumption of the *benchmarks* and the *surrogate* are multiplied to estimate the energy consumption of the *target program* p_T (equation 6.26).

$$\mathbf{X} = \begin{pmatrix} 1.00 & 0.10 & 0.10 \\ 0.20 & 1.00 & 0.50 \\ 0.00 & 0.00 & 1.00 \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} 1.00 \\ 2.00 \\ 3.00 \end{pmatrix} \tag{6.23}$$

$$\mathbf{X}^{+} = \begin{pmatrix} 1.02 & -0.10 & -0.05 \\ -0.20 & 1.02 & -0.49 \\ 0.00 & 0.00 & 1.00 \end{pmatrix} \tag{6.24}$$

$$\beta = \mathbf{X}^{+}\mathbf{y} = \begin{pmatrix} 1.02 & -0.10 & -0.05 \\ -0.20 & 1.02 & -0.49 \\ 0.00 & 0.00 & 1.00 \end{pmatrix} \begin{pmatrix} 1.00 \\ 2.00 \\ 3.00 \end{pmatrix} = \begin{pmatrix} 0.66 \\ 0.37 \\ 3.00 \end{pmatrix} \tag{6.25}$$

$$\mathbf{x} = \begin{pmatrix} 10.10 & 2.00 & 2.50 \end{pmatrix}$$

$$\mu_E(p_T) = \mathbf{x}\beta = \begin{pmatrix} 10.10 & 2.00 & 2.50 \end{pmatrix} \begin{pmatrix} 0.66 \\ 0.37 \\ 3.00 \end{pmatrix} = 14.93 \tag{6.26}$$

Example for target program

Consider now building the model to characterize the *target resource* Energy, using the same data as in the previous example. The resulting **X** and **y** vectors are shown in equation 6.27. The *surrogate* of Energy (equation 6.29) shows how much Energy is consumed by each unit of the other *resources*: approximately 10 energy units are used by each CPU unit, 1 energy unit are used for both each memory and I/O units. The measures of the program p_T for CPU, memory, and I/O are multiplied by the *surrogate* of the Energy to estimate the energy consumption of the *target program* p_T (equation 6.30). The prediction for the *target resource* of the *target program* is the same as in the previous example (equation 6.26).

$$\mathbf{X} = \begin{pmatrix} 1.00 & 0.20 & 0.00 \\ 0.10 & 1.00 & 0.00 \\ 0.10 & 0.50 & 1.00 \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} 10.10 \\ 2.00 \\ 2.50 \end{pmatrix} \tag{6.27}$$

$$\mathbf{X}^{+} = \begin{pmatrix} 1.02 & -0.20 & 0.00 \\ -0.10204082 & 1.02 & 0.00 \\ -0.05102041 & -0.49 & 1.00 \end{pmatrix} \tag{6.28}$$

$$\beta = \mathbf{X}^{+}\mathbf{y} = \begin{pmatrix} 1.02 & -0.20 & 0.00 \\ -0.10204082 & 1.02 & 0.00 \\ -0.05102041 & -0.49 & 1.00 \end{pmatrix} \begin{pmatrix} 10.10 \\ 2.00 \\ 2.50 \end{pmatrix} = \begin{pmatrix} 9.80 \\ 1.02 \\ 1.01 \end{pmatrix} \tag{6.29}$$

$$\mathbf{x} = \begin{pmatrix} 10.1 & 2.00 & 2.50 \end{pmatrix}$$

$$\mu_E(p_T) = \mathbf{x}\beta = \begin{pmatrix} 10.1 & 2.00 & 2.50 \end{pmatrix} \begin{pmatrix} 9.90 \\ 1.01 \\ 1.00 \end{pmatrix} = 14.93 \tag{6.30}$$

Example for target resource

In the first example the *surrogate* describes p_T as a linear combination of p_1, p_2, and p_3. Choosing representative *programs* as *benchmarks* will ensure that the *surrogate* will provide information about the nature of the *target program.* In this example it roughly shows the proportion of CPU vs memory vs I/O usage.

In the second example the *surrogate* shows how much Energy is consumed by each measured unit of the CPU metric, memory metric, and I/O metric. In chapter 13 it will be shown how this approach can be used to estimate the time needed by each processor instruction, or by each memory access.

The $\mathbf{X}^{+}$ matrix can be seen as a linear function that maps y (the measures of the *target program*, or the measures of the *target resource*) to its surrogate β.

6.2.4 Hardware and Software intrinsic relationship

The previous example shows the close relationship between the use of the model to characterize a *target program*, and a *target resource*. As already stated in section 5.2.3, the models used to characterize the *target program* and the *target resource* have many connections:

- the matrix $\mathbf{X}$ used in the model to characterize the *target program* is equal to the transpose of the matrix $\mathbf{X}$ used in the model to characterize the *target resource*;
- the vector $\mathbf{y}$ used in the model to characterize the *target program* is equal to the transpose of the vector $\mathbf{x}$ used in the model to characterize the *target resource*;
- the vector $\mathbf{x}$ used in the model to characterize the *target program* is equal to the transpose of the vector $\mathbf{y}$ used in the model to characterize the *target resource*.

As shown in the previous example, if the number of rows is equal to the number columns of $\mathbf{X}$ (i.e. is a square matrix), the prediction of the *target resource* for the *target program* in the two models is the same.

6.2.5 Assumptions and limits

Ordinary linear regression is sensitive to outliers in the dependent variable (Anscombe, 1973). Completion time is likely to contain outliers, mainly caused by certain sporadic effects that are not considered in the model or to the instability of the system where the measurement is performed. When it is reasonable to expect the presence of outliers in the measures, robust variants of the ordinary least squares should be used instead of ordinary least squares. This will help mitigate the effect of outliers that can have a severe, negative impact on the quality of the model built. As shown by Hoaglin et al. (2011) and Fox (1997), iterative approaches can help when noise is present in the data.

As discussed in section 6.2.6, linearity is assumed between the dependent and the independent variables. Is not always evident if linearity is a reasonable choice, depending on the *target program* or *target resource* being characterized. In chapter 13 we show that, with the right choice of explanatory variables, even a complex non linear phenomenon such as completion time in a highly parallel environment, such heterogeneous GPU computing, can be successfully modelled by linear regression.

Linear regression assumes that the observed values $\mathbf{y}$ differ from the linear combination of $\mathbf{X}\beta$ by an additive noise, of independent, identically distributed Gaussian distribution with zero mean. The residuals therefore are assumed to be independent and zero centred. This can be a strong assumption when dealing with measures of physical quantities, e.g. time or energy. If an instrument used to measure is biased, the error will not be zero centred, and linear regression will bias the solution accordingly. If an unexpected correlation between error is present in the data, linear regression will offload that information on some combination of the independent variables. Residual analysis can usually help identify these conditions, showing non uniform or not zero centred distribution of residuals.

Linear regression also assumes constant variance of the residuals, also called homoskedasticity. This assumption is often not verified. For example completion time could be influenced by unexpected processes running in the operating system at the same time as the measured *program*. In chapter 13 we address a non constant variance in measurement errors normalizing measures by their variance. This is the most common approach to solve this problem. Residual analysis shows the presence of heteroskedasticity.

6.2.6 Residuals analysis

When the number of rows of $\mathbf{X}$ is larger than the number of columns, i.e. when $m \gg n$, is useful to analyse the residuals ϵ.

One of the assumptions of linear regression is that error, i.e. the vector ϵ of the regression residuals, needs to have zero mean, and should not have any correlation with the explanatory variables. Whenever the residuals violate the assumptions, the error can not be seen as noise, indicating that the regression model is not capturing all the relevant information.

To illustrate this phenomenon, consider the example shown in figure 6.3. We generated two vectors with 1000 random numbers, uniformly distributed between 0 and 1000. The dependent variable is non linearly depending on both the vectors: $y = k_1x_1^2 + k_2x_2^2$. We also generated two vectors $x_3 = x_1^2$, and $x_4 = x_2^2$. A uniformly distributed noise, between -100 and 100, was added to y.

In the first row of figure 6.3, linear regression attempts to model a non linear behaviour using a linear combination of linear regressors x_1 and x_2.

The residuals are not uniformly distributed around zero, a non linear trend is clear: the predicted values are initially overestimated (until approximately 2.5e6), then underestimated. The presence of such a strong trend indicates that the information not captured by the linear regression can not be explained as noise. Therefore, the independent variable contains a non linear behaviour not explained by the regressors.

In the second row of figure 6.3 the non-linear vectors x_3 and x_4 were used as regressors. The predictions have the same accuracy for all ranges of values. The residuals are uniformly distributed around zero. Such a distribution of the residuals indicates that the information not captured by the linear regression can be considered as noise. Therefore, all the interesting behaviour of the independent variable is correctly modelled by the regressors and the regression coefficients.

This example shows how important is the choice of the basis. It is often possible to measure a relevant behaviour of the *computational environment* using different but related metrics. Consider for example an algorithm that performs naive matrix multiplication on a couple of matrices passed as arguments. If we use the size of the matrix as a regressor, the completion time will not be succesfully modelled, because they are not in a linear relationship. Instead, using the number of CPU instructions as a regressor, completion time will be succesfully modelled.

6.2.7 Algebraic characterization of computational patterns

In this section we show how a *target program* can be expressed as a linear combination of *benchmarks*, under the assumption that every *programs* can be expressed as linear combinations of *computational patterns*.

Consider the matrix $\mathbf{W}$, containing the resource consumption of every *computational pattern*. Every row of $\mathbf{W}$ corresponds to a *resource* and every column of $\mathbf{W}$ to a *computational pattern*. Consider now the matrix $\mathbf{H}$, containing the composition of *benchmarks*, expressed as linear composition of *computational patterns*. Each row of $\mathbf{H}$ corresponds to a *computational pattern* and every column to a *benchmark*. Then equation 6.31, already presented in section 5.3, shows how $\mathbf{W}$ and $\mathbf{H}$ can be used to define the matrix $\mathbf{X}$ of the measures of the *benchmarks*, as defined in section 5.2.1.

$$\mathbf{X} = \mathbf{WH} \tag{6.31}$$

Benchmarks as compositions of patterns

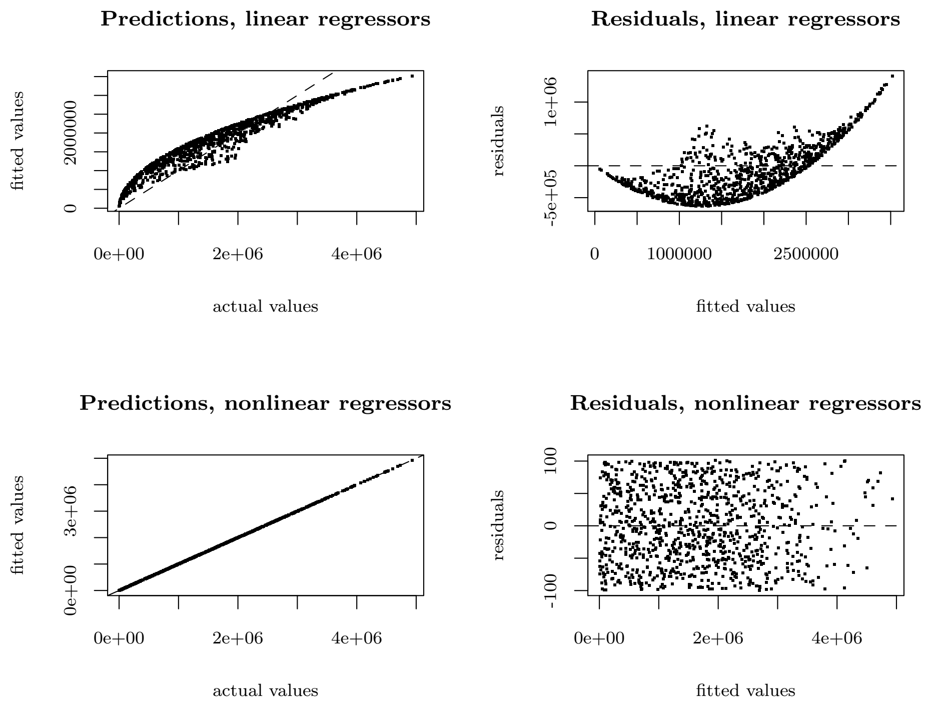


Figure 6.3: Example of predicted values and residuals using linear and non linear regressors

If every *program* can be expressed linearly in terms of *computational patterns*, then a vector **h** that expresses the *target program* as a linear combination of the *computational patterns* must also exist, as show in equation 6.32.

$$\mathbf{y} = \mathbf{W}\mathbf{h} \tag{6.32}$$

Target program as composition of patterns

As explained in section 5.3, *computational patterns* are supposed to be linearly independent. They are abstract programs, ideally in a number much larger than both the measures *resources* and the number of used *benchmarks.*

If **H** is full rank, i.e. the *benchmarks* are composed of different combinations of *computational patterns*, then linear regression can find two vectors β and ϵ that satisfy equation 6.33.

$$\mathbf{h} = \mathbf{H}\beta + \epsilon' \tag{6.33}$$

Surrogate from compositional patterns

W is a linear transformation from the *computational pattern space*, containing linear combinations of *computational patterns*, to the *resource space*, containing measures of *resources* used by the corresponding *program.* It maps the matrix **H** into **X** and **h** into **y**. Equation 6.34 shows that the β vector, from equation 6.33, is the *surrogate*, as presented in section 6.2.

$$\begin{aligned} \mathbf{h} &= \mathbf{H}\beta + \epsilon' \\ \mathbf{W}\mathbf{h} &= \mathbf{W}\mathbf{H}\beta + \mathbf{W}\epsilon' \\ \mathbf{y} &= \mathbf{X}\beta + \epsilon \\ \text{where} \quad \epsilon &= \mathbf{W}\epsilon' \end{aligned} \tag{6.34}$$

Equivalence of surrogates

As explained in section 5.3, *compositional patterns* are not actual *programs*, and they can not be directly measured. Therefore neither **W**, **H** nor **h** can be calculated. In section 6.3 Non Negative Matrix Factorization is used to model those matrices, assuming to know the number of *hidden factors* (the *computational patterns*).

6.3 Non negative matrix factorization

Non Negative Matrix Factorization (NMF) is a recent technique widely used to find hidden features in data in very different fields, from music analysis (Smaragdis and Brown, 2003) to document clustering (Xu et al., 2003).

For example NMF could be used to analyse newspapers articles to find topics, given the occurrence of words, i.e. certain combinations of words tend to appear together. NMF takes as input a matrix containing the frequency of each word, for each article, and outputs two matrices, one matrix containing all possible topics, with the frequency of each word for each topic, and the other matrix containing the occurrences of topics in articles.

NMF is defined by equation 6.35, where **W** is called the *weights* matrix (or *meta-genes* matrix), and **H** is called the *hidden factors* matrix (or *meta-gene expression profiles* matrix), and **X** is the matrix for which we want to find the hidden factors.

$$\mathbf{W}\mathbf{H} \approx \mathbf{X} \tag{6.35}$$

NMF

Where **X** is a $n \times p$ matrix, and **W** and **H** are respectively $n \times r$ and $r \times p$, and r is usually $r \ll \min(n, p)$.

NMF minimizes a cost function that includes the distance D between **X** and **WH**, and an optional regularization function R that ensures certain properties on **W** and **H**, as shown in equation 6.36.

$$\arg\min_{\mathbf{W},\mathbf{H}}(D(\mathbf{X}, \mathbf{W}\mathbf{H}) + R(\mathbf{W}, \mathbf{H}))) \tag{6.36}$$

NMF cost function

Typical distance functions D are the Frobenius distance or the Kullback-Leibler divergence (LEE, 2001).

The regularization function R can be used when **W** and **H** should be chosen, within the solution space, to have certain properties such as smoothness or sparsity (Cichocki et al., 2008).

NMF is implemented with iterative algorithms, using multiplicative update rules, that are shown to find local optima (LEE, 2001).

The choice of the initial values for **W** and **H** is crucial for the performance of the algorithm and the quality of the solution (that is not guaranteed to be a global optimum). Several seeding methods are used: random values from a uniform distribution; the result of Independent Component Analysis (Marchini et al., 2013); non negative double singular value decomposition (Boutsidis and Gallopoulos, 2008); manually provided fixed seeds.

As we have shown in section 6.2.7, that the *weights* matrix $\mathbf{W}$ and the *hidden factors* matrix $\mathbf{H}$ can be interpreted as a decomposition of the *programs* in terms of *computational patterns*. Equation 6.35 can be seen as an approximation of equation 5.13, where $\mathbf{X}$ contains the measures of the *programs*, $\mathbf{W}$ contains the measures of the *computational patterns* and $\mathbf{H}$ the decomposition of each *program* (each column of $\mathbf{X}$) in linear combinations of *computational patterns*.

6.3.1 The surrogate

When using *NMF* as a *solver* to characterize a *target program*, the input matrix for the *NMF* algorithm is a matrix $\bar{\mathbf{X}}$ made by concatenating the vector $\mathbf{y}$ as a new column of the measurement matrix $\mathbf{X}$, by convention the rightmost column. The *NMF* algorithm outputs the matrices $\mathbf{W}$ and $\bar{\mathbf{H}}$. The rightmost column $\mathbf{h}$ of the $\bar{\mathbf{H}}$ matrix contains the vector that expresses the *target program* as a linear combination of the *computational patterns*. The vector $\mathbf{h}$ has the same meaning as in equation 6.34. We call $\mathbf{H}$ the matrix $\bar{\mathbf{H}}$ without the rightmost column. The columns of $\mathbf{H}$ are the vectors that express the *benchmarks* as linear combinations of the *computational patterns*.

The *surrogate* β is the pair of matrices $\mathbf{W}$ and $\bar{\mathbf{H}}$, as shown in equation 6.37.

$$\mathbf{W}\bar{\mathbf{H}} \approx \bar{\mathbf{X}} \tag{6.37}$$

NMF for the *solver*

The matrix $\mathbf{W}$ can be seen as a function that maps a vector that expresses a *benchmark* as a linear combination of *computational patterns* into a vector that contains the *resource consumption* of that *benchmark*. The columns of $\mathbf{W}$ can be seen as the characterization of the *computational patterns* in terms of *resource consumption*.

It is interesting to notice that the *surrogate* created using *NMF* does not only characterize the *target program* (or the *target resource*), it also attempts an estimation of the *computational patterns*, and characterizes the *benchmarks* (or the other *resources*). In section 10.1.2 we show the expressiveness of the *surrogate* created with *NMF*.

6.3.2 The predictor

The *predictor* is found by determining a new row $\mathbf{w}$ of the matrix $\mathbf{W}$, such that the vector $\mathbf{x}$ containing the *measures* of the *programs* for the *target resource* is equal to the multiplication of the vector $\mathbf{w}$ and the matrix $\mathbf{H}$, as shown in equation 6.38 and equation 6.39, where $\mathbf{H}^{+}_{\text{right}}$ is the right pseudo-inverse of $\mathbf{H}$.

$$\mathbf{x} = \mathbf{wH} \tag{6.38}$$

$$\mathbf{w} = \mathbf{xH}^{+}_{\text{right}} \tag{6.39}$$

$$p = \mathbf{wh} \tag{6.40}$$

predictor using NMF

The vector $\mathbf{w}$ can be seen as an additional row of the matrix $\mathbf{W}$. When the *solver* is used to characterize a *target program*, the vector $\mathbf{w}$ contains the measures of the *target resource* for the *computational patterns* (when the *solver* is used to characterize a *target resource*, the vector $\mathbf{w}$ contains the decomposition of the *target program* in *computational patterns*). By right multiplying $\mathbf{w}$ by $\mathbf{H}$ we obtain the measures of the *target resource consumption* of the *benchmarks* (when the *solver* is used to characterize a *target resource*, we obtain the measures of the *target program* for the *resources*). This information is known, therefore we can compute $\mathbf{w}$ inverting $\mathbf{H}$. Once both $\mathbf{w}$ and $\mathbf{h}$ are known, the prediction can be computed multiplying $\mathbf{w}$ by $\mathbf{h}$ (as defined in 6.3.1), as show in equation 6.40. This gives us the prediction of the *target resource* for the *target program* (this is the same also when the *solver* is used to characterize a *target resource*).

The right pseudo-inverse of $\mathbf{H}$ can be computed noting that $\mathbf{H}$ has more columns (the number of *benchmarks*) than rows (the number of *computational patterns*), because *NMF* requires the number of hidden factors to be smaller than the columns of $\mathbf{X}$. Assuming that the *benchmarks* contain enough variety, the hidden factors will not be linearly dependent, and $\mathbf{H}$ will be full row rank. Matrices with full row rank have right inverses $\mathbf{H}^{\mathbf{-1}}_{\text{right}}$ with $\mathbf{HH}^{\mathbf{-1}}_{\text{right}} = \mathbf{I}$. The right pseudo-inverse can then be computed with equation 6.43.

The pseudo-inverse can be estimated using the *QR* decomposition (equation 6.41), resulting in equation 6.47: in equation 6.42 we transpose $\mathbf{QR}$; in equation 6.43 we show the formula for the right pseudoinverse of $\mathbf{H}$; in equation 6.44 we substitute equations 6.41 and 6.42 into equation 6.43; in equation 6.45 we simplify $\mathbf{Q}^\top\mathbf{Q}$; in equation 6.46 we apply the inverse operator to $\mathbf{R}^\top\mathbf{R}$; in equation 6.47 we simplify $\mathbf{R}\mathbf{R}^{\mathbf{-1}}$.

$$\mathbf{H}^\top = \mathbf{Q}\mathbf{R} \tag{6.41}$$

$$\mathbf{H} = \mathbf{R}^\top\mathbf{Q}^\top \tag{6.42}$$

$$\mathbf{H}^{\mathbf{-1}}_{\text{right}} = \mathbf{H}^\top(\mathbf{H}\mathbf{H}^\top)^{-1} \tag{6.43}$$

$$\mathbf{H}^{\mathbf{-1}}_{\text{right}} = \mathbf{Q}\mathbf{R}(\mathbf{R}^\top\mathbf{Q}^\top\mathbf{Q}\mathbf{R})^{-1} \tag{6.44}$$

$$\mathbf{H}^{\mathbf{-1}}_{\text{right}} = \mathbf{Q}\mathbf{R}(\mathbf{R}^\top\mathbf{R})^{-1} \tag{6.45}$$

$$\mathbf{H}^{\mathbf{-1}}_{\text{right}} = \mathbf{Q}\mathbf{R}\mathbf{R}^{\mathbf{-1}}\mathbf{R}^{-\top} \tag{6.46}$$

$$\mathbf{H}^{+}_{\text{right}} = \mathbf{Q}\mathbf{R}^{-\top} \tag{6.47}$$

Right pseudoinverse estimation

6.3.3 Hardware and Software intrinsic relationship

The intrinsic relationship between the characterization of hardware and software is particularly evident in the *surrogate* created by the *NMF solver*. The *surrogate* is composed of two matrices $\mathbf{W}$ and $\mathbf{H}$. Consider the following:

- when NMF is used to characterize a *target program*:
  - $\mathbf{X}_{\text{SW}}$ contains the measures of the *benchmarks* for the *resources*, every *resource* is a row, every *benchmark* is a column
  - NMF finds $\mathbf{W}_{\text{SW}}$ and $\mathbf{H}_{\text{SW}}$ such that $\mathbf{X}_{\text{SW}} = \mathbf{W}_{\text{SW}}\mathbf{H}_{\text{SW}}$
  - $\mathbf{W}_{\text{SW}}$ contains the *resource* consumption of *computational patters*: the rows of $\mathbf{W}_{\text{SW}}$ are *resources*, the column are *computational patterns*
  - $\mathbf{H}_{\text{SW}}$ contains the *benchmarks* expressed as linear combinations of *computational patterns*: the rows are *computational patterns*, the columns are *benchmarks*
- when NMF is used to characterize a *target resource*:
  - $\mathbf{X}_{\text{HW}}$ contains the measures of the *benchmarks* for the *resources*, every *benchmark* is a row, every *resource* is a column
  - NMF finds $\mathbf{W}_{\text{HW}}$ and $\mathbf{H}_{\text{HW}}$ such that $\mathbf{X}_{\text{HW}} = \mathbf{W}_{\text{HW}}\mathbf{H}_{\text{HW}}$
  - $\mathbf{H}_{\text{HW}}$ contains the *resource* consumption of *computational patters*: the columns of $\mathbf{H}_{\text{HW}}$ are *resources*, the rows are *computational patterns*

– $\mathbf{W}_{\text{HW}}$ contains the *benchmarks* expressed as linear combinations of *computational patterns*: the columns of $\mathbf{W}_{\text{HW}}$ are *computational patterns*, the rows are *benchmarks*

$$\mathbf{X}_{\text{HW}} = \mathbf{W}_{\text{HW}}\mathbf{H}_{\text{HW}} \tag{6.48}$$
$$\mathbf{X}_{\text{SW}} = \mathbf{X}_{\text{HW}}^{\top} \tag{6.49}$$
$$\mathbf{X}_{\text{SW}} = (\mathbf{W}_{\text{HW}}\mathbf{H}_{\text{HW}})^{\top} \tag{6.50}$$
$$\mathbf{X}_{\text{SW}} = \mathbf{H}_{\text{HW}}^{\top}\mathbf{W}_{\text{HW}}^{\top} \tag{6.51}$$
$$\mathbf{W}_{\text{SW}} = \mathbf{H}_{\text{HW}}^{\top} \tag{6.52}$$
$$\mathbf{H}_{\text{SW}} = \mathbf{W}_{\text{HW}}^{\top} \tag{6.53}$$
$$\mathbf{X}_{\text{SW}} = \mathbf{W}_{\text{SW}}\mathbf{H}_{\text{SW}} \tag{6.54}$$
$$\tag{6.55}$$

As shown in equations 6.52 and 6.53, with simple algebraic transformations, the *surrogate* for the characterization of the *target resource* is also a *surrogate* for the characterization of the *target program.* Equation 6.48 shows the definition of *surrogate* for $\mathbf{X}_{\text{HW}}$: equation 6.49 notes that the two models have the same $\mathbf{X}$ matrix, simply transposed; equation 6.50 applies a simple substitution to equation 6.49, using equation 6.48; equation 6.51 applies a simple property of transpose matrices: $(AB)^{\top} = B^{\top}A^{\top}$; equations 6.52 and 6.53 simply rename the matrices of the *surrogate*; in equation 6.54 we show that the *surrogate* of $\mathbf{X}_{\text{HW}}$ can be used to build a *surrogate* for $\mathbf{X}_{\text{SW}}$.

6.3.4 Limits of NMF

We have shown how to enrich the output of *NMF* to create a *surrogate* and a *predictor*, not only characterizing the *target program* and predicting its usage of the *target resource*, but also finding a theoretical set of possible *computational patterns* and the decomposition of the *programs.* In practice, however, the information contained in $\mathbf{X}$ will not be sufficient to really capture all the interesting behaviours, both in terms of *programs* (that need to use different *computational patterns*) and *resources* (that need to measure the effect of the different *computational patterns*). Therefore, NMF will not be able to find all the *computational patterns*, but only a combination of them.

6.4 Conclusion and future work

In this chapter we have presented 3 *solvers* for our benchmarking model. All the presented *solvers* assume that *programs* can be explained as linear composition of *computational patterns*.

- The *simplex solver* assumes that is possible to design and measure *benchmarks* that perfectly embody *computational patterns*. Under this assumption finding the *surrogate* of the *target program* can be formulated as a linear programming problem, and solved using the simplex algorithm.

- Because *computational patterns* are ideal programs, is reasonable to assume that they can be directly measured only in controlled settings. Real-world programs are too complex to be analysed with such assumption. We show that if both the *benchmarks* and the *target program* can be assumed to be linear compositions of *computational patterns*, then linear regression can be used to find a representative *surrogate*.

- *Non Negative Matrix Factorization* can be used to characterize the *resources* and the *benchmarks* in terms of *hidden factors*, that can be interpreted as *computational patterns*.

Other *solvers* could be developed used with out benchmarking model. Bayesian Regression and Support Vector Regression are natural extensions that fit well in the general setting of our benchmarking model.

6.4.1 Bayesian regression

In section 6.2 we have argued that an important feature of the *linear solver* is the ability to interpret the *surrogate* to verify its quality. This information could be used to inform the regression process.

The general setting for linear regression is $\mathbf{y} = \mathbf{X}\beta + \epsilon$. β is estimated with the pseudo-inverse $\hat{\beta} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$; it can be noted that all the information comes from the measures $\mathbf{X}$ and $\mathbf{y}$. Ordinary least squares, used in the *linear regression solver* presented in the section 6.2, assumes that the information contained in the matrix $\mathbf{X}$ and vector $\mathbf{y}$ is enough to find the regression coefficient β. This can be seen as a frequentist approach.

In Bayesian regression, additional information is provided in the form of prior distribution over β, and this distribution is combined with the data $\mathbf{X}$

and $\mathbf{y}$ to build a posterior distribution over the error ϵ and the coefficients β, following Bayes theorem.

The Bayesian approach is particularly useful when there is prior knowledge of the possible values that the *surrogate* β can have. For example, when characterizing the completion time using performance counters, information about the machine provided by the hardware vendor can be used to build a prior. Consider a computer with a 2 GHz CPU; the regression coefficient associated to the "CPU-cycles" *resource* needs to be positive (executing cycles takes time), and is probably around $\frac{1}{2\times 10^6}$. This information could be used to build a prior distribution for β.

Bayesian regression is a generalization of standard regression: ordinary least squares can be seen as a particular case of Bayesian regression that uses a uniform prior distribution (that assigns the same probability to every possible value of β).

The error ϵ is assumed to be a Gaussian random variable with mean 0 and standard deviation σ. $P(\mathbf{X}, \mathbf{y}, \beta, \sigma^2)$ is the likelihood; it contains the information about β and σ^2 that can be extracted from the data (equivalent to linear regression).

As shown in equation 6.56, the posterior distribution $P(\beta, \sigma^2|\mathbf{X}, \mathbf{y})$ combines the information form the prior distribution of β and σ^2 with the likelihood.

$$P(\beta, \sigma^2|\mathbf{X}, \mathbf{y}) \propto P(\mathbf{X}, \mathbf{y}|\beta, \sigma^2)P(\beta, \sigma^2) \tag{6.56}$$

Posterior distribution

Finding an analytical computation of the posterior distribution is usually difficult, but using Markov Chain MonteCarlo (MCMC) algorithms (Gilks, 2005), such as the Gibbs sampler (Smith and Roberts, 1993), or the Metropolis-Hastings sampler (Geyer, 1992), it can be approximated.

The advantages of a Bayesian approach over a frequentist approach are:

- in the frequentist approach we assume that the sample is large enough to be representative;
- in the Bayesian approach all the assumptions are explicit
- the Bayesian approach follows the sequential nature of experimentation: prior knowledge combined with new data leads to posterior knowledge. The next experiment will use the posterior knowledge of the previous experiment as prior knowledge: old posterior knowledge combined with new data leads to a new posterior knowledge;

- Bayesian regression does not only offer the most probable values of β and σ, it also describes their distributions. This allows to verify statements such as "what is the probability that β is within a certain range".

6.4.2 Support Vector Regression

All the *solvers* presented in the previous sections assume the existence of *computational patterns* that compose linearly into *programs*. This assumption seems to be verified by the good characterization and prediction performance of the model, as shown in the experimental section. However, the model can be used even without assuming linearity.

Support Vector Machines (SVM) are a supervised learning model, widely used for classification and regression. SVMs map the input data into a higher dimensional space. When used for classification SVMs find the hyperplane that better separates the data into two classes, i.e. the hyperplane that defines the largest margin between the sets of points (Cortes and Vapnik, 1995). The maximum-margin hyperplane can be applied in a non-linearly transformed feature space. This allows to separate points non-linearly separable. Meyer et al. (2003) compares SVMs and other classifiers.

The same principle used to classify non linearly separable points can be applied to regression (Drucker et al., 1997), called Support Vector Regression (SVR). The hyperplane that best fits the data is calculated in the transformed feature space.

Bishop and Tipping (2003) explains the relationship between linear regression, Bayesian regression, and Support Vector Machines.

SVR could therefore be used instead of linear regression, to account for non-linear relationships between the *programs*. However, we believe that SVR also has important drawbacks with respect to linear regression:

- the *surrogate* created by the *linear regression solver* has intuitive interpretation, and can therefore be easily verified; the *surrogate* that would be created by SVR would have no direct interpretation, because the mapping to a higher-dimensional non-linear feature space makes the regression coefficient non easily related to the original independent variables.

- approaches based on SVMs are prone to over-fitting: is common practice to try several non-linear transformations until a good fit is achieved, and the resulting model is rarely verified because of the lack of clarity in the *surrogate*.

In chapter 13 we present an experiment where good results are achieved with a linear model, in a similar setting where SVMs were used. We show that renouncing to non-linearity does not compromise the prediction results, and that the model created is easily verifiable. Nonetheless, it would be desirable to study when the use of non-linear spaces makes the model more accurate.

Chapter 7

Experimental complexity of software

In the previous chapters we have considered applying different inputs to the same algorithm as different *programs*. In this chapter we extend the model adding the concept of *experimental complexity*, that describes the evolution of the *surrogate* of a program as the input size changes.

7.1 Characterization through the surrogate

Studying how the *surrogate* β of a *program* changes with the input size can reveal the nature of a computation and can be used to predict what bottleneck the program will suffer as the input becomes too large.

Consider the following example. The basis is composed by two programs:

- p_{cpu} a CPU intensive program, it does not use memory
- p_{mem} a memory intensive program, it does not perform CPU intensive tasks

and using them to characterize a *target program* p where the memory usage grows linearly and the CPU usage polynomially. Imagine also using performance counters closely related to CPU and memory usage, like "instructions" and "cache-misses", as our *resources*. Imagine the *resources* usage of the basis and the target program to be what reported in table 7.1.

Because p_{cpu} and p_{mem} are close to what we could call "CPU usage" and the "memory usage" *computational patterns*, and because the chosen *resources* accurately measure their usage, our model can characterize the

program	instructions	cache-misses
p_{cpu}	100	1
p_{mem}	10	10
$p(\|x\| = 10)$	501	10
$p(\|x\| = 20)$	1004	40
$p(\|x\| = 30)$	1509	90
$p(\|x\| = 40)$	2016	160
$p(\|x\| = 50)$	2525	250
$p(\|x\| = 60)$	3036	360
$p(\|x\| = 70)$	3549	490
$p(\|x\| = 80)$	4064	640
$p(\|x\| = 90)$	4581	810
$p(\|x\| = 100)$	5100	1000

Table 7.1: Example of instructions and cache-miss usage for the basis and the target program

evolution of the surrogate in an intuitive fashion. Figure 7.1 shows the evolution of the components of β extracted by our model from the data in table 7.1, using the performance counters as *resources*, p_{cpu} and p_{mem} as the basis, and creating a surrogate for every input size of the *target program p*. Is immediately visible that β_{mem} grows linearly with the input size, while β_{cpu} grows polynomially, revealing the different behaviour of the *target program* with respect to the *computational patterns* of interest.

This example is an oversimplification of reality. The available data is usually not as auto-explicative as in table 7.1 (that would allow us to characterize the *target program*'s behaviour even without looking at the *surrogate*). In section 10.2 we show experiments with real algorithms, starting from measures that do not reveal the nature of the *target program* until we analyse it through the evolution of the *surrogate*.

7.2 Definition of experimental complexity

In the previous section we showed how the evolution of the *surrogate* can reveal how the behaviour of a *program* changes as the input size grows. In this section we formalize that intuition, introducing the concept of *experimental complexity of software*, as a function that describes the relation between the *surrogate* and the size of the input to the *target program*.

Definition 12 (Experimental complexity of a program)**.** *The experimental*

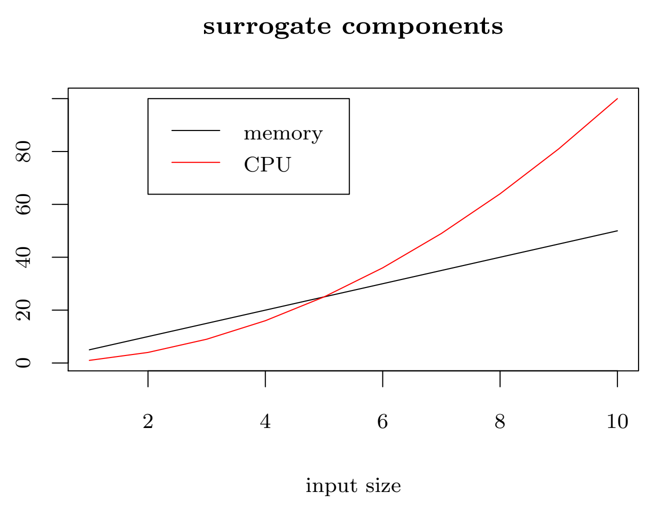


Figure 7.1: Example of evolution of surrogate components as the input size grows

complexity ξ of a program is a vector valued function that maps the size of the input data $\|x\|$ of a program into its corresponding surrogate β.

Equation 7.1 shows that the vector valued function ξ is defined as a vector of functions ξ_i, each describing how the i^{th} coefficient of β changes as the input of the *program* grows.

$$\xi(\|x\|) = \begin{pmatrix} \xi_1(\|x\|) \\ \xi_2(\|x\|) \\ \vdots \\ \xi_n(\|x\|) \end{pmatrix} = \begin{pmatrix} \beta_{1_x} \\ \beta_{2_x} \\ \vdots \\ \beta_{n_x} \end{pmatrix} = \beta_x \tag{7.1}$$

Experimental complexity

ξ_i can be found using curve fitting on a predefined set of functions. Consider the example of the previous section, ξ_{cpu} and ξ_{mem} need to fit the curves shown in figure 12. We define a set of possible interesting functions:

- $f_{\text{lin}} = \|x\|$

- $f_{\text{log}} = \log(\|x\|)$
- $f_{\text{square}} = \|x\|^2$
- $f_{\text{cubic}} = \|x\|^3$
- $f_{\text{exp}} = e^{\|x\|}$
- ...

We then generate a vector for every f_i, using the input size as the argument of f_i. At this point we run linear regression using the generated vectors as independent variables, and the values of β_{mem} as dependent variable. The regression coefficient will show what combination of f_i best fits the values of β_{mem}. For example in our case all the coefficients are extremely small except for f_{lin}, that has a value of 0.5, showing that the *target program* is linear with respect to the *benchmark* that expresses memory access.

We then repeat using the values of β_{cpu} as dependent variables, and find that all the coefficient are close to zero except for f_{square} that has a coefficient of 0.01, showing that the *target program* is quadratic with respect to the *benchmark* that expresses the CPU intensive task.

The experimental complexity can be used to interpolate (or even extrapolate) resource consumption of the *target program* for input sizes for which we have no measures for any *resource*, and we can not therefore apply the *solver* to build a *surrogate*. In the previous example we could be interested in predicting the "instructions" and "cache-misses" values for an input size of 110. We can calculate $\beta(\|x\| = 110)$ as shown in equation 7.2, then predict the "instructions" and "cache-misses" as shown in equation 7.3.

$$\begin{aligned}
\beta(\|x\| = 110) &= \xi(\|x\|) = \\
&= \begin{pmatrix} \xi_{\text{mem}}(110) \\ \xi_{\text{cpu}}(110) \end{pmatrix} = \\
&= \begin{pmatrix} 0.5 * 110 \\ 0.01 * 110^2 \end{pmatrix} = \\
&= \begin{pmatrix} 55 \\ 121 \end{pmatrix}
\end{aligned} \tag{7.2}$$

$$\begin{pmatrix}\mu_{\text{cache-misses}}(\|x\| = 110) \\ \mu_{\text{instructions}}(\|x\| = 110)\end{pmatrix} = \begin{pmatrix}p_{\text{mem}}^T & p_{\text{cpu}}^T\end{pmatrix} \beta(\|x\| = 110) = = \begin{pmatrix}1210 \\ 5621\end{pmatrix} \tag{7.3}$$

7.3 Relation with computational complexity

Unlike theoretical computational complexity approaches, such as the widely used Big-O notation, *experimental complexity* is an empirical metric, therefore it depends on the measures available upon model creation. Like other empirical analysis of algorithms performance, *experimental complexity* attempts to describe the behaviour of *programs* with respect to a set *computational resources* of interest. The limitation of empirical approaches is typically the inability do abstract the characterization of the software from the hardware where the program is measured. However, as shown in the previous chapters, our approach is capable of combining information coming from different hardware and resource usage measures, isolating the underlying structure of the *target program*. The advantage of empirical approaches is the fact that characterizations are directly applicable to real-world scenarios, like scheduling and resource allocation, whereas theoretical approaches are more suitable to the study asymptotic behaviour, the orders of growth, ignoring the constant factors. In practice, however, constant factors are important, because hardware has limited resources.

Our approach describes the *programs* in terms of features, relying on regression to fit the features to actual measures. A similar approach has been recently proposed by Goldsmith et al. (2007), based on basic blocks extraction and clustering. Our model is considerably simpler, because it looks at programs as black boxes, which makes it also easily portable.

7.4 Bottlenecks

The definition of measure requires countable additivity if the resource being measured is not shared by two programs, or countable sub-additivity if they share part of it. It could also happen that the measure of the sum of two programs is larger than the sum of the measures of the programs taken individually. This can happen if running the programs simultaneously results in bottleneck e.g. thrashing the system.

We can consider the combination of programs as another program. For example if we want to run program p_1 and program p_2, we can define program $p_{1,2}$ as their combination: $p_{1,2} = \{p_1, p_2\}$. Our model will still apply. The discussion about bottlenecks caused by the simultaneous execution of programs can be reduced to the more general discussion of bottleneck caused by a program on a computational environment.

Bottlenecks happen when the demand for a computational resource is higher than the computational environment can provide. Usually the operating system will serialize the requests, and the computation will have to wait for the resource to become available. E.g. if a program uses more memory than physically available, the system will start swapping, and most of the computational time will be spent waiting for data to be exchanged between RAM and disk. When a bottleneck occurs the structure of the computation changes, sometimes dramatically.

7.4.1 Grace area

The *grace area* of a *computational environment* is the combination and amount of *resources* that is safe to consume on that particular environment without occurring in significant bottlenecks. When a *program* is outside the *grace area* its *experimental computational complexity* on that *computational environment* changes, and the sub-additivity property does not hold.

In the *resource* measurement consumption space, restricted to *resource* of a particular *computational environment* only, the *grace area* is the polytope where *programs* do not thrash that machine.

Every machine will have different bottleneck conditions, therefore different *grace areas*.

Let us now consider a set of identical CPU bound benchmarks. If we execute them simultaneously, sub-additivity will hold only as long as there are available processors. Imagine a computational environment with n processors. Because the benchmarks are CPU bound, then can run each on a separate processor in parallel. The completion time of the benchmark defined as the combination of the first n benchmarks $p_{1\ldots n} = \bigcup_{i=1}^{n} p_i$ will be equal to $t(p_{1\ldots n}) = \max(t(p_1), \ldots, t(p_n)) = t(p_1)$. Subadditivity holds, because $t(p_{1\ldots n}) \leq \sum_{i=1}^{n} t(p_i)$. If we now run $p_{1\ldots n}$ ns p_{n+1} simultaneously, we will have more active CPU bound programs than processors, and some time will be spent switching context. The completion time of $t(p_{1\ldots n} + p_{n+1}) > t(p_{1\ldots n}) + t(p_{n+1})$, violating the subadditivity constrain. A resource is therefore valid only if is not acting as a bottleneck.

7.5 Compositionality of *Surrogates* and *Experimental complexity*

In this section we discuss the compositional properties of the *surrogate* β and the *experimental complexity* ξ.

In section 10.2 we verify the compositionality of β and ξ using simple toy-benchmarks and sorting algorithms, to compare the experimental results with theoretical time complexity analysis.

7.5.1 Linear composition of *surrogates* and *experimental complexity*

By definition, any linear composition of the basis $\mathbf{X}$ can be expressed by the *linear regression solver* using a *surrogate* β. Imagine a *program* c being defined as the linear combination of two programs a and b: c consists of k_a invocations of the *program* a and k_b invocations of *program* b. The *resource* usage of c will also be a linear combination of the *resource* usage of a and b, as shown in equation 7.4.

$$\begin{aligned} p_c &= k_a p_a + k_b p_b \\ \mu_{p_c} &= k_a \mu_{p_a} + k_b \mu_{p_b} \qquad (7.4) \\ &\qquad c \text{ is a linear combination of } a \text{ and } b \end{aligned}$$

The *surrogate* β_c of the *program* c can be written as a linear combination of the *surrogates* β_a and β_b of *programs* a and b, as shown in equation 7.11:

- equation 7.5 follows from equation 7.4, because y_c is composed of measures of the *program* c;
- equations 7.6, 7.7, and 7.8 simply report the formula of the estimation of β_a, β_b, and β_c using linear regression;
- equation 7.9 uses equation 7.5 to substitute y_c with $y_a + y_b$
- equation 7.10 uses the distributive property of matrices
- equation 7.11 simplifies equation 7.10, using equations 7.6 and 7.7, showing that the estimation of the *surrogate* of a linear combination of programs is the linear combination of the *surrogates*.

$$\mathbf{y_c} = k_a \mathbf{y_a} + k_b \mathbf{y_b} \tag{7.5}$$
$$\hat{\beta}_a = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y_a} \tag{7.6}$$
$$\hat{\beta}_b = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y_b} \tag{7.7}$$
$$\hat{\beta}_c = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y_c} \tag{7.8}$$
$$\hat{\beta}_c = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T(k_a\mathbf{y_a} + k_b\mathbf{y_b}) \tag{7.9}$$
$$\hat{\beta}_c = k_a(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y_a} + k_b(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y_b} \tag{7.10}$$
$$\hat{\beta}_c = k_a\hat{\beta}_a + k_b\hat{\beta}_b \tag{7.11}$$

Compositionality of *surrogates*

The compositional property of β does not depend on the input size. *Surrogates* will therefore be compositional as the input size grows. As shown in equations 7.12 to 7.15, this also makes the *experimental complexity* ξ compositional: equation 7.12 and 7.13 simply state that applying $\|x\|$ to ξ, the corresponding β can be found; equation 7.14 states that *surrogates* are compositional, independently of the input size; substituting equations 7.12 and 7.13 into equation 7.14 we can obtain a valid ξ_c, independently of the input size.

$$\xi_a(\|x\|) = \beta_{a(x)} \tag{7.12}$$
$$\xi_b(\|x\|) = \beta_{b(x)} \tag{7.13}$$
$$\beta_{c(x)} = k_a\beta_{a(x)} + k_b\beta_{b(x)} \quad \forall x \tag{7.14}$$
$$\xi_c(\|x\|) = \beta_{c(x)} = k_a\xi_a(\|x\|) + k_b\xi_b(\|x\|) \quad \forall x \tag{7.15}$$

Compositionality of *experimental complexity*

7.5.2 Function composition

In this section we explore how the composition of functions reflects on the *surrogates* and the *experimental complexity*. Imagine a *program* c composed of an algorithm a that in turn calls another algorithm b, following a certain function $f(x)$. For example the algorithm a could call b on every element of the input, in this case $f(x) = \|x\|$, or it could call b on every element visited during a binary search, in this case $f(x) = \log_2 \|x\|$.

Assuming linearity in the composition of programs, as discussed in the previous section, the resource consumption of c will follow equation 7.16.

$$\begin{aligned} p_c &= p_a + f(x)p_b \\ \mu_{p_c} &= \mu_{p_a} + f(x)\mu_{p_b} \end{aligned} \tag{7.16}$$

c is $a \circ b$

Equations 7.17 to 7.21 show that, like in the previous section, the *surrogates* compose in the same way as the measures of the *programs*. Equation 7.22 shows that, because 7.21 holds for any input size, *computational complexity* is compositional as well.

$$\mathbf{y_c} = \mathbf{y_a} + f(x)\mathbf{y_b} \tag{7.17}$$

$$\hat{\beta}_c = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y_c} \tag{7.18}$$

$$\hat{\beta}_c = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{y_a} + f(x)\mathbf{y_b}) \tag{7.19}$$

$$\hat{\beta}_c = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y_a} + f(x)(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y_b} \tag{7.20}$$

$$\hat{\beta}_c = \hat{\beta}_a + f(x)\hat{\beta}_b \tag{7.21}$$

$$\xi_c(\|x\|) = \beta_{c(x)} = \beta_{a(x)} + f(x)\beta_{b(x)} = \xi_a(\|x\|) + f(x)\xi_b(\|x\|) \quad \forall x \tag{7.22}$$

Compositionality of functions

7.6 A single number is not enough

As already noted by Smith (1988), characterizing performance with a single number is error prone. Nonetheless, most of the benchmarks, including the most used, represent performance with a uni-dimensional metric. This is seen as "necessary evil" (Smith, 1988), to make the characterization easy to interpret by the user. As discussed in section 1.1, especially referring to the "war of the benchmark means" (Mashey, 2004), the choice of the best single number metric is still an unresolved issue. Over the years the benchmarking community has used several different single number metrics (like MIPS, GFLOPS, etc.), leading to controversial results.

Using a single number is equivalent to either using a single *resource* (without any further characterization), or to using our model with a single *program* as the basis.

In this chapter we discuss why using a single metric to characterize software or hardware is inadequate and inevitably leads to significant error.

Consider figure 7.2, where we show three *programs* in the *resource* space (the axes are the measure of the *resource* consumption). If only p_1 is used as

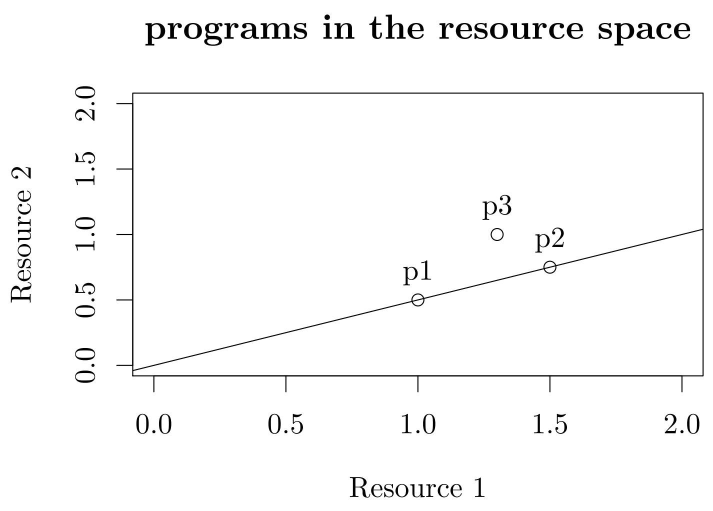


Figure 7.2: Example of points in the resource space

the basis to represent the other *programs*, only the *programs* with a *resource* consumption multiple of p_1 will be modelled without error. Geometrically it means that, if the basis is only composed of p_1, only the points that lie on the line that passes through p_1 and the origin (because of the *null empty set* property of *measures*) can be expressed without error. The *program* p_2 lies on the line, it can therefore be modelled with $\beta = 1.5$, because $p_1\beta = p_2$. However, *program* p_3 does not lie on the same line as p_1, therefore it can not be modelled by any value of β without significant error.

This condition similarly applies to *resource spaces* of higher dimensions: only the points that lies on the line that passes through the origin and the single *program* in the basis can be modelled without significant error.

Using a single dimensional metric different than a program's resource consumption is still equivalent to trying to find a line in the resource consumption space to express all possible programs.

In the rest of the chapter we analyse under which conditions this will happen, in terms of *computational patterns.* For simplicity, we will limit the discussion to 2 *resources*, and to metrics that use the resource consumption of programs.

Equations 7.23 and 7.24 show the generic *resource* consumption of 2 *computational patterns a* and *b*, for 2 *resources.* For example, the *resources* could be completion time on two different machines, *computational pattern* a could be pure CPU arithmetic instructions, and b memory usage.

$$a = \begin{pmatrix} \mu_{\mathrm{a1}} \\ \mu_{\mathrm{a2}} \end{pmatrix} \tag{7.23}$$

$$b = \begin{pmatrix} \mu_{\mathrm{b1}} \\ \mu_{\mathrm{b2}} \end{pmatrix} \tag{7.24}$$

Resource consumption of *computational patterns*

Equations 7.25 and 7.26 show the *programs* p_1 and p_2 are linear combination of the *computational patterns* a and b.

$$p_1 = k_{\mathrm{a1}}a + k_{\mathrm{b1}}b = k_{\mathrm{a1}} \begin{pmatrix} \mu_{\mathrm{a1}} \\ \mu_{\mathrm{a2}} \end{pmatrix} + k_{\mathrm{b1}} \begin{pmatrix} \mu_{\mathrm{b1}} \\ \mu_{\mathrm{b2}} \end{pmatrix} = \begin{pmatrix} k_{\mathrm{a1}}\mu_{\mathrm{a1}} + k_{\mathrm{b1}}\mu_{\mathrm{b1}} \\ k_{\mathrm{a1}}\mu_{\mathrm{a2}} + k_{\mathrm{b1}}\mu_{\mathrm{b2}} \end{pmatrix} \tag{7.25}$$

$$p_2 = k_{\mathrm{a2}}a + k_{\mathrm{b2}}b = k_{\mathrm{a2}} \begin{pmatrix} \mu_{\mathrm{a1}} \\ \mu_{\mathrm{a2}} \end{pmatrix} + k_{\mathrm{b2}} \begin{pmatrix} \mu_{\mathrm{b1}} \\ \mu_{\mathrm{b2}} \end{pmatrix} = \begin{pmatrix} k_{\mathrm{a2}}\mu_{\mathrm{a1}} + k_{\mathrm{b2}}\mu_{\mathrm{b1}} \\ k_{\mathrm{a2}}\mu_{\mathrm{a2}} + k_{\mathrm{b2}}\mu_{\mathrm{b2}} \end{pmatrix} \tag{7.26}$$

Programs composition and *resource* consumption

Imagine using the *program* p_1 as the basis for our model, and p_2 as the *target program*. Because the basis is composed of only one *program*, the *surrogate* β will be a simple scalar. Therefore, for the model to be accurate, p_2 will have to lie on the same line of p_1 in the plane. This condition is true if the ratio between the consumption of the two *resources* is equal for p_1 and p_2, as shown in equation 7.27. Equations 7.28 to 7.34 are simple arithmetic transformations: 7.28 multiplies both members of the equation by $(k_{\mathrm{a2}}\mu_{\mathrm{a2}}+k_{\mathrm{b2}}\mu_{\mathrm{b2}})(k_{\mathrm{a1}}\mu_{\mathrm{a2}}+k_{\mathrm{b1}}\mu_{\mathrm{b2}})$; equation 7.29 expands the multiplication; equation 7.30 rearranges the terms to make the simplification in equation 7.31 more evident, where we subtract $k_{\mathrm{a1}}k_{\mathrm{a2}}\mu_{\mathrm{a1}}\mu_{\mathrm{a2}}$ and $k_{\mathrm{b1}}k_{\mathrm{b2}}\mu_{\mathrm{b1}}\mu_{\mathrm{b2}}$ to both sides; in equation 7.32 we multiply both sides by $\frac{1}{k_{\mathrm{a2}}\mu_{\mathrm{a2}}k_{\mathrm{b2}}\mu_{\mathrm{b2}}}$; in equation 7.34 we factor the 4 terms into the multiplication of 2 differences; in equation 7.35 we show the conditions under which the equation is satisfied.

$$\frac{k_{a1}\mu_{a1} + k_{b1}\mu_{b1}}{k_{a1}\mu_{a2} + k_{b1}\mu_{b2}} = \frac{k_{a2}\mu_{a1} + k_{b2}\mu_{b1}}{k_{a2}\mu_{a2} + k_{b2}\mu_{b2}} \tag{7.27}$$

$$(k_{a1}\mu_{a1} + k_{b1}\mu_{b1})(k_{a2}\mu_{a2} + k_{b2}\mu_{b2}) = (k_{a2}\mu_{a1} + k_{b2}\mu_{b1})(k_{a1}\mu_{a2} + k_{b1}\mu_{b2}) \tag{7.28}$$

$$\begin{aligned} &k_{a1}\mu_{a1}k_{a2}\mu_{a2} + k_{a1}\mu_{a1}k_{b2}\mu_{b2} + k_{b1}\mu_{b1}k_{a2}\mu_{a2} + k_{b1}\mu_{b1}k_{b2}\mu_{b2} = \\ &= k_{a2}\mu_{a1}k_{a1}\mu_{a2} + k_{a2}\mu_{a1}k_{b1}\mu_{b2} + k_{b2}\mu_{b1}k_{a1}\mu_{a2} + k_{b2}\mu_{b1}k_{b1}\mu_{b2} \end{aligned} \tag{7.29}$$

$$\begin{aligned} &k_{a1}k_{a2}\mu_{a1}\mu_{a2} + k_{a1}k_{b2}\mu_{a1}\mu_{b2} + k_{a2}k_{b1}\mu_{a2}\mu_{b1} + k_{b1}k_{b2}\mu_{b1}\mu_{b2} = \\ &= k_{a1}k_{a2}\mu_{a1}\mu_{a2} + k_{a2}k_{b1}\mu_{a1}\mu_{b2} + k_{a1}k_{b2}\mu_{a2}\mu_{b1} + k_{b1}k_{b2}\mu_{b1}\mu_{b2} \end{aligned} \tag{7.30}$$

$$k_{a1}k_{b2}\mu_{a1}\mu_{b2} + k_{a2}k_{b1}\mu_{a2}\mu_{b1} = k_{a2}k_{b1}\mu_{a1}\mu_{b2} + k_{a1}k_{b2}\mu_{a2}\mu_{b1} \tag{7.31}$$

$$\frac{k_{a1}\mu_{a1}}{k_{a2}\mu_{a2}} + \frac{k_{b1}\mu_{b1}}{k_{b2}\mu_{b2}} = \frac{k_{b1}\mu_{a1}}{k_{b2}\mu_{a2}} + \frac{k_{a1}\mu_{b1}}{k_{a2}\mu_{b2}} \tag{7.32}$$

$$\frac{k_{a1}\mu_{a1}}{k_{a2}\mu_{a2}} - \frac{k_{b1}\mu_{a1}}{k_{b2}\mu_{a2}} + \frac{k_{b1}\mu_{b1}}{k_{b2}\mu_{b2}} - \frac{k_{a1}\mu_{b1}}{k_{a2}\mu_{b2}} = 0 \tag{7.33}$$

$$\left(\frac{k_{a1}}{k_{a2}} - \frac{k_{b1}}{k_{b2}}\right)\left(\frac{\mu_{a1}}{\mu_{a2}} - \frac{\mu_{b1}}{\mu_{b2}}\right) = 0 \tag{7.34}$$

$$\frac{k_{a1}}{k_{a2}} = \frac{k_{b1}}{k_{b2}} \quad \text{or} \quad \frac{\mu_{a1}}{\mu_{a2}} = \frac{\mu_{b1}}{\mu_{b2}} \tag{7.35}$$

Condition for an accurate model

This shows that with a single dimension metric, the model can only be accurate in two cases:

- $\frac{k_{a1}}{k_{a2}} = \frac{k_{b1}}{k_{b2}}$: if the ratio between the presence of *computational patterns* in p_1 and p_2 is the same, which means that p_1 and p_2 have the same structure. In other words, they are the same program, only run multiple times or on a larger input size.
- $\frac{\mu_{a1}}{\mu_{a2}} = \frac{\mu_{b1}}{\mu_{b2}}$: if the ratio of the measures of the *computational patterns* are the same, which means that they have the same behaviour on the

resources. In the case of the two resources being completion time on two different machines, it can only happen if the machines have the same architecture, simply one machine being slower than the other in every aspect of the architecture. In the section 10.4 we show an experiment where the same algorithm is measured on different architectures (AMD vs ATOM), leading to different *surrogates* β.

The previous demonstration naturally extends to higher dimension *resource spaces*. With *n resources*, the ratio between the consumption of all the *resources* would have to be the same for basis and the *target program*, as shown in equation 7.36. In general, with n-dimensional *resource spaces* the model can only describe the programs that lie in the hyperplane identified by the basis span. If the basis contain only 1 *program*, the hyperplane is 1-dimensional (a line).

$$\frac{\mu_{r_i}(p_1)}{\mu_{r_j}(p_1)} = \frac{\mu_{r_i}(p_2)}{\mu_{r_j}(p_2)} \quad \forall i, j \tag{7.36}$$

Condition for accurate model in n-dimensions

7.7 Conclusions

In this chapter we have extended our model to consider the relationship between *surrogates* of the same program with a different input size. We have introduced the notion of *experimental computational complexity*. We have discussed its compositional properties, and we have shown why a performance metric based on a single dimension can not be a representative characterization of the performance of software.

Experimental computational complexity is an empirical performance analysis metric, an attempt to bridge the gap between theoretical time complexity (Big-O) and the black-box measurement-based approaches to software performance analysis.

In chapter 10 we will validate its expressiveness with experiments based on sorting algorithms.

Chapter 8

Computational energy model

As shown in the previous chapters, our benchmarking model can characterize and predict the consumption of generic *resources*. In our work we are particularly interested in two specific *resources*: completion time and energy consumption. In this chapter we present a simple and high level energy model that further describes the relationship between time and energy.

The ability to predict the energy needed by a system to perform a task, or several concurrent parallel tasks, allows the scheduler to enforce energy-aware policies, while providing acceptable performance.

The approaches in literature to model energy consumption of tasks usually focus on low-level descriptors and require invasive instrumentation of the computational environment.

We developed an energy model and a methodology to automatically extract features that characterize the computational environment relying only on a single power meter that measures the energy consumption of the whole system. In chapter 12 we show that once the model has been built, the energy consumption of concurrent parallel tasks can be calculated, with a statistically insignificant error, even without any power meter.

We show that our model can predict with high accuracy, even only using the utilization time of the cores in an HPC enclosure, without using performance counters. Hence, the model could be easily applicable to heterogeneous systems, where collecting representative performance counters can be problematic.

8.1 The energy model

In this section we will present the energy model, starting with the most general form, the we will refine and simplify the model assuming property of the computational power and execution setting.

8.1.1 General form

Similarly to Feng et al. (2005) and Wang et al. (2011), in its most abstract form, the energy consumed by a computational environment to complete a set of tasks can be written as equation 8.1:

$$\mathrm{E} = \int_{t_{\text{start}}}^{t_{\text{end}}} \mathrm{P}(t)\,\mathrm{d}t \tag{8.1}$$

where t_{start} and t_{end} are starting and ending time of the set of tasks, $\mathrm{P}(t)$ is the function of the instant power with respect to time, as written in equation 8.2:

$$\mathrm{P}(t) = \sum_{i \in \text{res}} \mathrm{P}_i \alpha_i(t) \tag{8.2}$$

Equation 8.2 represents instant power in its most general form, where the instant power consumption is just the sum of the instant power consumption of each computational resource (e.g. cores, memory, network). $\alpha_i(t)$ is the function of the utilization factor for the i^{th} resource at time t, and P_i the peak power consumption of the i^{th} resource.

We define t_i as the integral of $\alpha_i(t)$, as shown in equation 8.3.

$$t_i = \int_{t_{\text{start}}}^{t_{\text{end}}} \alpha_i(t)\,\mathrm{d}t \tag{8.3}$$

Therefore, t_i can also be defined as the average value of $\alpha_i(t)$ in the timespan from $t_{\text{start}}$to t_{end}, multiplied by $t_{\text{wall}} = t_{\text{end}} - t_{\text{start}}$.

Starting from the general energy definition 8.1 it is possible to replace equation 8.2 to separate the time-dependent components from the peak power as in 8.5. The sum and integral can be swapped by linearity to get 8.6. Finally the equation can be simplified replacing the definition 8.3.

$$
\begin{aligned}
\mathrm{E} &= \int_{t_{\mathrm{start}}}^{t_{\mathrm{end}}} \mathrm{P}(t)\,\mathrm{d}t = && (8.4)\\
&= \int_{t_{\mathrm{start}}}^{t_{\mathrm{end}}} \sum_{i\in\mathrm{res}} \mathrm{P}_i \alpha_i(t)\,\mathrm{d}t = && (8.5)\\
&= \sum_{i\in\mathrm{res}} \mathrm{P}_i \int_{t_{\mathrm{start}}}^{t_{\mathrm{end}}} \alpha_i(t)\,\mathrm{d}t = && (8.6)\\
&= \sum_{i\in\mathrm{res}} \mathrm{P}_i t_i && (8.7)
\end{aligned}
$$

Equation 8.1 can therefore be simplified to equation 8.7, where only the peak power and average utilization factor of the resources are needed to calculate the energy consumption and no integral is explicitly needed.

However, equation 8.2 can be further refined by thinking about the nature of the various P_i. In particular we can identify two distinct elements that contribute in defining the power consumption:

Infrastructure : that comprises everything that is constantly turned on during all the computation, therefore not distinguishable from the fixed cost of turning on the computational environment. Its power consumption is written as $\mathrm{P}_{\mathrm{infr}}$

Active machines : the number of active computers in a computational environment, such as a cluster. The power consumption of each machine is written as P_{m} and refers to the overhead needed to power the machine. If the machines are not identical, each group of identical machine will have a different P_{m}. For simplicity we will assume that all machines have identical overhead.

It is worth noticing that when the computational environment consists of a single machine, $\mathrm{P}_{\mathrm{infr}}$ will include P_{m}, as they will not be distinguishable.

Equation 8.8 shows the refinement of equation 8.2, with $\mathrm{P}_{\mathrm{infr}}$and P_{m}factored out.

$$\mathrm{P}(t) = \mathrm{P}_{\mathrm{infr}} + \mathrm{P}_{\mathrm{m}} m(t) + \sum_{i\in\mathrm{res}} \mathrm{P}_i \alpha_i(t) \qquad (8.8)$$

Machines could be turned on and off during the execution of jobs. In the previous equation the number of machines that are active at time t is described by the $m(t)$ term.

As an example let us consider the case of an HPC cluster where we complete a task A with one dual-core machine active, then turn on an additional dual-core machine (identical to the first machine) to execute a second task B, therefore $m(t)$ will be equal to 1, during the execution of task A and 2 during the execution of task B. $\mathrm{P}(t)$will change over time, in particular when we turn on the additional machine $\mathrm{P}_{\mathrm{m}}m(t)$ will double suddenly. If we model the consumption of cores, $\alpha_i(t)$ will also change over time. Let us assume task A to be strictly serial, it will then use only one core, and task B to use all the cores available. $\alpha_i(t)$ will also rise suddenly when task B starts.

Now we discuss how the general model can be refined further by adding reasonable hypothesis on its various terms in order to obtain variants useful for analyzing the energy consumption and predicting the energy usage of a given system.

8.1.2 Fixed number of active machines and limited considered resources

Under the assumption that the number of active machines does not change during the execution of the workload of interest, the function $m(t)$ becomes a constant m.

$$E = t_{\mathrm{wall}}(\mathrm{P}_{\mathrm{infr}} + m\mathrm{P}_{\mathrm{m}}) + \sum_i \mathrm{P}_i t_i \tag{8.9}$$

Is often unfeasible to attempt to model all the resources in a computational environment, and typically the desired level of accuracy allows to restrict our attention on a limited set of resources.

Focusing on cores of homogeneous machines

Equation 8.10 shows the refinement of equation 8.9 focusing only on the power consumed by cores, where $\mathrm{P}_{\Delta\mathrm{c}}$is the difference between the power consumption when performing useful job and idle state (idle state includes job performed by the operating system, i.e. anything not directly related to the observed tasks) and t_k is the time that the core k has been active for the observed tasks.

$$E = t_{\mathrm{wall}}(\mathrm{P}_{\mathrm{infr}} + m\mathrm{P}_{\mathrm{m}}) + \mathrm{P}_{\Delta\mathrm{c}} \sum_{k \in \mathrm{cores}} t_k \tag{8.10}$$

Assuming that cores can only be in either active or idle state is an oversimplification, as CPUs have several possible active states. However, in the experimental section we will show that this simple model already achieves the desired descriptive and predictive results, therefore we are not interested in modeling the time spent by cores in all the possible active state. For these reasons we decided to opt for the simplest model, restricting the state to either active or idle. The model can be refined to express the time spent in several possible states, should this level of detail be considered important.

Equation 8.10 assumes that all cores have identical power consumption, i.e. we are modeling an homogeneous system. If we want to model an heterogeneous system, where different sets of cores have different power consumption, we simply have to include multiple $\mathrm{P}_{\Delta\mathrm{c}}$in equation 8.10. This is true also for P_{m}, that is assumed to be identical for all machines. If this assumption does not hold, it is sufficient to add multiple $\mathrm{P}_{\mathrm{m}}$to equation 8.10.

The case of a single parallel job

If we consider the execution of a single parallel job running in an HPC cluster, we can assume that the task will start with a pre-processing phase running on a single core (e.g. decomposing the mesh), then execute the parallel job on all the cores for the same amount of time each, then a post processing phase on a single core (e.g. recomposing the mesh). If we assume that all the machines are turned on for the whole task, and that during the parallel phase we will use n cores, the formula becomes:

$$E = t_{\mathrm{wall}}(\mathrm{P}_{\mathrm{infr}} + m\mathrm{P}_{\mathrm{m}} + \mathrm{P}_{\Delta\mathrm{c}}) + (n-1)\mathrm{P}_{\Delta\mathrm{c}}t_{\mathrm{job}} \tag{8.11}$$

where $t_{\mathrm{job}}$is the time spent in the parallel phase.

The case of a single machine

To model the simple case of a job running on a single machine, we can simplify equation 8.11, defining $\mathrm{P}_{\mathrm{infr+m}} = \mathrm{P}_{\mathrm{infr}} + Pm$, as shown in equation 8.12. Merging $\mathrm{P}_{\mathrm{infr}}$ and P_{m} is unavoidable because the inability to turn off the machine makes it indistinguishable from the infrastructure.

$$E = t_{\mathrm{wall}}(\mathrm{P}_{\mathrm{infr+m}} + \mathrm{P}_{\Delta\mathrm{c}}) + (n-1)\mathrm{P}_{\Delta\mathrm{c}}t_{\mathrm{job}} \tag{8.12}$$

Sequential job on a single machine

The simplest case to model is that of a sequential job on a single machine. Defining $\mathrm{P}_{\mathrm{infr+m+\Delta c}} = \mathrm{P}_{\mathrm{infr+m}} + \mathrm{P}_{\Delta\mathrm{c}}$, equation 8.12 can be simplified to equation 8.13.

$$E = t_{\mathrm{wall}} \mathrm{P}_{\mathrm{infr+m+\Delta c}} \tag{8.13}$$

8.1.3 Automatic characterization of hardware

In this section we will show how refinements of the energy model can be used to automatically extract a characterization of the computational environment.

The refinements presented in formulae 8.9, 8.10, 8.12, and 8.13, can be generalized to:

$$E = \sum_i \mathrm{P}_i t_i \tag{8.14}$$

where P_i and t_i are the power consumption and the utilization time of the i^{th} resource.

We will assume t_i to be known, for every considered resource. The scheduler usually has this information, at least for relevant computational resources.

We can consider each P_i as a random variable. We can then run a set of benchmarks, measuring t_i and the energy consumption, with a simple ammeter at the Power Distribution Unit (PDU), for each run. We can then consider each run as an independent experiment, and use a statistical approach to estimate P_i, for every i.

Equation 8.15 shows the general setting of regression analysis.

$$f(\mathbf{X}, \beta) + \epsilon = \mathbf{y} \tag{8.15}$$

where:

X is a matrix with the measured t_i of the benchmarks, a row for every experiment, a column for every i.

y is a vector with the measures of the energy consumption of the benchmarks, in the same order as the rows of **X**.

β is a vector that contains the regression coefficients. Every coefficient represents a P_i. The i^{th} coefficient corresponds to the i^{th} column of **X**.

ϵ contains the fitting residuals, the regression error.

$f()$ is a statistical multivariate function that predicts y_i using the values of the i^{th} row of $\mathbf{X}$ and β.

When the underlying model can be assumed to be linear, ordinary least square (linear regression) can be used as the regression algorithm, simple vector dot product can be used as $f()$. Linear regression finds the β that minimizes the norm of the vector ϵ from equation 8.16.

$$\mathbf{X}\beta + \epsilon = \mathbf{y} \tag{8.16}$$

Linear regression is an adequate statistical regression tool only if the model exposes a linear behaviour. This assumption usually holds for high-level models, whereas a linear approximation is usually not feasible on low-level models. For example, performance counters measure low-level events that interact in non-linear ways. If we build a linear model using performance counters as t_i, those non-linear interactions will not be modelled, and a large error will arise.

As an example, imagine attempting to measure the completion time of low level instructions such as *arithmetic sums*, measuring a benchmark with very few branch mis-prediction on a super-scalar processor. The super-scalar pipeline will execute several sums in parallel. If we try to use this model do predict a program that contains only a few *arithmetic sums* between branch mis-predictions, the super-scalar pipeline will often be empty, and the sums will not be executed in parallel, resulting in a very different completion time.

Measures of energy and completion time are subject to noise, for example caused by tasks executed in background by the operating system. For this reason, some outliers are expected to be present in the measures. To mitigate their effect on the results of the regression, we used a robust linear regression algorithm (iterated re-weighted least squares), as described by Hoaglin et al. (2011) and Fox (1997).

In chapter 12 we present experimental results of automatic characterization using the method described in this section.

8.2 Discussion

The power model presented in this chapter is a simple and coarse grain model that describes the energy consumption of *computational environments* at the cluster level. Notwithstanding the simplicity of the mode, we show in chapter 12 that it achieves remarkably precise power and energy consumption predictions. Figure 8.1 shows a realistic example for a cluster where $\mathrm{P_{infr}}$ is equal to 100W, $\mathrm{P_m}$ to 90W and $\mathrm{P_{\Delta c}}$ to 10W, on machines with 16 cores. Even using only 1 active core requires consuming $\mathrm{P_{infr}} + \mathrm{P_m} + \mathrm{P_{\Delta c}}$ power, then the required power grows linearly with the number of active cores. When we use more cores than available on a single machine, we turn on the second machine. Using 17 cores the power consumption is equal to $\mathrm{P_{infr}} + 2\mathrm{P_m} + 17\mathrm{P_{\Delta c}}$.

We model the completion time of the program using Amdahl's law (Amdahl, 1967), shown in equation 8.17: $T(1)$ is the completion time for the program using a single processing unit; $T(n)$ is the completion time using n processing units; s is the portion of the computation inherently serial (a number between 0, completely parallel computation, to 1, completely serial computation); n is the number of used processing units. Asymptotically, the completion time with a very large number of processing units tends to $sT(1)$.

$$T(n) = T(1)(s + \frac{1-s}{n}) \tag{8.17}$$

Amdahl's law

Figure 8.1 also shows the energy consumption as the number of processing units grows. The energy is simply the multiplication of the instant power and the completion time. Because the power overhead of using additional cores grows with the number n of cores, and the completion time decreases proportionally to $\frac{1}{n}$, the energy consumption will initially decrease, then when the power increases faster than the time decreases, the energy needed will start to rise.

Figure 8.2 considers the ideal scenario where it is possible to build a perfect computational environment that does not waste any power, and only uses energy for useful work, we'll have $\mathrm{P_{infr}} = \mathrm{P_m} = 0$ (because no energy is wasted in overhead), and some non-zero $\mathrm{P_{\Delta c}}$. Power now grows linearly with the number of active cores.

Considering also the ideal scenario where we are able to build a perfectly parallel computation, with no inherently serial portion ($s = 0$), the completion time will keep decreasing as we add processing units, asymptotically tending to 0.

The resulting energy consumption is constant with the number of processing units used. Adding units saves time, but consumes more power, in the same ratio.

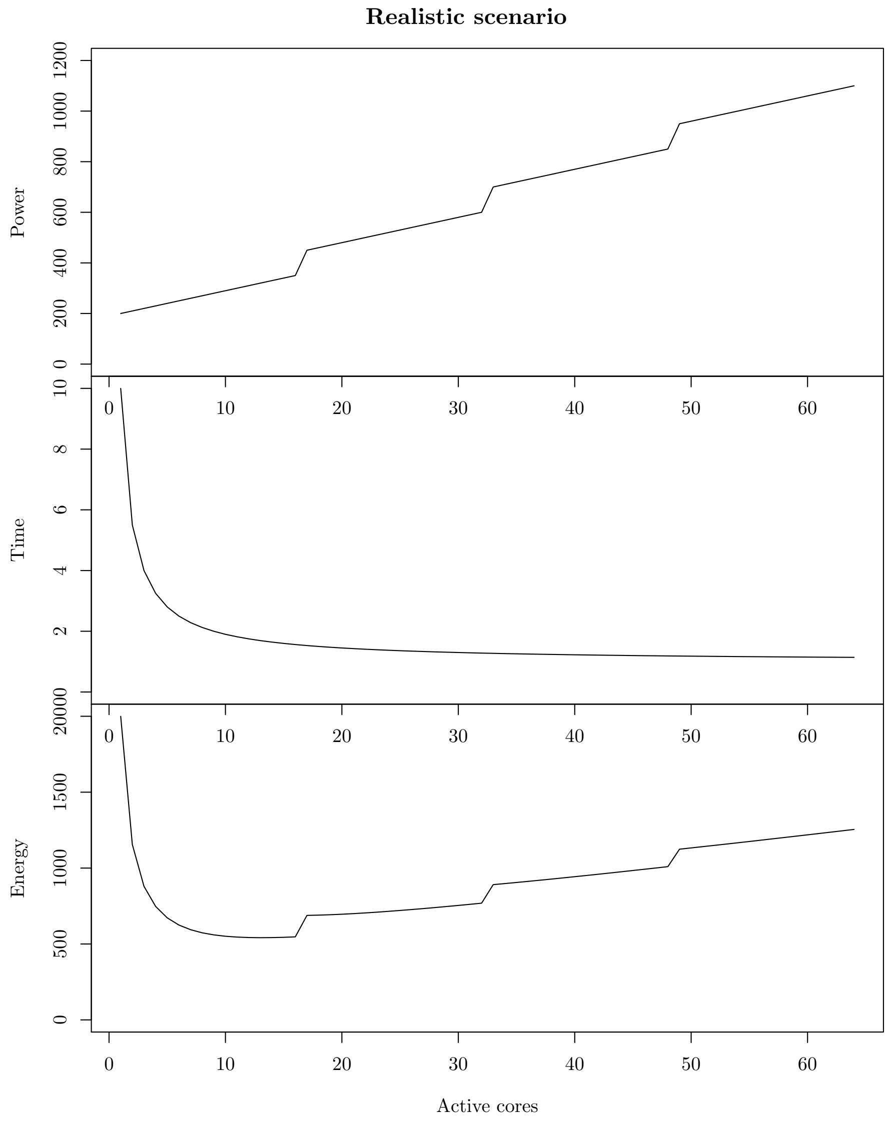


Figure 8.1: Realistic scenario: power model with $P_{infr} = 100$, $P_m = 90$, $P_{\Delta c} = 10$; completion time following Amdahl's law, with 10% algorithm serial; and power model combined with Amdahl's law

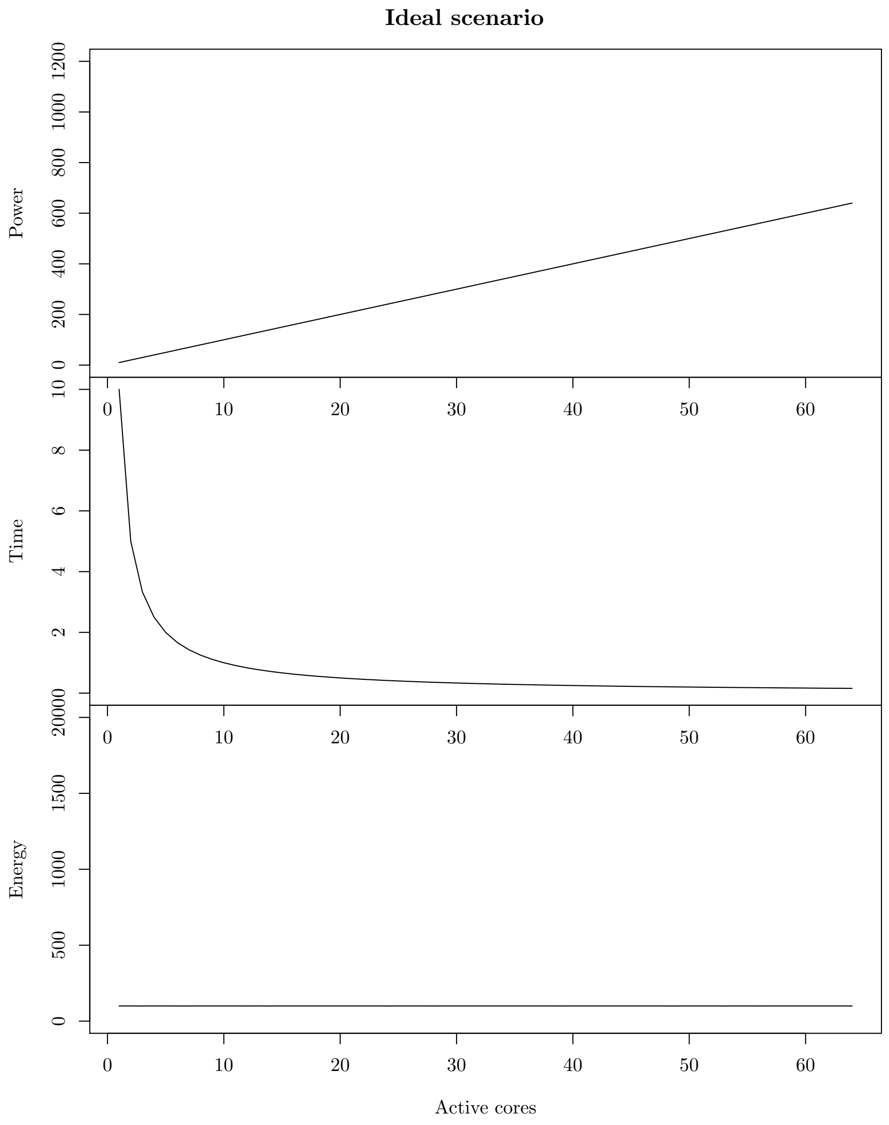


Figure 8.2: Idea scenario: power model with $P_{infr} = P_m = 0$, $P_{\Delta c} = 10$; completion time following Amdahl's law, with 0% algorithm serial; and power model combined with Amdahl's law

8.3 Conclusions

We presented a coarse-grain energy model for parallel concurrent tasks, with several refinements of the general formula to show how it can be applied to different cases ranging from no restrictions to single machine, single task, multiple tasks on a single machine, limited considered resources.

We showed how linear regression can be applied to the formula to characterize the hardware. The model was accurate enough to allow us to predict peak power absorption with high statistical confidence.

We also showed how the model and the regression coefficient can accurately estimate the energy consumption of a system executing concurrent parallel tasks, only using the information available at runtime to the scheduler, which could then optimize the energy consumption while preserving acceptable performance. This approach could also be used in cloud systems to account for energy usage of tasks when a single computational environment is used for concurrent tasks.

We show that the predictions obtained from our model have high accuracy, even only using the utilization time of the cores in an HPC enclosure, without using performance counters. Hence, the model could be easily applicable to heterogeneous systems, where collecting representative performance counters can be problematic.

Finally, while the estimate of the energy consumption relies on time measurements, the prediction of the instantaneous power can be performed using only the linear model coefficients. This makes it possible for a cluster management system to decide how many nodes and cores can be activated while respecting pre-determined power usage constraints.

As we will show in chapter 12, the energy model can be used to model the energy consumption of concurrent and parallel tasks, running at the same time on the same *computational environment*, e.g. a cluster. We show that once the *computational environment* has been profiled and the energy model coefficients ($\mathrm{P_{infr}}$, $\mathrm{P_m}$, and $\mathrm{P_{\Delta c}}$) have been extracted, the energy consumed by each individual task can be calculated with a small error. This could be used by a cloud provider to account for energy consumption of the tasks.

Chapter 9

Conclusions

In this part of the thesis we have presented our benchmarking model.

In chapter 5 we have introduced the general approach. The model is resource agnostic, in the sense that different and heterogeneous *resources* can be used to build the model, and can be the target of *resource* usage prediction. The model can therefore be used to characterize completion time, as well as energy consumption, and other interesting aspects of *programs*. The model can be used to characterize both hardware and software. In section 5.2.3 we discuss how the characterization of software is the dual of characterization of hardware. Our model exploit the inherent relationship and interdependency of hardware and software to characterize *program*'s and *computational environments* behaviour. Our model is black-box, because it does not require access to neither the source code of the analysed *program*, or the specifications of the *computational environment* used to execute the *program*. It only relies on measures of consumption of *resources*.

In chapter 6 we have presented three solvers: algorithms that can be used to characterize the *target program* or the *target resource* using measures of *benchmarks*. The characterization can be used to understand the nature of the *target program*, explained in terms of linear combination of *benchmarks*; or to understand the nature of the *target resource*, explained in terms of linear combination of the other *resources*. The model can the use the characterization to create predictions of the consumption of the *target resource* by the *target program*. For example, our benchmarking model could be used to predict the completion time of a *program* on a new architecture.

In this part we have also introduced the concept of *computational patterns* to explain and discuss the properties of our model, and justify why a simple linear approach can be used to build analytical models of complex non-linear phenomena such as the *resource* consumption of *programs* on different architectures.

In chapter 7 we have also introduced the concept of *experimental computational complexity*, an empirical metric of software complexity, that can be used to describe how the *resource* consumption changes as the input size grows. With respect to other empirical metrics, *experimental computational complexity* is resource agnostic, and can be used to describe the consumption of arbitrary *resources*. We use it to demonstrate why a single dimensional metric for performance analysis can only be accurate if the architecture being analysed does not change, or if the *benchmark* used to analyse the machine is the same *program* whose performances we are interested in.

In our work we support the characterization and prediction of generic *resources*, but we focus on completion time and energy consumption in particular. In chapter 8 we introduced our energy model, a simple high level approach that is capable of describing the energetic behaviour of concurrent parallel *programs*. We show how the energy model can be refined to describe different kind of *computational environments*, from single core machines to HPC clusters. The energy model is an attempt to describe the relationship between the two *resources* of particular interest for our work: completion time and energy.

Part III

Experimental validation

In this part we present the experimental validation of our benchmarking model, using both micro-benchmarks and real-world applications.

In chapter 10 we test our model with a small number of micro-benchmarks, designed to verify the expressiveness and the properties of the *surrogate* and the *experimental complexity*, presented in the previous part.

In chapter 11 we use our model to predict the completion time of CPU SPEC, using the huge database of experiment results available on the SPEC website.

In chapter 12 we use our energy model to predict the power consumption of a small cluster, using the widely used OpenFOAM suite.

In chapter 13 we use our model in conjunction with static analysis techniques, to predict the best device to run OpenCL kernels on an heterogeneous machine.

Chapter 10

Validating the expressiveness of the model and experimental complexity using micro-benchmarks

In this chapter we present a list of experiments performed with simple micro-benchmarks we created and measured to show the capabilities of our model in a simplified environment. In this chapter:

- we show that the *surrogate* β can be used to characterize the *target program*;
- we characterize and predict the behaviour of algorithms using *experimental complexity of software*, verifying the properties described in chapter 7.5;
- we show that energy is a representative *resource* that reflects the computational complexity of *programs* using sorting algorithms;
- we explore the use of a basis composed of a single *benchmark*, verifying the theoretical results presented in chapter 7.6, showing that the use of a single *program* as basis necessarily leads to large characterization error when the architecture changes.

10.1 Expressiveness of the model

10.1.1 Using the simplex solver

Experimental setup

In Morelli and Cisternino (2012), we showed how to use our model to characterize *mergesort*, using two micro-benchmarks as basis, measuring only completion time and consumed energy as *resources*.

Here we present an extended version of the same experiments. We measured the completion time and the energy consumption of a small set of programs running on a desktop computer equipped with a CoreDuo processor with 2MB L2 cache and 1 GB RAM. To measure the energy consumption we attached a simple ammeter to the computer power plug (therefore measuring the whole machine power absorption), calculated the average instant power during the execution of each program, then multiplied by the completion time. We prepared two synthetic benchmarks:

cpu is a simple *add* assembler instruction executed 2^{20} times

mem is a program that sums a fixed number of random locations from a large array

We used *cpu* and *mem* as our *benchmarks* and measured *mergesort* sorting arrays of different sizes (1M, 2M, 4M, 8M, 16M, 32M).

Table 10.1.1 shows the measured completion time and energy consumption for the program used as the basis, table 10.1.1 shows the measures relative to the *target program mergesort*, changing the input size from 1M to 32M.

	cpu	mem
time	2.14 s	7.26 s
energy	81.46 J	304.00 J

Table 10.1: Completion time and energy consumption for *cpu* and *mem*

	p(1M)	p(2M)	p(4M)	p(8M)	p(16M)	p(32M)
time	0.22 s	0.33 s	0.67 s	1.39 s	2.85 s	5.79 s
energy	8.60 J	13.20 J	27.49	58.29 J	121.79 J	254.82 J

Table 10.2: Completion time and energy consumption for *mergesort*

Referring to the high level model defined in section 5.2, we defined the $\mathbf{X}$ matrix using the first two columns of the above table, and we used the measurement vectors for *mergesort* at various input sizes as $\mathbf{y}$ vectors.

Using the *simplex* solver presented in section 6.1 we created a *surrogate* β for every input size, such that $\mathbf{y_i} = \mathbf{X}\beta_i$. We decided to use the *simplex* solver because we were confident that the program in the basis can be representative of actual *computational patterns*.

$$\mathbf{X} = \begin{pmatrix} 2.14 & 7.26 \\ 81.46 & 304.00 \end{pmatrix}$$

$$\mathbf{y_{1M}} = \begin{pmatrix} 0.22 \\ 8.60 \end{pmatrix} \quad \mathbf{y_{2M}} = \begin{pmatrix} 0.33 \\ 13.20 \end{pmatrix} \quad \mathbf{y_{4M}} = \begin{pmatrix} 0.67 \\ 27.49 \end{pmatrix}$$
$$\mathbf{y_{8M}} = \begin{pmatrix} 1.39 \\ 58.29 \end{pmatrix} \quad \mathbf{y_{16M}} = \begin{pmatrix} 2.85 \\ 121.79 \end{pmatrix} \quad \mathbf{y_{32M}} = \begin{pmatrix} 5.79 \\ 254.82 \end{pmatrix}$$

Expressiveness of β

The resulting β vectors (calculated using the *simplex solver*) are:

$$\beta_{1M} = \begin{pmatrix} 0.075118 \\ 0.008161 \end{pmatrix} \quad \beta_{2M} = \begin{pmatrix} 0.075862 \\ 0.023093 \end{pmatrix} \quad \beta_{4M} = \begin{pmatrix} 0.069347 \\ 0.071845 \end{pmatrix}$$
$$\beta_{8M} = \begin{pmatrix} 0 \\ 0.248125 \end{pmatrix} \quad \beta_{16M} = \begin{pmatrix} 0 \\ 0.528191 \end{pmatrix} \quad \beta_{32M} = \begin{pmatrix} 0 \\ 0.780954 \end{pmatrix}$$

The evolution of the *surrogate* β is also shown in figure 10.1, where it can be seen that for small input sizes the computation is dominated by CPU, and for large input sizes it quickly gets dominated by memory usage. This is expected because as the array grows it will not fit into cache and a lot of cache miss will occur, therefore most of the time and energy will be spent accessing memory.

It is worth noticing that only black box measures were taken on both the *programs* used in the basis and the *target program*, in particular only completion time and energy consumption. Nonetheless the model was able to reveal the increasing memory usage of the sorting algorithm as the input size grows. This result indicates that, given a representative set of programs, non obvious behaviour of programs can be extracted from high level measures, such as energy consumption, not necessarily directly correlated with the described phenomena.

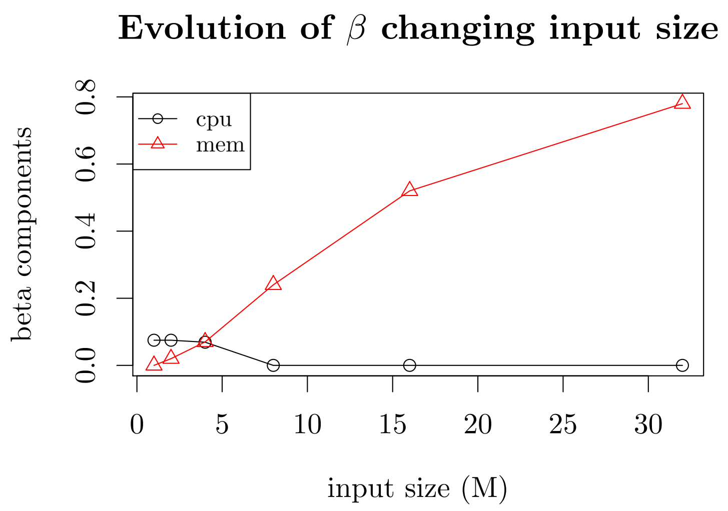


Figure 10.1: β for *mergesort*

10.1.2 Using the linear regression and NMF solvers

In this section we characterize the memory access pattern, using two different solvers: *linear regression* and *NonNegative Matrix Factorization (NMF).*

Experiment setup

We wrote two simple micro-benchmarks:

cpu sums 2^{20} random numbers between 0 and 1024, extracted using the "rand()" C function, and forced between 0 and 1024 using the modulo "%" operator. The *cpu* program uses only simple CPU instructions, it does not allocate, read, or write memory. We used *cpu* as one of the programs in the basis, meant to represent CPU-only computations.

mem(n) allocates a buffer of 2^{24} integers and, for $\forall i \in [0, 2^{24}]$ assigns i to the $(i \times n)\%2^{24}$ element of the buffer, where n is of the form $n = 2^k + 1$ (a power of 2, plus 1). Because the buffer size is a power of 2 and n is a power of 2 + 1, all the items of the buffer are accessed by the algorithm, if $n = 1$ the buffer is accessed sequentially, if $n = 3$ the algorithm skips 2 locations between each access, and so on. The *mem* program can be used to test the impact of sequential versus non-sequential memory access. We tested *mem* for values of n from $2^1 + 1$ to $2^{20} + 1$. We selected *mem(33)* as one of the programs in the basis, $n = 33$ ensures a large amount of cache-misses. This makes it a good representative of memory intensive computations.

We ran the benchmarks on a machine with an Intel(R) Core(TM)2 Duo CPU E6550, 32K L1 cache, 4MB L2 Cache, 8GB RAM, running Linux Fedora 22, kernel 4.0.2-300.

We measured the *cpu-cycles*, *instructions*, *cache-misses*, and *cache-references* performance counters using the "perf" kernel tool. We repeated each run 30 times, reporting here only the mean value.

We used *cpu* and *mem(33)* (from now on simply called *mem*) as basis for the model. All the other programs (and cases for *mem(n)*) are expressed in terms of the basis. In other terms, β_i describes the program i as a linear combination of *cpu* and *mem*.

	cpu	mem
cpu-cycles	36133625.00	956232936.00
instructions	70765551.00	361442598.00
cache-misses	315.00	16944612.00
cache-references	35286.00	17414679.00

Table 10.3: Basis resource consumption

Table 10.3 shows the *resource* usage of the basis. The two *programs* have very different behaviour with respect to the measured *resources*: the number of cache-misses and cache-references in *mem* is higher by several order of magnitudes than in *cpu*; the ratio between cache-misses and cache-references in *mem* is approximately 1, meaning that almost every access in cache resulted in a miss, whereas for *cpu* the ratio is 1 miss every 25 references; the ratio between cpu-cycles and instructions (CPI) is almost 0.5 for *cpu*, meaning that the super-scalar pipeline is efficiently used, whereas CPI drops at almost 3 for *mem*, because most of the cpu-cycles are spent waiting for cache-misses.

We measured *mem(n)* for values of $n = 2^k + 1$, for k going from 1 to 20, on the same computational environment as in the previous section, using the same performance counters (cpu-cycle, instructions, cache-misses, cache-references).

Figure 10.2 shows the measured resource consumption of *mem(n)*, as the number of items of the buffer as skipped between accesses. Large n values result in a sparse memory access patterns, small n values result in a more *sequential* access pattern. Large n values result in frequent cache misses. The number of cache-misses is affected in non-obvious ways by the L1 (32KB) and L2 (4MB) cache sizes. For $n = 33$ the number of cache misses is already approximately equal to the number of memory accesses, but for n between 2048 and 32768 the number of cache-misses reduces by an order of magnitude, because the iterator cycles so quickly that the blocks in memory are not replaced before they are used again the next time the iterator reaches the same area of the buffer.

Characterization using linear regression

To characterize the memory access pattern, as n changes from $n = 2^1 + 1$ to $2^{20} + 1$, we used the *linear regression solver*, with *cpu* and *mem(33)* as the *benchmarks* used for the basis. The $\mathbf{X}$ matrix is therefore equal to the measures reported in table 10.3, normalized by row, and the $\mathbf{y_i}$ vectors are the measures of *mem(i)*. The *surrogate* β_i refers to the $\mathbf{y_i}$ vector.

Figure 10.3 shows the evolution of β as n grows. For small values of n, the memory access pattern is mainly sequential, therefore the computation is dominated by CPU usage. For extremely large values of n the computation is dominated by cache-misses, therefore the *surrogate* assigns a large value to the *mem* component of β, and a small negative value to the *cpu* component. This indicates that for those values of n, *mem(n)* is even closer to the *memory computational pattern* than *mem(33)*. A possible interpretation of $\beta_{2^{20}+1} \approx (-0.92, 1.18)$ is that *mem(*$2^{20}+1$*)* can be explained as 1.18 instances of *mem(33)* removing the CPU usage of 0.92 instances of *cpu*.

Characterization using NMF

To characterize *mem(n)* we also used the *NonNegative Matrix Factorization (NMF) solver*, presented in section 6.3. We ran *NMF* on the same matrix $\mathbf{X}$ used in the previous section (containing the measures reported in table 10.3, normalized by row). The *surrogate* found by the *NMF* solver explains the measures of the *programs* in terms of 2 matrices $\mathbf{W}$ (the basis components matrix) and $\mathbf{H}$ (the mixture coefficients matrix).

We used the *nmf* function from the *NMF R* package (Gaujoux and Seoighe, 2010). The following options were used:

- the positive parts of the output of Independent Component Analysis (ICA) was used as the seeder (Marchini et al., 2013)

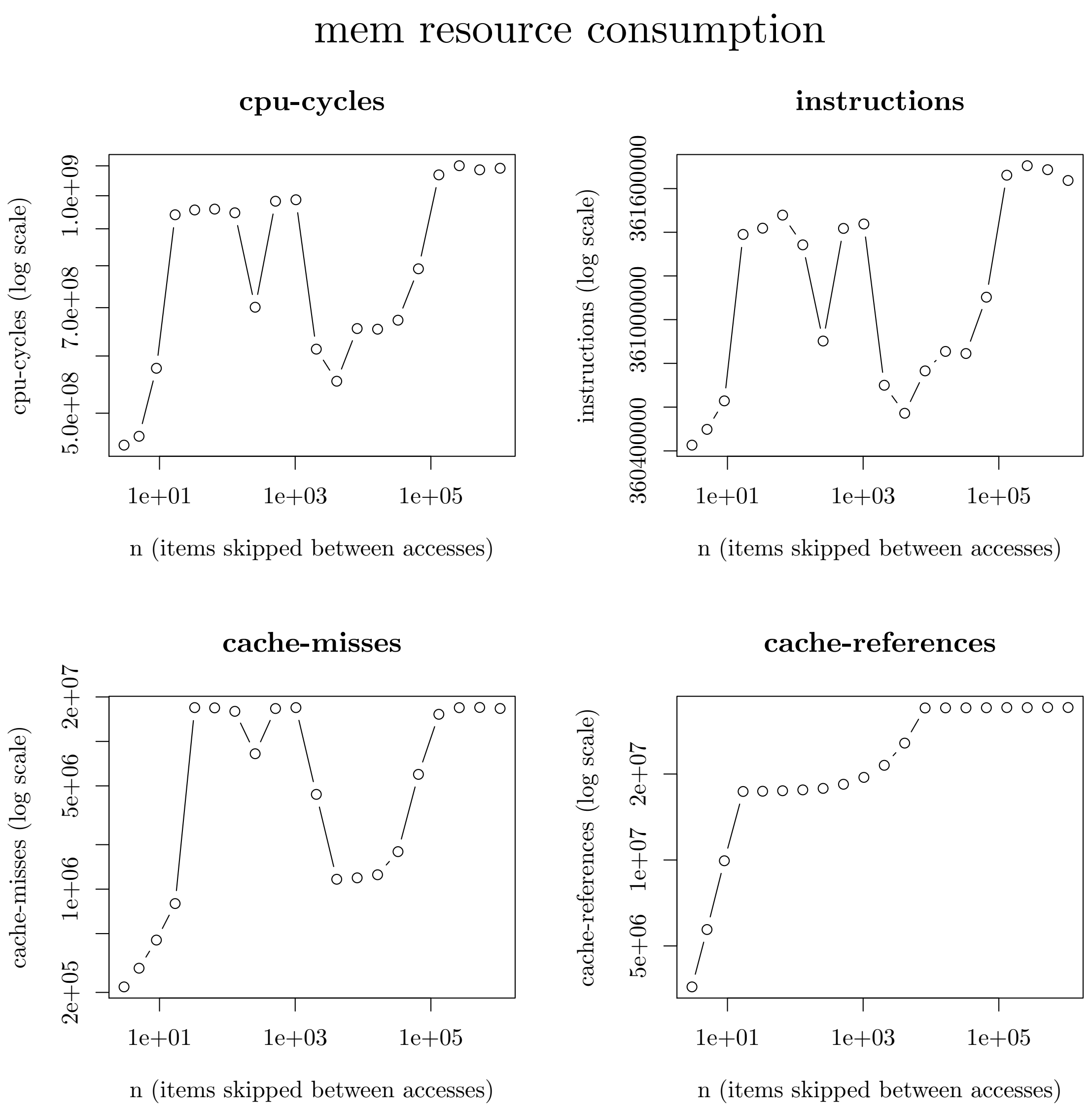


Figure 10.2: Measured resource consumption of *mem*

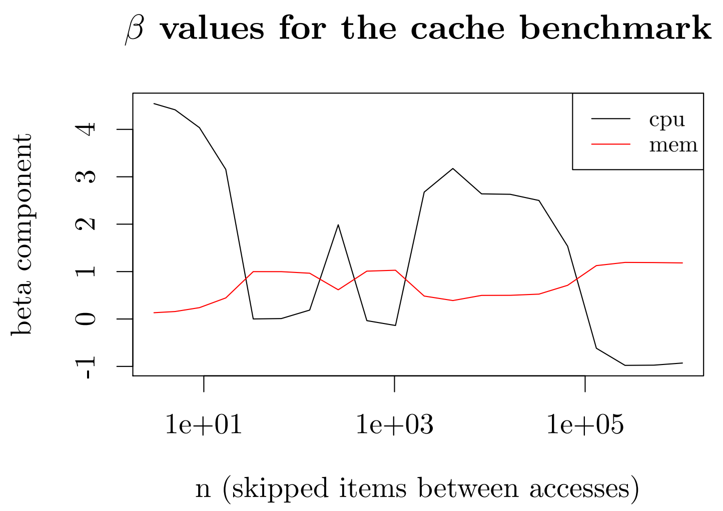


Figure 10.3: β for *mem*

- "Nonsmooth NMF" (Pascual-Montano et al., 2006) was used a to produce sparser factors (to better separate the influence of hidden factors in the programs)

- he number of hidden factors was set to two (assuming that the only interesting *computational patterns* present in the measured *programs* were accessing memory efficiently and randomly)

Figure 10.4 shows the *surrogate* found using both the basis (*cpu* and *mem(33)*) and all the cases for the *mem(n)* program.

The columns of the basis components matrix (the **W** matrix) report the estimated measures of the two potential *computational patterns* present in the *programs*. The first hidden factor captures the absence of cache-misses, with a large number of cache-references and instructions. The second hidden factor captures the cache-misses. This indicates that the *surrogate* found by *NMF* found hidden factors close to the desired *computational patterns*. Clustering the performance counters with respect to their importance to find the hidden factors, cache-misses is the most important factor.

The columns of the mixture coefficients matrix (the **H** matrix) report the estimated presence of hidden factors in each *program*. The first row represents the first hidden factor, the second row represents the second hidden factor. Clustering programs by their hidden factors composition we can see 2 groups. One group contains *cpu* and the instances of *mem(n)* for which the number of cache misses is small, for small values of n (sequential memory access) and for values of $2^{12}+1 < n < 2^{17}+1$, as previously discussed. The second group contains the *mem* basis program and the instances of *mem(n)* that resulted in a large number of cache-misses.

This confirms that the *surrogate* found by *NMF* can extract useful information regarding the behaviour of *programs*, indicating what combination of *computational resources* constitute the hidden factors that might be close to *computational patterns.*

10.2 Experimental computational complexity

In this section we validate the expressiveness of the experimental computational complexity ξ. We created and measured a few micro-benchmarks designed to stress CPU only, memory only, and both CPU and memory intensive computations. The goal of the test is to demonstrate that the experimental complexity is capable of describing the behaviour of programs with respect to interesting resource usage, and that it is compositional.

10.2.1 The experimental setup

We wrote a small number of simple micro-benchmarks:

cpu the same program presented in section 10.1.

mem(n) the same program presented in section 10.1.

f(n) consists of a cycle of length $log_2 n$, in each iteration are performed 2^{20} integer sums, 2^{20} integer multiplications, 2^{21} integer modulo operations, and a random number extraction. We tested f for values of n from 2^{16} to 2^{23}, with increments of 2^{16}.

g(n) allocates a buffer of n integers, initially set to 0, assigns n random numbers to n random locations of the buffer, sums the values of n random locations of the buffer. The g program performs n memory reads, n memory writes, n integer sums, $2n$ integer modulo operations, $3n$ random number extractions. We tested g for values of n from 2^{16} to 2^{23}, with increments of 2^{16}.

f+g(n) a program that sums the output of *f(n)* and *g(n)*. We tested *f+g* for values of n from 2^{16} to 2^{23}, with increments of 2^{16}.

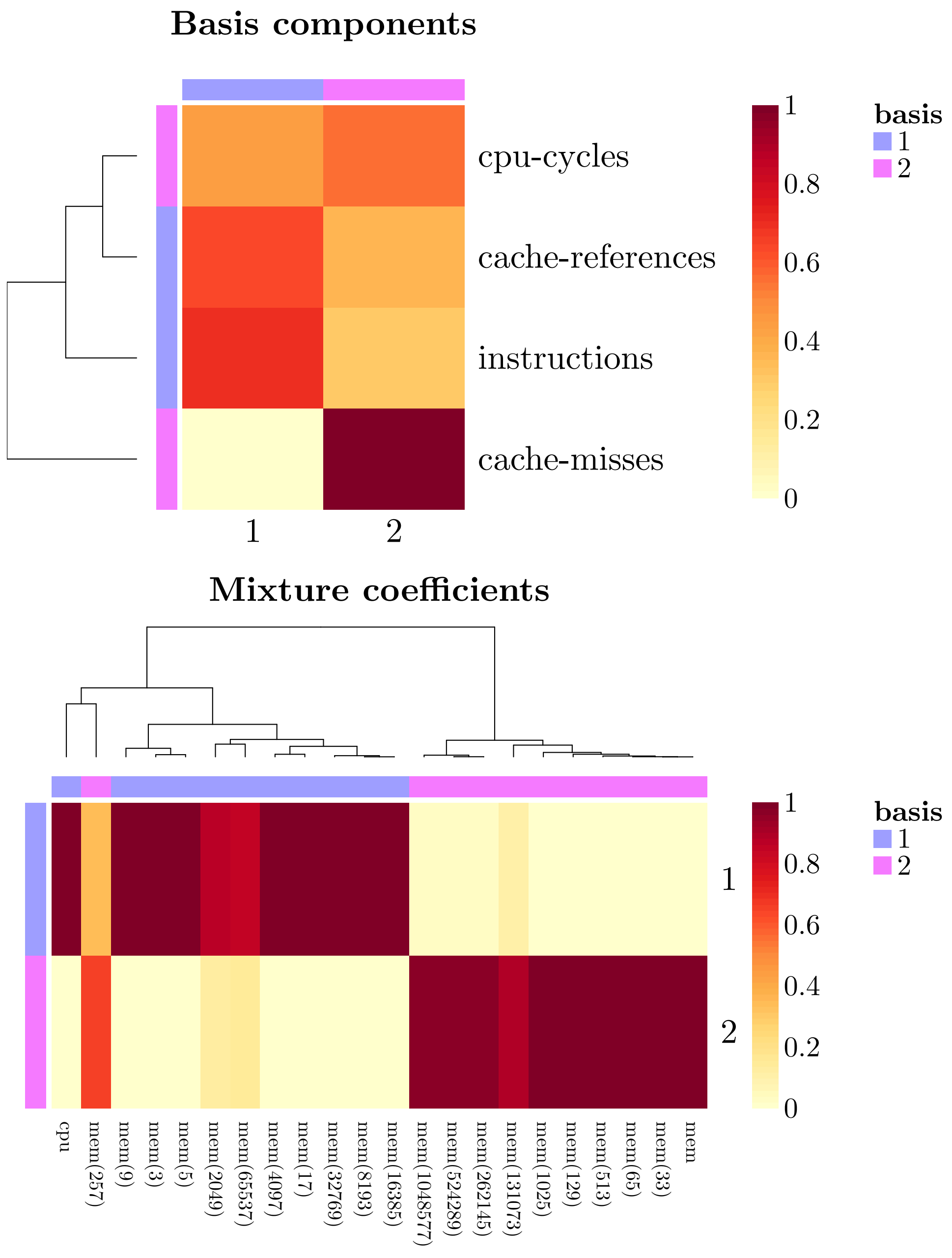


Figure 10.4: Characterization of *mem* using *NMF*

fg(n) a program with the same code as *f(n)*, with the addition of a call to *g(n)* in each iteration of the cycle (executed $log_2 n$ times). We tested *fg* for values of n from 2^{16} to 2^{23}, with increments of 2^{16}.

We ran the benchmarks on the same machine as the the 10.1 section, we also measured the same performance counters(*cpu-cycles*, *instructions*, *cache-misses*, and *cache-references*), repeating each run 30 times, reporting here only the mean value.

We used *cpu* and *mem(33)* (from now on simply called *mem*) as basis for the model. All the other programs (and cases for *mem(n)*) are expressed in terms of the basis. In other terms, β_i describes the program i as a linear combination of *cpu* and *mem*. Table 10.3 reports the measures of the basis, and section 10.1.2 discusses those measures.

Figure 10.5 shows the *resource* consumption of the *f, g, fg* and *f+g programs.*

The program f is CPU-bound and similar to the *cpu* program. as expected, the number of cpu-cycle and instructions grow logarithmically with the input size. Like *cpu*, the number of cache references and misses is extremely low, because it does not make use of memory. However, the CPI is slightly above 2, whereas the *cpu* CPI is approximately 0.5. This is probably caused by a nested loop in f that interferes with the super-scalar pipeline: the outer loop is repeated $log_2 n$ times, the inner loop is repeated 2^{20} times; *cpu* only has a single loop of 2^{20} iterations. This makes f non trivially explainable with *cpu* alone.

The program g is memory-bound. The number of cpu-cycles, instructions, and cache-references performance counters grow linearly with the input size. The number of cache-misses has a different behaviour for small values of n (approximately below 2^{20}), where the buffer is contained in cache, therefore resulting in a limited number of misses, and for large values of n, because the buffer is not contained in cache, and because we access random location of the buffer, the larger the buffer the more likely to require data not contained in cache.

As expected, the performance counters of f+g program as simply the sum of the performance counters of f and g. The performance counters of fg show the composition of the f and g functions: the values of the performance counters are the sum of the values for f with the multiplication of the times g is performed (g is called inside the loop cycle of length $log_2 n$) and the values of the performance counters of g.

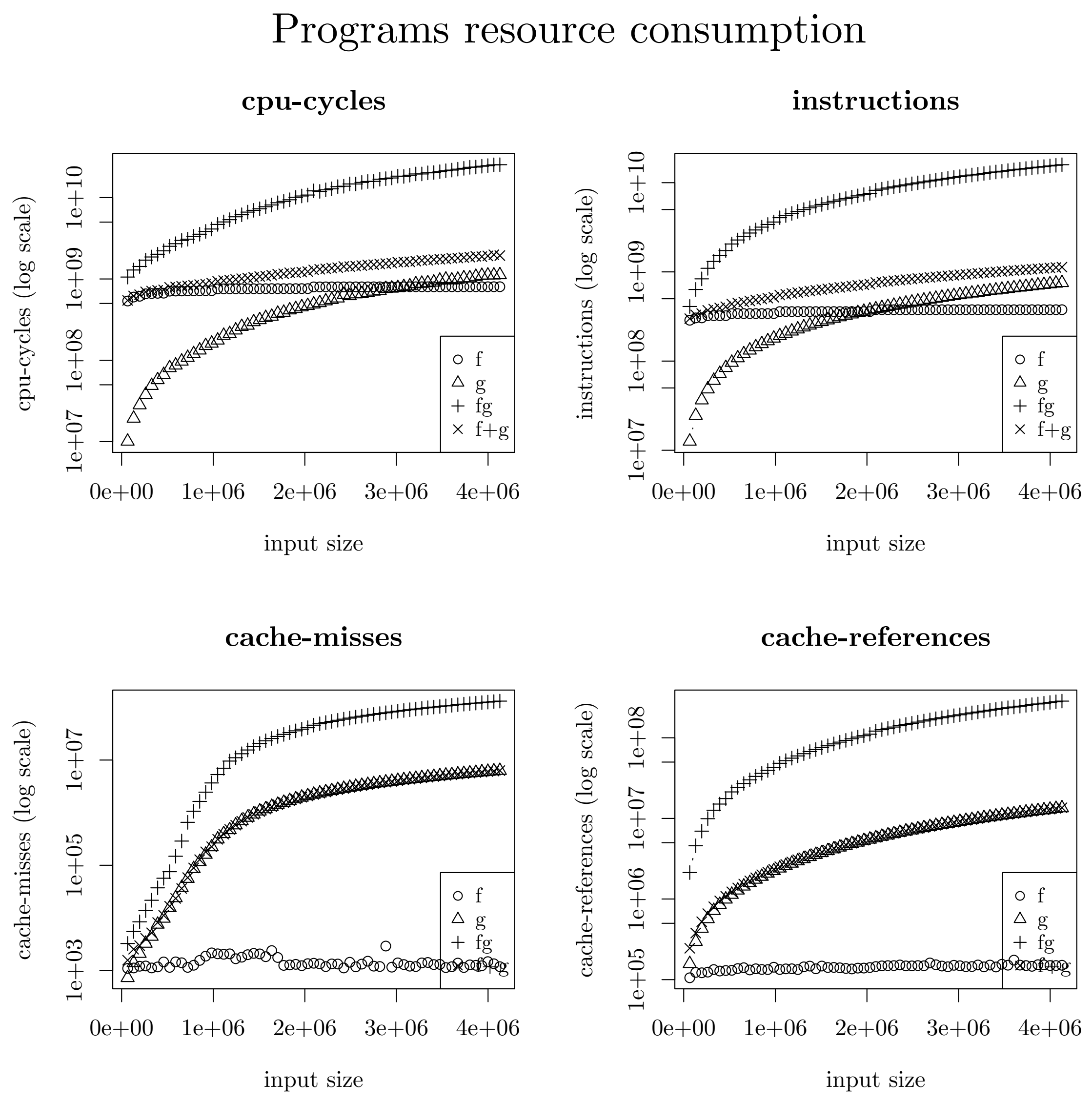


Figure 10.5: Resource (cpu-cycles, instructions, cache-misses, and cache-references performance counters) usage of the programs

10.2.2 ξ is compositional

We used our model to find the experimental complexity ξ of f and g, and we combined ξ_f and ξ_g to predict the values of the surrogate β of $f{+}g$ and fg, to show that the experimental complexity is compositional.

We used *cpu* and *mem(33)* as the the basis for the model. To find the experimental complexity ξ we used the same curves used in the previous sections (linear, quadratic, cubic, logarithmic, log-linear)

To find the experimental complexity ξ we performed curve fitting, with linear regression, using the following functions as independent functions:

linear $y = k_1 x$. This function captures loops of the form "for(in i=0; i < n; i++)".

quadratic $y = k_2 x^2$. This function captures two level nested linear loops.

cubic $y = k_3 x^3$. This function captures three level nested linear loops.

logarithmic $y = k_4 \lceil \log_2 x \rceil$. This function captures loops of the form "for (i=n; i>1; i = i/2)".

log-linear $y = k_4 x \lceil \log_2 x \rceil$. This function captures linear loops combined with logarithmic loops.

Figure 10.6 shows the β values for f and the relative ξ. The values of both the *cpu* and *mem* components of β are well described by the logarithmic curve with a small linear component.

Figure 10.7 shows the β values for g and the relative ξ. The *cpu* component of β is well described by the linear curve, the *mem* component by a combination of logarithmic and negative linear curves.

We manually defined $\hat{\xi_{f+g}} = \xi_f + \xi_g$ and $\hat{\xi_{fg}} = \xi_f + \lceil \log_2 n \xi_g \rceil$, from the manual analysis of the source code of the $f{+}g$ and fg programs. We then predicted the β values applying equation 7.3 to $\hat{\xi_{f+g}}$ and $\hat{\xi_{fg}}$. Figures 10.8 and 10.9 show the predicted and actual β values. As can be seen they are very close, confirming the compositionality of ξ.

10.3 Energy and experimental computational complexity: sorting algorithms

In this section we explore the use of energy consumption as the only measured *resource*, trying to characterize the experimental complexity of algorithms only looking at their energy consumption. The goal of this exper-

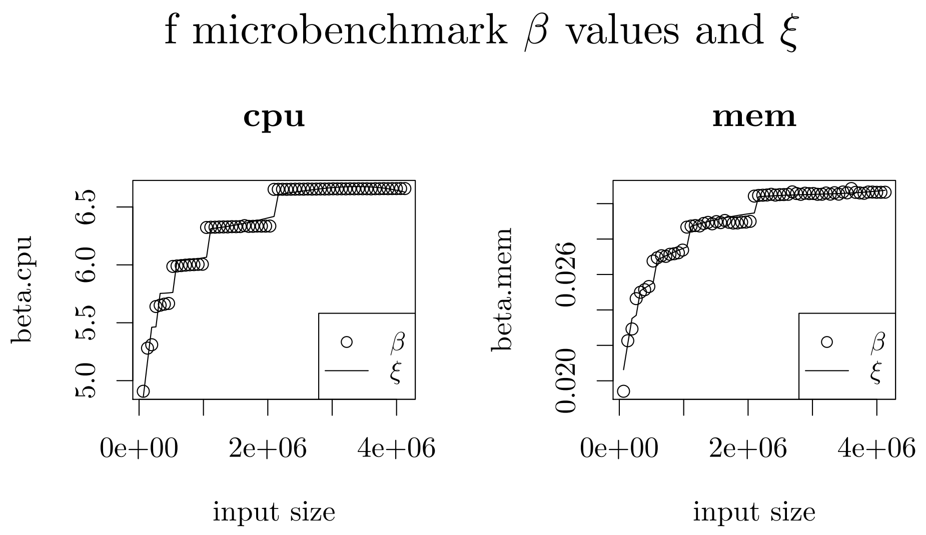


Figure 10.6: β and ξ for f

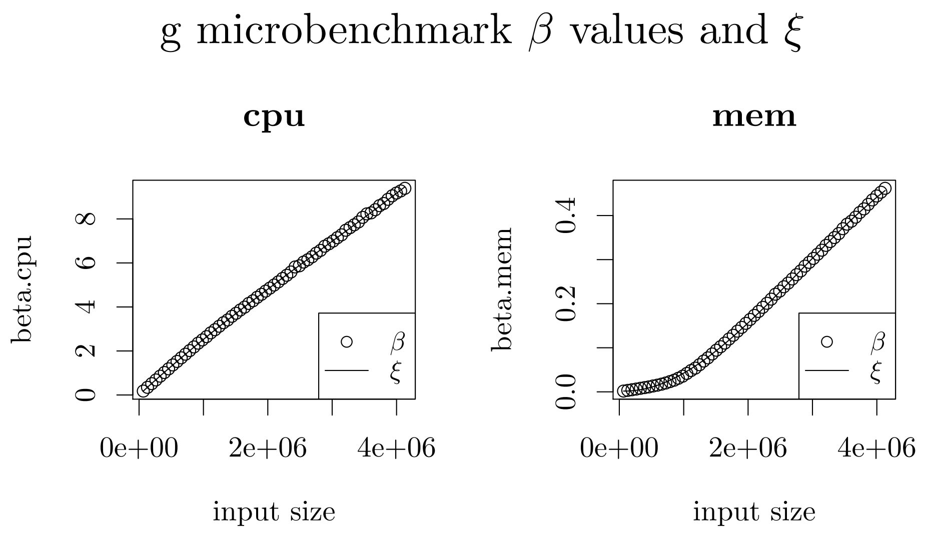


Figure 10.7: β and ξ for g

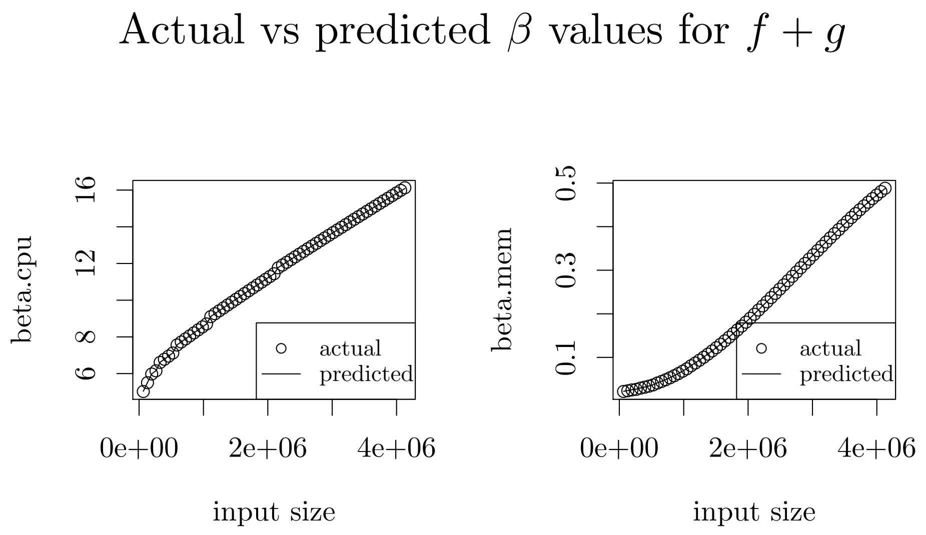


Figure 10.8: Predicted and actual β values for *f+g*

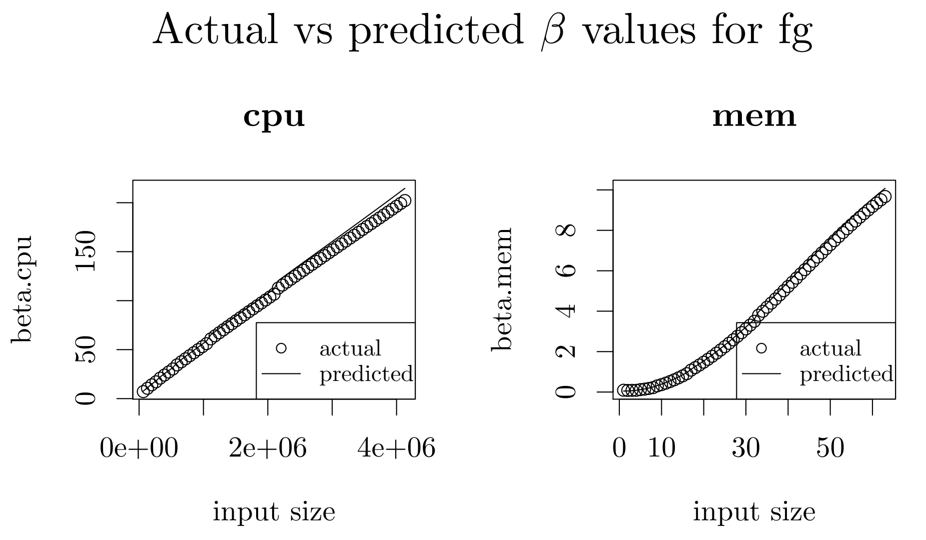


Figure 10.9: Predicted and actual β values for *fg*

iment is to show that energy is a good metric, capable of capturing the complexity of algorithms.

10.3.1 Experiment setting

We created a simple micro-benchmark (called *CPU*), to be used as basis, that performs 2^{20} integer sums. We measured the power consumption using an ammeter at the power plug, of a computer with an AMD Opteron SledgeHamer 1.4 GHz, Socket 940, 2 GB RAM, running Windows Vista Home Edition. The declared CPU power consumption is 89 W. The energy used by the *CPU* micro-benchmark is 217.54 J. We wrote an measured 3 sorting algorithms, *mergesort*, *heapsort*, and *quicksort*, written in C, using vectors of dimension from 1M to 41M elements.

Because we are measuring a single *resource* and the basis is composed of a single *benchmark*, the equation 6.17 simplifies from matrix to scalar operations only, as shown in equations 10.1 and 10.2, where μ_{CPU} is the measured energy consumption of *CPU*, and $\mu_{p(i)}$ is the energy consumption of the *target program*.

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

$$\mu_{p(n)} = \mu_{\text{CPU}}\beta_n \tag{10.1}$$

$$\beta_n = \frac{\mu_{p(n)}}{\mu_{\text{CPU}}} \tag{10.2}$$

For every input size n we found the corresponding β_n, simply dividing the energy consumption of the algorithm by the energy consumption of *CPU*.

To find the experimental complexity ξ we performed curve fitting, with linear regression, using the same curves described in section 10.2.2 (linear, quadratic, cubic, logarithmic, and log-linear).

10.3.2 Experimental results

Figure 10.10 shows the single component of β_n (simply the ration between *heapsort* and the *CPU* benchmark) as the input size changes, and the ξ. The best fitting curve was *log-linear*, with $R^2 = 0.9999$ and multiplicative factor 4.74×10^{-09}. The complete formula of ξ is shown in equation 10.3.

$$\xi(n) = 4.74 \times 10^{-09} n \log_2 n \tag{10.3}$$

ξ of heapsort

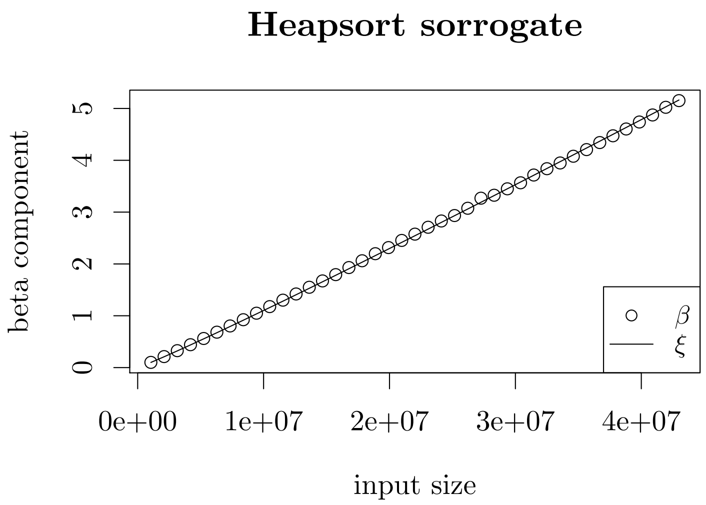


Figure 10.10: β and ξ for *heapsort*

Figure 10.11 shows the single component of β_n (simply the ration between *mergesort* and the *CPU* benchmark) as the input size changes, and the ξ. Like for *heapsort*, the best fitting curve was *log-linear*, with $R^2 = 0.9998$ and multiplicative factor 1.08×10^{-08}. The complete formula of ξ is shown in equation 10.4.

$$\xi(n) = 1.08 \times 10^{-08} n \log_2 n \tag{10.4}$$

ξ of mergesort

Figure 10.12 shows the single component of β_n (simply the ration between *quicksort* and the *CPU* benchmark) as the input size changes, and the ξ. The best fitting curve was *quadratic*, with $R^2 = 0.9934$ and multiplicative factor 2.38×10^{-14}. The complete formula of ξ is shown in equation 10.5.

$$\xi(n) = 2.38 \times 10^{-14} n^2 \tag{10.5}$$

ξ of quicksort

For all sorting algorithms the experimental computational complexity ξ was the same as the theoretical time complexity O. Interestingly, the ξ of

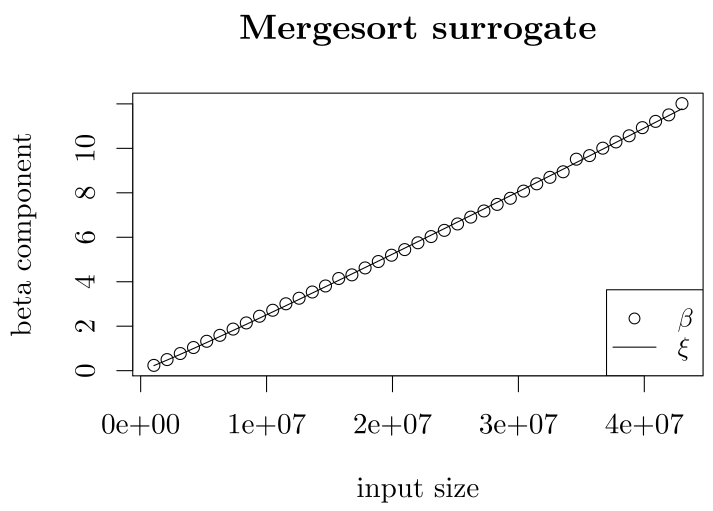


Figure 10.11: β and ξ for *mergesort*

quicksort was the same curve as the theoretical worst case $O(n^2)$ instead of the average case $O(n \log n)$. However, as can be seen in figure 10.12 the growth of the *surrogate* is not as steep as n^2. It lies between $n \log n$ and n^2, a curve not modelled by ξ. Allowing linear combinations of curves, the best fit is the sum of quadratic and log-linear. Figure 10.13 shows the fitting of the *surrogate* and the ξ described in formula 10.6.

$$\xi(n) = 1.54 \times 10^{-14} n^2 + 3.17 \times 10^{-9} n \log_2 n \tag{10.6}$$

ξ of quicksort, with quadratic and log-linear combined

10.4 Micro-architecture independence: limits of a single dimension

To verify the limits of using a single value as *surrogate* presented in chapter 7.6, we compared the results for *mergesort* characterized using *CPU* only presented in the previous section, with the same *programs* (the *CPU* benchmark and *mergesort*) on different machines. We expect the *surrogate* to show small differences for similar architectures, and large differences for different architectures. In the first experiment we use a machine of the same

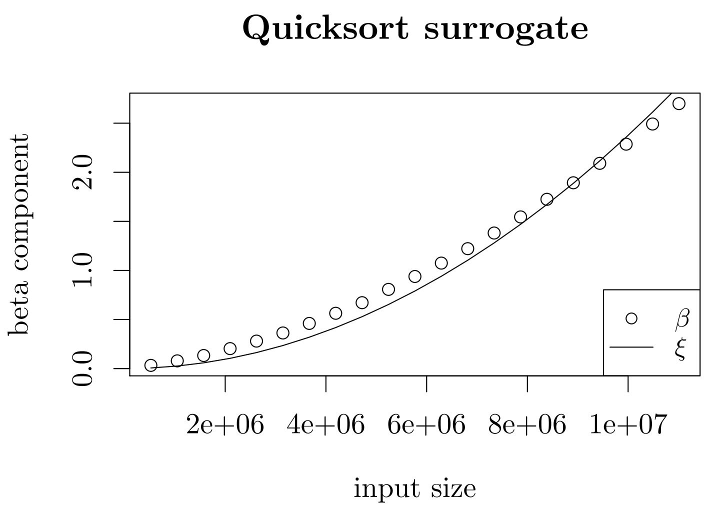


Figure 10.12: β and ξ for *quicksort*

family, but with a slightly newer processor. In the second experiment we use a completely different architecture. We present the results of characterizing using measures from each architecture separately, as well as in the same model. We conclude this section showing that using the *target program* to characterize itself, the model error disappears, and that using multiple *programs* as basis (instead of just a single program), the model error becomes much smaller.

10.4.1 Different architectures

In the first experiment we used the same micro-architecture as in the previous section, but with a slightly newer processor: AMD Athlon 64 X2 Toledo 4200++, 2.20 GHz, Socket 939, 2 GB RAM, running Windows Vista Home Edition. The differences between the Opteron machine (presented in the previous section) and the Athlon machine are:

- the socket 939 is similar to socket 940, but is revised removing the need for buffered memory, doubling peak memory bandwidth;
- the clock rate of the Athlon machine is faster than the Opteron (2.2 GHZ versus 1.4GHz);

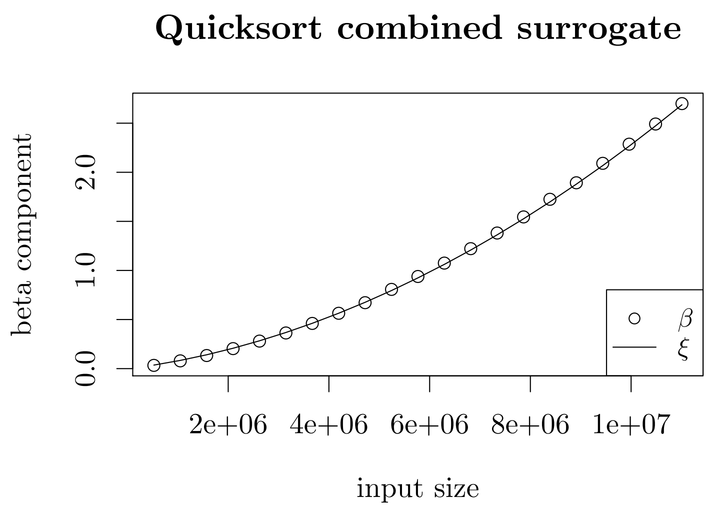


Figure 10.13: β and ξ for *quicksort*, allowing linear combination of curves

- the Opteron is single core, the Athlon is dual core;

We measured the energy consumption of the *CPU* micro-benchmark (177.25 J), and of *mergesort.* Because the micro-architecture is similar, the *computational patterns* will have a similar *resource* consumption on the Athlon and Opteron machines. However, because the CPU is not the same, the *computational patterns resource* consumption not be exactly the same. In particular we expect the CPU to run faster, while the cost of accessing memory will be similar, thus violating the conditions expressed in equation 7.35. We expect therefore the model created using a single *program* in basis to produce inconsistent *surrogates.*

$$\xi(n) = 5.44 \times 10^{-9} n \log_2 n \tag{10.7}$$

ξ of mergesort on Athlon

In the second experiment we used a different micro-architecture: an Intel ATOM N230, 512KB L2, 4MB RAM. Measured the energy consumption of the *CPU* micro-benchmark (96.62 J), and of *mergesort.* Because the architectures of the Opteron and ATOM machines are substantially different, the *computational patterns* will also have a largely different *resource* con-

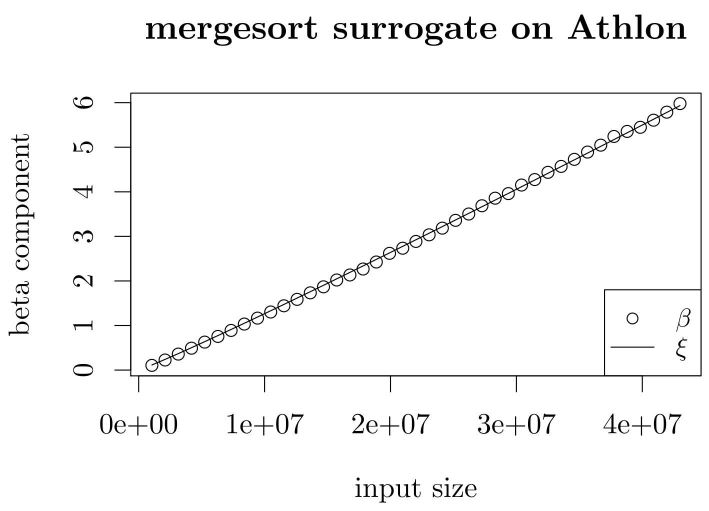


Figure 10.14: β and ξ for *mergesort* on Athlon

sumption, violating the conditions expressed in equation 7.35. We expect therefore the model to produce inconsistent *surrogates*.

Figure 10.15 shows the single component of β_n (simply the ration between *mergesort* and the *CPU* benchmark) as the input size changes, and the ξ. Like for *mergesort* on the AMD machine, the best fitting curve was *log-linear*, with $R^2 = 0.9998$ and multiplicative factor 1.79×10^{-9}. The complete formula of ξ is shown in equation 10.8.

$$\xi(n) = 1.79 \times 10^{-9} n \log_2 n \tag{10.8}$$

ξ of mergesort on ATOM

The ξ of *mergesort* found using the measures on both the Athlon, Opteron, and ATOM follows the same curve $n \log n$. However, the multiplicative factors are different. In particular, on the ATOM is an order of magnitude smaller than on the Opteron. Predictions made with the β found using measures on Opteron alone, would produce a large error on the ATOM machine.

Figure 10.16 shows the *surrogates* calculated independently on the 3 machines. We can verify that the *surrogate* calculated on Opteron is close but not equal to the *surrogate* calculated on the Athlon. The distance

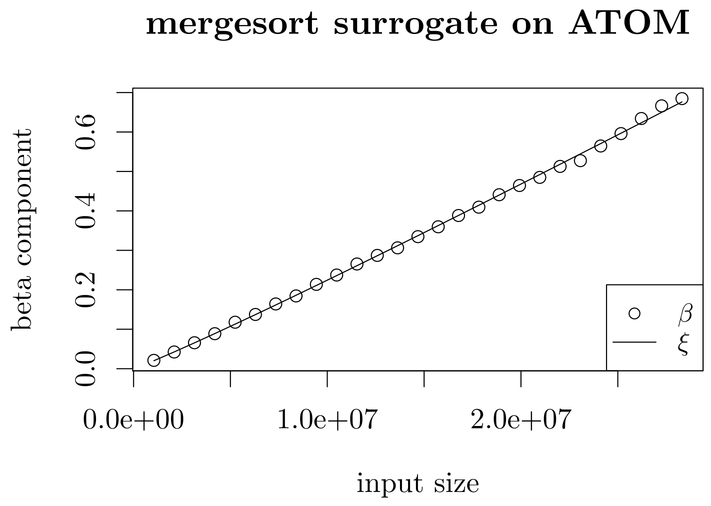


Figure 10.15: β and ξ for *mergesort* on ATOM

increases as the difference between architectures becomes significant: the *surrogates* on Opteron and ATOM are radically different.

10.4.2 Using all the measures in the same model, only 1 *program* as basis

As expected from the theoretical results in chapter 7.6, combining the measures from different architectures in the same model, but using a single *program* as basis, with a different structure than the *target program*, will lead to an imprecise model. We combined the measures from the previous experiments, using *CPU* as basis, and *mergesort* as the *target program*, on Athlon, Opteron and ATOM.

For every input size we build a *surrogate*, using the measures of the *CPU* micro-benchmark on the 3 architectures as matrix $\mathbf{X}$ (single column), and the measures of *mergesort* on all architectures as $\mathbf{y}$. In this experiment the *surrogate* is the same for all the architectures.

To test the goodness of the model, we plotted the actual values versus the fitted values, as shown in figure 10.17. The fitted values show the representation that the model makes of the actual data. The characterization error is large, as evident by the difference between the actual values and the fitted values, for all the architectures.

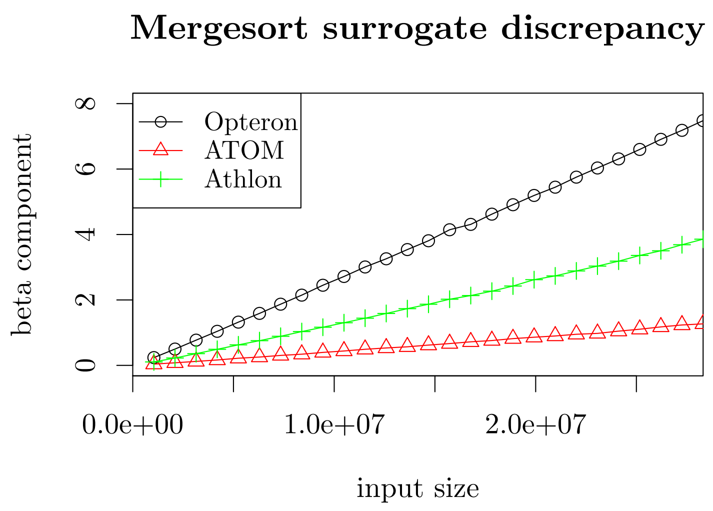


Figure 10.16: Discrepancy in *surrogates* between Athlon, Opteron and ATOM

10.4.3 Using all the measures in the same model, with a larger basis

To further verify the theoretical model, we added one program to the basis. We measured *quicksort* on a single input size $n = 1M$, on all the architectures. We used *quicksort* as an example of *program* different than the *target program* (they have different computational complexity), composed of different *computational patterns* than the *CPU* micro-benchmark (*quicksort* uses memory, while *CPU* only performs arithmetic operations).

For every input size we build a *surrogate*, using the measures of the *CPU* micro-benchmark and *quicksort(1M)* on the 3 architectures as matrix $\mathbf{X}$ (2 columns), and the measures of *mergesort* on all architectures as $\mathbf{y}$. In this experiment the *surrogate* is the same for all the architectures.

As expected, the characterization using a basis with more than a single *program* is more accurate. The differences between the Opteron, Athlon, and ATOM architectures are better captured than in the previous experiment, even using a small basis (2 *programs*).

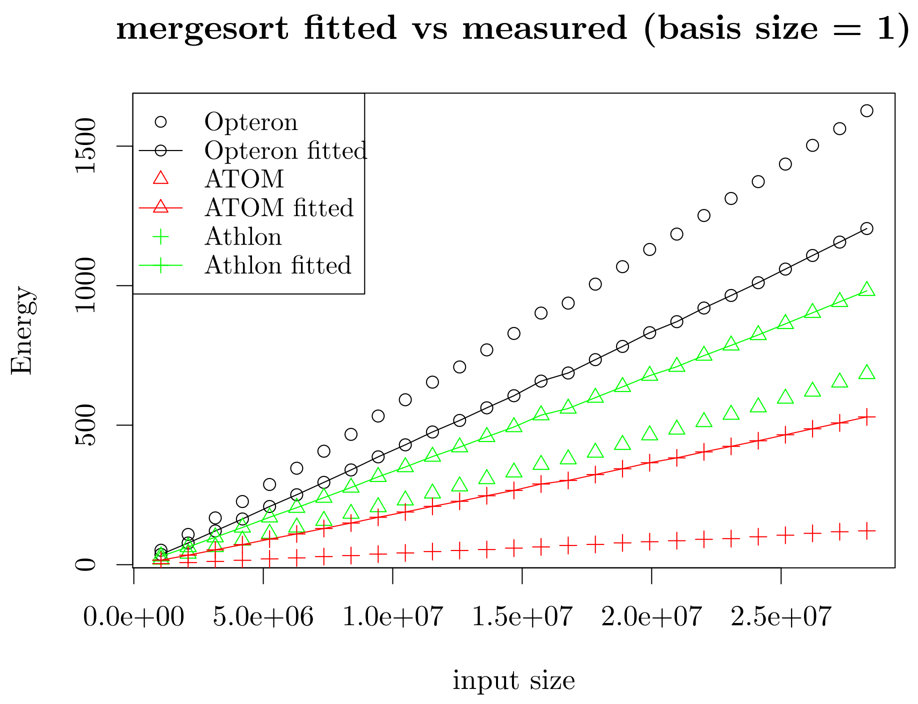


Figure 10.17: Measured versus fitted values using a single *program* as basis on 3 different architectures

10.4.4 Using the *target program* as basis

To further confirm the theoretical results in chapter 7.6, we verified that using the *target program* as the only *program* in the basis, the *surrogates* found independently on the 3 architectures will be the same.

The *surrogates* as the input size grows are shown in figure 10.19.

10.4.5 Discussion

In this section we have verified that using a single *program* as basis can only lead to precise model in 2 cases:

- the architecture is substantially the same
- the *target program* is the same as the *program* in the basis

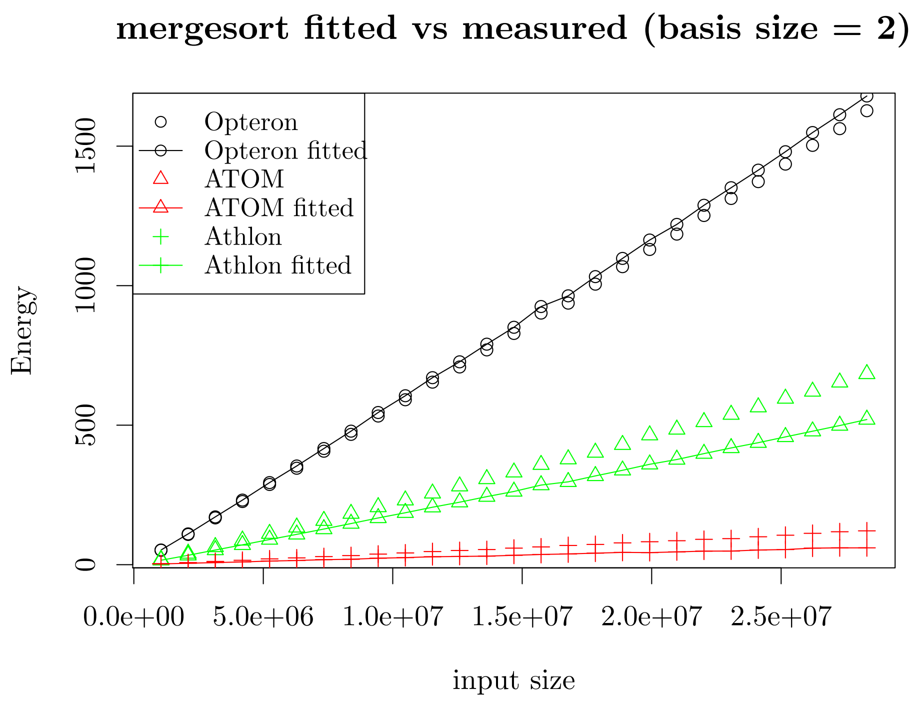


Figure 10.18: Measured versus fitted values using 2 *programs* as basis on 3 different architectures

In chapter 11 we will show that using a sufficiently large number of *benchmarks* and *resources* from different *computational environments* the model becomes accurate, and the prediction has a small error.

Most of the approaches to benchmarking used nowadays attempt to use a single number to characterize computer systems. This experiment shows that this approach inherently leads to inaccurate models. In chapter 13 we will show that using more than a single dimension to characterize hardware (the dimensionality of β), we can build expressive models, and the predictions can be accurate enough to chose the best device for the *target program*. The dimensionality of β does not need to be large, in fact it should be as small as possible to ensure readability of the *surrogate*, and avoid over-fitting.

This verifies the the theoretical result described in chapter 7.6: the model

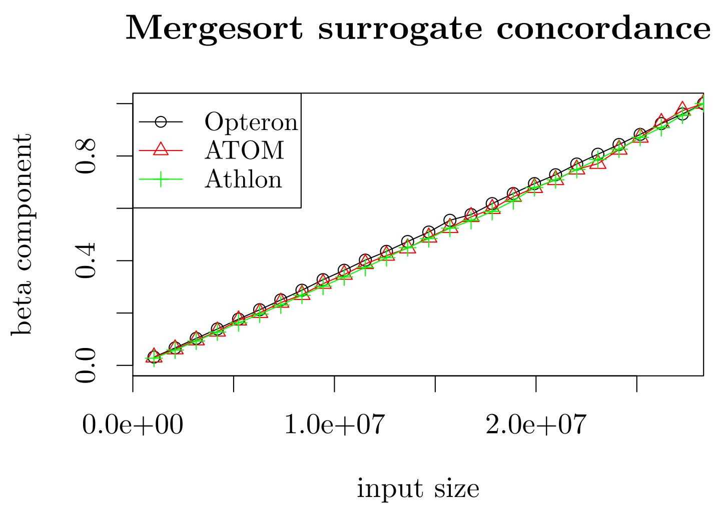


Figure 10.19: Concordance in *surrogates* between Athlon, Opteron and ATOM, using the *target program* as basis

created using a single metric can not be accurate when the underlying *computational environment* changes significantly.

10.5 Conclusions

In this chapter we have presented a few experiments using toy benchmarks, designed to verify the expressiveness, the properties and the limits of the *surrogate* β and the *experimental computational complexity* ξ. We have seen that characterizing a *target program* using a single *program* as basis leads to a large error as we apply the same model to different architectures. We have verified that the *surrogate* and the *experimental complexity of software* are expressive and compositional. We have also seen that energy is a *resource* that reveals the theoretical complexity of algorithms.

Chapter 11

Validating performance prediction on multiple architectures using CPUSPEC

In this chapter we show the prediction accuracy of our model, using the widely used benchmarking suite SPEC CPU, and predicting the completion time of the suite on a machine not used to train the model. This demonstrates that the model is capable of accurate cross-architecture *resource* consumption predictions.

11.1 CPUSPEC

We tested our model using the SPEC CPU2006 data publicly available on the SPEC website [1], using the completion time on each report as a different computational resource, choosing a subset of the suite as the benchmarks and the rest of the suite as the target programs.

We decided to use the SPEC CPU suite because it has been proven to be a representative workload for real world applications, it has been used to prove the performance of several other performance predictors, and all the data needed to replicate the experiment is publicly available, allowing repeatability of the presented results.

[1] `https://www.spec.org`

We downloaded the data from the SPEC website, we kept only the complete results that had both SPEC INT and SPEC FP, creating one row for each machine and one column for each program of the suite. The resulting matrix contained 1133 rows and 29 columns. The software and scripts used to download and aggregate the data, to create the models, to make predictions and to check the prediction errors is available with open source license [2].

11.1.1 Bias error

Performance measures will have bias error, as shown by Mytkowicz et al. (2009), because of unexpected phenomena. Our dataset consists of 4082 different SPEC INT or SPEC FP reports, of which some from the same machine, some from different. 2634 reports from SPEC INT and 2568 from SPEC FP. In some case we have more than one report from a machine (repeated experiment). We extracted the reports that have been run on machines with the same components and analyzed the difference in the measures to estimate the bias error. We calculated the expected measure for a program as the average value $\bar{x} = \frac{\sum x}{m}$ where m is the number of measure we have for that program, the relative error as $e = \frac{x-\bar{x}}{\bar{x}}$ and the bias error of a program as the Root Mean Squared Error (RMSE) of the relative errors $RMSE = \sqrt{\frac{\sum e^2}{m}}$.

The total bias error (RMSE) of the CPU SPEC is equal to 0.0330, the bias error of the programs varies from 1% to more than 7%. Therefore we expect an error when trying to predict the program's completion time of the same order of magnitude as its bias error. We will study the correlation between the prediction error and the bias error. We expect he prediction error to be positively correlated with the bias error (guessing the correct completion time will be harder for those programs with a large bias error).

11.2 Predicting completion time using the linear regression solver

The primary objective of our experiment is to test the prediction accuracy of our prediction model using the data from CPU SPEC 2006. In each experiment we proceeded as follows:

[2] `https://github.com/vslab/Energon`

Table 11.1: Programs bias error

type	program	bias error
SPEC INT	445.gobmk	0.0101
SPEC INT	458.sjeng	0.0107
SPEC FP	444.namd	0.0108
SPEC FP	453.povray	0.0113
SPEC INT	464.h264ref	0.0113
SPEC INT	473.astar	0.0126
SPEC INT	403.gcc	0.0129
SPEC FP	454.calculix	0.0129
SPEC FP	416.gamess	0.0132
SPEC INT	401.bzip2	0.0133
SPEC FP	447.dealII	0.0146
SPEC INT	400.perlbench	0.0154
SPEC INT	483.xalancbmk	0.0158
SPEC FP	450.soplex	0.0164
SPEC INT	429.mcf	0.0170
SPEC INT	456.hmmer	0.0181
SPEC FP	433.milc	0.0207
SPEC INT	471.omnetpp	0.0216
SPEC FP	482.sphinx3	0.0218
SPEC FP	465.tonto	0.0226
SPEC FP	481.wrf	0.0280
SPEC FP	434.zeusmp	0.0340
SPEC FP	435.gromacs	0.0351
SPEC INT	462.libquantum	0.0432
SPEC FP	437.leslie3d	0.0535
SPEC FP	459.GemsFDTD	0.0593
SPEC FP	470.lbm	0.0681
SPEC FP	410.bwaves	0.0701
SPEC FP	436.cactusADM	0.0768

1. we chose the number m of resources used to build the model (*training size*), and the number n of benchmarks (*basis size*)

2. we picked a program form the CPU SPEC suite as the target program

3. we randomly selected m results from the CPU SPEC 2006 database. This represents the knowledge we have when we build the model. We extracted one results from the rest of the CPU SPEC 2006 database to test the model. We created a matrix $\mathbf{M_{model}}$ with the selected m CPU SPEC results, each system is a row of the matrix and each measure of the program is a column

4. we normalized each row of $\mathbf{M_{model}}$ (to make sure each computational environment had the same weight in the model). This matrix constitutes the training set of our model.

5. we pick one of the programs of CPUSPEC as the *target program*

6. we selected a random subset of n programs in the CPU SPEC suite to be used as the *benchmarks*, the basis for our model. The probability of choosing a program as basis is proportional to its correlation with the *target program*. Similar programs will therefore be preferred as basis.

7. we created the matrix $\mathbf{X}$ keeping only the columns of $\mathbf{M_{model}}$ relative to the n *benchmarks*

8. we found the *surrogate* β of the *target program*, using Robust Least Squares as the solver, such that $\mathbf{X}\beta = \mathbf{y}$, where $\mathbf{y}$ is the vector of measures of the target program for the systems used to build the model

9. we predicted the resource consumption of the target program on 100 randomly selected systems from the CPU SPEC database that were not used in the training set ($\mathbf{M_{model}}$) by estimating the resource consumption of the target program ($\mathbf{p}$), multiplying the vector containing the measures of resource consumption of the benchmarks on the new system $\mathbf{x}$ by the surrogate $\mathbf{p} = \mathbf{x}\beta$

10. we calculated the relative error between the predicted resource consumption $\mathbf{p}$ and the real resource consumption $\mathbf{t}$ as $\frac{\mathbf{p-t}}{\mathbf{t}}$

11. we also calculated Relative Absolute Error (RAE), defined in equation 11.1 s, where p_i is the predicted value, μ_i is the measured value and $\hat{\mu}$ is the mean of the measured values.

12. we repeated this experiment for every program in the CPU SPEC suite, 100 times for each program (to ensure statistical significance, selecting random *benchmarks* and random *resources*), for basis sizes varying from 2 to 25 *benchmarks*, and training sizes from 40 to 200. A total of 1670400 predictions have been performed.

$$E_{\mathrm{RAE}} = \frac{\sum |p_i - \mu_i|}{\sum |\mu_i - \hat{\mu}|} \tag{11.1}$$

RAE

11.2.1 Completion time prediction accuracy

In this section we report the prediction accuracy of our model choosing the basis (the programs used as benchmarks). Figure 11.1 shows the RAE changing the amount of information available when building the model, both in terms of small training set (40 machines) to large training set (200 machines) and small set of benchmarks (10) to a large set of benchmarks (25). RMSE is a good measure of the overall prediction error because it includes both the bias and the variation of the errors. Every combination of basis size and training set shows the RMSE of a large number of experiments: for each program (29) were repeated 100 experiments, each including 100 predictions, for a total of 290 thousands predictions.

As expected, with limited information available when building the model, the predictions contain a large error.

With a limited number of *benchmarks* in the basis, RAE is consistently high even using a large amount of measures. This can be explained by the fact that the programs in CPU SPEC are very different. Therefore, using a small set of randomly selected *benchmarks* as the basis, is unlikely that they will contain representative aspects of the *target program*. The resulting model will not be able to characterize the important characteristics of the *target program* and the predictions will have a large error.

For small values of training size is noticeable a minimum in RAE for values of basis size approximately a quarter of the training size. This is an expected phenomenon, because a large basis with not enough data points results in a model that over-fits the data, with little residuals, but large prediction error.

The number of *benchmarks* used in the basis should therefore be chosen depending on the amount of measures available for the training set, making

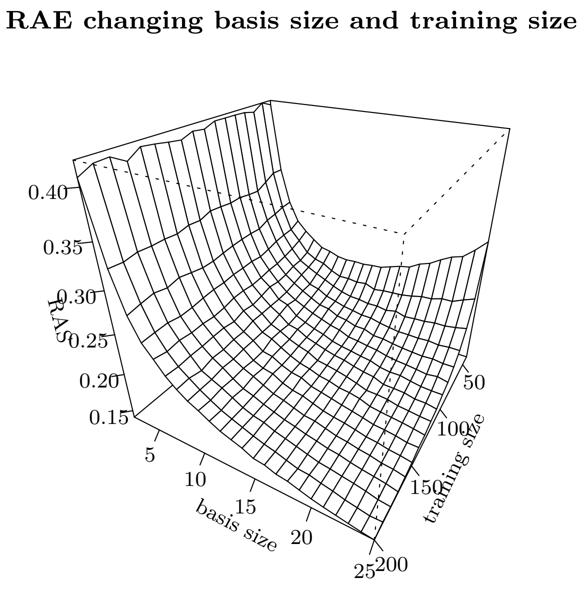


Figure 11.1: Completion time prediction RAE for different basis sizes and training set sizes

sure that the number of rows in the matrix **X** is considerably larger than the number of rows.

Predictions accuracy for each program

In this section we will explore the detailed predictions in two cases: a limited amount of information available when building the model (using both a small basis size and a small training set), and a large amount of information (using a large basis and a large training set).

Figure 11.2 shows the distribution of the relative prediction errors and the fitting residuals, using 5 *benchmarks* as basis, and 50 different machines, for each program.

Figure 11.3 shows the distribution of the relative prediction errors and the fitting residuals, using 25 *benchmarks* as basis, and 100 different machines, for each program.

Programs are ordered by their bias error, to show that there is a negative correlation between the prediction accuracy the bias error.

The overall RAE of the case with 5 *benchmarks* and 50 machines is 0.27, whereas with 25 *machines* and 100 machines RAE is only 0.16.

The improvement is noticeable especially in the benchmarks with high bias error, such as 459.GemsFDTD (where the prediction error mean and standard deviation went from -0.05 and 0.32, to -0.01 and 0.16), or 470.lbm (that went from -0.11 and 0.78, to -0.03 and 0.26). This improvement is expected, because with a larger basis more detailed behaviour can be captured, and with a larger training set the risk of over-fitting decreases.

11.2.2 Fitting residuals, bias, and prediction error

The prediction error and the regression residuals are closely related. Their correlation is as high as 0.97. This is also evident from figures 11.2 and 11.3 and tables 11.2 and 11.3 where, for each program, the distribution of prediction error and fitting residuals are shown. Programs with large residuals have poor performance predictions (e.g. 436.cactusADM or 410.bwaves), programs with small residuals have good predictions (e.g. 445.gpbmk or 458.sjeng).

Also, as expected, bias error is positively correlated with prediction error (0.89). The programs in figures 11.2 and 11.3 are ordered by bias error, and the prediction error is generally very small in the first programs and large in the last programs.

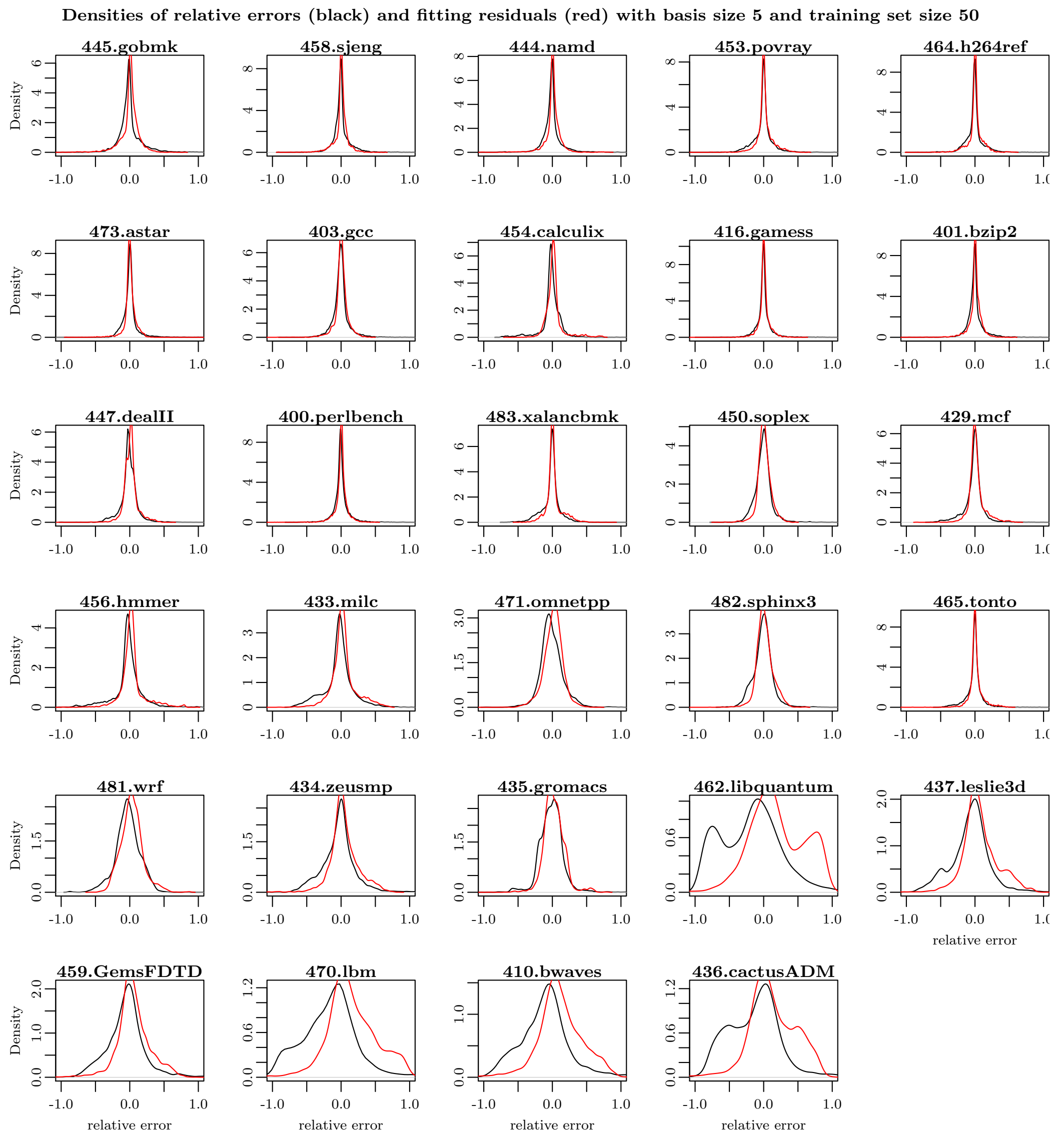


Figure 11.2: Completion time prediction accuracy with basis size 5 and training size 50, ordered by bias error

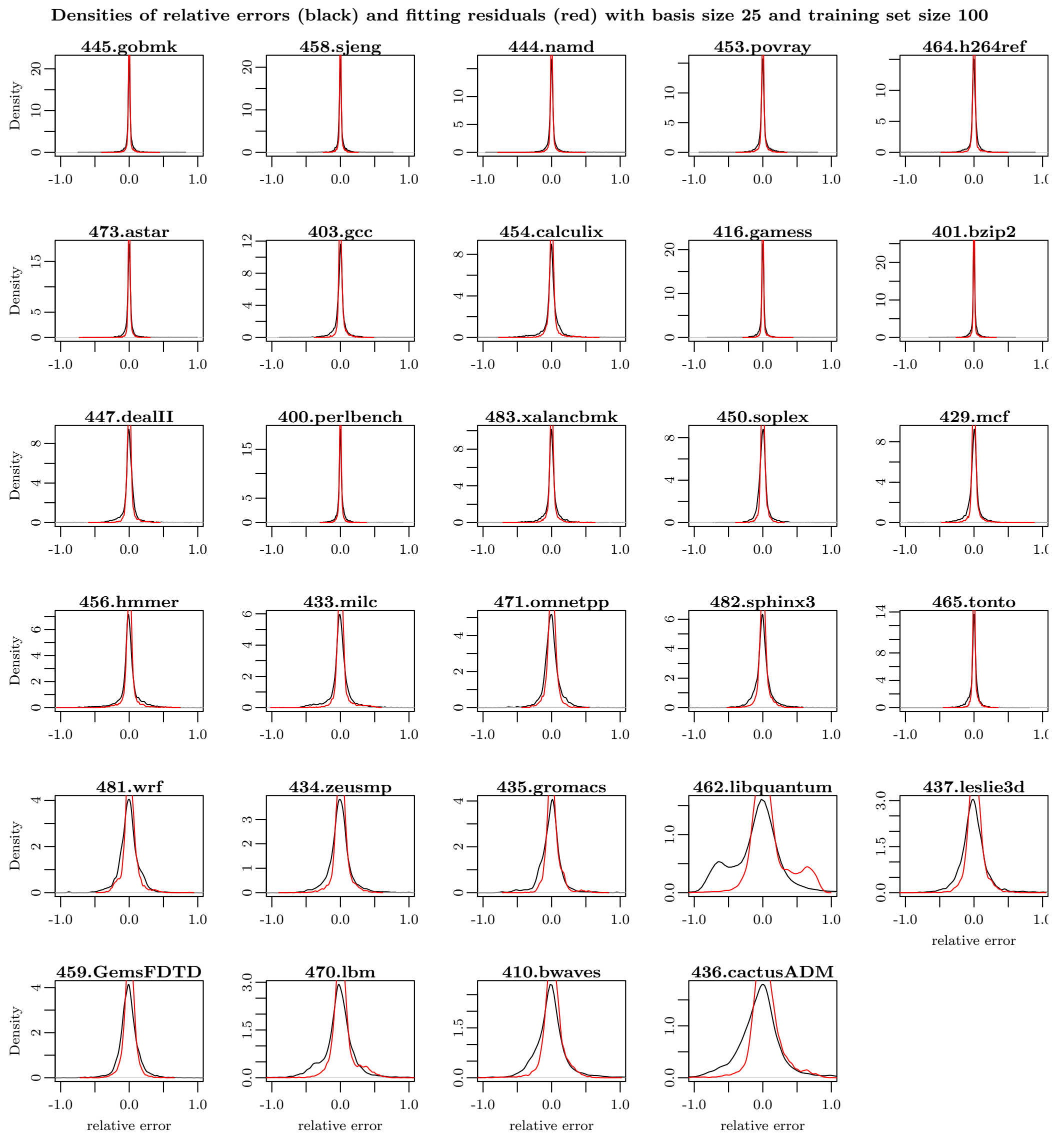


Figure 11.3: Completion time prediction accuracy with basis size 25 and training size 100, ordered by bias error

Therefore, fitting residuals, bias error, and prediction error, are closely related. Knowing the bias error of a program gives an immediate idea of the expected quality of the prediction quality. Looking at the distribution of the regression residuals is possible to estimate the prediction uncertainty.

11.2.3 Discussion

Recently an analysis of the redundancy of the CPU SPEC 2006 suite (Kareem and Singh, 2015) showed that 429.mcf, 471.omnetpp, 403.gcc, and 462.libquantum exhibit different behaviour with respect to the rest of the suite. The authors used Principal Component Analysis to cluster the programs, similarly to what previously done by Phansalkar et al. (2005).

From the existing literature we might conclude that those programs that do not fit into any cluster found by PCA will have poor performance predictions, because of the reduced similarity with the programs in the basis. However, with our experiment, we found that this is generally not true. With the exception of 462.libquantum, all the programs that are outside the main PCA clusters (429.mcf, 471.omnetpp, and 403.gcc) have accurate performance prediction.

11.3 Conclusions

The experiment presented in this chapter shows that our model can be used to predict the *resource* consumption (in particular the completion time) of *programs* using a black box approach. We tested the model using the data from the SPEC CPU 2006 suite, using a subset of the suite as *surrogates* and predicting the completion time of the remaining *programs*, with an increasing accuracy as we use more data to build the model. The model has been extensively tested with this data (290000 predictions), making it a reliable measure of accuracy. We have also shown that the fitting residuals can be used as a reliable estimation of the prediction error.

This model can also be used to characterize the behaviour of a program, only using measures of its resource usage.

Linear regression, despite its simplicity, offers comparable or superior prediction accuracy than more complicated approaches (as described in Sharkawi et al. (2009)), using only completion time instead of a large set of performance counters (as describe in Phansalkar et al. (2005); Sharkawi et al. (2009)). It should therefore be preferred to more complicated models.

This model could be used in an HPC scheduler (where the source code of the tasks is seldom available) to better allocate the nodes (the right number of nodes, with the right amount of memory); or to predict the resources needed by a task in a cloud, to consolidate the virtual machines while keeping the required SLA; or in an operative system scheduler, because once the model has been built, the prediction is computationally not expensive.

	Error mean	Error SD	Residuals mean	Residuals SD
445.gobmk	0.01	0.21	-0.00	0.16
458.sjeng	0.01	0.13	-0.00	0.10
444.namd	0.03	0.31	-0.01	0.24
453.povray	-0.01	0.15	0.02	0.12
464.h264ref	-0.00	0.15	0.00	0.11
473.astar	0.01	0.14	0.00	0.09
403.gcc	-0.00	0.14	0.00	0.11
454.calculix	-0.02	0.15	0.02	0.13
416.gamess	0.00	0.14	0.00	0.10
401.bzip2	0.00	0.14	-0.00	0.11
447.dealII	-0.01	0.15	0.01	0.10
400.perlbench	0.00	0.13	0.00	0.10
483.xalancbmk	-0.02	0.13	0.02	0.12
450.soplex	-0.01	0.13	0.01	0.10
429.mcf	-0.02	0.14	0.02	0.12
456.hmmer	-0.02	0.25	0.03	0.19
433.milc	-0.03	0.23	0.04	0.18
471.omnetpp	-0.01	0.18	0.01	0.14
482.sphinx3	-0.01	0.15	0.02	0.12
465.tonto	-0.01	0.12	0.01	0.10
481.wrf	-0.03	0.19	0.03	0.16
434.zeusmp	-0.03	0.27	0.04	0.22
435.gromacs	-0.02	0.20	0.02	0.16
462.libquantum	-0.17	0.50	0.19	0.40
437.leslie3d	-0.06	0.32	0.07	0.26
459.GemsFDTD	-0.05	0.32	0.05	0.26
470.lbm	-0.11	0.78	0.14	0.41
410.bwaves	-0.06	0.51	0.10	0.37
436.cactusADM	-0.09	0.50	0.12	0.37

Table 11.2: Errors and residuals for test size 50 and basis size 5, ordered by bias error

	Error mean	Error SD	Residuals mean	Residuals SD
445.gobmk	0.00	0.07	0.00	0.03
458.sjeng	0.00	0.05	0.00	0.03
444.namd	0.01	0.11	0.00	0.04
453.povray	-0.00	0.08	0.00	0.04
464.h264ref	-0.00	0.09	0.00	0.04
473.astar	-0.00	0.07	0.00	0.03
403.gcc	-0.00	0.10	0.00	0.05
454.calculix	-0.01	0.14	0.01	0.08
416.gamess	-0.00	0.06	0.00	0.03
401.bzip2	-0.00	0.05	0.00	0.03
447.dealII	0.00	0.11	0.00	0.06
400.perlbench	0.00	0.07	0.00	0.03
483.xalancbmk	-0.00	0.11	0.00	0.06
450.soplex	0.00	0.09	0.00	0.04
429.mcf	-0.01	0.12	0.01	0.06
456.hmmer	0.00	0.20	0.01	0.09
433.milc	-0.00	0.18	0.01	0.10
471.omnetpp	-0.00	0.14	0.00	0.07
482.sphinx3	0.00	0.14	0.00	0.07
465.tonto	-0.00	0.08	0.00	0.04
481.wrf	-0.01	0.15	0.01	0.11
434.zeusmp	-0.00	0.19	0.01	0.11
435.gromacs	-0.01	0.20	0.02	0.11
462.libquantum	-0.10	0.51	0.10	0.28
437.leslie3d	-0.01	0.21	0.01	0.12
459.GemsFDTD	-0.01	0.16	0.00	0.09
470.lbm	-0.03	0.26	0.03	0.16
410.bwaves	-0.02	0.35	0.02	0.16
436.cactusADM	-0.03	0.42	0.05	0.22

Table 11.3: Errors and residuals for test size 100 and basis size 25, ordered by bias error

Chapter 12

Validating the energy model for concurrent parallel tasks using OpenFOAM

In the previous chapters we have used our model to characterize and predict the performance of micro-benchmarks and complex programs (CPUSPEC). In this chapter we present a set of experiments that we conducted to test the energy model presented in chapter 8. Using programs used for Computer Fluid Dynamics running on a small cluster we measured the power and energy consumption. We used our model to predict the peak power of a cluster running parallel tasks. Moreover, we attempt to model the energy consumption of concurrent parallel tasks running on the same *computational environment*.

12.1 Predict peak power

As discussed in section 2.3 the ability to predict the peak power consumption of a complex computational environment such as a cluster, or a subset of enclosures of a cluster, is important to avoid over-utilization of Power Distribution Units (PDU) and to avoid over-heating. More generally, characterization of power absorption is an important task to estimate battery life in mobile devices, such as smartphones or netbooks. Almost the entirety of those devices have multiple processing units. The energy model presented in chapter 8 can be applied to both those *computational environments*, and could be used by the Operating System to predict the power absorption.

12.1.1 Experiment setting

We designed a series of experiments to check if the energy model is capable of extracting the power consumption coefficients of our energy model ($\mathrm{P_{infr}}$, $\mathrm{P_m}$and $\mathrm{P_{\Delta c}}$) only using the total energy consumption and the information available at runtime to the scheduler (t_{wall}, t_{job}, n and m).

We measured a small set of real-world programs running on a 4 nodes enclosure of an HPC cluster, and used linear regression to extract $\mathrm{P_{infr}}\mathrm{P_m}$and $\mathrm{P_{\Delta c}}$. To check the accuracy of the estimation, we tried to predict peak power absorption.

OpenFOAM is an open source Computational Fluid Dynamics (CFD) and structural analysis tool, widely used in HPC clusters. We tested our model measuring the completion time and energy consumption of 4 cases of the tutorials included in the OpenFOAM CFD suite, running on an enclosure in the IT Center data center at the University of Pisa, with 4 compute nodes, each node equipped with 2 Intel(R) Xeon(R) X5670 CPUs (2.93GHz), hyper threading disabled, each CPU has 6 cores. We measured the instant power consumption of the whole enclosure (at the power socket) using a Phidgets 1122 ammeter [1], that has a range of 30A and 0.042A of resolution on AC, corresponding to a measurement error of approximately 9W. One compute node was running Windows HPC server 2008 with the measurement framework we wrote [2] to control the experiment and measure the energy consumed by the enclosure. The remaining 3 compute nodes were installed with CentOS, Kernel 2.6.32, we installed OpenFOAM from the RHEL RPM package available on the OpenFOAM website [3]. We modified 4 of the tutorials included in the OpenFOAM distribution as follows:

1. case *cavity* with the *icoFoam* solver, augmenting the mesh density 900 times, 100 iterations

2. case *pitzDaily* with the *adjointShapeOptimizationFoam* solver, augmenting the mesh density 400 times, 10 iterations

3. case *squareBump* with the *shallowWaterFoam* solver, augmenting the mesh density 6400 times, 90 iterations

4. case *mixerVesselAMI2D* with the *pimpleDyMFoam* solver, augmenting the mesh density 1000 times, 10 iterations

[1] `http://www.phidgets.com/`

[2] `https://github.com/vslab/Energon`

[3] `http://www.openfoam.org`

We measured the completion time t_{wall}(real time elapsed from the start of the job to its completion on all nodes), time elapsed in parallel execution t_{job}(real time spent during the parallel phase of the computation), and energy consumed (the product of average instant power, measured by the ammeter at power distribution unit level, and completion time) by the 4 programs running on 1 (12 cores), 2 (24 cores) and 3 nodes (36 cores).

The programs have different behaviour, indicating different underlying computational patterns:

- *squareBump* achieves the best performance with 36 cores, indicating a CPU bound computation, that is not penalized by communications
- *cavity* and *pitzDaily* reach the minimum at 24 cores, and the performance remain the same using the full 26 available cores, showing an I/O bound computation, that is penalized by frequent communications
- *mixerVesselAMI2D* achieves the best performance with 12 cores, indicating an even stronger I/O bound computation.

Equation 8.11 can be rewritten as equation 12.1:

$$E = \mathrm{P}_{\text{infr}} t_{\text{wall}} + \mathrm{P}_{\text{m}} m t_{\text{wall}} + \mathrm{P}_{\Delta\text{c}}(t_{\text{wall}} + (n-1)t_{\text{job}}) \qquad (12.1)$$

For every program we measured n, m, t_{wall}, t_{job} and E

We tried to estimate P_{infr}, $\mathrm{P}_{\text{m}}$and $\mathrm{P}_{\Delta\text{c}}$using linear regression.

To get an immediate idea of the quality of our estimates we calculated the reference values of P_{infr}, $\mathrm{P}_{\text{m}}$and $\mathrm{P}_{\Delta\text{c}}$as follows:

experiment A : we measured the measurement system alone, with no active compute node. The power consumption was 161.08 W.

experiment B : we measured the measurement system alone, with 1 active but idle compute node. The power consumption was 261.85 W.

experiment C : we measured the measurement system alone, with 1 active compute node with 1 processor running at 100%, the remaining 11 idle. The power consumption was 273.40 W.

We defined the reference $\mathrm{P}_{\text{infr}} = 161.08W$ as the power consumption measured during experiment A, the reference $\mathrm{P}_{\text{m}} = 100.77W$ as the difference between the power consumption of the experiment B and the experiment A; the reference $\mathrm{P}_{\Delta\text{c}} = 11.56W$ as the difference between the power consumption during experiment C and experiment B.

The experiments in the following two sections have been designed to study the accuracy of the energy model, that was designed to be as simple as possible while retaining high accuracy, using formula 12.1. The accuracy is high enough to allow us to avoid modeling other aspects of the computational system, focusing only on active cores.

To verify the quality of the model, we tested the difference between the predicted and the measured peak power in each experiment, using:

F-test to test whether two normally distributed populations can be considered to have the same variance.

T-test to test whether the differences between the measured and predicted values can be considered purely the effect of the noise present in the measured values. We used the variant known in statistics as *paired t-test*, or *repeated measures t-test*.

p-value the probability that the F-test or the T-test results happened by chance. The null hypothesis of the F-test is that the two populations have the same variance; the null hypothesis of the T-test is that the two populations have the same mean. We reject the null hypotheses only if their p-values are lower than 0.05, i.e. the probability that the results of the experiment happened by chance are less than 5%.

R^2 the coefficient of determination, indicating how well data fit a statistical model. R^2 indicates the amount of information in the data explained by the model, values near to 1 indicate a good fit. For example, an $R^2 = 0.99$ indicates that 99% of the data is explained by the model.

12.1.2 Modelling power using $\mathrm{P_{infr+m}}$ and $\mathrm{P_{\Delta c}}$, without separating $\mathrm{P_{infr}}$ and $\mathrm{P_m}$

In our first experiment we tried to explain the energetic behaviour of our computational environment only in terms of infrastructure and active cores, neglecting the power usage differences related to turning on the machines. The goal of this experiment is to show that separating $\mathrm{P_{infr+m}}$ into $\mathrm{P_{infr}}$ and $\mathrm{P_m}$ is necessary if multiple machines are used. We rewrote 8.12 into 12.2.

$$E = \mathrm{P_{infr+m}} t_{\mathrm{wall}} + \mathrm{P_{\Delta c}}(t_{\mathrm{wall}} + (n-1)t_{\mathrm{job}}) \tag{12.2}$$

We can get an estimate of $\mathrm{P_{infr+m}}$ and $\mathrm{P_{\Delta c}}$ from the measurements of OpenFOAM executions, using formula 12.2: we build a matrix $\mathbf{X}$ where

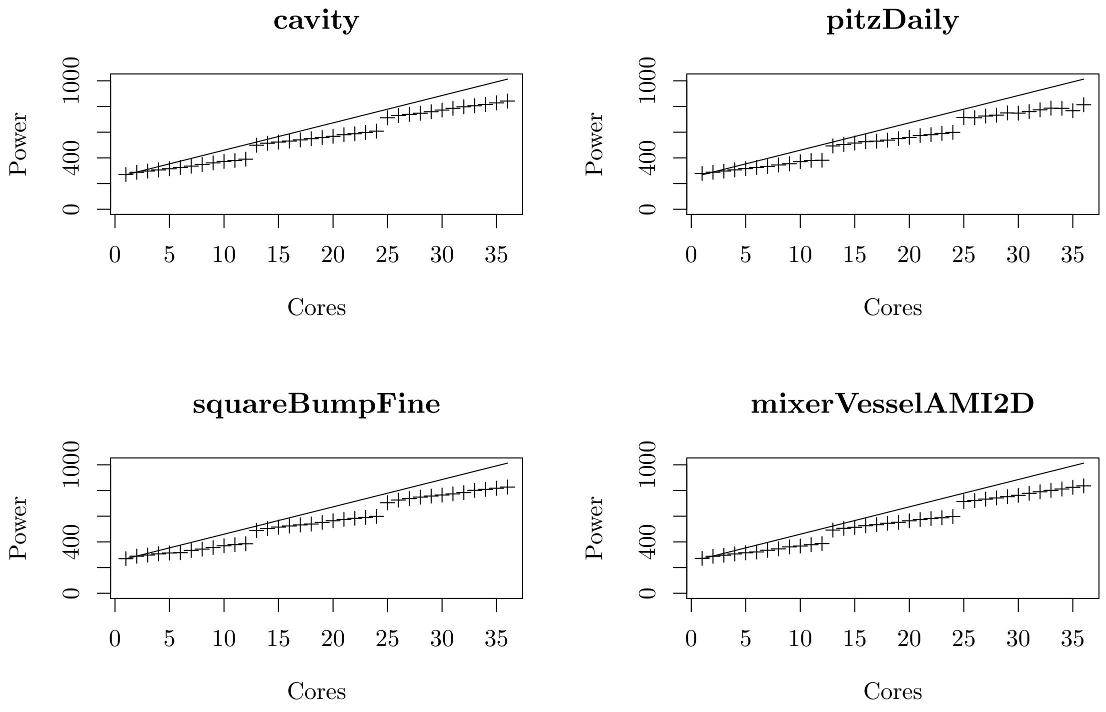


Figure 12.1: Measured power versus estimated power with fitted $\mathrm{P}_{\mathrm{infr+m}}$ and $\mathrm{P}_{\Delta\mathrm{c}}$

every row is a program, the first column contains t_{wall}, and the second $(t_{\mathrm{wall}} + (n-1)t_{\mathrm{job}})$.

We also prepare a vector $\mathbf{y}$ with the E, the energy consumption of all the programs, in the same order as the rows of the matrix $\mathbf{X}$.

The coefficient of the regression, the values of β, are our estimates for $\mathrm{P}_{\mathrm{infr+m}}$ and $\mathrm{P}_{\Delta\mathrm{c}}$.

Regression assigns $\mathrm{P}_{\mathrm{infr+m}} = 247.03$, and $\mathrm{P}_{\Delta\mathrm{c}} = 21.28$. $R^2 = 0.8209$, showing that the fitting is not very accurate.

Figure 12.1 shows the estimated peak power using formula 12.1 with $\mathrm{P}_{\mathrm{infr+m}}$ and $\mathrm{P}_{\Delta\mathrm{c}}$ fitted with the regression (the continuous line), as well as the measured values (the points) sampled in the middle of the execution, when all the cores are active (peak power consumption).

It is evident how inaccurate peak power absorption is modeled by formula

12.2, in a computational environment where the number of active machines can change.

F-test for the estimated and the measured peak power consumption reports a ratio of variances of 1.54 and a p-value of 0.01, showing that the difference in the variances of the measured and predicted peak power was relevant. We also ran a T-test to check if the mean of the absolute values of the differences between predicted and measures peak powers was less than 9W (the expected power measurement error). The T-test reported a p-value below 0.01, forcing us to reject the null hypothesis. This indicates that the probability that the difference of the means of the measured and predicted peak powers was within measurement error was very low. Both the F-test and the T-test found statistically relevant differences in the two datasets, showing that the predicted peak powers were not accurate.

12.1.3 Modelling power using $\mathrm{P_{infr}}$, $\mathrm{P_{\Delta c}}$, and $\mathrm{P_m}$

To verify that 12.1 can describe accurately the energy consumption of a parallel task, we added an independent variable to the regression, whose coefficient represents $\mathrm{P_m}$.

Similarly to what we did in the previous experiment, we build a matrix $\mathbf{X}$ where every row is a row is a program, the first column contains t_{wall}, the second contains mt_{wall} and the third $(t_{\mathrm{wall}} + (n-1)t_{\mathrm{job}})$. This is the same matrix as the one built for the previous experiment, with the addition of a column.

The same vector $\mathbf{y}$ is the same as the previous experiment.

The coefficient of the regression, values of $\mathbf{b}$, are our estimates for $\mathrm{P_{infr}}$, $\mathrm{P_m}$ and $\mathrm{P_{\Delta c}}$. This time $R^2 = 0.9993$, showing that the regression could fit the data with a much higher confidence than it was possible in the previous experiment.

	Estimated	Reference	Relative error
$\mathrm{P_{infr}}$	163.76	161.08	0.02
$\mathrm{P_m}$	97.84	100.77	-0.03
$\mathrm{P_{\Delta c}}$	10.51	11.56	-0.09

Table 12.1: Estimated $\mathrm{P_{infr}}$, $\mathrm{P_m}$and $\mathrm{P_{\Delta c}}$

Table 12.1 reports the values of the estimated $\mathrm{P_{infr}}$, $\mathrm{P_m}$and $\mathrm{P_{\Delta c}}$, as well as the reference values, and the relative errors.

Figure 12.2 shows the estimated power using formula 12.1 with $\mathrm{P_{infr}}$, $\mathrm{P_m}$ and $\mathrm{P_{\Delta c}}$ fitted with the regression, as well as the measured values. The

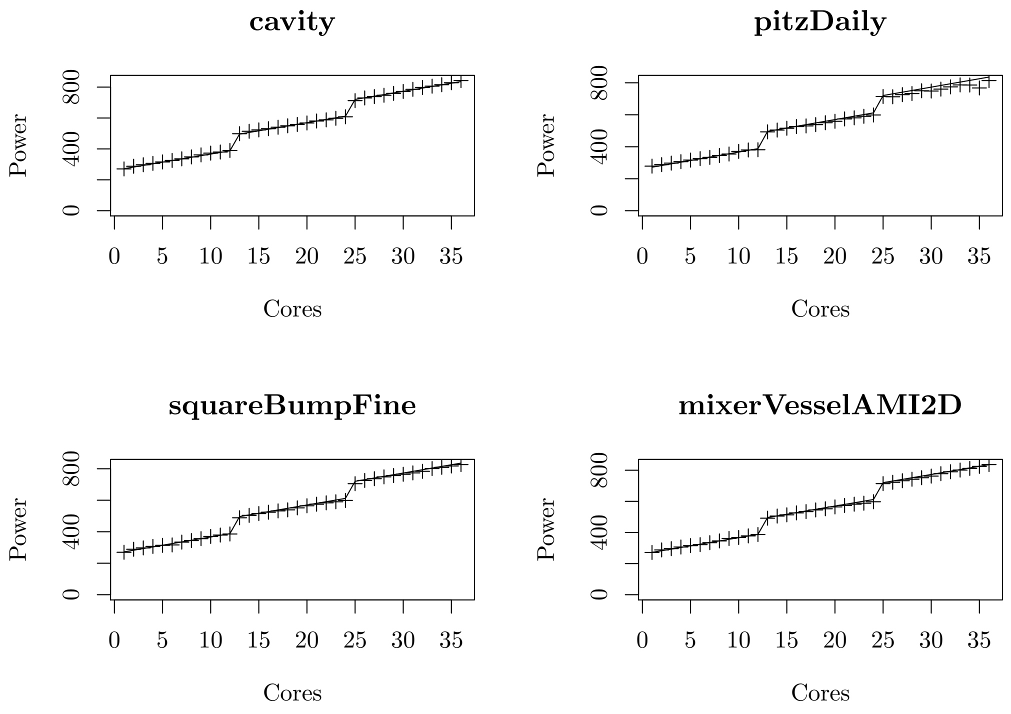


Figure 12.2: Measured power versus estimated power with fitted P_{infr}, P_{m} and $P_{\Delta\mathrm{c}}$

estimated values are extremely close to the measured values.

F-test for the estimated and the measured peak power consumption reports a ratio of variances of 0.96 and a *p*-value of 0.80. We ran a T-test, to test the null hypothesis that the mean of the absolute values of the differences between predicted and measured peak power was less than 9W (the measurement error). The T-test reported a *p*-value close to 1, allowing us to accept the null hypothesis. This shows that the model accurately predicted peak powers, as both the F-test and T-test could not find statistically relevant differences in the mean and the variances of the predicted and measured peak power.

Figures 12.3, 12.4, 12.5, and 12.6 show the measured instant power for most of the experiments. The green and red lines indicate the predicted power consumption during the serial and parallel phases, the dotted lines indicate the uncertainty introduced by the ammeter (9W). This shows how precisely the power absorption can be predicted only using values known to the scheduler (m and n) and the architecture characterization, i.e. the coefficients extracted by our model ($\mathrm{P_{infr}}$, $\mathrm{P_m}$, and $\mathrm{P_{\Delta c}}$).

12.2 Modelling energy consumption of concurrent programs

Energy is one of the main expenses in a datacenter. Most of the cloud infrastructure offer a pay-per-use cost model. Therefore, is important for cloud providers to be able to account for the energy consumption of each task (Kim et al., 2011). However, in a virtualized environment, several tasks run on the same machine at the same time.

Several energy model have been proposed to characterize the energy consumption of tasks (Goiri et al., 2010; Kim et al., 2011), also parallel tasks have been modelled (Garg et al., 2009; Li, 2012; Wang et al., 2010). Most of the research focuses on Dynamic Voltage Scaling (DVS) of the CPU. However, not many models attempt to describe the power and energy consumption of concurrent parallel tasks, running on the same *computational environment*. Our energy model can be used to model concurrent execution of tasks. In this section we validate the characterization and prediction of energy consumption in the case of several tasks running on at the same time. This information could be used for accounting in datacenters and cloud environments.

We designed an experiment to check if our model can accurately describe the energy consumption of two concurrent parallel programs, running

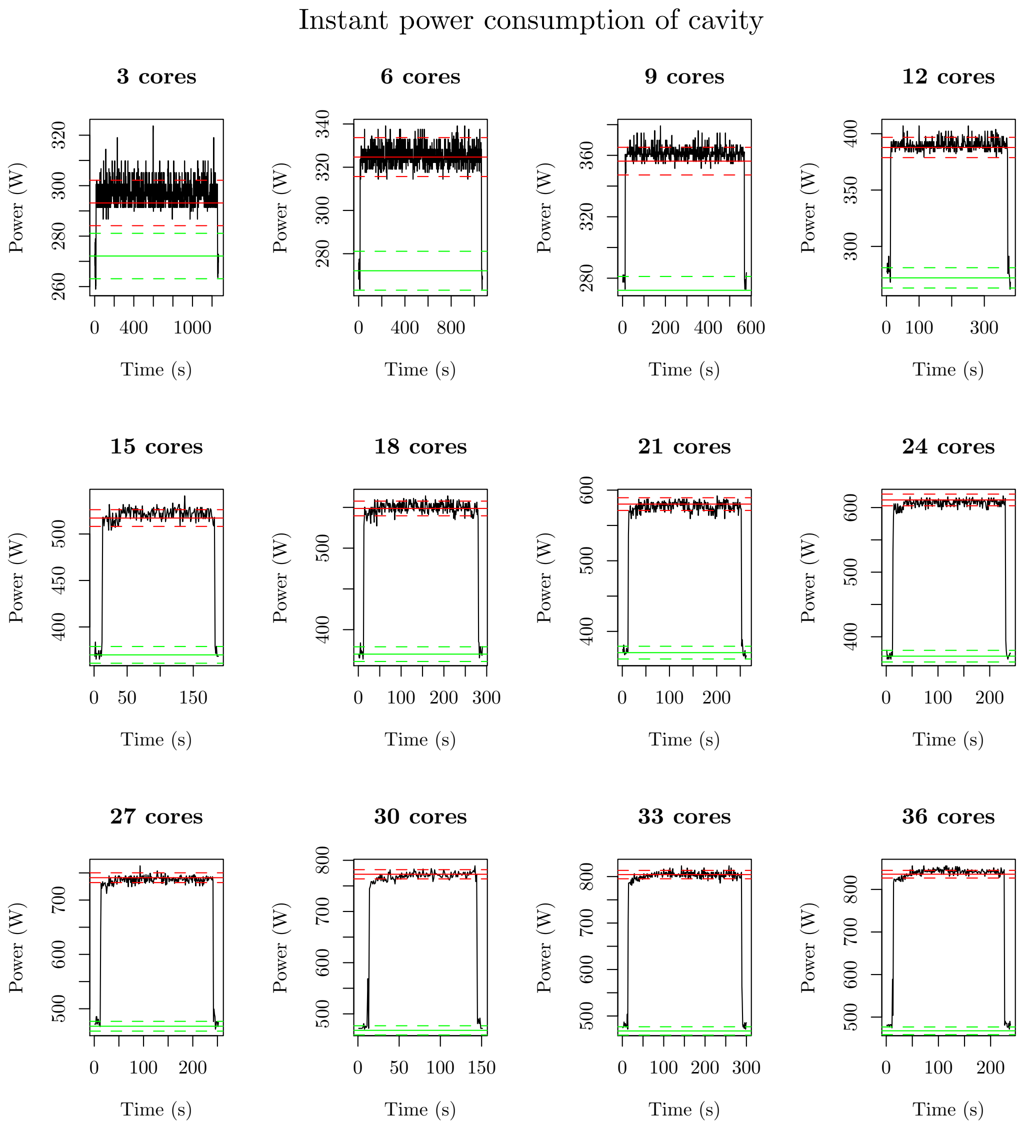


Figure 12.3: Predicted and measured instant power during serial (green) and parallel (red) phases: cavity

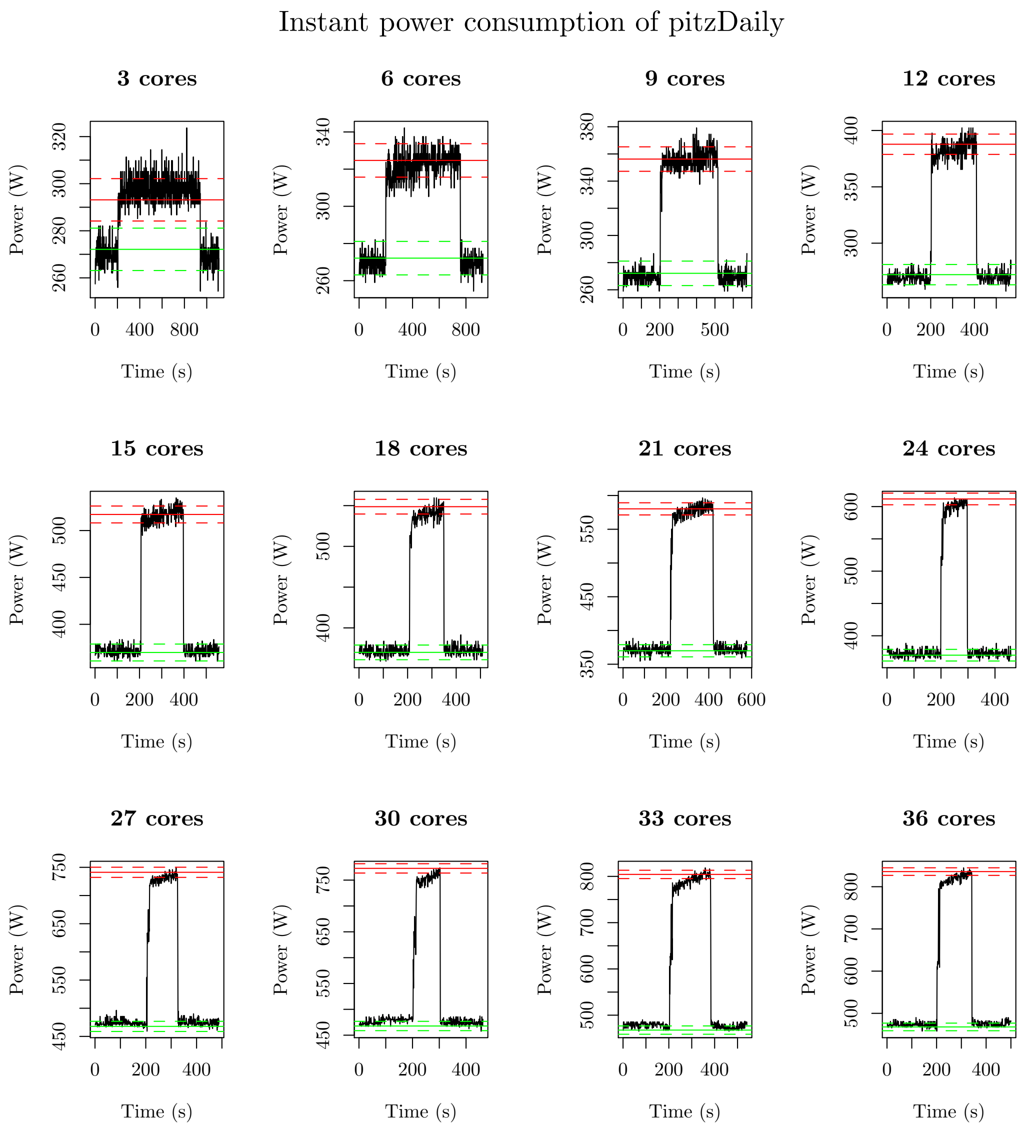


Figure 12.4: Predicted and measured instant power during serial (green) and parallel (red) phases: pitzDaily

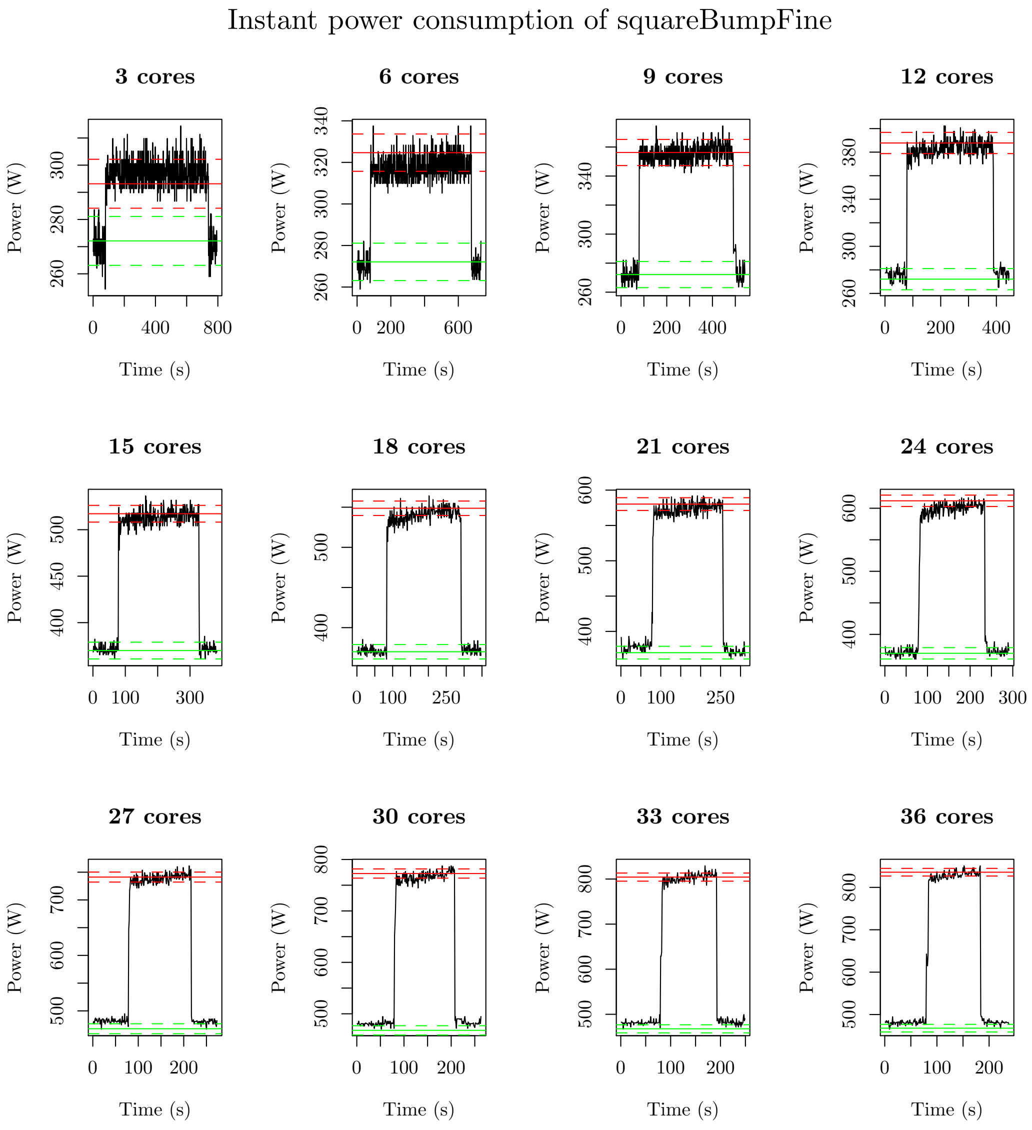


Figure 12.5: Predicted and measured instant power during serial (green) and parallel (red) phases: squareBump

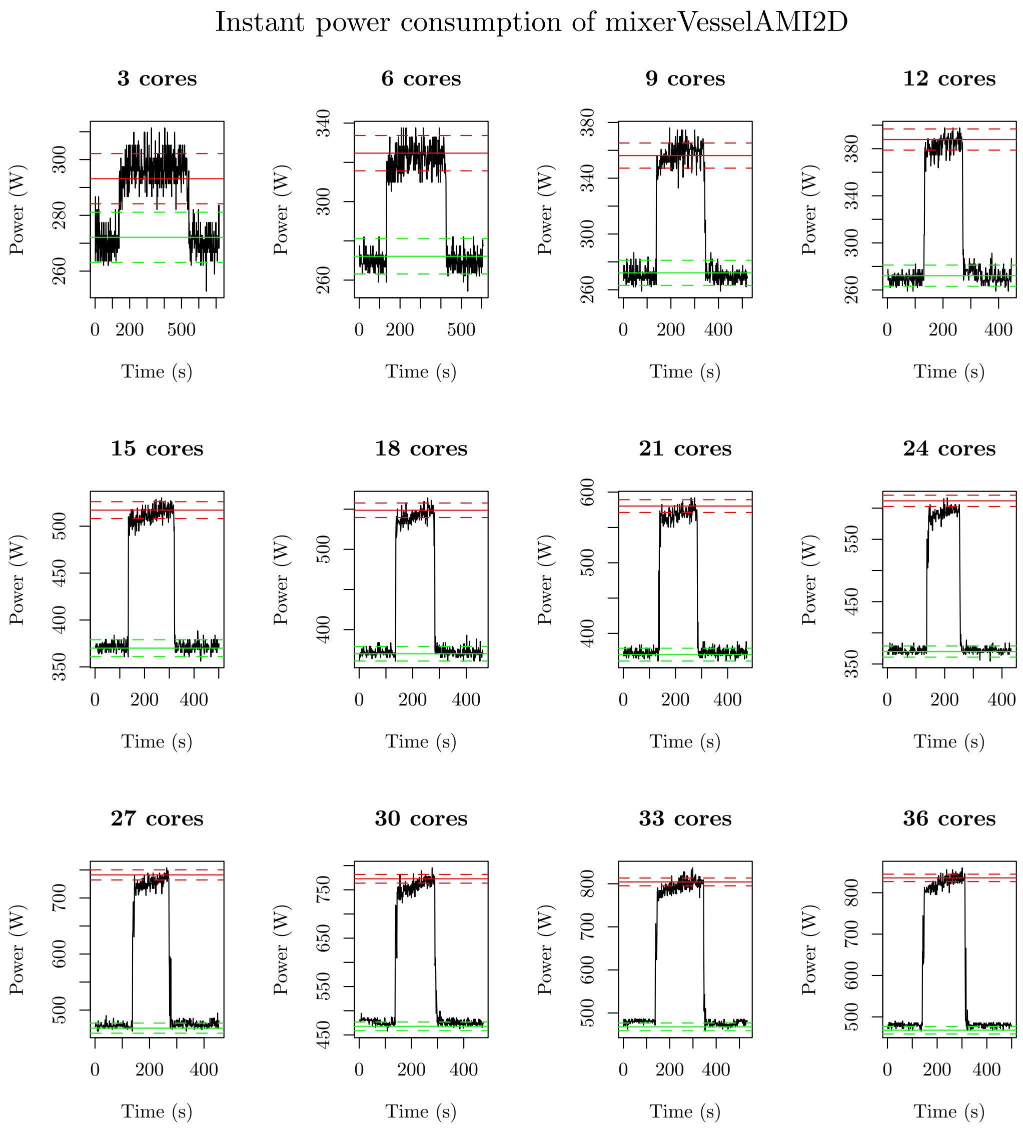


Figure 12.6: Predicted and measured instant power during serial (green) and parallel (red) phases: mixerVesselAMI2D

simultaneously on the same machine. We tested the model assigning 1, 3 or 6 cores to *cavity*, *mixerVesselAMI2D*, *squareBumpFine* and *pitzDaily* on the same machine used in the previous section, testing all programs pairs and combinations of number of cores (1, 3 or 6). We ran a total of 90 experiments.

Under the assumption that the machine will not thrash, equation 8.9 can be refined into equation 12.3 as follows:

- setting $m = 1$ because we use only one machine
- we measure $t_{\text{wall}}$and t_{job_1} and t_{job_2}, information available to the scheduler at runtime
- $\mathrm{P}_{\Delta\mathrm{c}} \sum_{i \in \text{cores}} t_i$ can be refined as the sum of the number of cores used by each program multiplied by their t_{job}

$$E = (\mathrm{P}_{\text{infr}} + \mathrm{P}_{\text{m}} + 2\mathrm{P}_{\Delta\mathrm{c}}) \max(t_{\text{wall}_1}, t_{\text{wall}_2}) + \mathrm{P}_{\Delta\mathrm{c}}((n_1 - 1)t_{\text{job}_1} + (n_2 - 1)t_{\text{job}_2}) \tag{12.3}$$

Figure 12.7 shows the energy prediction error distribution for all tests (90), and filtering by program (36 each). Table 12.2 reports a statistical analysis of the prediction accuracy and relative errors. 99% of the predictions have an error below 0.04.

The F-test on the predicted and measured values report a p-value of 0.86 and a ratio of variances equal to 0.96. We also ran the T-test, with the null hypothesis that the absolute values of the differences of the predicted and measured values were less than 9W (the measurement error). The T-test reported a p-value of 0.18, allowing us to accept the null hypothesis. This shows that the differences in the predictions and the measures are not statistically relevant, and they should be assumed to have same variance and difference of mean within measurement error. This confirms the accuracy of the model.

Measure	Mean	Median	10 perc	50 perc	90 perc	99 perc
Relative accuracy	1.0004	1.0004	0.9802	1.0004	1.0208	1.0255
Relative errors	0.0004	0.0004	-0.0198	0.0004	0.0208	0.0255
Absolute percentage errors	0.0129	0.0109	0.0026	0.0109	0.0241	0.0353

Table 12.2: Accuracy measures

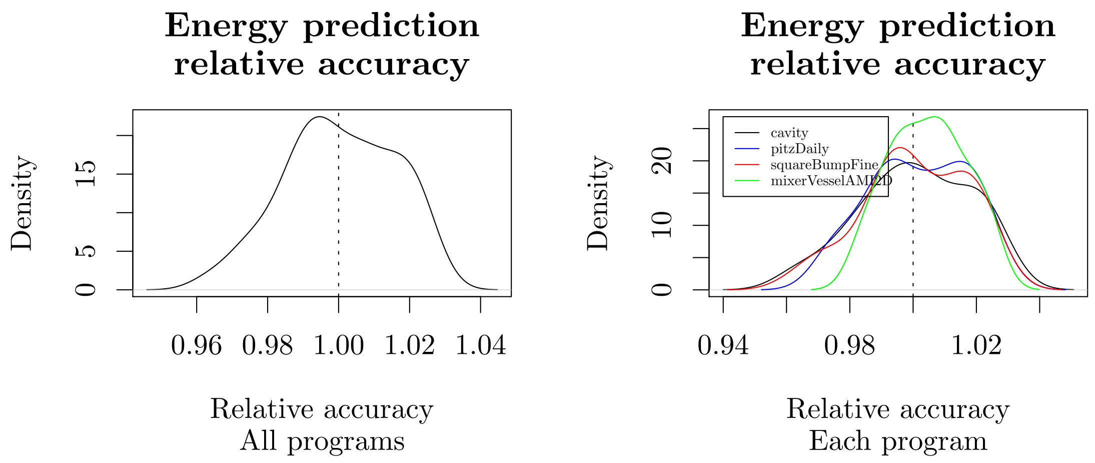


Figure 12.7: Distribution of Energy prediction error running two tasks in parallel

12.3 Conclusions

In this chapter we have tested the accuracy of our energy model. We could characterize the power coefficients P_{infr}, P_{m}, and $P_{\Delta \mathrm{c}}$, using regression on a set of measures of a real world program (OpenFOAM) running in a small cluster, only using information available to the scheduler: completion time, number of machines, the number of core used, and readings from an ammeter at the power plug level (information available on most PDU units). Using the power coefficients we could predict the peak power absorption of the cluster reliably.

Moreover, to test the ability of the model to describe the energy consumption of concurrent parallel programs running at the same time on the same cluster, we predicted the energy consumption of 90 different combinations of programs and number of user cores. The predictions had a small error. The model is therefore precise, and could be used to build an energy-based accounting system for a datacenter, without requiring any substantial modification to the Operating System or the scheduler, simply processing the scheduler logs.

Chapter 13

Combining static analysis with the prediction model: best device scheduling in heterogeneous environments

The last few years have seen an increasing number of programming models, languages and frameworks to address the wide heterogeneity and broad availability of parallel computing resources by raising the level of programming abstraction and enhancing portability across different platforms. Despite the ability to run parallel algorithms across different devices, each device is generally characterised by a restricted set of computations for which it outperforms others. On increasingly popular multi-device heterogeneous platforms, the problem is therefore to schedule computations in a way that automatically chooses the best device among the available ones, each time a computation has to be run.

The broad availability and affordability of multi-device systems introduces a dynamism in both the system configuration and in the set of computations to run that make traditional scheduling approaches to be unfit because of restrictions on the computational structure or of the overhead introduced by the scheduling policy.

In this chapter we combine our benchmarking model with runtime code analysis, to efficiently address the problem of dynamically exploiting the computing power of heterogeneous, parallel platforms through a scheduling strategy based on algorithmic feature extraction, completion time prediction and classification. The goal of this experiment is to prove that our benchmarking model can be adapted with sophisticated techniques, in this case runtime analysis of programs, to predict the most performance device for a given instance of a program, at runtime.

In this chapter:

- we describe a strategy to efficiently extract programs features at runtime;
- we combine our benchmarking model, more specifically the *linear regression solver* with code analysis, to estimate the completion time of parallel programs on heterogeneous platforms;
- we show how to use the *surrogate* to analyse the device characterization;
- we show how to use the *linear regression solver* to predict the completion time on several different devices;
- we apply the performance prediction model to dynamically best-schedule algorithms on heterogeneous platforms based on completion time. Since completion time is a metric, we introduce the chance to easily refine the scheduling policy to take into account data-transfer overhead and sub-optimal scheduling.

13.1 Introduction

In the last few years computing has become increasingly heterogeneous, offering a broad portfolio of different parallel devices, ranging from CPUs, through GPUs to APUs[1] and coprocessors. On the laptop and desktop CPUs market, the recent trend sees embedding a multicore CPU and a GPU on the same chipset (Ketan Paranjape et al., 2014) [2] . From the GPUs point of view we have been heading toward system equipped with multiple cards, exploiting proprietary inter-communication technologies, such as SLI, or effectively doubling the entire GPU architecture on a single card. Intel recently released an hybrid device called "Xeon Phi", a coprocessor for cluster nodes and desktop systems, which is becoming quite popular and affordable for a broad audience. Given this, today's desktop computers are effectively

[1] APU is the term used by some processors brands to refer to a CPU and a GPU integrated into the same die

[2] `http://www.slideshare.net/PankajSingh137/amd-9th-intlsocconf-presentation`

equipped with one or more multicore CPUs, multiple GPUs and possibly highly-parallel coprocessors.

In this chapter we show how our resource usage characterization and prediction model can be used to schedule parallel computation on heterogeneous platforms. Our approach is device-aware and computation-based, which means it focuses on the specific characteristics of each available device and on the structure of each computation.

13.2 Code analysis and feature extraction

Recently OpenCL [3] has become one of the most popular approaches for heterogeneous parallel programming, thanks to which parallel algorithms can run on CPUs, GPUs and other kind of accelerators with nearly hundred-percent portability. One of the major OpenCL limitations is the lack of any support to exploit the heterogeneity of a system in a device-aware fashion.

F# to OpenCL (FSCL) (Cocco, 2015) [4] is a framework for high-level programming and execution on heterogeneous platforms that addresses the problem of raising abstraction over both OpenCL programming, making it possible to program OpenCL kernels from within F#, while helping to dynamically exploit the heterogeneity of a platform through a transparent, device and computation aware scheduling approach.

The main problem in code analysis for feature extraction is that for most computations the value of a feature depends on the input.

FSCL statically [5] pre-computes features, analysing the AST of the kernel and building a *finalizer* for each feature, which completes the evaluation of the feature value as soon as the input is known.

Feature finalizers are computed the first time a kernel is seen and stored in a data-structure for future use (fig. 13.1). Once built, a feature finalizer can be applied to multiple, different sets of kernel arguments to retrieve the matching feature value. Thanks to the static analysis of the kernel AST and the construction of a lambda that generally contains very lightweight code[6], the overhead of completing feature evaluation at kernel execution time, which corresponds to applying the lambda to the kernel arguments, is mostly irrelevant.

[3]`https://www.khronos.org/opencl/`

[4]`http://fscl.github.io/FSCL.Compiler/`

[5]With "statically" we mean at kernel compilation time, that is the first time a particular kernel is seen

[6]For features counting particular construct in a kernel, such as the number of memory accesses, the finalizer code contains only few arithmetic operations

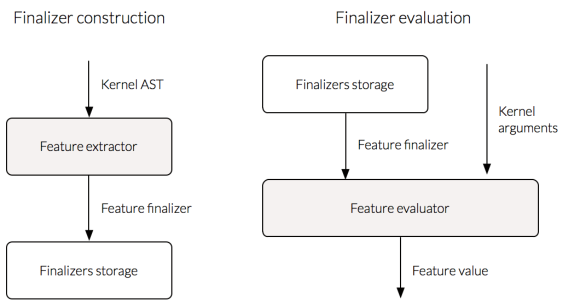
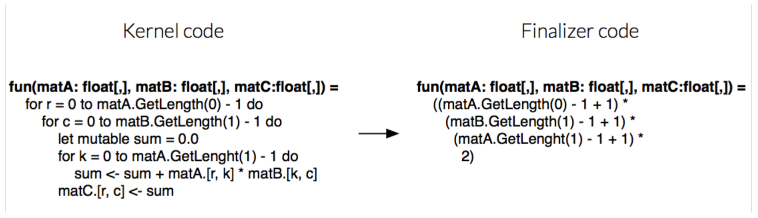


Figure 13.1: Finalizer construction and evaluation

Building a feature finalizer for a particular kernel consists in mapping the kernel to a lambda function, preserving the set of parameters but replacing the body. Figure 13.2 shows this mapping for a sequential matrix multiplication kernel and a feature that counts the number of accesses to the elements of the input arrays.


Kernel code
Finalizer code

```
fun(matA: float[,], matB: float[,], matC:float[,]) =
   for r = 0 to matA.GetLength(0) - 1 do
      for c = 0 to matB.GetLength(1) - 1 do
         let mutable sum = 0.0
         for k = 0 to matA.GetLenght(1) - 1 do
            sum <- sum + matA.[r, k] * matB.[k, c]
         matC.[r, c] <- sum
```

```
fun(matA: float[,], matB: float[,], matC:float[,]) =
   ((matA.GetLength(0) - 1 + 1) *
     (matB.GetLength(1) - 1 + 1) *
      (matA.GetLenght(1) - 1 + 1) *
       2)
```

Figure 13.2: Kernel to finalizer mapping for a feature that counts memory reads

This approach combines static analysis of code with actual parameters used at runtime. For example, the finalizer code that counts the number of arithmetic expression with a dependency on the kernel actual arguments, is shown in figure 13.2.

13.3 Prediction model

We will use linear regression as the *solver* for this experiment, as presented in section 6.2.

As discussed in section 6.2.5, ordinary linear regression is sensitive to outliers in the dependent variable. Completion time is likely to contain outliers, mainly caused by sporadic effects that may not be considered in the model and to the instability of the system where the measurement is performed. Outliers can be introduced also by an heavy task in the system that steals computing resources from the running tests or by driver instability. For this reason, our prediction model employs a robust regression method instead of ordinary regression (Hoaglin et al., 2011; Fox, 1997).

The linear model is build running the benchmarks on each available OpenCL device in the system. Each model correlates a set of features to the completion time on a specific device. For each device a linear model is separately built starting from the following data:

- A set of benchmarks: for each benchmark we consider several cases by varying the input size. Each benchmark case is executed on the device to obtain the corresponding completion time. With respect to the concept of *programs*, as defined in section 5.1, each benchmark case is considered a different *program*.

- A set of features: each feature captures a certain aspect of a program. The chosen features are extracted from each benchmark case. The features are here considered as *resources*, as defined in section 5.1. The completion time is the *target resource*.

- A matrix $\mathbf{A}$ where each column is a feature and each row is a benchmark case.

- A vector $\mathbf{t}$ with the completion times of the benchmark cases on the considered device, in the same order of the cases in $\mathbf{A}$

After having defined a set of benchmark cases and features, a matrix $\mathbf{A}$ and a vector $\mathbf{t}$, we build the matrix of the explanatory variables $\mathbf{X}$ from $\mathbf{A}$.

Linear regression assumes homoskedasticity (i.e. constant variance) in the error terms. In particular, the error on a feature should not be correlated to the completion time. In our model we instead expect the errors on features to be affected by such a correlation. To deal with this issue, we employ the Weighted Least Squares method, which is a generalization of ordinary least squares that relaxes this assumption by normalizing the equations using the variance of the completion time. To estimate the standard deviation of the completion time of each benchmark case, we repeated every experiment 100 times. For our model, we set the dependent variable $\mathbf{y}$ as the component-wise normalization of $\mathbf{t}$ by its standard deviation (i.e. $y[i] = t[i]/\sigma(t[i])$, so that the error can be assumed to be identically distributed on all of the samples. Consistently, we also normalize each row of **A** in the same way, as shown in equation 13.2.

Finally, we add to the resulting matrix an unary column, normalized like **t**, which constitutes the intercept of the linear regression. Conceptually, the intercept represents the time needed to start any computation, independently from the specific benchmark case being measured.

$$\mathbf{y} = \begin{pmatrix} {}^{t_1}\!/_{\sigma(t_1)} \\ {}^{t_2}\!/_{\sigma(t_2)} \\ \vdots \\ {}^{t_m}\!/_{\sigma(t_m)} \end{pmatrix} \tag{13.1}$$

Normalised regressand

$$\mathbf{X} = \begin{pmatrix} {}^{1}\!/_{\sigma(t_1)} & {}^{a_{1,1}}\!/_{\sigma(t_1)} & {}^{a_{1,2}}\!/_{\sigma(t_1)} & \cdots & {}^{a_{1,n}}\!/_{\sigma(t_1)} \\ {}^{1}\!/_{\sigma(t_2)} & {}^{a_{2,1}}\!/_{\sigma(t_2)} & {}^{a_{2,2}}\!/_{\sigma(t_2)} & \cdots & {}^{a_{2,n}}\!/_{\sigma(t_2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ {}^{1}\!/_{\sigma(t_m)} & {}^{a_{m,1}}\!/_{\sigma(t_m)} & {}^{a_{m,2}}\!/_{\sigma(t_m)} & \cdots & {}^{a_{m,n}}\!/_{\sigma(t_m)} \end{pmatrix} \tag{13.2}$$

Normalised regressors

We can now apply **y** (as defined in the equation 13.1) and **X** (as defined in the equation 13.2) to the regression equation (equation 6.17 presented in section 6.2).

The resulting *surrogate* β (the set of regression coefficients) describes the completion time on the considered device in terms of a linear combination of the features.

Completion time is an unreliable measure, subject to unpredictable errors due to the presence of running processes and tasks in the system during the execution of the measured program. Since we expect the presence of outliers in the data, as already said we employ a robust regression method, which uses the Iteratively Re-weighted Least Squares algorithm (Holland and Welsch, 1977) to identify and discard the outliers.

Once β has been calculated, we can use formula 6.22 presented in section 6.2 to predict the completion time of a target program (case) on the device considered. In this formula, **x** is a vector that contains the same features used in **X** but calculated on a *target program*.

The *solver* is performed independently for each OpenCL device available, building **y** in equation 13.1 starting from the completion times of the benchmark cases on the device.

In the context of our benchmarking model, the features extracted at runtime on the OpenCL kernels are the measured *computational resources*; the model is used to characterize the completion time *target resource*, the *surrogate* (the regression coefficients) will express the time taken to complete every unit of reported feature.

13.4 Experimental validation

In order to validate the model, we define a set of relevant features to extract and a set of programs that form the training set. Then, we extract the chosen features from each training sample case. Finally, we build a device model (i.e. set of regression coefficients) independently for each device in the running system. Given a device d, we run each sample case on d to obtain the corresponding completion time. With the set of feature values and completion times for all the sample cases, we prepare the model described in section 5.2, consisting of the matrix of the explanatory variables $\mathbf{X}$ and a dependent vector $\mathbf{y}$, which contains the completion time of the cases on the specific device. We finally apply linear regression to $\mathbf{X}$ and $\mathbf{y}$ to obtain the device model. While feature extraction is performed once, linear regression is repeated for each device in the system.

Once the set of device models has been built, we use it to predict the completion time of a set of testing programs. Each test is also executed to measure its actual completion time. Finally, we calculate the accuracy in selecting the best device for each test in terms of the ratio between the completion time of the estimated fastest device and the measured lowest completion time across the set of devices.

13.4.1 Experiment setup

To validate the prediction model we setup an heterogeneous system equipped with an APU and a discrete GPU. The APU is a chipset that includes a CPU and a GPU on-die. With “D-GPU” we indicate the discrete GPU and with “I-GPU” the on-die GPU. The system configuration is reported below.

- AMD Fusion A10-5800K (CPU with an integrated AMD HD 7660D GPU)

- AMD HD 7970 (discrete GPU)
- 4GB DDR3 Ram - 1333 Mhz
- Windows 7 64 bit

In the definition of the training set, we focus on including computations that stress only specific features and device characteristics with a minimal effect on the others. Since this set of samples constitute the "basis" used to predict the completion time of other computations, we want to start from a minimal set of samples (i.e. small basis) capable of describing a possibly wide set of other algorithms and progressively refine it by introducing further samples.

Vector addition This kernel performs an element-wise sum of two vectors, where each work-item sums the elements matching its own global index. Given the very short execution time and the extremely lightweight nature of the computation, this sample allows to focus on the data-transfer time and on the contribution of the work-space size to the completion time. We execute the kernel varying the input size from 1MB up to 128MB with 1MB step.

Matrix multiplication naive A matrix multiplication kernel where each work-item in a 2D work-space multiplies a row of the first matrix by a column of the second. The first matrix is accessed with a cache-friendly pattern, while the second is accessed with a matrix-width stride that may incur many cache misses. For this reason, matrix multiplication allows to capture the impact of cache misses on completion time. We run this kernel starting from 64x64 elements matrices up to 2048x2048, with a 64-elements step.

Logistic map A logistic map performed on each element on an input vector filled with random floating point values. Since each work-item performs one only memory access and many arithmetic operations, this sample captures the impact of floating point operations on the completion time. We execute the kernel varying the input size from 1MB up to 128MB with 1MB step and the number of iterations per-work-item from 1000 to 10000.

For each training sample we produce 30 cases characterized by different input sizes. Each case is then run 100 times to get the average completion time and the standard deviation.

The selection of the features to extract is related to the aspects stressed by the set of training samples. We consider memory accesses and operations (arithmetic, logic and transcendent) two of the most relevant features to use in order to characterize the completion time. While a given operation takes a fixed same amount of time to complete, memory accesses have a different impact depending on whether the data accessed is in cache or not. For this reason, instead of considering the amount of memory accesses we estimate the number of cache misses. Cache misses are estimated by detecting the size of the data cache and cache line on a specific device[7] and by computing the strides of memory accesses in the sample. A first approximation introduced is considering each array separately as like as each array was stored in a separate, private data cache. At kernel-compilation time we pre-compute the access strides to each (global) array and we count the number of accesses with a certain stride by assessing the total trip-count of the loops containing the memory access operations. A second approximation is introduced by considering the largest stride among the memory accesses in a loop, as if the block of memory between the lowest and the highest addresses was entirely used. At kernel-execution time we complete the evaluation of the strides and the relative trip count. These information are then coupled with the cache line and the size of the data cache to estimate the number of cache misses.

Given a certain number of operation and memory accesses per-work-item, the completion time should be affected by the total number of work-items launched. Therefore, we add the work-space size to the set of the computed features.

Total number of instructions executed This feature represents the computation as if all of the operations had the same cost and were executed in a sequential order.

Number of instructions executed per thread This feature represents the computation as if all of the work-items could perform operations in a pure parallel fashion.

Cache misses This feature captures the time spent waiting for memory accesses that do not hit the cache.

Global work-items This feature corresponds to the amount of work-items spawn to execute a computation.

[7]This information can be retrieved through the OpenCL device-query API or running micro-benchmarks

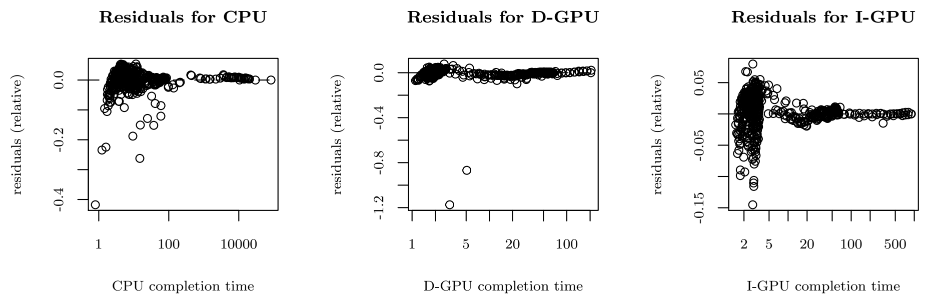


Figure 13.3: Fitting residuals

Kernel-launch overhead The cost of starting a kernel. Since the value of this features is always 1 (intercept), linear regression charges it with the part of the completion time that it is not able to properly describe in terms of the other features (non-linear terms).

This set of features should be sufficient to investigate the costs of launching a kernel, the amount of time spent performing arithmetic, logic, or transcendent operations and the overhead of memory accesses. Several other factors may contribute to the completion time, but we want to start from the coarsest model and progressively refine it as like as for the set of training samples.

Figure 13.3 shows the residuals of the fitting for both CPU, discrete GPU and integrated GPU. Since we normalize $\mathbf{X}$ by $\mathbf{t}$ (section 13.3), the residuals are shown as relative values. The analysis of the residuals reveals the following:

- Most of the residuals lie near 0, which in general indicates a good fit.
- Residuals are larger for completion times close to zero. This is expected, because cases that have a very short completion time are more likely to be affected by measurement errors induced by the influence of external and unpredictable effects, such as the presence of other processes running in the system. This also suggests that we should expect a relevant prediction error for programs that complete in a short time.

13.4.2 Predicting the completion time

To evaluate the error in predicting the completion time of computations using the models built for the devices in the running platform, we define a set of test samples.

Sum of matrix rows This kernel sums the rows of a two-dimensional matrix, producing a vector whose size is equal to the matrix width. Each work-item performs a reduction along a column. As for all the matrix-based samples, we run this kernel starting from 64x64 elements matrices up to 2048x2048, with a 64-elements step.

Sum of matrix columns This kernel sums the columns of a two-dimensional matrix, which results in a vector matching the matrix height. Each work-item performs a reduction along a row.

Sobel filtering A Sobel 3x3 filtering algorithm on a 2D matrix.

Convolution filtering A generalization of Sobel filtering, with a generic input filter varying in size from 3x3 to 19x19 elements.

Matrix transpose naive Matrix transpose performed by making each work-item to transpose a single element of the matrix.

Matrix transpose advanced Matrix transpose that exploits local memory to make successive work-items to read and write successive matrix elements, enabling coalescing and reducing channel/bank conflicts.

In the set of test samples we also include the training samples used to build the device models. Using a set of samples to predict themselves provides, in addition to the set of residuals, an insight on the quality of the fitting.

In figures 13.4 and 13.5 we compare the measured completion time and the estimated completion time for each test sample by varying the input size. The left column reports the measured completion time on CPU, discrete GPU and integrated GPU, while the right column shows the predicted completion time on the same devices. To be easily compared with each other, both the completion times are expressed in the same logarithmic scale.

The first three samples illustrated in figures 13.4 and 13.5 are the training samples used to build the device models. As expected, the predicted completion times of these programs is very close to the measured completion times.

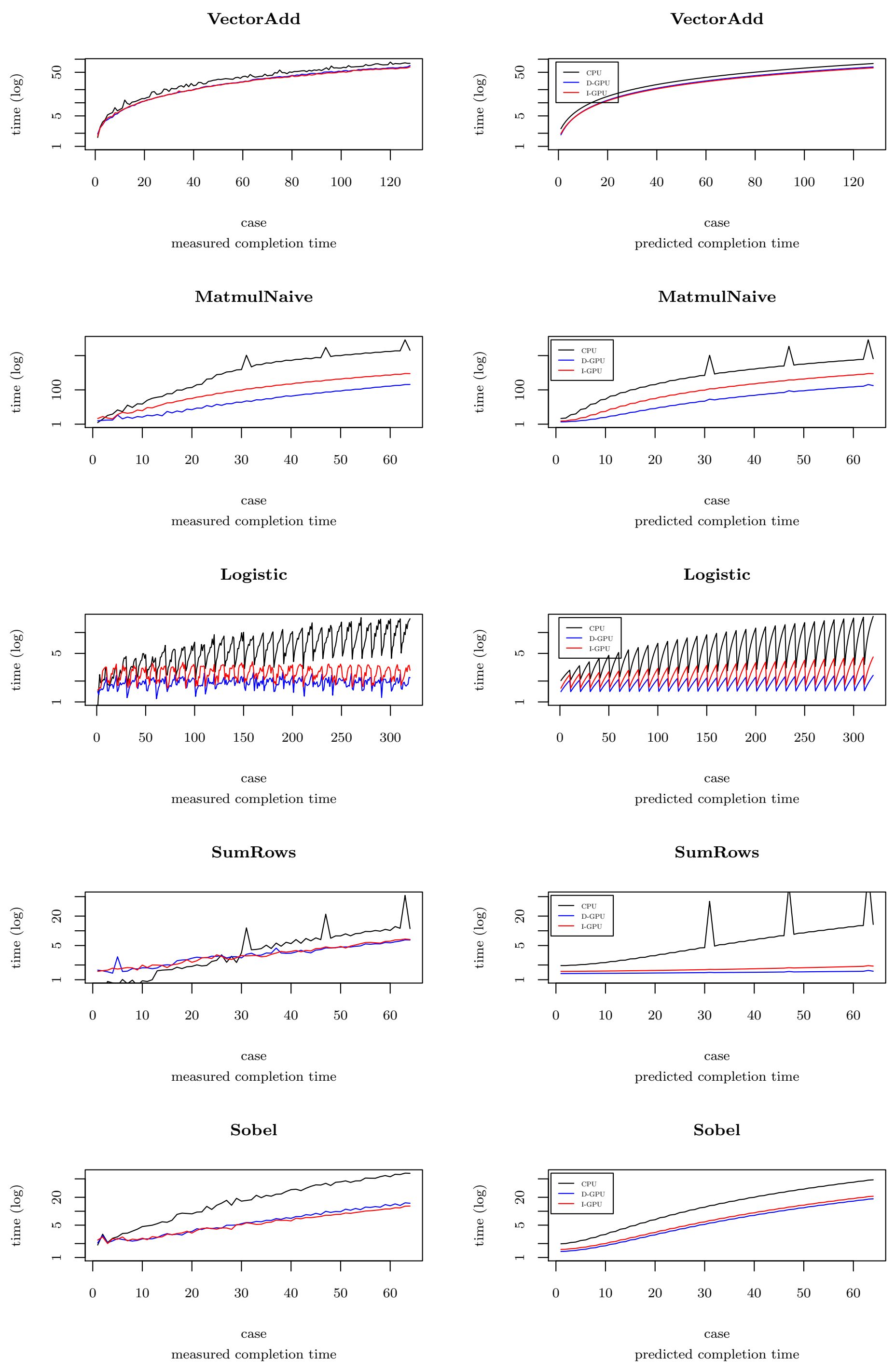


Figure 13.4: Measured and predicted completion time

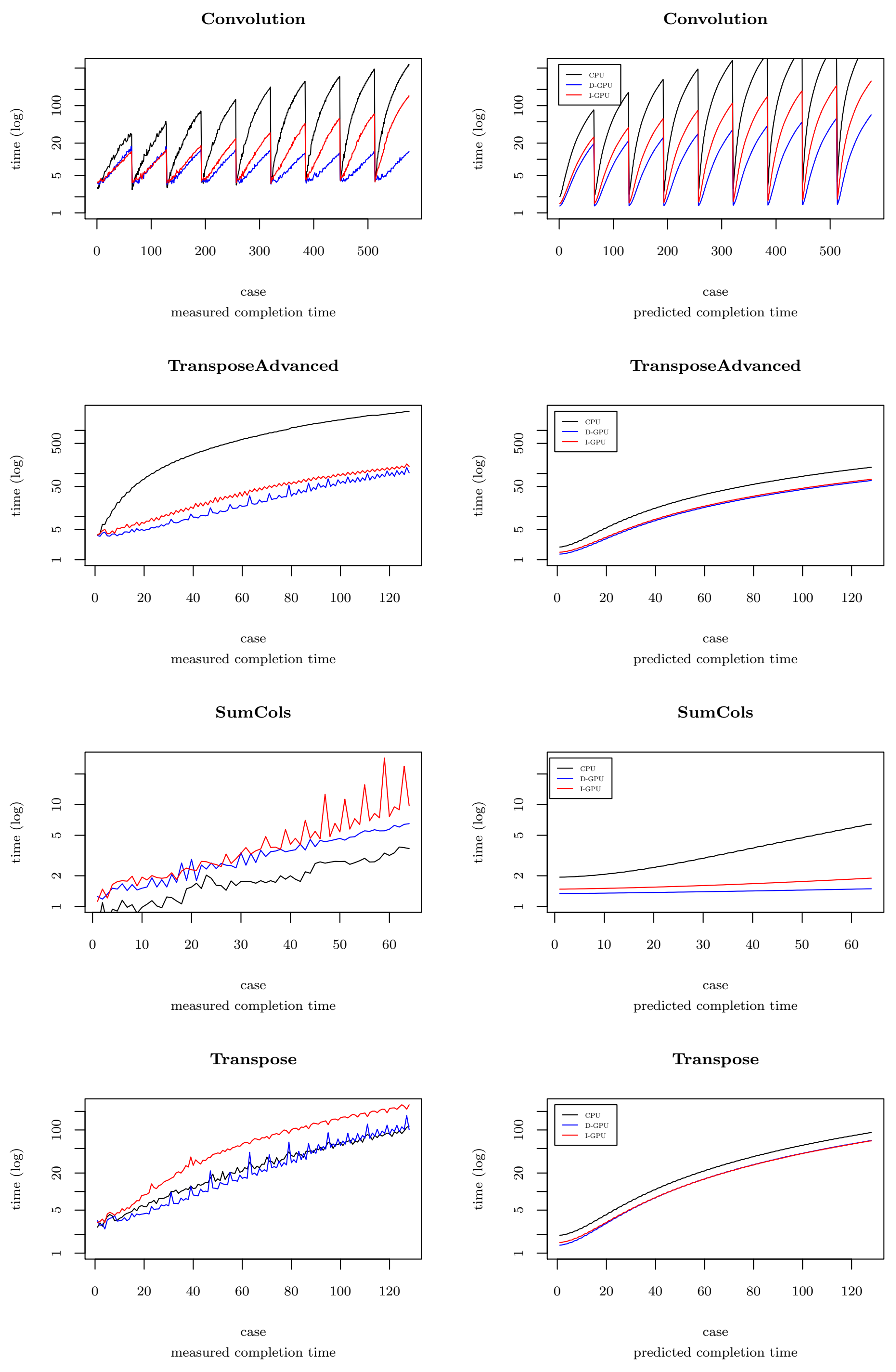


Figure 13.5: Measured and predicted completion time

The overall behaviour of most of the samples shows a simple relation between the input size and the completion time. Two noticeable exceptions are *MatMulNaive* and *SumRows*, whose non-monotonicity is well above measurement noise. A possible explanation for the evident spikes in the graphs is the eviction of many cache lines when accessing memory with a stride of 4KB, 6KB and 8KB. These spikes are correctly estimated thanks to the the cache-miss-estimation feature. The same behaviour is correctly predicted in *SumRows*, even though it does not belong to the set of training samples. This is consistent with our expectation: the aforementioned spikes are indeed caused by a particularly aggressive cache eviction.

The prediction of CPU completion times is generally more accurate than the prediction of GPUs. This is mainly due to the fact that the cache-miss-estimation feature models the cost of accessing memory from the CPU with sufficient accuracy, while it does not properly fit the GPUs, where the cost of memory accesses depends on the access pattern in a different way. To accurately predict the completion time on GPUs, additional information must be retrieved and analysed, such as coalescing in reading and writing memory, ALU fetch ratio[8] and channel/bank conflicts. In addition, information about the usage of LDS (Local Data Share) memory are needed to improve the prediction of *TransposeAdvanced*.

13.4.3 Best-device prediction

The quality of completion-time prediction is only partially related to the quality of best-device guessing. A very precise prediction of the measured completion times leads to a reliable best-device guess, but errors that may affect the completion time prediction not necessarily imply a specific error in guessing the most efficient device. It is sufficient to consider a linear model that overestimates the completion time of a constant factor independently from the input size and the device. In such a case, the quality of the completion time estimation is low, but the one of best-device guessing may be instead very high.

For this reason, we evaluate the accuracy of best-device prediction. Figure 13.6 shows the frequency of the relative prediction accuracy, measured as the ratio between the completion time of the device predicted by our algorithm and the actual optimal device. We also show the geometric mean of the relative accuracy (red line). A geometric mean equal to 1 corresponds to situations where the algorithm always predicts the correct device. When

[8]The occupancy of GPU ALUs during the time a memory request is served

the geometric mean is near to 1 the algorithm does not always predict the best device across the input sizes, but the performance degradation is low, hence the error is small. Higher values of the geometric mean correspond to situations where the difference between the predicted device and the best device is relevant. For example, a geometric mean close to 2 means that the predicted device usually takes twice the time than the best device.

The programs in the basis (*VectorAdd*, *MatMulNaive* and *Logistic*) show an ideal behaviour, with a geometric mean very close to 1, or exactly 1. *Convolution* also shows an ideal prediction error. *SumRows*, *Sobel* and *TransposeAdvanced* have a very low geometric mean, with the vast majority of the predictions being accurate. *Transpose* is plotted on a different x scale, because it is the only algorithm for which we obtain prediction errors larger than 2. The geometric mean for this test sample is therefore high (close to 2), even though most of the predictions were accurate (the frequency of the bin relative to 2 is considerably higher than the others combined together).

13.4.4 Interpretation of the regression coefficients

Each linear regression coefficient can be easily interpreted as the time needed to perform one unit of the corresponding feature. As explained in section 5.1 a *valid* feature "counts" the occurrences of a certain phenomenon, while the corresponding coefficient quantifies the time needed for each occurrence. The linear regression applied to the training samples creates a linear model for each device, with a coefficient for each feature. In our experiments, the coefficient for the *total number of instructions executed* feature for the CPU is $0.24e-9$. If we consider this coefficient, the value of the corresponding feature and the completion time on the CPU, we can estimate the number of operations per second of the CPU. The particular value of the regression coefficient for the total number of instructions leads to an estimated $4.2e9$ operations per second, which means $2.1e9$ operations per second on each of the two CPU cores. This number is very close to the declared operating frequency of the CPU, between 3.8 and 4.2 GHz. A similar evaluation can be done for the GPUs.

Given that we count high-level instructions that may not have a one-to-one match with low level executable code, we expected a larger estimation error. Moreover, not all of the low-level instructions require exactly a single clock cycle and many other relevant behaviours are not modelled, such as super-scalarity and the effect of branches. Nonetheless, the coefficients and their corresponding physical values are surprisingly close. This shows that our method can accomplish a twofold purpose: predict the completion

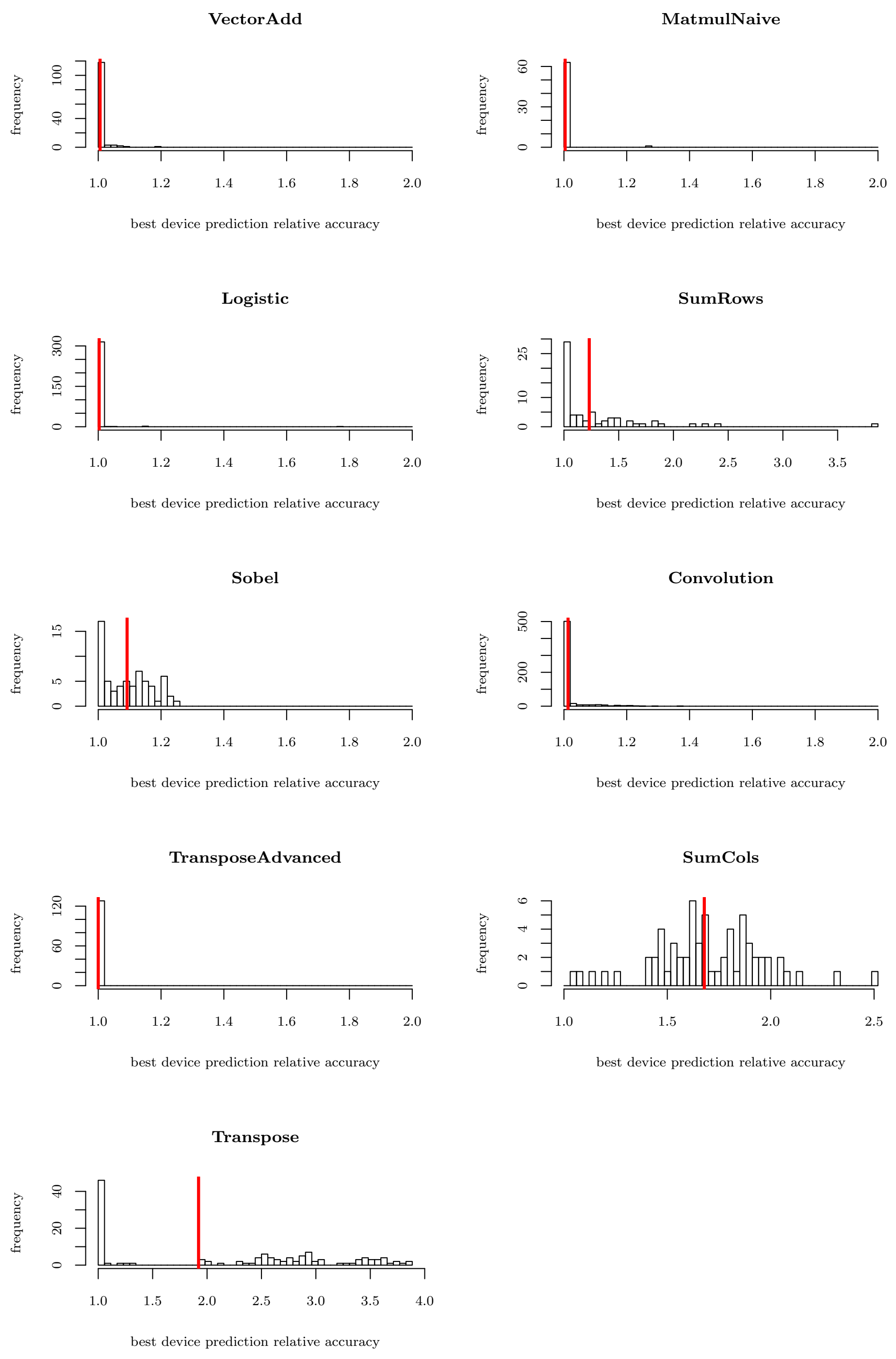


Figure 13.6: Best device prediction relative accuracy

time of computations running on heterogeneous platforms and estimate the characteristics of the available devices.

13.4.5 Limits

As shown in the previous sections, linear regression can successfully describe and predict complex behaviours of hardware and software, but only if the following conditions are met:

- The features must provide an adequate model of the costs incurred by target programs. The model can be high-level, i.e. it does not need to explicitly account for compiler optimization or runtime details, but it must capture all of the aspects that characterize the computation. For example, in our experiments we can successfully predict the CPU cache miss behaviour of *Sum of matrix rows* because a similar behaviour is shown by *Matrix multiplication naive*, that is present in the basis. Conversely, with our set of features we would not have been able to evaluate I/O costs, as none of the features was related to them.
- The basis must exercise the features independently, to avoid multicollinearity between the input data. In our experiments, the *Logistic map* kernel was designed so that the number of work items and the amount of arithmetic operations could be chosen independently. In the other kernels we used, both the number of work items and the number of operations to be performed depend on the size of the data.

If none of the features captures an important behaviour of the target program, predictions will contain a large error. The opposite is also usually true, that is if predictions have a large error the target program is characterized by a relevant uncaught behaviour. This is interesting because it helps understanding what is worth investigating when trying to describe a target program.

If two algorithms are similar in every feature we measure but they differ on an important aspect that we do not measure, then we will not be able to capture their difference in the model, and the predictions will not be able to discriminate between them.

The prediction model can only capture linear relationships between completion time and features. Only features that "count" events that are relevant for the completion time will affect the prediction.

Static code analysis shows some limitations in case of complex control flows, such as loops where the trip count depends on the content of an array or while-loops conditioned by multiple variables. Some advanced techniques, like LLVM branch probability estimation and loop-unrolling algorithms, may be employed in the future to widen the range of computations that can be covered.

13.5 Conclusions

In this chapter we presented a simple but effective completion time prediction approach that uses the *linear regression solver* to create a model for each device exposed by arbitrary heterogeneous platforms. We evaluated the quality of the approach to predict the completion time of an heterogeneous set of devices and the effectiveness in determining the best device where to schedule a kernel.

The prediction accuracy of our model has been tested against more complex models, like the one described by Wen (2014), based on SVM, and despite its simplicity, it resulted more accurate. Following Occam's Razor, the simpler model should be chosen over a complex one. Moreover, the regression coefficients found by the linear model have an intuitive physical explanation, as shown in section 13.4.4.

Part IV

Conclusions

As discussed in chapters 1.1 and 2, the field of software and hardware performance analysis and energy characterization is fragmented and lacks of a systematic approach with the properties of *accuracy*, *consistency*, *broad scope*, *simplicity*, and *fruitfulness*, listed by Kuhn (1977) as requisites for a good scientific model. In this work we attempt to provide a simple, generic, and abstract model that adheres to the aforementioned characteristics.

Chapter 14

Contributions

In this work we have introduced the concept of *computational pattern*, a theoretical basic unit of computation. We postulate the existence of a large number of *computational patterns*, one for each computational behaviour with respect to *computational resource* consumption. We also postulate that real *programs* are composed of combinations of *computational patterns*, and that the different *resource consumption* of the *patterns* on different machines can explain the different behaviour of *programs* on different machines. *Computational patterns* are not directly measurable, because real *programs* are composed of several different *patterns*.

We have presented our benchmarking model, an attempt to formally define a well founded, black-box, unified hardware and software, based upon generic resources model, capable of both characterizing and predicting *target program's* and *target resource's* consumption.

The model is black-box both from the point of view of the hardware and the software:

hardware : it is sufficient to observe the consumption of resources from outside the *computational environment*, i.e. energy consumption alone is enough to characterize, as shown in chapter 10.

software : *programs* can be characterized without analysing their source code, simply observing and performing statistical analysis of their resource usage.

Our benchmarking model offers a unified view on hardware and software. In literature, hardware and software have traditionally been modelled with different approaches and methodologies. We argue that, due to their intrinsic relationship (one depends on the other), they should be modelled by a unified approach. Our benchmarking model can be used to characterize a *target program*, or a *target resource*, with simple algebraic transformations (a matrix transpose on the matrix $\mathbf{X}$).

Our work also attempts to offer a simple approach, valid for the broadest possible set of *computational resources*. Traditionally, profoundly different models have been proposed for different *computational resources*. Our model treats all *resources* in the same way, assuming a few properties are respected (the mathematical properties of measures). Examples of *resources* are completion time, energy consumption, and performance counters.

We have described how different *solvers* can be used to characterize and to predict the *resource consumption* of the *target program*:

Simplex is straightforward under the assumption that the *benchmarks* used to build the matrix $\mathbf{X}$ embody *computational patterns*, and that the *target program* is composed of *computational patterns* present in the used *benchmarks*;

Linear Regression is the simplest and more generic solver, that does not assume the *benchmarks* to embody *computational patterns*, therefore more suited for real world *programs* and *computational environments*. The only assumption used in this solver is that the underlying *computational patterns* compose linearly into *programs*. We argue that non-linear independent variables can model complex behaviour of dependent variables. In the experimental validation we report several experiments where the *linear solver* was able to capture elusive behaviours, such as the effect of cache-eviction on completion time;

NonNegative Matrix Factorization is a natural choice, having postulated that *programs* are linear combinations of *computational patterns*. NMF offers a possible characterization of the *benchmarks* in terms of hidden factors, possibly *computational patterns*, identifying their *resource consumption*.

Despite the simplicity of the *solvers* presented, in the experimental section we show that we were able to predict the performance of *programs*, with an accuracy equal if not superior to other approaches in literature, that make use of complex models. As stated by Newton at the beginning of the 3rd book of the "Principia": "We are to admit no more causes of natural things than such as are both true and sufficient to explain their appearances. Therefore, to the same natural effects we must, as far as possible, assign the same causes" (Newton, 2011).

We introduce the notion of *experimental computational complexity* ξ, as a curve fitting process on the *surrogates* of *programs*, as the input size grows. We show the relationship with traditional time complexity. We also show some of the properties of compositionality of ξ. It can be used to characterize a *target program*, but also to predict the *surrogate* of a *program* for which we have no measures (by interpolation, or extrapolation).

Most of the benchmarking approaches in literature attempt to describe systems and programs using a single metric. In section 7.6 we discussed why this approach inevitably leads to a large characterization (therefore prediction) error. We show that such approach can only lead to small error if either the model is used on a single architecture, or the model is normalized using the same program that we want to characterize

The natural tendency in industry to offer a single number to characterize the performance of systems and program could be a reason why prediction models in literature are usually designed to describe a single architecture, with little portability. Our approach not only can be ported from an architecture to another, but is more precise if the matrix $\mathbf{X}$ contains measures coming from different architectures.

Lastly, we introduced a simple energy model, capable of accurately characterize the power and energy consumption of *computational environments* from the *resource* to the cluster level. We also showed how to automatically characterize the power consumption of the factors in the model. In chapter 12 we presented an experiment that makes use of the energy model. The energy model is capable of describing the energy consumption of concurrent parallel programs.

We have validated our contributions with extensive testing on several different machines, using both micro-benchmarks and real-world programs, as well as widely used benchmarking suites.

Chapter 15

Future work

As discussed in section 6.4, more solvers could be explored. We think that a Bayesian approach to regression would be beneficial, because it makes the assumptions behind the model explicit, and offers a characterization in terms of random distribution, more expressive than single numbers in presence of noise. The model could also be used without assuming linearity in the combination of *computational patterns* into *programs*, using non-linear regression methods, such as Support Vector Regression.

Our benchmarking model analyses *programs* only considering their whole execution. It would be interesting to apply the model to program phases. Programs behaviour characterization has been explored by Duesterwald et al. (2003), Sherwood et al. (2002), Wunderlich et al. (2003), and Perelman et al. (2003). An elementary first attempt could be to split the *program*'s execution in time windows, and use apply our model to the measured *resource* consumption in every window independently. The *surrogate* would then become the evolution of the *surrogates* of each time window. The model could then be enhanced allowing uneven window sizes, detecting a substantial change in the *surrogate* as the trigger to declare the beginning of a new program's phase. However, this approach treats every window independently, neglecting the underlying structure between program's phases. To overcome this limitation, an Hidden Markov Model (HMM) could be used to model the evolution of the program, assuming the phase as the hidden state. The *surrogate* would then not only identify the program's phases, but also characterize each phase, and describe the transition probability between phases. To pursue this approach, we would need to redefine the *resource consumption* to allow partial usage (in a time window, as opposed to the whole program's execution); then using the cosine similarity between defined in section 5.4.3 we could separate program's phases; then encode the transitions with an HMM.

We tested our model on several types of devices, including netbooks, workstations, HPC cluster enclosures, heterogeneous platforms equipped with CPU, GPU, and APU. However, our model should also apply to embedded devices and FPGAs. It would be interesting to run experiments on a broader range of devices to verify if the model is still capable of accurate predictions.

Our benchmarking model is abstract enough to be applied to other fields. Potentially it could be used to characterize and predict the behaviour of systems where the interaction between the components can be approximated with linear interactions of smaller, simpler elements.

For example, it could be applied to the field of business benchmarking. Modelling the use of *resources* by explanatory business processes to complete tasks, our model could be used to characterize a particular business process in terms of other processes (the rows of the matrix $\mathbf{X}$ being *resources* and the columns business processes, the vector $\mathbf{y}$ as the target business process). Also, it would be interesting to model a performance metrics in terms of consumption resources by business processes (the rows of the matrix $\mathbf{X}$ being the processes, the columns the resources, the vector $\mathbf{y}$ being the analysed performance metric).

Bibliography

A. R. Alameldeen and D. A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006. URL `http://www.cecs.pdx.edu/~alaa/ece588/papers/alameldeen_ieeemicro_2006.pdf`.

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*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: http://doi.acm.org/10.1145/1465482.1465560. URL `http://doi.acm.org/10.1145/1465482.1465560`.

G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP 2004Object-Oriented Programming*, pages 172–196. Springer, 2004. URL `http://link.springer.com/chapter/10.1007/978-3-540-24851-4_8`.

F. J. Anscombe. Graphs in statistical analysis. *The American Statistician*, 27(1):17–21, 1973. URL `http://amstat.tandfonline.com/doi/pdf/10.1080/00031305.1973.10478966`.

D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, and others. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991. URL `http://hpc.sagepub.com/content/5/3/63.short`.

A. Beloglazov and R. Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '10, pages 4:1–4:6, New York, NY, USA, 2010a. ACM. ISBN 978-1-4503-0453-5. doi: http://doi.acm.org/10.

1145/1890799.1890803. URL `http://doi.acm.org/10.1145/1890799.1890803`.

A. Beloglazov and R. Buyya. Energy Efficient Resource Management in Virtualized Cloud Data Centers. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 826–831, Washington, DC, USA, 2010b. IEEE Computer Society. ISBN 978-0-7695-4039-9. doi: http://dx.doi.org/10.1109/CCGRID.2010.46. URL `http://dx.doi.org/10.1109/CCGRID.2010.46`.

M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, and others. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of High Performance Computing Applications*, 3(3):5–40, 1989. URL `http://hpc.sagepub.com/content/3/3/5.short`.

R. Bianchini and R. Rajamony. Power and energy management for server systems. *IEEE Computer*, 37(11):68–74, 2004. URL `ftp://athos.rutgers.edu/cs/pub/technical-reports/work/dcs-tr-528.pdf`.

W. L. Bircher and L. K. John. Analysis of dynamic power management on multi-core processors. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 327–338, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: http://doi.acm.org/10.1145/1375527.1375575. URL `http://doi.acm.org/10.1145/1375527.1375575`.

W. L. Bircher and L. K. John. Complete system power estimation using processor performance events. *Computers, IEEE Transactions on*, 61 (4):563–577, 2012. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5714687`.

C. M. Bishop and M. E. Tipping. Bayesian regression and classification. *Nato Science Series sub Series III Computer And Systems Sciences*, 190: 267–288, 2003. URL `http://research.microsoft.com/pubs/67158/bishop-nato-bayes.pdf`.

A. Blanco-Fernndez, A. Colubi, M. Garca-Brzana, and M. Montenegro. A Linear Regression Model for Interval-Valued Response Based on Set Arithmetic. In R. Kruse, M. R. Berthold, C. Moewes, M. . Gil, P. Grzegorzewski, and O. Hryniewicz, editors, *Synergies of Soft Computing and*

Statistics for Intelligent Data Analysis, number 190 in Advances in Intelligent Systems and Computing, pages 105–113. Springer Berlin Heidelberg, Jan. 2013. ISBN 978-3-642-33041-4, 978-3-642-33042-1. URL `http://link.springer.com/chapter/10.1007/978-3-642-33042-1_12`.

J. L. Bonebakker. Finding representative workloads for computer system design. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2007.

C. Boutsidis and E. Gallopoulos. SVD based initialization: A head start for nonnegative matrix factorization. *Pattern Recognition*, 41(4):1350–1362, 2008. URL `http://www.sciencedirect.com/science/article/pii/S0031320307004359`.

G. E. Box and N. R. Draper. *Empirical model-building and response surfaces.* John Wiley & Sons, 1987. URL `http://psycnet.apa.org/psycinfo/1987-97236-000`.

D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM. ISBN 1-58113-232-8. doi: http://doi.acm.org/10.1145/339647.339657. URL `http://doi.acm.org/10.1145/339647.339657`.

D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997. URL `http://dl.acm.org/citation.cfm?id=268810`.

R. Buyya, A. Beloglazov, and J. Abawajy. Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges. *arXiv preprint arXiv:1006.0308*, 2010. URL `http://arxiv.org/abs/1006.0308`.

A. S. Cassidy and A. G. Andreou. Beyond Amdahl's Law: An Objective Function That Links Multiprocessor Performance Gains To Delay and Energy. *IEEE TRANSACTIONS ON COMPUTERS*, 2011.

F. Chang, K. Farkas, and P. Ranganathan. Energy-Driven Statistical Sampling: Detecting Software Hotspots. In B. Falsafi and T. Vijaykumar, editors, *Power-Aware Computer Systems*, volume 2325 of *Lecture Notes in Computer Science*, pages 105–108. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-01028-9.

J.-J. Chen and C.-F. Kuo. Energy-Efficient Scheduling for Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 28–38, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2975-5. doi: http://dx.doi.org/10.1109/RTCSA.2007.37. URL `http://dx.doi.org/10.1109/RTCSA.2007.37`.

S. Cho and R. Melhem. Corollaries to Amdahl's Law for Energy. *IEEE Comput. Archit. Lett.*, 7(1):25–28, Jan. 2008. ISSN 1556-6056. doi: 10.1109/L-CA.2007.18. URL `http://dl.acm.org/citation.cfm?id=1383041.1383084`.

S. Cho and R. G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *Parallel and Distributed Systems, IEEE Transactions on*, 21(3):342–353, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4798160`.

A. Cichocki, R. Zdunek, and S.-I. Amari. Nonnegative matrix and tensor factorization [lecture notes]. *Signal Processing Magazine, IEEE*, 25 (1):142–145, 2008. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4408452`.

G. Cocco. *FSCL: Homogeneous programming and execution on heterogeneous platforms*. PhD dissertation, University of Pisa, 2015.

M. Colyvan. *The indispensability of mathematics*. Oxford University Press, 2001. URL `http://espace.library.uq.edu.au/view/UQ:1352`.

C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3): 273–297, 1995. URL `http://link.springer.com/article/10.1007/bf00994018`.

H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976. URL `http://comjnl.oxfordjournals.org/content/19/1/43.short`.

M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 157–166, New York, NY, USA, 2006. ACM. ISBN 1-59593-282-8. doi:

http://doi.acm.org/10.1145/1183401.1183426. URL http://doi.acm.org/10.1145/1183401.1183426.

C. Davis. Theory of positive linear dependence. *American Journal of Mathematics*, pages 733–746, 1954. URL http://www.jstor.org/stable/2372648.

G. Dhiman, G. Marchetti, and T. Rosing. vGreen: A System for Energy-Efficient Management of Virtual Machines. *ACM Trans. Des. Autom. Electron. Syst.*, 16(1):6:1–6:27, Nov. 2010. ISSN 1084-4309. doi: http://doi.acm.org/10.1145/1870109.1870115. URL http://doi.acm.org/10.1145/1870109.1870115.

J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK users' guide*, volume 8. Siam, 1979. URL https://books.google.it/books?hl=en&lr=&id=AmSm1n3Vw0cC&oi=fnd&pg=PR5&dq=linpack&ots=EEGawHct3u&sig=t9rebtIN7_taRAKjvf1S-oF6sNg.

H. Drucker, C. J. Burges, L. Kaufman, A. Smola, V. Vapnik, and others. Support vector regression machines. *Advances in neural information processing systems*, 9:155–161, 1997. URL https://books.google.it/books?hl=en&lr=&id=QpD7n95ozWUC&oi=fnd&pg=PA155&dq=vapnik+support+vector+machine&ots=iCmrlFYR9w&sig=sjPNZuzMMuH3vyVLrpLOUSmyS08.

E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 220–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2021-9. URL http://dl.acm.org/citation.cfm?id=942806.943853.

P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler. Energy Metering for Free: Augmenting Switching Regulators for Real-Time Monitoring. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 283–294, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3157-1. doi: http://dx.doi.org/10.1109/IPSN.2008.58. URL http://dx.doi.org/10.1109/IPSN.2008.58.

L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Workload Design: Selecting Representative Program-Input Pairs. *Parallel Architectures and*

Compilation Techniques, International Conference on, 0:83, 2002. ISSN 1089-795X. doi: http://doi.ieeecomputersociety.org/10.1109/PACT.2002.1106006.

F. Farahnakian, P. Liljeberg, and J. Plosila. LiRCUP: Linear Regression Based CPU Usage Prediction Algorithm for Live Migration of Virtual Machines in Data Centers. In *2013 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 357–364, Sept. 2013. doi: 10.1109/SEAA.2013.23.

X. Feng, R. Ge, and K. W. Cameron. Power and Energy Profiling of Scientific Applications on Distributed Systems. 2005. URL `http://www.mscs.mu.edu/~fengx/web/pubs/IPDPS2005PowerProfiling.pdf`.

J. Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, WMCSA '99, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0025-0. URL `http://dl.acm.org/citation.cfm?id=520551.837522`.

J. Fox. *Applied regression analysis, linear models, and related methods.* Sage Publications, Inc, 1997. URL `http://psycnet.apa.org/psycinfo/1997-08857-000`.

S. Gal-On and M. Levy. Exploring CoreMarkA Benchmark Maximizing Simplicity and Efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012. URL `https://backdraft-technologies.com/techlit/coremark-whitepaper.pdf`.

S. K. Garg, C. S. Yeo, A. Anandasivam, and R. Buyya. Energy-efficient scheduling of HPC applications in cloud computing environments. *arXiv preprint arXiv:0909.1146*, 2009. URL `http://arxiv.org/abs/0909.1146`.

R. Gaujoux and C. Seoighe. A flexible R package for nonnegative matrix factorization. *BMC bioinformatics*, 11(1):367, 2010. URL `http://www.biomedcentral.com/1471-2105/11/367`.

C. J. Geyer. Practical markov chain monte carlo. *Statistical Science*, pages 473–483, 1992. URL `http://www.jstor.org/stable/2246094`.

W. R. Gilks. *Markov chain monte carlo.* Wiley Online Library, 2005. URL http://onlinelibrary.wiley.com/doi/10.1002/0470011815.b2a14021/full.

I. Goiri, F. Julia, R. Nou, J. L. Berral, J. Guitart, and J. Torres. Energy-aware scheduling in virtualized datacenters. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 58–67. IEEE, 2010. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5600320.

S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 395–404. ACM, 2007. URL http://dl.acm.org/citation.cfm?id=1287681.

S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982. URL http://dl.acm.org/citation.cfm?id=806987.

C. Gu, H. Huang, and X. Jia. Power Metering for Virtual Machine in Cloud ComputingChallenges and Opportunities. 2014. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6905704.

S. Hawking. *The grand design.* Random House LLC, 2011.

S. Hemmert. Green hpc: From nice to necessity. *Computing in Science & Engineering*, 12(6):0008–10, 2010. URL http://www.computer.org/csdl/mags/cs/2010/06/mcs2010060008.html.

J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach.* Morgan Kaufman, 2003.

J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, July 2000. ISSN 0018-9162. doi: 10.1109/2.869367. URL http://dl.acm.org/citation.cfm?id=619053.621510.

D. C. Hoaglin, F. Mosteller, and J. W. Tukey. *Exploring data tables, trends, and shapes*, volume 101. John Wiley & Sons, 2011. URL http://books.google.it/books?hl=en&lr=&id=pzf2u-vHk3gC&oi=fnd&pg=PR13&dq=+Robust+regression.+In+Exploring+Data+Tables,+Trends,+and+Shapes&ots=NFyYThiBgM&sig=IOlYI9kw4N2ZOM1GROHFg-O5Eb8.

P. W. Holland and R. E. Welsch. Robust regression using iteratively reweighted least-squares. *Communications in Statistics-Theory and Methods*, 6(9):813–827, 1977. URL `http://www.tandfonline.com/doi/abs/10.1080/03610927708827533`.

L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *NIPS*, pages 883–891. Curran Associates, Inc., 2010. URL `http://dblp.uni-trier.de/db/conf/nips/nips2010.html#HuangJYCMN10`.

C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: simulating shared-memory multiprocessors with ILP processors. *Computer*, 35(2):40–49, 2002. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=982915`.

C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi: http://dx.doi.org/10.1109/MICRO.2006.8. URL `http://dx.doi.org/10.1109/MICRO.2006.8`.

T. Isobe, E. D. Feigelson, M. G. Akritas, and G. J. Babu. Linear regression in astronomy. *The Astrophysical Journal*, 364:104–113, Nov. 1990. ISSN 0004-637X. doi: 10.1086/169390. URL `http://adsabs.harvard.edu/abs/1990ApJ...364..104I`.

M. A. Iverson, F. Ozguner, and L. C. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 99–111. IEEE, 1999. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=765115`.

R. Jain. The art of computer system performance analysis: techniques for experimental design, measurement, simulation and modeling. *New York: John Willey*, 1991.

A. P. Kareem and R. A. Singh. Principal component and cluster analysis of SPEC CPUint2006 Benchmarks. 2015. URL `http://hypatia.teiath.gr/xmlui/handle/11400/4987`.

Ketan Paranjape, Steve Hebert, and Bob Masson. Heterogeneous Computing in the Cloud: Crunching Big Data and Democratizing HPC Access for the Life Sciences. Technical report, 2014.

N. Kim, J. Cho, and E. Seo. Energy-based accounting and scheduling of virtual machines in a cloud system. In *Green Computing and Communications (GreenCom), 2011 IEEE/ACM International Conference on*, pages 176–181. IEEE, 2011. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6061323`.

N. Kim, J. Cho, and E. Seo. Energy-credit scheduler: An energy-aware virtual machine scheduler for cloud systems. *Future Generation Computer Systems*, 32:128–137, 2014. URL `http://www.sciencedirect.com/science/article/pii/S0167739X1200115X`.

V. Kindratenko and P. Trancoso. Trends in high-performance computing. *Computing in Science & Engineering*, 13(3):92–95, 2011. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5756280`.

T. S. Kuhn. Objetivity, value judgment, and theory choice. 1977. URL `http://books.google.it/books?hl=en&lr=&id=iGpd3xLGNbYC&oi=fnd&pg=PA74&dq=Objectivity,+Value+Judgment,+and+Theory+Choice.&ots=1ocI5l6Z-2&sig=i9ZAx202dZigqsaNDEQRg1rjTrE`.

T. S. Kuhn. *The structure of scientific revolutions*. University of Chicago press, 2012. URL `http://books.google.it/books?hl=en&lr=&id=3eP5Y_OOuzwC&oi=fnd&pg=PR5&dq=The+Structure+of+Scientific+Revolutions&ots=xUYOD8lJnN&sig=aLYSOzSLoodXkE2Tt2FMF7SqNjk`.

M. Kuperberg, K. Krogmann, and R. Reussner. Performance prediction for black-box components using reengineered parametric behaviour models. In *Component-Based Software Engineering*, pages 48–63. Springer, 2008. URL `http://link.springer.com/chapter/10.1007/978-3-540-87891-9_4`.

A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. *SIGPLAN Not.*, 35(11):105–116, Nov. 2000. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/356989.356999. URL `http://doi.acm.org/10.1145/356989.356999`.

D. D. LEE. Algorithms for nonnegative matrix factorization. *Advances in Neural Information Processing Systems*, 13:556–562, 2001. URL `http://ci.nii.ac.jp/naid/10020951848/`.

L. Lefvre and A.-C. Orgerie. Designing and evaluating an energy efficient Cloud. *J. Supercomput.*, 51(3):352–373, Mar. 2010. ISSN 0920-8542. doi: http://dx.doi.org/10.1007/s11227-010-0414-2. URL `http://dx.doi.org/10.1007/s11227-010-0414-2`.

K. Li. Energy efficient scheduling of parallel tasks on multiprocessor computers. *The Journal of Supercomputing*, 60(2):223–247, 2012. URL `http://link.springer.com/article/10.1007/s11227-010-0416-0`.

T. Li and L. K. John. Run-time modeling and estimation of operating system power consumption. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):160–171, 2003. URL `http://dl.acm.org/citation.cfm?id=781048`.

D. J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005. URL `http://books.google.it/books?hl=en&lr=&id=R8RLniX5DNQC&oi=fnd&pg=PR11&dq=Measuring+computer+performance:+a+practitioner%27s+guide&ots=irGuTxFwtx&sig=Vuq9DQCE_oUdzkijHmv3ItWjry0`.

M. Lindner, F. Galn, C. Chapman, S. Clayman, D. Henriksson, and E. Elmroth. The cloud supply chain: A framework for information, monitoring, accounting and billing. In *2nd International ICST Conference on Cloud Computing (CloudComp 2010)*, 2010. URL `https://www.ee.ucl.ac.uk/~sclayman/docs/CloudComp2010.pdf`.

H. Liu, C.-Z. Xu, H. Jin, J. Gong, and X. Liao. Performance and energy modeling for live migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 171–182, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0552-5. doi: http://doi.acm.org/10.1145/1996130.1996154. URL `http://doi.acm.org/10.1145/1996130.1996154`.

X. Ma, M. Dong, L. Zhong, and Z. Deng. Statistical power consumption analysis and modeling for GPU-based computing. In *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009. URL `http://www.sigops.org/sosp/sosp09/papers/hotpower_6_ma.pdf`.

J. L. Marchini, C. Heaton, and B. D. Ripley. fastICA: FastICA algorithms to perform ICA and Projection Pursuit. *R package version*, pages 1–2, 2013.

A. D. Martin, K. M. Quinn, and J. H. Park. Mcmcpack: Markov chain monte carlo in r. *Journal of Statistical Software*, 42(9):1–21, 2011. URL `https://www.law.berkeley.edu/files/jstatsoftMCMCpack.pdf`.

A. J. Martin. Towards an energy complexity of computation. *Inf. Process. Lett.*, 77(2-4):181–187, Feb. 2001. ISSN 0020-0190. doi: 10.1016/S0020-0190(00)00214-3. URL `http://dl.acm.org/citation.cfm?id=375434.375482`.

J. R. Mashey. War of the benchmark means: time for a truce. *ACM SIGARCH Computer Architecture News*, 32(4):1–14, 2004. URL `http://dl.acm.org/citation.cfm?id=1040137`.

F. H. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical report, Lawrence Livermore National Lab., CA (USA), 1986. URL `http://www.osti.gov/scitech/biblio/6574702`.

E. Metz and R. Lencevicius. Performance Data Collection: Hybrid Approach. In *Proceedings of the 2nd International Workshop on Dynamic Analysis*, pages 48–51. Citeseer, 2004. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.3951&rep=rep1&type=pdf#page=52`.

D. Meyer, F. Leisch, and K. Hornik. The support vector machine under test. *Neurocomputing*, 55(1):169–186, 2003. URL `http://www.sciencedirect.com/science/article/pii/S0925231203004314`.

R. Mishra, N. Rastogi, D. Zhu, D. Moss, and R. Melhem. Energy aware scheduling for distributed real-time systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9–pp. IEEE, 2003. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1213099`.

D. Morelli and A. Cisternino. A compositional model to characterize software and hardware from their resource usage. In A. V. Jones, editor, *2012 Imperial College Computing Student Workshop*, volume 28 of *OpenAccess Series in Informatics (OASIcs)*, pages 95–101, Dagstuhl, Germany, 2012. Schloss DagstuhlLeibniz-Zentrum fuer Informatik. ISBN 978-3-939897-48-4. doi: http://dx.doi.org/10.4230/OASIcs.ICCSW.2012.95. URL `http://drops.dagstuhl.de/opus/volltexte/2012/3771`.

D. Morelli and A. Cisternino. Accurate Blind Predictions of Open-FOAM Energy Consumption Using the LBM Prediction Model. In *Euro-Par 2014: Parallel Processing Workshops*, pages 400–411. Springer, 2014. URL `http://link.springer.com/chapter/10.1007/978-3-319-14313-2_34`.

T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34 (4):52–58, 2001. URL `http://www.computer.org/csdl/mags/co/2001/04/r4052.pdf`.

S. S. Mukherjee, S. V. Adve, T. Austin, J. Emer, and P. S. Magnusson. Performance simulation tools. *Computer*, (2):38–39, 2002. URL `http://www.computer.org/csdl/mags/co/2002/02/r2038.pdf`.

K. G. Murty. *Linear programming*, volume 57. Wiley New York, 1983.

T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong. In *In Proc. of Intl Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 265–276. ACM, 2009.

A. Nair and R. Lysecky. Non-intrusive dynamic application profiler for detailed loop execution characterization. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 23–30. ACM, 2008. URL `http://dl.acm.org/citation.cfm?id=1450102`.

D. Nellans, V. K. Kadaru, and E. Brunvand. ASIM-An asynchronous architectural level simulator. In *Proceedings of GLSVLSI*. Citeseer, 2004. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.131.6213&rep=rep1&type=pdf`.

N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007. URL `http://dl.acm.org/citation.cfm?id=1250746`.

R. Neugebauer and D. Mcauley. Energy is just another resource: energy accounting and energy pricing in the Nemesis OS. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 67 – 72, May 2001. doi: 10.1109/HOTOS.2001.990063.

I. Newton. *Philosophi Naturalis Principia Mathematica.* Henry Pemberton, 3rd edition, 2011. ISBN 978-1-60386-435-0.

A. Pascual-Montano, J. M. Carazo, K. Kochi, D. Lehmann, and R. D. Pascual-Marqui. Nonsmooth nonnegative matrix factorization (nsNMF). *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28 (3):403–415, 2006. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1580485`.

R. Patel and A. Rajwat. A survey of embedded software profiling methodologies. *arXiv preprint arXiv:1312.2949*, 2013. URL `http://arxiv.org/abs/1312.2949`.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: the hardware/software interface*. Morgan Kaufman, 2008.

E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 244–255. IEEE, 2003. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1238020`.

A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 10–20, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-8965-4. doi: 10.1109/ISPASS.2005.1430555. URL `http://dl.acm.org/citation.cfm?id=1317536.1318392`.

A. S. Phansalkar. *Measuring program similarity for efficient benchmarking and performance analysis of computer systems*. PhD thesis, University of Texas at Austin, Austin, TX, USA, 2007. AAI3285977.

K. Popper. *The logic of scientific discovery*. Routledge, 2014. URL `http://books.google.it/books?hl=en&lr=&id=LWSBAgAAQBAJ&oi=fnd&pg=PP1&dq=The+Logic+of+Scientific+Discovery+&ots=pyDi-Z1EeM&sig=d1sUegeiMYbpJbNBO7-o71PPnnQ`.

W. J. Price. A benchmark tutorial. *Micro, IEEE*, 9(5):28–43, 1989. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=45825`.

R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No ”power” struggles: coordinated multi-level power management for the data center. *SIGARCH Comput. Archit. News*, 36(1):48–59, Mar. 2008.

ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1353534.1346289. URL `http://doi.acm.org/10.1145/1353534.1346289`.

K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 302–313, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: http://doi.acm.org/10.1145/1555754.1555793. URL `http://doi.acm.org/10.1145/1555754.1555793`.

S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: a balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 365–376, New York, NY, USA, 2007a. ACM. ISBN 978-1-59593-686-8. doi: http://doi.acm.org/10.1145/1247480.1247522. URL `http://doi.acm.org/10.1145/1247480.1247522`.

S. Rivoire, M. A. Shah, P. Ranganathan, C. Kozyrakis, and J. Meza. Models and Metrics to Enable Energy-Efficiency Optimizations. *Computer*, 40 (12):39–48, Dec. 2007b. ISSN 0018-9162. doi: 10.1109/MC.2007.436. URL `http://dl.acm.org/citation.cfm?id=1339817.1339896`.

S. M. Rivoire. *Models and metrics for energy-efficient computer systems*. PhD thesis, Stanford University, Stanford, CA, USA, 2008. AAI3313649.

P. E. Roundy and W. M. Frank. Applications of a Multiple Linear Regression Model to the Analysis of Relationships between Eastward- and Westward-Moving Intraseasonal Modes. *Journal of the Atmospheric Sciences*, 61 (24):3041–3048, Dec. 2004. ISSN 0022-4928. doi: 10.1175/JAS-3349.1. URL `http://journals.ametsoc.org/doi/abs/10.1175/JAS-3349.1`.

J. Russell and M. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings. International Conference on*, pages 328 –333, Oct. 1998. doi: 10.1109/ICCD.1998.727070.

R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14(4): 344–384, Nov. 1996. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/235543.235545. URL `http://doi.acm.org/10.1145/235543.235545`.

V. Salapura, R. Bickford, M. Blumrich, A. A. Bright, D. Chen, P. Coteus, A. Gara, M. Giampapa, M. Gschwind, M. Gupta, S. Hall, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, M. Ohmacht, R. A. Rand, T. Takken, and P. Vranas. Power and performance optimization at the system level. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05, pages 125–132, New York, NY, USA, 2005. ACM. ISBN 1-59593-019-1. doi: http://doi.acm.org/10.1145/1062261.1062262. URL `http://doi.acm.org/10.1145/1062261.1062262`.

V. Sekar and P. Maniatis. Verifiable resource accounting for cloud computing services. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 21–26. ACM, 2011. URL `http://dl.acm.org/citation.cfm?id=2046666`.

C. Seo, G. Edwards, D. Popescu, S. Malek, and N. Medvidovic. A framework for estimating the energy consumption induced by a distributed system's architectural style. In *Proceedings of the 8th international workshop on Specification and verification of component-based systems*, SAVCBS '09, pages 27–34, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-680-9. doi: http://doi.acm.org/10.1145/1596486.1596493. URL `http://doi.acm.org/10.1145/1596486.1596493`.

G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of Java programs. In *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 38. Proceedings*, pages 85–101. IEEE, 2001. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=911758`.

L. Shannon and P. Chow. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 190–199. ACM, 2004. URL `http://dl.acm.org/citation.cfm?id=968308`.

S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. Taylor, and X. Wu. Performance projection of HPC applications using SPEC CFP2006 benchmarks. In *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*, pages 1–12, 2009. doi: 10.1109/IPDPS.2009.5161057.

A. Shenoy, J. Hiner, S. Lysecky, R. Lysecky, and A. Gordon-Ross. Evaluation of dynamic profiling methodologies for optimization of sensor networks. *Embedded Systems Letters, IEEE*, 2(1):10–13, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5430950`.

T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. Technical report, University of California at San Diego, La Jolla, CA, USA, 2001.

T. Sherwood, E. Perelman, a. G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36 (5):45–57, Oct. 2002. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/635508.605403. URL `http://doi.acm.org/10.1145/635508.605403`.

Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 134–139. IEEE, 1999. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=781298`.

A. Sinha and A. P. Chandrakasan. JouleTrack - A web based tool for software energy profiling. In *In Design Automation Conference*, pages 220–225, 2001.

P. Smaragdis and J. C. Brown. Non-negative matrix factorization for polyphonic music transcription. In *Applications of Signal Processing to Audio and Acoustics, 2003 IEEE Workshop on.*, pages 177–180. IEEE, 2003. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1285860`.

A. F. Smith and G. O. Roberts. Bayesian computation via the Gibbs sampler and related Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 3–23, 1993. URL `http://www.jstor.org/stable/2346063`.

J. E. Smith. Characterizing computer performance with a single number. *Communications of the ACM*, 31(10):1202–1206, 1988. URL `http://dl.acm.org/citation.cfm?id=63043`.

S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855610.1855620`.

S. P. T. Srinivasan and U. Bellur. Novel Power and Completion Time Models for Virtualized Environments. *arXiv:1411.3201 [cs]*, Nov. 2014. URL `http://arxiv.org/abs/1411.3201`. arXiv: 1411.3201.

S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *in Proc. Int. Wkshp Power and Timing Modeling, Optimization and Simulation (PATMOS*, 2001.

V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, ICCAD '94, pages 384–390, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-89791-690-5. URL `http://dl.acm.org/citation.cfm?id=191326.191500`.

J. G. Tong and M. A. Khalid. Profiling tools for FPGA-based embedded systems: Survey and quantitative comparison. *Journal of Computers*, 3 (6):1–14, 2008. URL `http://www.academypublisher.com/ojs/index.php/jcp/article/viewArticle/03060114`.

N. Vijaykrishnan, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. pages 95–106, 2000.

L. Wang, G. Von Laszewski, J. Dayal, and F. Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with DVFS. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 368–377. IEEE, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5493462`.

S. Wang, H. Chen, and W. Shi. SPAN: A software power analyzer for multicore computer systems. *Sustainable Computing: Informatics and Systems*, 1(1):23–34, 2011. URL `http://www.sciencedirect.com/science/article/pii/S221053791000003X`.

R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984. URL `http://dl.acm.org/citation.cfm?id=358283`.

Y. Wen. Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. 2014.

D. H. Woo and H.-H. S. Lee. Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era. *Computer*, 41(12):24–31, Dec. 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.494. URL `http://dl.acm.org/citation.cfm?id=1495784.1495841`.

R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 84–95. IEEE, 2003. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1206991`.

W. Xu, X. Liu, and Y. Gong. Document clustering based on non-negative matrix factorization. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 267–273. ACM, 2003. URL `http://dl.acm.org/citation.cfm?id=860485`.

T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Measuring Similarity of Large Software Systems Based on Source Code Correspondence. In F. Bomarius and S. Komi-Sirvi, editors, *Product Focused Software Process Improvement*, volume 3547 of *Lecture Notes in Computer Science*, pages 179–208. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-26200-8. URL `http://dx.doi.org/10.1007/11497455_41`.

C.-Y. Yang, J.-J. Chen, and T.-W. Kuo. An approximation algorithm for energy-efficient scheduling on a chip multiprocessor. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 468–473. IEEE Computer Society, 2005. URL `http://dl.acm.org/citation.cfm?id=1049151`.

F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 374–382. IEEE, 1995. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=492493`.

L. Ye, G. Lu, S. Kumar, C. Gniady, and J. H. Hartman. Energy-efficient storage in virtual machine environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, pages 75–84, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-910-7. doi: http://doi.acm.org/10.1145/1735997.1736009. URL `http://doi.acm.org/10.1145/1735997.1736009`.

H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: managing energy as a first class operating system resource. *SIGOPS Oper. Syst. Rev.*, 36(5):123–132, Oct. 2002. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/635508.605411. URL `http://doi.acm.org/10.1145/635508.605411`.

Y. Zhang, Y. Hu, B. Li, and L. Peng. Performance and power analysis of ATI GPU: A statistical approach. In *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pages 149–158. IEEE, 2011. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6005434`.