Analysis, Modeling and Optimization of Execution Time and Energy for Parallel Programs

PhD thesis in Computer Science

by

Philipp Gschwandtner

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of doctor of philosophy

advisor: Prof. Dr. Thomas Fahringer, Institute of Computer Science

Innsbruck, February 20, 2017

Certificate of Authorship and Originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Philipp Gschwandtner, Innsbruck, February 20, 2017

Abstract

Traditionally, high performance computing (HPC) considered execution time as the most important objective for optimizing parallel programs. However, the power wall and also the subsequent rise of multi- and many-core designs forced the scientific community and industry to shift their focus towards additional concerns, such as energy consumption, power consumption or thermal budgets. Nowadays, these concerns often pose the limiting factor when designing faster hardware or optimizing software.

Contrary to much related work in this field, targeting hardware design, scheduling, or resource management, the research presented in this thesis investigates several non-functional parameters (including execution time and energy consumption) from a compiler perspective. Employing the unique capabilities of a compiler, it tackles three research use cases. First, we examine the conflicting nature of three non-functional parameters in the context of multi-objective auto-tuning. Second, we investigate the concept of code significance and how it can be leveraged to drive program execution on unreliable hardware in order to reduce energy consumption. Third, we discuss the benefits of compiler-assisted predictive models with regard to overhead reduction.

To investigate these open research problems, a hardware and software model suitable for compilers is established in this doctoral thesis, enriched with key properties with respect to the aforementioned tasks. Implemented and integrated into the Insieme compiler and runtime system framework, it supports the automatic identification of target code regions, enables the specification of key properties, extensible by additional concepts such as user-defined metrics.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Dr. Thomas Fahringer for his continuous support of my Ph.D study and research, for his patience, motivation, knowledge and experience. His guidance helped me in all the time of research and writing of this thesis. Additionally, I would also like to thank my second supervisor Prof. Dr. Sabine Schindler for her support and the opportunity to participate in the Doctoral School Computational Interdisciplinary Modelling, the external reviewers for their valued perspective on my work, and the rest of the thesis committee.

I also thank Dr. Juan Durillo for his highly valued assistance in major parts of my research, and Dr. Radu Prodan for his insightful comments and experience. Additionally, I would like to express my gratitude to both Dr. Dimitrios Nikolopoulos and Dr. Bernd Mohr for their highly esteemed guidance, support, and access to research facilities abroad — they helped me broaden my horizon in science, work, and also culture.

My sincere thanks also go to all my fellow students and colleagues, most notably Ferdinando Alessi, Luis Ayuso, Alex Hirsch, Bernhard Höckner, Matthias Janetschek, Herbert Jordan PhD, Klaus Kofler, Stefan Moosbrugger, and Peter Zangerl. They all contributed to our common projects and goals, and we shared a great many productive and entertaining discussions about work-related topics and also virtually everything else that came to mind during our time in the office. I particularly thank Dr. Peter Thoman for our fruitful and fun discussions in the office, my car, and on the trail, and for his and Herbert's demands for high-quality work.

Finally, I would like to thank both my parents, my sister, and Lisa for supporting me in all my endeavors. They encouraged me to pursue my dreams throughout all these years, and continue to do so.

Contents

Chapter 1

Introduction

1.1 Motivation

Originally, research in the field of hardware design and software engineering mainly targeted high computational performance, the reduction of execution time being the chief concern of processor architects and software developers. However, increasing clock frequencies imposed a limit regarding sequential speed improvement, as processor designs of this era hit a power wall that made any further advances in this direction technically infeasible. This marks an important occurrence of increased non-functional interests in high performance computing, as additional concerns such as power consumption, energy costs and heat dissipation started to pose limiting factors in new hardware designs. Researchers and engineers shifted to multi-core and later many-core computing as the means to further increase performance partly due to these constraints. While this paradigm change toward increased parallelism offers large potential, providing high performance at reduced power, energy and heat expenses, these non-functional concerns are still the main design limitation of today's architectures [48], and consequently also software design and optimization. Also, they opened up new branches of research that tackle previously non-existent problems or exploit new potential. These non-functional concerns, specifically analysis and optimization of energy in addition to execution time, represent the focus of the research presented in this thesis.

The original first-class citizen among non-functional parameters is high performance or short execution time, and it has been targeted by various branches of computer science research since the beginning of high performance computing. The increasing memory gap [139], which is the time delay for a memory access to be completed, has led to memory hierarchies composed of caches. Optimizing their

usage gave rise to many fields of research, with examples in the area of source code transformations such as loop tiling [23], optimizing data cache usage, or program repositioning [85], which aims at reducing instruction cache misses. Similar endeavors followed every other new development in hardware design.

However, in the past two decades, also power and energy have become increasingly important entities, evident by a growing amount of literature dating back to the 1990s dealing with power modeling [22] and simulation [9], or energy modeling [108], and also a plethora of optimization techniques addressing power and energy. Although not as long-established as performance optimization, energy and especially power represent the main constraint when designing new hardware. This entailed technologies such as dynamic voltage and frequency scaling (DVFS), one of the most ubiquitous means of providing a trade-off between execution time and power or energy, and more recently Turbo Boost [25]. The importance of power (and also energy in part) has been further aggravated by recent events such as the 20 megawatt power limit [131]. It states that supercomputing centers must not consume more than 20 MW of power in order to be feasible with regard to infrastructure costs. Combined with the strive for exascale computing, a redoubling of efforts is required to achieve these goals.

1.2 State of the Art

There is an abundance of related work regarding performance, power, energy, or other non-functional parameters in the field of HPC, investigating instrumentation, measurement, analysis, modeling, and optimization. However, this section will focus on the most recent work directly related to the research presented in this thesis in Chapters 4 to 6. For clarity, only an outline is given here, with more detailed discussions of related work presented in the respective section of the individual chapters.

1.2.1 Multi-objective Auto-Tuning

Despite the rising of new concerns such as power and energy over the past two decades, high performance has always remained among the most important goals of research in HPC hardware and software optimization. In order to address this presence of multiple optimization goals, a lot of related work evaluates their new optimization methods in a multi-objective manner. Examples include hardware design exploration [94], self-tuning libraries for linear algebra [135, 137] or signal processing [42, 100], or application-specific auto-tuning [4, 30].

Open Problems

Many works that address multi-objective problems assign fixed weights to these objectives, effectively reducing the problem to a mono-objective one and masking any potential trade-offs between conflicting objectives. The energy-delay-product (EDP) [48] is among the most popular of such fixed-weight functions. While, over time, additional metrics were established that balance the importance of the individual objectives, they fail at capturing the true trade-off between conflicting objectives but rather sample this trade-off at pre-defined points. This results in limited flexibility in case the preference regarding the objectives changes. Hence, very few works apply true multi-objective optimization. Furthermore, there are additional objectives such as resource usage [65] that might need to be considered besides execution time and energy. However, at this time, there are no works in HPC that consider more than two objectives.

For this reason, we present a true multi-objective auto-tuner in Chapter 4, published in [53], that optimizes parallel programs for three conflicting objectives, execution time, resource usage and energy consumption. Additionally, based on a combined compiler-runtime framework, our method allows for both compile-time parameters (such as source code transformations) and run-time parameters (such as degree of parallelism and DVFS) to be combined in the same optimization process, whereas many works are limited to one of the two. Finally, we obtain generalized guidelines that can aid in tuning parallel programs for these three conflicting objectives.

1.2.2 Significance-driven Optimization of Code Execution

One of the results of the industry's shift towards including energy and power concerns is the development of efficient tools such as DVFS. It provides the means to manage the trade-off between time and power or energy by reducing the frequency and voltage. However, DVFS as deployed throughout most of today's commodity hardware is limited to operating well above the threshold voltage that is required for transistors to function reliably. On the other hand, the concept of near-threshold voltage (NTV) operates hardware closer to this threshold voltage, offering much larger power reductions at the expense of performance. Nevertheless, while NTV has been a topic of interest for some time [69], its entailed higher probabilities of hardware faults have prevented its wide-spread deployment in HPC beyond research projects. However, popular goals such as exascale performance at 20 megawatts might depend on such innovative methods. As a result, the scientific community has studied hardware and software mechanisms that render programs fault-resilient [34, 68, 69, 70]. Also, there

is research investigating the effect of faults on software [35] but it ignores the impact of fault recovery on energy or how to leverage fault resilience for energy reduction. Also, very few works deal with partial protection schemes [78].

Open Problems

Although there is research in both NTV operation and the effect of faults on codes in the context of HPC, few works combine these aspects. Those that do employ protection mechanisms that are explicitly designed to cope with the unreliability of hardware when operated at NTV. Contrary to that, we investigate the effect of this unreliability on unprotected codes in the context of reducing energy consumption in Chapter 5, published in [52]. Specifically, we examine the behavior of iterative solvers that naturally converge to solutions. We aim at reducing overall energy consumption by reducing power via NTV and mitigating the performance impact via parallelism. Moreover, we investigate partial protection mechanisms by evaluating the significance of individual parts of the target program and its data.

1.2.3 Compiler-assisted Execution Time and Energy Modeling

A lot of literature dealing with non-functional parameters investigates their predictive modeling. This is often required by optimization methods that open large parameter search spaces to be explored. For this reason, many works tend to employ predictive models that aid in the navigation of these search spaces. Establishing a basis, there are many analytical works that attempt to model non-functional parameters or their relationships. Prominent examples are the roofline model for execution time [140] and energy [26], Wattch [22] or ECM [115]. Then, a vast number of predictive models is built on top of this basis. These include models requiring user input [18, 113, 117], as well as automatic approaches [19, 55, 77]. While most of them aim at performance prediction, some also target power or energy prediction, and many stochastic approaches support additional non-functional parameters.

Open Problems

The majority of predictive methods uses run-time information such as performance counters to generate or train their models. Often, a parameter study involving multiple search space samples is required for example to obtain enough support nodes for function fitting. Hence, these methods require multiple target program executions. However, static information can help in this process and reduce the number of samples required for achieving a desired level of accuracy. Few works

incorporate static information, nevertheless they still partially rely on stochastic methods that require multiple target program executions [19] or employ the user to provide key information regarding important program parameters. Furthermore, many of the models found in literature are problem-size or machine-size specific and need to be re-trained if these parameters change.

Contrary to these works, in Chapter 6, we investigate a modeling approach based on the capabilities of a compiler that employs a single execution of distributed memory parallel programs for model generation. We show that using a compiler's data flow analysis, we can obtain models parametrized for both problem size and machine topology without depending on user directives. In addition, we show how automatic source code transformations can be used to reduce the overhead of model generation. Finally, we build and evaluate these models for both execution time and energy.

1.3 Organization

This thesis is structured as follows. Chapter 2 introduces a formal model that describes the hardware entities, software entities and their relationship this research is based on. The realization of this formal model in a practical non-functional instrumentation and measurement framework within the Insieme Compiler is described in Chapter 3. Thereafter, Chapters 4 to 6 present some of the research based on this model.

First, Chapter 4 discusses auto-tuning trade-offs of parallel programs between execution time, resource usage, and energy consumption in a multi-objective context using iterative compilation. Second, Chapter 5 introduces the notion of code significance, how it can be attributed to code regions of parallel programs and their data, and investigates whether code significance can drive code execution on unreliable hardware. Third, Chapter 6 shows how compiler knowledge can facilitate the generation of analytical models for execution time and energy prediction and reduce their training overhead. Finally, Chapter 7 provides a conclusion to the research done thus far and lists potential future work to be explored.

Chapter 2

Model

In order to model the power, energy and time behavior of hardware and software in the context of today's high performance computing systems and applications, a model regarding their behavior and technical background with respect to the physics involved is required. Section 2.1 will introduce the hardware model on which the remainder of the thesis is based. Subsequently, Section 2.2 will establish the corresponding software model used to represent the programs targeted in this work.

This chapter is based on work on performance and energy benchmarking and analysis, published by under the titles *Performance Analysis and Benchmarking of the Intel SCC*, see [50], and *Modeling CPU Energy Consumption of HPC Applications on the IBM POWER7*, see [51].

2.1 Hardware Model

The major non-functional parameters of interest in this work are the execution time and energy consumption of parallel programs. However, they are actually properties of the hardware that executes these programs. For this reason, we require a hardware model. First, we describe the physical relationship between computer hardware, time, power and energy consumption. Second, a hardware topology model is defined that describes the type of hardware components, their connections and semantics. Finally, a list of key properties is established with respect to the goals of this thesis.

2.1.1 Physics Background

In this section, a brief overview of the relationship between time, power and energy is presented within the context of computer hardware.

Energy E is defined as the integral of power over time, i.e.

$$E = \int_0^t P_{\text{inst}} \, \mathrm{d}t \tag{2.1}$$

where P_{inst} is the instantaneous power. However, for several reasons such as the highly dynamic nature of power and measurement limitations, etc, instantaneous power is usually difficult to work with in practical contexts. Instead, Equation (2.1) is often numerically approximated by using average power P_{avg} instead:

$$E = t \cdot P_{\text{avg}} \tag{2.2}$$

For brevity, throughout this thesis, P always denotes P_{avg} over a given interval, unless specified otherwise. The majority of today's high performance computing hardware comprises up to billions of transistors [12]. These transistors and their interconnects are the main causers for execution time, power dissipation and hence energy consumption[1]. For any piece of transistor hardware, P can be divided into its static and dynamic component

$$P = P_{\text{static}} + P_{\text{dynamic}} \tag{2.3}$$

where P_{static} denotes the static power drawn by the hardware without doing any actual work (e.g. no clock signal, no state changes, etc.) and P_{dynamic} denotes the dynamic power drawn by the transistors processing the actual workload. These terms are affected by parameters such as fabrication process size (e.g. 22 nanometers [29]), material choice and quality, supply voltage, temperature, and others, and may differ between multiple samples of the same integrated circuit.

Since P_{static} is a static component and not affected by the workload, we focus on P_{dynamic}, which can be further broken down into

$$P_{\text{dynamic}} = C \cdot V^2 \cdot F \cdot \alpha \tag{2.4}$$

where C is the electrical capacitance (a fixed property for a given component), V is the supply voltage, F is the clock frequency and α commonly known as the *switching factor* or *activity factor* [22], with $0 \leq \alpha \leq 1$. This switching factor denotes how often transistors switch upon a clock signal, with 0 meaning that transistors never

[1] Note that according to the first law of thermodynamics, energy can never be actually destroyed in an isolated system but only converted from one form to another (e.g. electric into thermal). Nevertheless, for brevity, the term "energy consumption" is used synonymously in this thesis to denote the conversion and dissipation of electric energy as waste heat.

switch and 1 meaning they switch at every clock signal. The term α represents a major impact software has on the power and hence also on the energy consumption of hardware. Normal workloads in high performance computing will cause moderate switching factors, and some works simply assume e.g. $\alpha = 0.5$.[22]

Equation (2.4) also evidently shows that the clock frequency F only affects the dynamic power consumption linearly, whereas the voltage appears as a power of 2 and hence has quadratic impact on P_{dynamic}. However, since higher clock frequencies imply shorter switching times as mentioned above, F and V are tightly coupled to a certain degree and changed in unison to ensure stable operation. Literature sometimes coarsens this to a term of power of 3 and refers to this as the "cube root rule" [14], where the power consumption is a cubic function of the speed of computer hardware.

Transistors work on the well-known basis of changing the current between one pair of terminals by applying voltage or current to another pair of terminals. This is commonly known as switching *on* or *off*. To operate, a certain minimum V, commonly referred to as the *threshold voltage* [47] or V_{TH}, is required at the controlling terminals for the transistor to switch states reliably within a specific time (dictated by the clock rate F), where shorter switching times require higher threshold voltages for reliable operation. Most of today's computer hardware supports operation at varying levels of F and V, with $V_{\text{TH}} << V$ to ensure stable operation.

2.1.2 Topology

The target hardware architecture for this work are distributed and shared memory parallel computers. This encompasses single-chip many-core systems as well as clusters comprised of a large number of multi-core-based nodes. This section defines a hardware topology model that specifies the types of hardware components, how they are connected, what their semantics are, and establishes several key characteristics and relationships among them. A very brief overview of the model is presented at the beginning, with detailed definitions and examples for actual HPC hardware following in the remainder of this section.

In short, a parallel computer is modeled as a directed graph with its vertices representing hardware components of three types (*functional units*, *caches*, *memory units*), with additional components such as *cores* or *nodes* that are obtained via composition. The edges of the graph denote the physical and logical links between these components. This topology model is then enriched in Section 2.1.3 with *non-functional properties* such as clock frequencies that affects the performance of model components with regard to *time* and *power*, and as a result also *energy*. Additionally,

domains are introduced that comprise graph components with combined properties. All parts of this model are focused on the research carried out during the course of this thesis. Hardware details that are omitted have no relevance for this research.

Definition 2.1 ***(Parallel Computer)***

Let $\mathcal{U}$ be a set of vertices that model *hardware units* or *hardware components*, and $\mathcal{L} \subseteq \mathcal{U}^2$ be a set of edges representing unidirectional *links* among them, where $l = (u_1, u_2)$ with $l \in \mathcal{L}$ and $u_1, u_2 \in \mathcal{U}$ denotes a hardware connection that can transfer instructions or data from u_1 to u_2. Then a parallel computer is defined as a connected, directed graph $\mathcal{M} = (\mathcal{U}, \mathcal{L})$. A hardware component $u \in \mathcal{U}$ is defined as an instance of any of the following types:

- Functional Unit: performs computations,
- Cache: caches memory locations, or
- Memory Unit: stores program or data.

These types are denoted by $u_{\text{func}} \in \mathcal{U}_{\text{func}}$, $u_{\text{cache}} \in \mathcal{U}_{\text{cache}}$ and $u_{\text{mem}} \in \mathcal{U}_{\text{mem}}$ respectively, and it is required that $\mathcal{U}_{\text{func}} \neq \emptyset$ and $\mathcal{U}_{\text{mem}} \neq \emptyset$.

As the remainder of this section will show, the three hardware component types of Definition 2.1 are sufficient for modeling distributed memory and shared memory parallel computers. The edges $l \in \mathcal{L}$ linking individual components represent both on-chip interconnects and network connections. While *caches* are not required for a purely functional model, we include them explicitly to model their non-functional effects on program execution. The individual components are further defined as follows.

Definition 2.2 ***(Functional Unit)***

A functional unit is a hardware component that performs actual computations on a specific type of data. It is always linked to at least one memory unit directly, or transitively via one or more caches. A functional unit fetches input data and instructions from memory units or caches, performs the requested computation, and writes output data back to memory units or caches.

Examples of frequently-occurring data types that functional units operate on are

- integer,
- vectorized integer,
- floating-point, or

- vectorized floating-point.

Non-vectorized functional units operating on integer or floating-point data are commonly referred to as *ALU* and *FPU* respectively. Vectorized functional units frequently occur in various HPC architectures (for example the Streaming SIMD Extensions (SSE) [119] in the x86 architecture, the Vector Multimedia Extensions (VMX) and Vector Scalar Extensions (VSX) [20] in the PowerPC architecture, etc.) to hide computation latencies and hence speed up computation throughput.

Definition 2.3 ***(Memory Unit)***

A memory unit is a hardware device in which program or data are stored during program execution. A parallel computer may consist of multiple memory units, some or all of which which may share physical address spaces.

A memory unit (most often RAM) is always managed by a memory controller. However since such a controller is always present, the hardware model presented here implicitly includes such a controller for every memory unit and memory controllers are not defined or listed separately.

Definition 2.4 ***(Cache)***

A cache is defined as a small but fast portion of memory that mirrors a subset of memory locations of memory units for faster access. Caches can be chained, however the ends of such a chain must be connected to a functional unit and a memory unit, respectively. Caches can either hold both program and data (*unified caches*), or be dedicated to only one of the two (*instruction caches* and *data caches*).

Caches are fast, usually on-chip types of memory providing quick access to data residing in memory units (RAM). There are usually multiple levels of caches present in a system, with lower levels holding less data at the benefit of lower latency. The first level (L1) is also typically split into dedicated instruction and data caches. Caches can be private to or shared among multiple execution units and other caches. In cache-coherent systems, complex protocols and policies are employed to ensure data consistency between multiple cores and CPUs, and these protocols and policies can differ between CPU models.

While these definitions are sufficient for modeling the majority of today's HPC hardware, we define a number of aggregated hardware components that denote commonly-occurring compositions of hardware components in HPC hardware.

Definition 2.5 ***(Core)***

A core consists of at least one functional unit and a number of *private caches* $u^{\text{core}}_{\text{cache}} \in \mathcal{U}^{\text{core}}_{\text{cache}}$ with $0 \leq |\mathcal{U}^{\text{core}}_{\text{cache}}|$ and $\mathcal{U}^{\text{core}}_{\text{cache}} \subset \mathcal{U}$. A cache is *private* if it is linked only to functional units or other private caches of the same core.

Definition 2.6 ***(CPU)***

A CPU consists of at least one core, and a number of *shared caches* $u^{\text{cpu}}_{\text{cache}} \in \mathcal{U}^{\text{cpu}}_{\text{cache}}$ with $0 \leq |\mathcal{U}^{\text{cpu}}_{\text{cache}}|$ and $\mathcal{U}^{\text{cpu}}_{\text{cache}} \subset \mathcal{U}$. A cache is *shared* if it is linked to two or more functional units or caches of different cores.

Definition 2.7 ***(Node)***

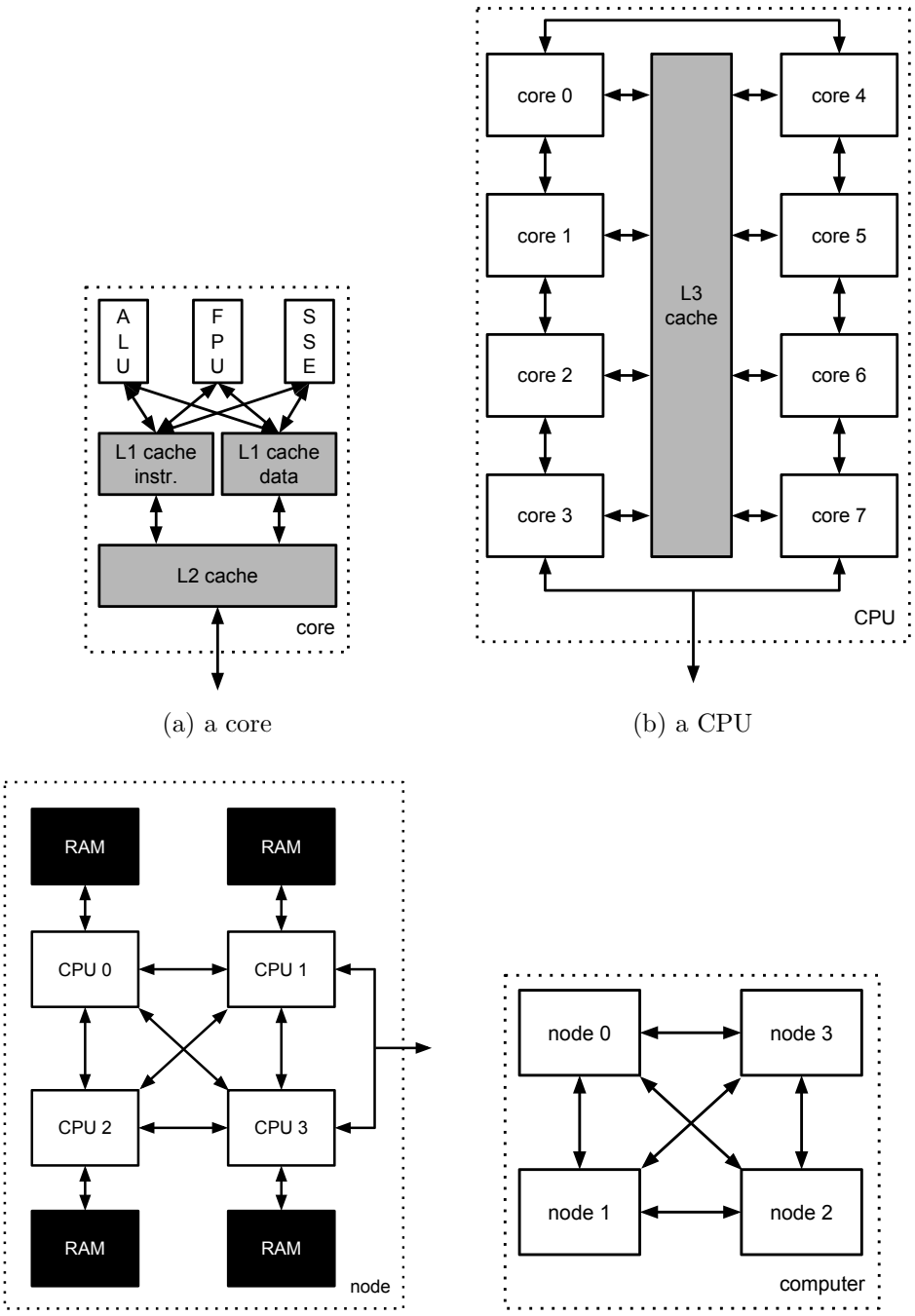
A node consists of at least one CPU and at least one memory unit.

Example 2.1 (Intel Xeon E5-4650)**.** Figure 2.1 illustrates the usage of aggregated hardware components as per Definitions 2.5 to 2.7, depicting a parallel computer consisting of 4 nodes, each holding 4 Intel Xeon E5-4650 [28] processors or CPUs. Each CPU comprises 8 cores and a shared, unified L3 cache. Every core is equipped with an ALU, an FPU and several functional units operating on vector types (for clarity, only one such vector unit for SEE instructions is shown). Additionally, every core holds a private, unified L2 cache and two private L1 caches, one for instructions and one for data.

Example 2.2 (Intel SCC)**.** Similarly, Figure 2.2 illustrates the model representation of the Intel Single-chip Cloud Computer, an experimental prototype with 48 Pentium cores created by Intel Labs [59, 83]. Contrary to most HPC architectures, the individual CPUs of the SCC are connected to their respective memory units via a fast mesh network instead of dedicated memory links and form a distributed memory parallel computer of 48 single-core, single-cpu nodes. The mesh network forwards all data loads/stores between CPUs and memory units, as well as messages passed between the CPUs.

2.1.3 Non-functional Hardware Properties

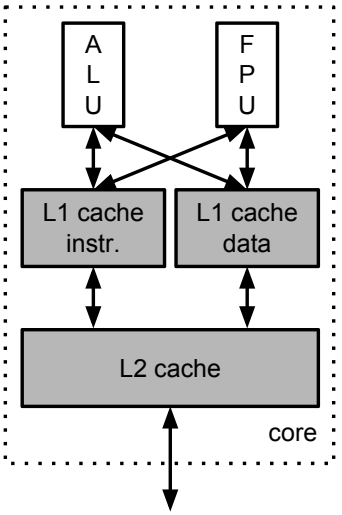
To reason about the performance or energy consumption of hardware, we need to attribute special properties to every hardware component $u \in \mathcal{U}$ and links $l \in \mathcal{L}$. These properties are the individual terms to compute P_{dynamic} of Equation (2.4). We omit C and α since the former is a fixed physical property, whereas α is a term



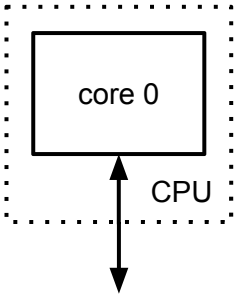
(a) a core (b) a CPU

(c) a node (d) a computer

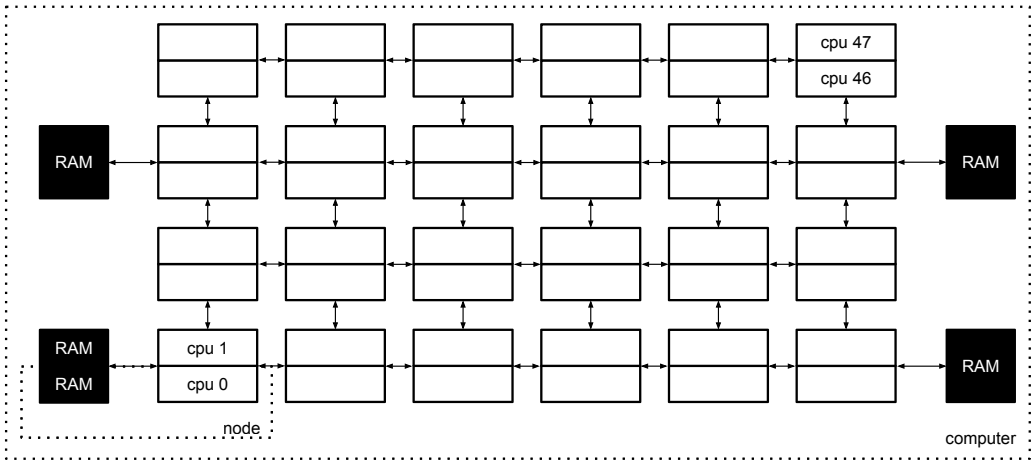
Figure 2.1: Hardware model representation for a parallel computer comprising four nodes each equipped with four Intel Xeon E5-4650 [28] CPUs. For clarity, not all edges are drawn.



(a) a core



(b) a CPU



(c) a node and the entire SCC

Figure 2.2: Hardware model representation for a single Intel Single-chip Cloud Computer (SCC) [59].

affected by the software that is executed. What remains, are the clock frequency F, which affects both performance and energy of a hardware device, and V^2, which has direct impact on its energy. However, since they are commonly changed in unison, V^2 also indirectly affects performance.

Definition 2.8 ***(Power Properties)***

Let $\mathcal{M} = (\mathcal{U}, \mathcal{L})$ be a parallel computer. Furthermore, let $\mathcal{P} \subseteq \mathcal{V} \times \mathcal{F}$ be a set of tuples (v, f), each representing a specific voltage V and a specific clock frequency F as per Equation (2.4). Then there is a $\mathcal{P}_x \neq \emptyset$ for all $x \in \mathcal{U} \cup \mathcal{L}$ that denotes valid operational clock frequency and voltage combinations for hardware component x, and every hardware component is always set to work at a specific (v_x, f_x). The tuples $(v, f) \in \mathcal{P}$ are also called *power states*.

Example 2.3 (Intel SCC)**.** The cores of the Intel SCC support 15 frequency levels that are derived by dividing a base clock frequency of 1600 MHz by divisors 2 through 15. Therefore, $\mathcal{F} = \{\frac{1600}{i} \in \mathbb{Q} | i \in \mathbb{N} \wedge 2 \leq i \leq 16\}$. Furthermore, the SCC cores support 208 voltage levels from 0 V to 1.3 V with a granularity of 6.25 mV. Thus, $\mathcal{V} = \{i \cdot 6.25 \in \mathbb{Q} | 0 \leq i \leq 208\}$. Note that while $\mathcal{V} \times \mathcal{F}$ leads to a total of 3120 possible frequency and voltage combinations, not all of them actually lead to stable system operation. For this reason, $|\mathcal{P}|$ is usually much less than $|\mathcal{V} \times \mathcal{F}|$.

Definition 2.9 ***(Domain)***

A *domain* of a specific non-functional parameter or observable is defined as a set $\mathcal{D} \subseteq \mathcal{U} \cup \mathcal{L}$ of hardware components and links that are all equally and simultaneously affected. Hence, they represent topological aggregations in a specific non-functional context.

Example 2.4 (DVFS Domains)**.** *DVFS domains* are an omnipresent example for non-functional parameter domains in HPC hardware. They encompass hardware components for which the voltage or frequency can only be set in unison, or more formally, $\forall x, y \in \mathcal{D} | (v_x, f_x) = (v_y, f_y)$ for any point in time.

An example of a DVFS domain is present on all Intel Sandy Bridge-EP based CPUs (including the Xeon E5-4650), where $\mathcal{D}_{\mathrm{VF}}$ for a CPU is the set of all cores of this CPU, since all cores adhere to both the same frequency and voltage — in contrast to the Haswell-EP, which supports per-core DVFS settings, leading to multiple $\mathcal{D}_{\mathrm{DVFS}}$ with $|\mathcal{D}_{\mathrm{DVFS}}| = 1$.

The Intel SCC is an example hardware where frequency and voltage domains are separate domains that do not match. For voltage, cores are grouped into 2x4 clusters, each forming a domain $\mathcal{D}_{\mathrm{V},i}$ with $|\mathcal{D}_{\mathrm{V},i}| = 8$ for $0 \leq i \leq 5$. Analogously, for frequency, cores are grouped into 1x2 clusters (also referred to as *tiles*), each representing a domain $\mathcal{D}_{\mathrm{F},j}$ with $|\mathcal{D}_{\mathrm{F},i}| = 2$ for $0 \leq j \leq 23$. As a result, the SCC has 6 domains for which the voltage can be set independently, and 24 domains for which the frequency can be set independently.

Example 2.5 (Measurement Domains)**.** A second example for domains are *measurement domains*. These are domains established by non-functional observables, which cannot be measured for individual hardware components. A prominent example is the domain spanned by the Intel Running Average Power Limit (RAPL) interface [31]. It is a hardware feature available on some Intel CPUs starting with the Sandy Bridge generation [31], that offers both the functionality to measure energy consumption and to set power caps that are not to be exceeded by the processor. For the Xeon E5-4650 model, this RAPL interface does not offer energy consumption readings for individual cores but rather all cores, leading to a domain $\mathcal{D}_{\mathrm{E}}$ holding all 8 cores.

Definition 2.10 ***(Time and Power of Computation)***

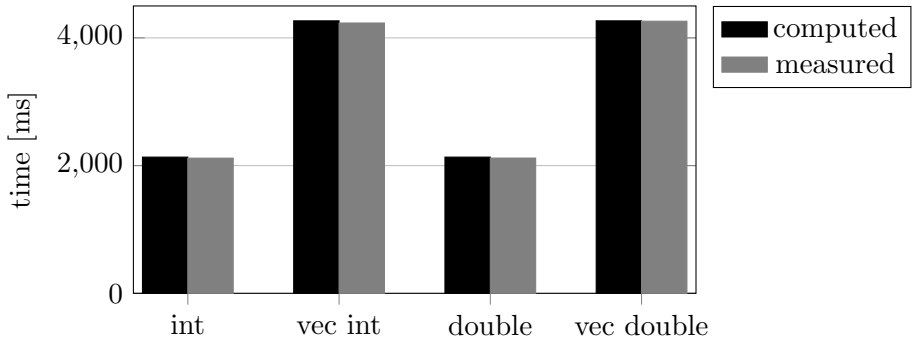
Let $u_{\text{func}} \in \mathcal{U}_{\text{func}}$ be a functional unit for a specific data type d and f the clock frequency this functional unit operates at. Let furthermore $(w_i, a_i) \in \mathbb{R}^2$ be a tuple where w_i denotes the *width* and a_i the average *instructions per clock* (CPI) of a specific instruction type i operating on data type d. The width w_i is the number of data type instances instruction i can simultaneously process, with $0 < w_i$. The average CPI a_i is the number of cycles between two processed instructions i_x and i_{x+1}.

Then the time it takes u_{func} to process i on n_d instances of type d can be approximated as $\frac{n_d}{w_i} \cdot \frac{a_i}{f}$. Similarly, every type of instruction i of width w_i can be associated with a specific average power consumption $P_{u_{\text{func}},i,w_i,v,f}$ of u_{func} at voltage level v and clock frequency f.

Example 2.6 (IBM POWER7 Computation Time)**.** The IBM POWER7 CPU [67] holds functional units capable of processing a fused multiply-add (FMA) instruction on integer and floating-point data types in a vectorized and non-vectorized fashion. Figure 2.3a shows the width and CPI information taken from [111]. The CPU was clocked at $f = 3.0$ GHz, and performed FMA instructions on 10^{11} elements of integer and double-precision floating point types. The CPU can compute 4 elements of integer type and 2 elements of double-precision floating point type per FMA, and requires an average of 0.25 and 0.125 CPI respectively. Setting the variables of Definition 2.10 to these values allows to compute the respective execution times. Figure 2.3b illustrates a comparison with measured values for the same workload.

	integer	double
not vectorized	$(1, 0.125)$	$(1, 0.125)$
vectorized	$(4, 0.125)$	$(2, 0.125)$

(a) Width and CPI in the form of (w, a) for fused multiply-add (FMA) instructions of the IBM POWER7 CPU [111].



(b) Computed and measured time for processing $1e11$ elements of integer and double precision floating-point type in both vectorized and non-vectorized FMA instructions at a clock frequency of $f = 3$ GHz.

Figure 2.3: Application of Definition 2.10 for the IBM POWER7 processor [51].

Definition 2.11 ***(Time and Power of Data Transfers)***

Let a connected, directed graph $\mathcal{M} = (\mathcal{U}, \mathcal{L})$ be a parallel computer. Furthermore, let $\omega : \mathcal{L} \to \mathbb{R}^2$ be a weight function assigning each link $l \in \mathcal{L}$ a tuple $(b, a) \in \mathbb{R}^2$ where b denotes bandwidth and a denotes CPI information. Then $\mathcal{M}$ can be extended to a *weighted*, connected, directed graph $\mathcal{G}' = (\mathcal{U}, \mathcal{L}, \omega)$. Given an amount of data d to be transferred from u_1 to u_2, $u_1, u_2 \in \mathcal{U}$ connected via $l_{u_1,u_2} = (u_1, u_2)$, then the time of transferring this data from u_1 to u_2 can be computed as $\frac{d}{b} + a$ where $(b, a) = \omega(l_{u_1,u_2})$. Similarly, every link l can be associated with a specific average power consumption $P_{l,v,f}$ when transferring data at voltage level v and clock frequency f.

Example 2.7 (SCC Memory and Mesh Network Bandwidth)**.** To show the applicability of Definitions 2.8, 2.9 and 2.11, we compute the average memory throughput of individual cores of the Intel SCC with regard to their physical location as depicted in Figure 2.2c and, therefore, the number of links involved to reach a memory unit. Note that first, the mesh network employs a simple x/y routing scheme, meaning that messages are first routed horizontally and then vertically to reach their destination. Second, the default mapping of memory units to cores follows a symmetrical,

horizontal and vertical bisection of the entire computer into 4 chunks, assigning one memory unit to a set of 12 adjacent cores and ensuring that no node requires more than 4 links to reach a memory unit.

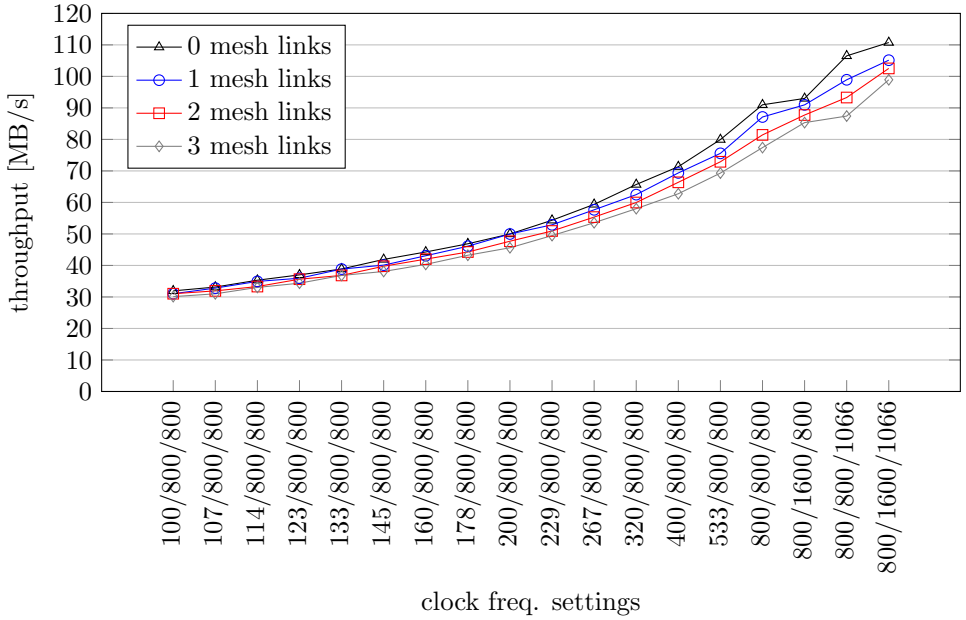
Combining our topology model of the SCC with documented information [74] regarding CPI and bandwidth ((b, a) of Definition 2.11), and clock frequencies ($\mathcal{F}$ of Definition 2.8), we can compute the maximum expected memory throughput. The bandwidth b is 16 bytes, and the CPI a is given as 40 core clock cycles for a core issuing a read or write request, 4 mesh network link clock cycles to forward a request or data, and 46 memory unit clock cycles for the memory unit to complete the request. Therefore, the overall CPI a of a single memory request and the corresponding answer can be computed as

$$a = \frac{40}{f_{\text{core}}} + \frac{4 \cdot 2 \cdot n}{f_{\text{mesh}}} + \frac{46}{f_{\text{memory}}}$$

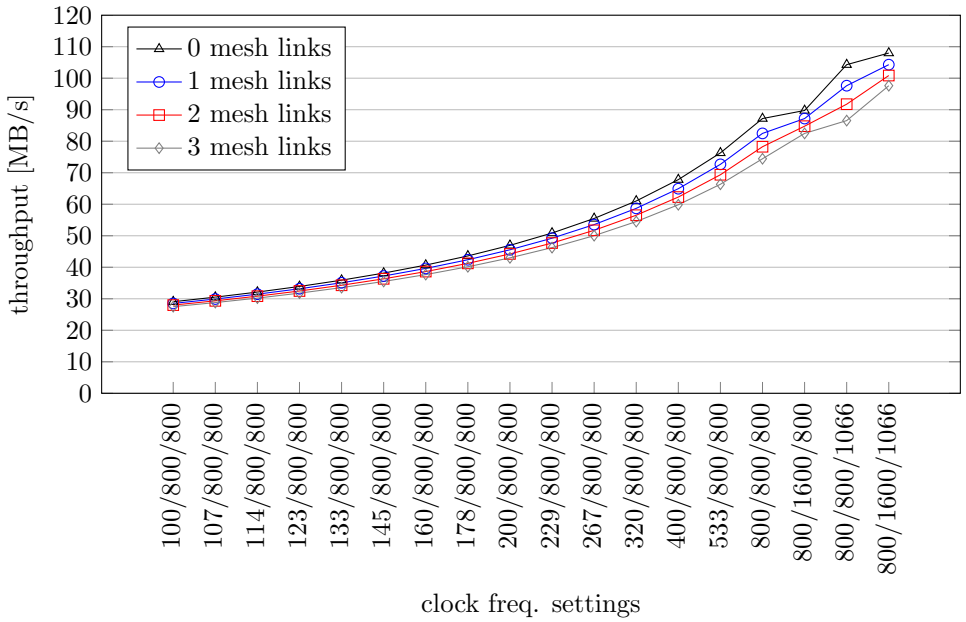
where f_{core}, f_{mesh}, and f_{memory} denote the clock frequencies of the cores, the mesh network links and the link to the memory unit to complete the request, and n denotes the number of mesh network link hops from source to destination. Figure 2.4 shows the computed and measured memory throughput for $0 < n \leq 4$ links involved for a set of core, mesh and memory frequencies. The measured data was obtained by running the stream memory benchmark [84] on an actual Intel SCC sample [50, 73].

Definition 2.12 ***(Cache Access)***

Let u_{func} be a functional unit, u_{cache} a cache unit and u_{mem} a memory unit connected via links $(u_{\text{func}}, u_{\text{cache}})$, $(u_{\text{cache}}, u_{\text{func}})$, $(u_{\text{cache}}, u_{\text{mem}})$, and $(u_{\text{mem}}, u_{\text{cache}})$. Read requests from u_{func} referring to memory locations in u_{mem} are first checked by u_{cache}, and if it holds a copy of the required memory location, its contents are returned immediately — this is called a *cache hit*. If not, it is a *cache miss*, and the contents are transferred from u_{mem} to u_{cache} and returned. Cache properties dictate replacement policies (i.e. which data to evict from the cache when loading new data) and whether write requests are cached or not.



(a) measured data



(b) computed data

Figure 2.4: Measured and computed memory throughput of the SCC for cores with varying distance from the memory units (0 to 3 mesh network links). The x axis entries represent clock frequency settings in the form of ($f_{\text{core}}/f_{\text{mesh}}/f_{\text{memory}}$).

2.2 Software Model

This section defines a software model required for discussing and analyzing non-functional parameters such as execution time or energy consumption in the context of parallel programs. First, the structure of software is defined, starting with *sequential programs*. They are represented as a graph of *statements* with *control flow edges*. This sequential representation is enriched with *parallel control flow edges* and *communication*, resulting in a *parallel program model*. Second, several properties on this software model are established, such as definitions of *code regions* or non-functional parameters or metrics such as *wall time* or *energy consumption*. The definitions of these metrics are vital due to their frequent use throughout the thesis.

2.2.1 Software Structure

Definition 2.13 ***(Sequential Program)***

Let $\mathcal{S}$ be a set of vertices denoting program statements and $\mathcal{E} \subseteq \mathcal{S}^2$ be a set of edges between them representing sequential control-flow. Then a sequential program is a directed graph $\mathcal{A} = (\mathcal{S}, \mathcal{E})$. A control flow edge $(s_1, s_2) \in \mathcal{E}$ with $s_1, s_2 \in \mathcal{S}$ enforces potential execution of s_2 after s_1, i.e. if there are two control edges $(s_1, s_2), (s_1, s_3) \in \mathcal{E}$, either s_2 or s_3 will be executed after s_1.

Example 2.8 (Sequential Program Example)**.** Figure 2.5 shows an example of a sequential program comprising 6 program statements and a number of control flow edges connecting them. Note two back edges, (s_4, s_3) and (s_6, s_2), representing loops.

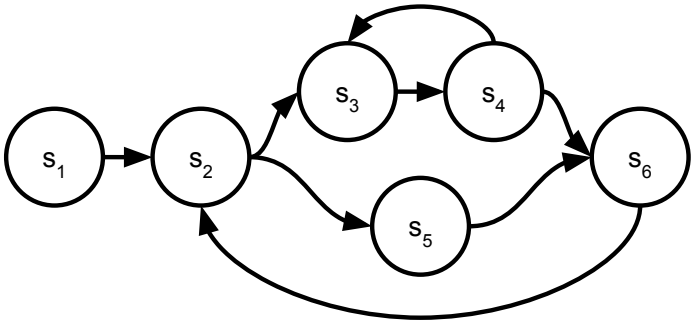


Figure 2.5: Software model representation of a sequential program.

Definition 2.14 ***(Parallel Program)***

Let *spawn* and *merge* be new types of program statements expressing parallelism and $\mathcal{S}_p$ be a set of vertices of these types.
Furthermore, let $\mathcal{E}_p \subseteq \mathcal{S}_p \times \mathcal{S}$ be a set of directed *parallel control flow edges*. For any parallel control flow edge $(s_1, s_2) \in \mathcal{E}_p$ it is required that either s_1 is of type *spawn* or s_2 is of type *merge*. Contrary to sequential control flow edges defined in Definition 2.13, all parallel control flow edges must be followed, i.e. if there are two parallel control flow edges $(s_1, s_2), (s_1, s_3) \in \mathcal{E}_p$ then both s_2 and s_3 are executed concurrently after s_1.
Additionally, let $\mathcal{E}_c \subseteq \mathcal{S}^2$ be a set of directed *communication edges*. Then $\mathcal{A}_p = (\mathcal{S} \cup \mathcal{S}_p, \mathcal{E} \cup \mathcal{E}_p, \mathcal{E}_c)$ represents a parallel program. Let $(s_1, s_2) \in \mathcal{S}$ be a control flow edge and $[s_2, s_3] \in \mathcal{E}_c$ a communication edge, then $[s_2, s_3]$ represents data being transferred from s_2 to s_3, with s_3 being executed after s_1 and completion of the data transfer.

Example 2.9 (Parallel Program Example)**.** Figure 2.6 illustrates an example of a parallel program consisting of 8 statements, including a *spawn* and a *merge* vertex. There are two edges denoting an increase in parallelism, (spawn, s_2) and (spawn, s_4), with the semantics of s_2 and s_4 being executed concurrently after *spawn*. Similarly, there are two edges denoting a decrease in parallelism, (s_3, spawn) and (s_5, merge), implying that s_6 will execute after s_3 and s_5. In addition, there are two communication edges shown, $[s_2, s_4]$ and $[s_5, s_3]$, denoting the transfer of data from s_2 to s_4 and s_5 to s_3 respectively.

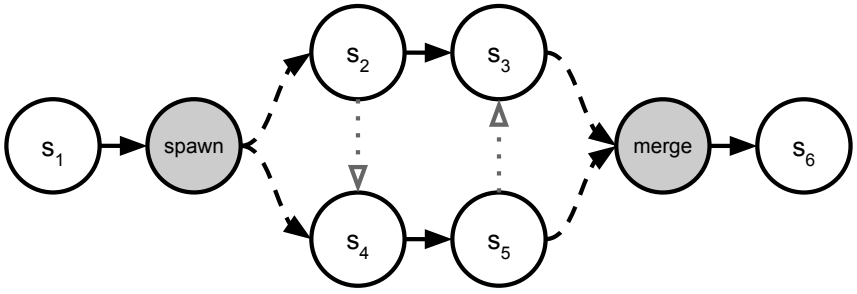


Figure 2.6: Software model representation of a parallel program.

Definition 2.15 ***(Entry and Exit Points)***

Every subgraph $\mathcal{A}' = (\mathcal{S}', \mathcal{E}') \subseteq \mathcal{A} = (\mathcal{S}, \mathcal{E})$ has special sets of vertices

$$\begin{aligned}
\mathcal{E}_e &= \{(s, s'_e) \in \mathcal{E} | s \in \mathcal{S} \setminus \mathcal{S}' \wedge s'_e \in \mathcal{S}'\}, \\
\mathcal{S}'_e &= \{s' \in \mathcal{S}' | (s, s') \in \mathcal{E}_e \wedge s \in \mathcal{S}\}, \\
\mathcal{E}_x &= \{(s'_x, s) \in \mathcal{E} | s \in \mathcal{S} \setminus \mathcal{S}' \wedge s'_x \in \mathcal{S}'\}, \text{ and} \\
\mathcal{S}'_x &= \{s' \in \mathcal{S}' | (s', s) \in \mathcal{E}_x \wedge s \in \mathcal{S}\}.
\end{aligned}$$

Then $\mathcal{S}'_e$ is called the set of *entry vertices* and $\mathcal{S}'_x$ is called the set of *exit vertices*. In the case of $\mathcal{A}' = \mathcal{A}$, $|\mathcal{S}'_e| = 1$.

Definition 2.16 ***(Sequential Code Region)***

A sequential code region is defined as a subgraph $\mathcal{A}' = (\mathcal{S}', \mathcal{E}') \subseteq \mathcal{A} = (\mathcal{S}, \mathcal{E})$ with exactly one *entry vertex* $\{s'_e\} = \mathcal{S}'_e$ and exactly one *exit vertex*, $\{s'_x\} = \mathcal{S}'_x$. This property is also called *single-entry single-exit*.

Definition 2.17 ***(Parallel Code Region)***

A parallel code region is defined as a subgraph $\mathcal{A}' = (\mathcal{S}', \mathcal{E}') \subseteq \mathcal{A} = (\mathcal{S} \cup \mathcal{S}_p, \mathcal{E} \cup \mathcal{E}_p)$ in which statements can be executed concurrently by different cores (see Definition 2.5).

2.2.2 Non-functional Software Properties

Definition 2.18 ***(Execution and Data Transfer Workloads)***

Let $\mathcal{S}$ be a set of vertices of a program. Furthermore, let n_i be the number of instances of instruction type i processed by a hardware unit u_{func}. Then each $s \in \mathcal{S}$ can be attributed with $n_{s,i}$ reflecting the workload s exerts on u_{func}.
Analogously, let n_d be the number of instances of data type d processed by hardware unit u_{func} and stored in a memory unit u_{mem} connected to u_{func} (potentially cached by several u_{cache} inbetween). Then each $s \in \mathcal{S}$ can be attributed with $n_{s,i}$ reflecting the amount of data that needs to be transferred to u_{func}.

Definition 2.19 ***(Degree of Parallelism)***

Let $r_i \in \mathcal{R}$ be a parallel code region with $i \in \mathbb{N}^+$. Then $|\mathcal{R}|$ denotes the *degree of parallelism*.

Definition 2.20 ***(Performance Metrics of Code Regions)***

Let $\mathcal{R}$ be a parallel code region, and let τ_{start} and τ_{end} be functions that return the entry and exit times of each $r \in \mathcal{R}$. Then the following definitions of metrics can be established:

- *walltime*:
$$t_{\text{wall}} = \max_{r \in \mathcal{R}}(\tau_{\text{end}}(r)) - \min_{r \in \mathcal{R}}(\tau_{\text{start}}(r))$$
- *resource usage* or *cpu time*:
$$t_{\text{cpu}} = \sum_{r \in \mathcal{R}} (\tau_{\text{end}}(r) - \tau_{\text{start}}(r))$$
- *speedup*:
$$\sigma = \frac{t_{\text{seq}}}{t_{\text{wall}}}$$
where t_{seq} denotes the execution time of a sequential code region counterpart of $\mathcal{R}$
- *efficiency*:
$$\epsilon = \frac{\sigma}{|\mathcal{R}|}$$

Definition 2.21 ***(Power and Energy Metrics of Code Regions)***

Let $\mathcal{R}$ be a parallel code region and let $\phi_{x,p}$ be a function that returns the average power consumption of a hardware unit or link $x \in \mathcal{U} \cup \mathcal{L}$ operating at power state $p \in \mathcal{P}_x$ when processing a statement $s \in \mathcal{R}$. Then the following definitions of metrics can be established:

- *average power*:
$$P_{x,p,\text{avg}} = \sum_{s \in \mathcal{R}} \left(\frac{\phi_{x,p}(s)}{|\mathcal{R}|} \right)$$
$$P_{\text{avg}} = \sum_{x \in \mathcal{U} \cup \mathcal{L}} P_{x,p,\text{avg}}$$
- *energy*:
$$E = P_{\text{avg}} \cdot t_{\text{wall}}$$

2.3 Summary

In this chapter a hardware and software model were established in order to discuss and analyze non-functional parameters such as execution time or energy consumption in the context of parallel programs. First, the physical relationship between computer hardware, time, power and energy consumption were described. Then, a hardware topology model was defined that describes the type of hardware components, their connections and semantics. Additionally, a list of key properties was established with respect to the goals of this thesis. Subsequently, a software model was defined, describing the structure of target programs within the scope of this thesis. Finally, several vital properties such as code regions and metrics such as wall time and energy consumption were presented.

Chapter 3

Insieme Measurement Framework

All work presented in this thesis has been built and integrated in the Insieme compiler and runtime system framework [65], or represents groundwork for it. Hence, the purpose of this chapter is to elaborate on the *instrumentation and measurement* components of Insieme, developed as part of this thesis for the presented work. Nevertheless, this chapter also briefly touches on many of Insieme's other components and capabilities, developed and maintained by the entire Insieme developer team [92], and the use cases of compiling, analyzing and optimizing parallel applications.

The source code for the Insieme compiler and runtime system, including the instrumentation and measurement components described in this chapter, are available online [118].

Insieme consists of two tightly integrated main components, namely the *Insieme compiler* and the *Insieme Runtime System (IRS)*. While, in theory, both can be utilized independently, most work —also beyond the scope of this thesis— is carried out by using both in a joint fashion. One main reason for this is their high degree of interoperability by means of transferring information from the compiler to the runtime system and vice versa. Due to the tight integration of both compiler and runtime system, the instrumentation and measurement framework is distributed among both of these components to provide non-functional program information. The compiler is responsible for instrumenting the program, issuing measurements, and retrieving results. The runtime system provides an execution environment for the program and performs the actual low-level measurements, offering non-functional information to the user or compiler for further use. Both of these aspects are addressed individually, first the compiler in Section 3.1 and second the runtime system in Section 3.2.

3.1 Compiler Component

This section outlines the compiler component of the measurement framework, focusing on the compiler's view of program *regions*, *metrics* and how the compiler conducts measurements using *executors* and retrieves non-functional run-time data.

3.1.1 Overview

The Insieme compiler comprises multiple components as sketched by Figure 3.1. Source code — be it C or C++, enriched with parallel constructs such as OpenMP pragmas or MPI communication primitives — is parsed by the *frontend* and converted into *INSPIRE* or *IR*, the unified intermediate representation of Insieme [64]. This intermediate representation of the input program allows analyses and transformations to be applied for a large number of use cases ranging from simple code region identification to complex loop transformations [53] or task optimizations [122]. The IR is then read by the *backend*, which again synthesizes C or C++ source code that is to be compiled by a backend compiler such as GCC [39]. It should be noted that Insieme offers multiple backends for code generation, however, only the *runtime backend* generates code that conforms to the application model of the runtime system and using it is mandatory for obtaining measurements with the measurement framework described in this chapter.

The process described above can be seen as a single invocation of the compiler, however all of the aforementioned components are highly modularized and can be applied in many ways (e.g. multiple invocations of the backend with differently transformed IR instances). For this reason, the instrumentation and measurement framework is highly modularized as well, offering a number of individual entities as follows:

- region specification,
- region identification,
- region instrumentation,
- metric specification,
- facilities for compiling and executing instrumented programs to obtain measurements, and
- parsing result data.

These components are described in detail throughout the remainder of this section.

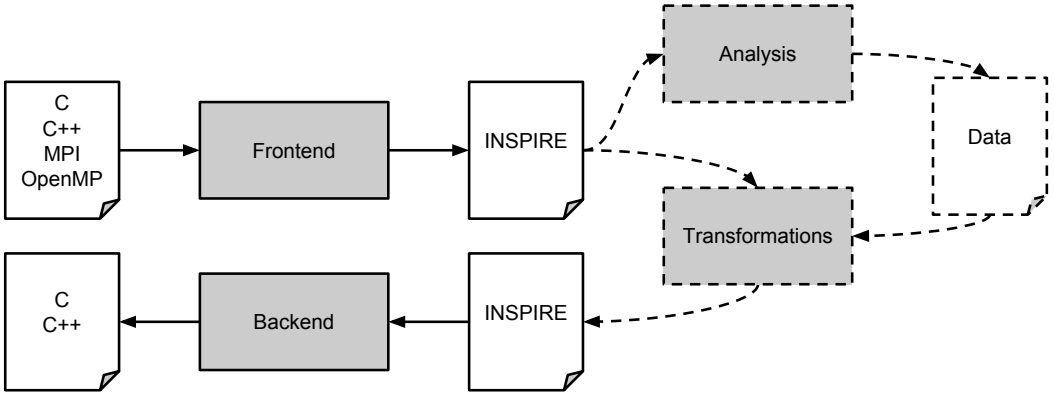


Figure 3.1: Insieme component interaction.

3.1.2 Region Specification, Identification and Instrumentation

Code regions from the software model point of view have been defined in Definitions 2.16 and 2.17. Within the compiler, they consist of a pair of INSPIRE statement addresses representing s_e and s_x, the entry and exit statements of the code region, as well as a linear index. A single statement address can also be instrumented, in which case $s_e = s_x$. Starting and ending a region via s_e and s_x is also referred to as *opening* and *closing* a region, and regions are called *open* if their entry statement has been executed but not the exit statement. Regions can be arbitrarily nested as long as an open region is not re-opened, and every exit statement must close the region that was most recently opened.

Example 3.1 (Compiler Regions)**.** Figure 3.2 illustrates an example of valid code regions within INSPIRE. Both R1 and R2 are cases where s_e and s_x refer to the same statement address, in this case a for statement. R3 illustrates the case of $s_e \neq s_x$.

```
1 for(x = 0..N:1) {
2   for(y = 0..M:1) {
3     counter += element[y];
4   }
5   sum[x] = counter;
6   counter = 0;
7 }
```

R2 (lines 2–4), R3 (lines 5–6), R1 (lines 1–7)

Figure 3.2: Example region identification for 3 regions: two loops (R1 and R2) and two assignments (R3).

There are three main ways of identifying code regions to be instrumented by the compiler:

Pragmas The Insieme compiler offers pragmas that direct the compiler to consider the following statement a code region. The syntax for such pragmas is `#pragma insieme region(id)` where *id* is the index of the region. The user is required to ensure continuous indices with $0 \leq \text{id} < N$ where N is the total number of regions. The same index may appear multiple times for user-defined aggregation, however the user must also ensure that a region is not nested in another with the same index. Also, this method is limited to $s_e = s_x$.

Analysis There are a number of utilities available within the compiler to identify code regions that match certain properties. These are called *region selectors*. Examples are specific statements such as `for` or `pfor` loops, function calls with specific signatures, or more complex regions such as covering non-blocking MPI communication and corresponding wait calls, or arbitrary region selection that matches a given size.

Program The Insieme compiler can be instructed to consider the entire program a code region.

Since the actual measurement of non-functional parameters takes place at runtime, any regions of interest must be marked for the runtime system. To that end, the instrumentation framework prepends an Insieme Runtime System directive `ir_inst_region_start(id)` to every start address (s_e) and appends the corresponding directive `ir_inst_region_end(id)` to every end address (s_x). These directives instruct the runtime system to perform measurements, as further detailed in Section 3.2.

3.1.3 Measurement Framework

The measurement framework is responsible for collecting a list of all metrics requested by the compiler (or the user), compiling the target program, conducting the actual measurement and parsing the measured data.

Metrics

Metrics are represented by a syntax tree structure, to allow the specification of user-defined metrics based on low-level metrics. Each user-requested metric, such as energy consumption, is denoted by the root node of such a tree, with its children consisting of aggregation operators of low-level metrics that represent the leaves.

Low-level metrics can be defined arbitrarily by the user, however each requires an implementation in the runtime system to collect data for it. Available aggregation operators include *addition*, *subtraction*, *multiplication*, *division*, *minimum*, *maximum*, and *average*. Additionally, for convenience, there is a *none* operator for specifying metrics that do not require any aggregation. Example 3.2 show examples of metrics using the *division* operator.

Example 3.2 (Compiler Metric Structure)**.**

$$\text{total\_energy} = \text{div}(\text{average\_power}, \text{wall\_time}) \tag{3.1}$$

$$\text{efficiency} = \text{div}(\text{div}(\text{cpu\_time}, \text{wall\_time}), \text{num\_workers}) \tag{3.2}$$

When specifying composed metrics for measurement, the compiler automatically resolves their leaves and composes a list of them to be forwarded to the runtime system. While doing so, the list of leaves is checked for special external instrumentation requirements, such as linking external libraries (e.g. the PAPI [88] library for hardware counter information) or ensuring specific system-level register access (e.g. Intel Running Average Power Limit (RAPL) [136, 54, 31]). This minimizes the software footprint and number of system privileges required to conduct measurements.

Executors

Once the metric leaves and the external instrumentation requirements have been identified, the program is compiled with a backend compiler such as GCC [39]. This binary is then handed to one of Insieme's *executors* along with the metric leaves and external instrumentation requirements. These executors are responsible for managing the program execution, i.e. transfer of the binary if necessary to a remote target system, executing it in a given environment with given program parameters, and transferring measurement result files back to the machine running the compiler. There are several executors available, including

local executor executes the binary on the local system (e.g. the one running the compiler),

SSH executor executes the binary on a remote system via ssh and ensures proper moving of files to and from the remote machine, and

MPI executor executes the binary on a remote system with additional mpi-specific settings such as target hosts, number of ranks, etc. .

When the runtime system executes and measures the program, it creates performance files that hold the results of the respective measurements (see Section 3.2). These performance logs are copied back by the executor if required, and then parsed by the compiler. The compiler then computes the originally requested, composed metrics.

3.2 Runtime Component

This section describes the Insieme Runtime System (IRS), with special attention paid to the *region-based* measurement component. It will elaborate on the specification of *metrics* and *metric groups* within the IRS and shows their importance in providing the means for *user-defined metrics* and *metric aggregation*.

3.2.1 Overview

The IRS [121] provides a shared memory parallel application model environment that implements INSPIRE semantics. It can execute INSPIRE programs that were synthesized by the *runtime backend* of the Insieme compiler. In order to adhere to the IRS application model, programs or program parts are encoded into *work items* which represent schedulable entities. These work items are executed by *workers*, which represent software threads that are user-managed from the perspective of the operating system. Workers form *work groups* that allow limiting synchronization in parallel contexts.

The IRS is implemented in C99 with selected component parts written in low-level assembler for more direct control and reduction of overhead (e.g. hardware timestamp retrieval or context switches). The workers are implemented using pthreads [89], however an abstraction layer allows the usage of alternative threading libraries such as Windows threads [138]. These abstraction layers also allow providing alternative implementations with respect to different instruction set architectures such as x86 or PowerPC.

3.2.2 Measurement Framework

There are two independent components in the IRS that provide measurement features:

region-based This component is used for measuring code regions that were identified and marked by the compiler or the user. It is capable of measuring non-functional data for both sequential and parallel regions identified via the

directives `ir_inst_region_start(id)` and `ir_inst_region_end(id)` as described in Section 3.1.2.

event-based This component is mostly used for functional and performance debugging of the IRS itself. It can record the time and origin of several types of events, such as a work item starting execution, a worker going to sleep due to a lack of work, or a work group completing a barrier.

Since the region-based component is much more often used in non-functional analysis and optimization than the event-based one, this section will further detail on the features and characteristics of the former.

The region-based instrumentation and measurement component of the IRS is designed to provide low-overhead data collection from a wide range of sources such as external libraries or hardware registers. To that end, the data collection within the IRS is centered around two main entities: *metrics* and *metric groups*. The former represents individual metrics such as *wall time* or *energy consumption*, while the latter represents a grouping of metrics that share common instrumentation or measurement code, usually because they are obtained from the same source (e.g. the same external library). The specifications of metrics and metric groups are subsequently described, with an illustration of their usage following in Algorithm 3.1.

Metrics

Contrary to the compiler, the IRS does not require the compiler's tree structure for representing metrics, since it only measures the metric leaves of this tree structure. For this reason, the IRS instead holds a list of metrics, all adhering to a common, generic specification. This generic specification allows new metrics to be implemented solely by providing their specification, and removes the need for modifications in components of the IRS other than the measurement component. The specification for metrics includes the following items:

name The name of the metric.

type The data type of the metric.

group The group of the metric. Each metric is part of exactly one group (e.g. execution time and wall time are both members of the *time* group). This allows the specification of group-specific code that should be executed only once upon opening or closing a region e.g. for metrics that require a common start/stop procedure.

scope Denotes the topological hardware scope scope, e.g. hardware thread, core, CPU, or node.

aggregator Denotes the aggregation method to be used over multiple region executions (i.e. average or sum).

work_item_start_action Instrumentation or measurement code to be executed when starting or resuming a work item that is part of a currently open region. This allows the specification of metrics that exclude work items which are not part of the currently open region but might have been scheduled by the IRS.

work_item_end_action Instrumentation or measurement code to be executed when stopping or suspending a work item that is part of a currently open region.

region_early_start_action Instrumentation or measurement code to be executed for each metric by the first worker entering a region. This allows the specification of metrics in parallel contexts with first-entry and last-exit characteristics.

region_late_end_action Instrumentation or measurement code to be executed for each metric by the last worker exiting a region.

output_conversion Denotes a conversion factor to be applied when writing previously measured data. This allows a reduction of measurement overhead for metrics that differ in their units as seen by the compiler or user, and the hardware or IRS (e.g. execution time in nanoseconds or clock cycles).

Within the action items, two pre-defined fields per metric are provided for user access:

last_⟨metric name⟩ The last value of this metric, usually written when starting a region, or starting or resuming a work item.

aggregated_⟨metric name⟩ The aggregated value of this metric, i.e. the quantity that has been observed so far, usually written when ending a region or ending or suspending a work item.

These two fields can be accessed by the user in any fashion, allowing user-defined aggregation of metrics. Example 3.3 illustrates a possible usage of these fields.

Example 3.3 (Execution Time)**.** A simple implementation for execution time based on a sum of values is given by

$$\text{last} = \text{get\_time}() \tag{3.3}$$

$$\text{aggregated} = \text{aggregated} + \text{get\_time}() - \text{last} \tag{3.4}$$

Metric Groups

In addition to metrics, the IRS also holds information about *metric groups*. They are required for full metric generality, since there may be multiple metrics that require a common initialization or measurement routine that may only be called once (e.g. PAPI [88] per-thread initialization). For this reason, similarly to metrics, also metric groups follow a specification that includes the following items:

name The name of the group.

global_decls Global variable declarations required by the group with the lifetime of the IRS process (e.g. buffers, state information, ...).

local_decls Local variable declarations required by the group with the lifetime of the current measurement code and value computation (e.g. temporary buffers).

global_init Per-process initialization code, executed upon startup of the IRS.

global_finalize Per-process finalization code, executed upon shutdown of the IRS.

worker_init Per-thread initialization code, executed upon worker creation.

worker_finalize Per-thread initialization code, executed upon worker shutdown.

work_item_start_action Measurement code, executed when starting or resuming a work item that is part of a currently open region.

work_item_end_action Measurement code, executed when stopping or suspending a work item that is part of a currently open region.

region_early_start_action Measurement code, executed for each group by the first worker entering a region.

region_late_end_action Measurement code, executed for each group by the last worker exiting a region.

Algorithm 3.1 Order of measurement actions performed by each IRS worker.

```
1:  worker_init(group)                                  } Initialization
2:  worker_init(metric)                                 }
3:  ...
4:  if first worker to enter a region then              } Entering
5:      region_early_start_action(group)                } a region
6:      region_early_start_action(metric)               }
7:  end if                                              }
8:  wi_start_action(group)                              }
9:  wi_start_action(metric)                             }
10: ...                                                 } Region
11: ...                                                 } content
12: ...                                                 }
13: wi_end_action(group)                                } Exiting
14: wi_end_action(metric)                               } a region
15: if last worker to exit a region then                }
16:     region_late_end_action(group)                   }
17:     region_late_end_action(metric)                  }
18: end if                                              }
19: ...
20: worker_finalize(group)                              } Finalization
21: worker_finalize(metric)                             }
```

Algorithm 3.1 briefly illustrates the order in which each worker uses metric and group specification items. Once the IRS has completed all measurements and shutting down, the measurements are written to disk. Each worker creates a separate file containing a list of all regions. Each region entry holds the ID of the region, the number of times it was executed, and a list of all metrics that were requested to be measured. If a worker did not execute a region, its number of executions is 0 and each metric value will hold 0.

Example 3.4 (Measurement File)**.** Table 3.1 shows the structure of a measurement file written by the IRS that can be parsed by the compiler. It holds two regions, where the first was executed 4 times and the second was never executed by the worker that wrote this file.

3.2.3 Platform-specific Features

Beyond the generic framework that allows user-defined metrics, the IRS already includes implementations for a few selected metrics that occur frequently, as well

Table 3.1: Example IRS measurement result file.

label	ID	no. of executions	total wall time	total cpu time	total cpu energy	total L2 misses
RG	0	4	2000	4000	42	9173
RG	1	0	0	0	0	0

as control features that allow modification of the hardware to some degree. This section briefly outlines these aspects of the IRS.

Time

The IRS offers a function for fine-grained time measurements with a minimum of overhead, `irt_time_ticks()`. It maps to low-level assembler instructions implemented for two wide-spread hardware instruction set architectures, x86 (which offers the `rdtsc` instruction to read a timestamp counter register of the CPU) and PowerPC (which also offers special purpose registers for time keeping, read via `mfspr`). Since most x86 CPUs offer one timestamp counter register per core, it is important to bind IRS workers to CPU cores. The IRS offers affinity policies for this purpose.

Energy

The IRS offers a low-level implementation for reading Running Average Power Limit (RAPL) [31, 63, 54] data from x86 microarchitectures that support it (Intel Sandy Bridge and newer HPC architectures). On HPC platforms, it offers energy consumption data for the entire CPU package, the CPU cores only or the CPU memory controller only. The model-specific registers (MSR) that hold this data act similar to the timestamp counter register, i.e. the value increases monotonically, with updates occurring roughly every millisecond. However, since they only use 32 bits for this information, these registers overflow every few minutes depending on the energy consumption of the measured hardware units. For this reason, the IRS features a *maintenance thread*, which is capable of executing code asynchronously at user-specific time intervals. The maintenance thread reads the RAPL registers approximately every 30 seconds and updates the IRS' RAPL data. This ensures RAPL register overflows to be captured correctly, even in the case of multiple overflows between the start and end of a measured code region.

3.2.4 Control features

In addition to its pure measurement capabilities, the IRS also offers facilities to control the hardware to some degree. Most notably, this includes a frequency and voltage scaling (DVFS) feature which maps to Linux' `cpufreq` infrastructure. The IRS allows setting the power state $p \in \mathcal{P}$ of Section 2.1 on a per-worker basis, i.e. the u_{core} a worker is currently executing on. For this reason, IRS workers must be bound to CPU cores for reliable operation.

3.3 Additional Research

Besides the research presented in this thesis, the Insieme compiler and runtime system framework is also connected with a number of additional works. These represent either fundamental research that was used to build the framework, or research that was achieved by using the framework. They include

OpenCL kernel splitting converting single-device OpenCL programs into multi-device OpenCL programs with input-sensitive task partitioning based on a machine learning model [49, 72, 125];

Automatic data layout optimization automatic conversion of array-of-structure data layouts to structure-of-array layouts for improved locality [71];

energy modeling and optimization energy analysis and modeling using random forest modeling, and controlling tunable hardware under timing constraints [3, 16];

Task-optimization optimizing the granularity of recursive tasks coupled with multi-versioning [120, 123, 124, 126, 127]; and

Intermediate Representation and Transformations research in intermediate representations for compilers and their transformation frameworks [64, 66].

3.4 Summary

This chapter presented the Insieme compiler and runtime system framework, which served as the basis for all the work presented in this thesis. Special attention was paid to the instrumentation and measurement components of Insieme, developed as part of the work presented. First, the compiler component was described, focusing on the compiler's view of program regions, metrics and how the compiler conducts

measurements using executors and retrieves non-functional run-time data. Second, the runtime system component was presented. The unified specifications for metrics and metric groups were defined and their usefulness in establishing user-defined metrics and user-defined metric aggregation was demonstrated.

Chapter 4

Multi-Objective Auto-Tuning

This chapter presents work on multi-objective, search-based auto-tuning using iterative compilation, published under the title *Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage*, see [53]. Using the Insieme framework for iterative compilation auto-tuning, a differential evolution algorithm optimizes several parallel programs with regard to three conflicting objectives. My contributions in this work are the development of the tuning framework, working with Dr. Juan Durillo on the auto-tuning algorithm, conducting measurements, analyzing the data, and deriving guidelines for efficient parallel program execution.

4.1 Introduction

The performance of a software application crucially depends on the quality of its source code. The increasing complexity and multi/many-core nature of hardware design have transformed code generation, whether done manually or by a compiler, into a complex, time-consuming, and error-prone task which additionally suffers from a lack of performance portability. To mitigate these issues, a new research field, known as *auto-tuning* [128], has gained increasing attention. *Auto-tuners* are an effective approach to generate high-quality portable code. They can produce highly efficient code versions of libraries or applications by generating many code variants which are evaluated on the target platform, often delivering high performance code configurations which are unusual or not intuitive.

Whilst earlier auto-tuning approaches were mainly targeted at execution time [30, 137], other optimization criteria such as energy consumption [129] or computing costs [65] are gaining interest nowadays. In this new scenario, a code *configuration*

that is found to be optimal for low execution time might not be optimal for another criterion or vice versa. Therefore, there is no single solution to this problem that can be considered optimal, but a set, namely the Pareto set [24], of solutions (i.e. code configurations) representing the optimal trade-off among the different optimization criteria. Solutions within this set are said to be non-dominated: any solution within is not better than the others for all the considered criteria.

This multi-criteria scenario demands further development of auto-tuners, which should capture these trade-offs and offer the user either the whole Pareto set or a solution within it. Although there is a growing amount of related work considering the optimization of several criteria [80, 129, 58, 103, 33], most of them consider two criteria simultaneously at most, and many fail in capturing the trade-off among the objectives and reduce the problem to a mono-objective one.

In this chapter we investigate the auto-tuning of shared-memory parallel codes using the Insieme compiler to optimize three different criteria: execution time, resource usage and energy consumption. For tuning the codes, we consider as optimization knobs: dynamic concurrency throttling (also known as DCT), loop tiling, and frequency and voltage scaling (also known as DVFS). We examine the obtained results in detail to analyze and illustrate the complex interactions between optimized software and hardware. Our main findings of this work demonstrate that:

- RS-GDE3 can be successfully applied to a three-objective optimization problem, and
- the trade-off between execution time and energy consumption, dependent on efficient parallelization, can be explained by investigating resource usage.

This chapter is structured as follows: Section 4.3 describes the auto-tuning infrastructure used for this work. Relevant related work is listed in Section 4.2. The experiment design, the objectives of interest, the target codes and hardware platform are outlined in Section 4.4. Section 4.5 presents our results and their detailed analysis. Finally Section 4.6 concludes.

4.2 Related Work

There is a wide range of related work pertaining to auto-tuning. One possible approach is Machine learning (ML) [1, 17]. The underlying idea of ML consists in off-line learning of the relationship among a set of code features and beneficial optimizations for a representative set of applications; the inferred knowledge is then reused to determine optimizations for unseen codes. The success of this approach

depends on the quality of the considered training data for learning, selected features, and representative applications. A wrong decision on any of these phases may imply ML to reproduce non-optimal behavior. In addition, despite some attempts to apply ML to optimize multiple criteria [43, 101], it has never been used in a truly multi-objective fashion, computing the whole set of Pareto efficient solutions.

Search-based methods pose an alternative to ML, consisting in iteratively evaluating different code versions of a target application. These methods are often used in self-tuning libraries such as ATLAS [137], OSKI [135], SPIRAL [100], FFTW [42], or application-specific approaches including Active Harmony [30], Sequoia [36], PetaBricks [5, 4], Patus [27], or OpenTuner [6]. Both, exact [75] and approximation search approaches [98], guarantee a local optimum for any application in many cases. Furthermore, they have been successfully applied for computing the whole set of Pareto efficient solutions in the past while optimizing up to two criteria, such as performance and efficiency, compilation time, or binary size [65, 58, 44, 82].

The recent concern for power and energy consumption is reflected in the growing amount of related work applying auto-tuning to optimize them. Whether they consider power or energy consumption, in addition to performance, most of these works fail to capture the trade-off between these two criteria and only compute a single solution to the problem. In [80], the authors apply dynamic programming to reduce the energy consumption while maintaining or improving the performance of message-passing and shared memory applications; the performed search relies on models for predicting performance and energy consumption, thus avoiding to evaluate different configurations of an application. Also a model for power consumption is employed in [102]. In this case, several combinations of power and performance based on preferences are optimized independently by means of a hierarchical search procedure and compared afterwards. In [129], however, real power measurements obtained from a dedicated power device are considered to guide the search using Active Harmony. As in the previous work, several combinations of energy and performance are also investigated independently. In [103], the authors apply auto-tuning to optimize independently for performance and power consumption; as the authors claim, code versions which are optimal for one of the objectives are not optimal for the other. Others works try to exploit slack time for example in MPI communication [114, 97] or OpenMP synchronization [33]. Despite all these efforts on optimizing performance and energy/power, almost none of these approaches compute the full Pareto set of solutions. Reducing this trade-off to a predefined number of solutions as frequently done in related work may limit the freedom of selecting a solution and render detailed trade-off analyses impossible. To the best of our knowledge, [40] is one of the few

works investigating that trade-off. More recently, also in [13], the authors point out the multi-objective nature of this problem, showing a fine-grained trade-off between performance and energy/power consumption.

Contrary to related work and to the best of our knowledge, this is the first application of an auto-tuner to an optimization problem consisting of three objectives: performance, resource usage and energy. Moreover, we provide a detailed analysis of the trade-offs identified by using the Insieme auto-tuning compiler. This has been done in several related works, which do not directly deal with optimization or auto-tuning of applications. They rather analyze the trade-off between performance and power or energy for changing hardware or software configurations. Many of them investigate DVFS or DCT [76, 21, 45, 41], while some evaluate application model changes [81]. Their main goal is to identify the general trade-off between performance and energy/power consumption with manually preselected solution samples, whereas we analyze automatically obtained results.

4.3 Insieme Compiler

4.3.1 Auto-Tuning Infrastructure

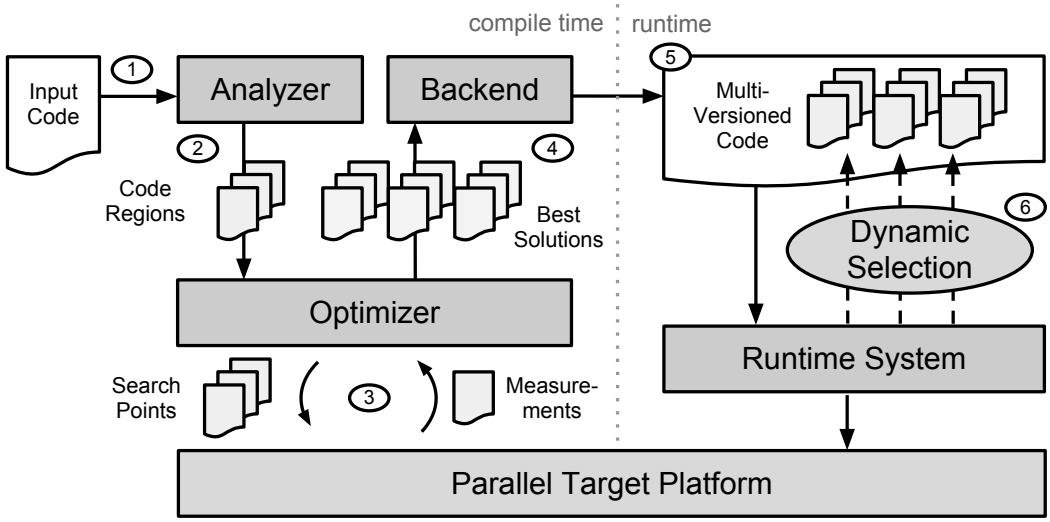


Figure 4.1: Detailed Insieme component interaction for the use case of search-based optimization, adapted from [65].

While the overall structure of the Insieme framework has already been described in Chapter 3, Figure 4.1 illustrates key components with regard to the specific use case of search-based optimization. The labels (1-6) follow the processing of a program

for this purpose, and are further described as follows: An input code is loaded by the frontend (1), analyzed and prepared to be tuned by the optimizer prior to execution.

The analyzer determines a set of *transformation skeletons* for a given parallel code region $\mathcal{R}$ per code which describe generic sequences of code transformations using unbound *parameters* for tunable properties. These encompass

loop tile sizes Controlled by the compiler to increase the effectiveness of cache units $u_{\text{cache}} \in \mathcal{U}_{\text{cache}}$. A dependency test based on the polyhedral model [15, 99] is performed beforehand to ensure the transformation can be applied.

number of cores Controlled by the runtime system via its number of workers parameter. Limits the degree of parallelism by limiting the set of cores $\mathcal{U}_{\text{core}}$ involved in the computation.

frequency setting Controlled by the runtime system via its hardware interaction capabilities, selects a power state $p_u \in \mathcal{P}_{\text{core}}$ with $u \in \mathcal{U}_{\text{core}}$.

The code, together with its associated transformation skeletons and some (optional) parameter constraints, is passed on to the optimizer (2). At this point, the optimizer conducts auto-tuning (for this reason we use *autotuner* as a synonym for optimizer) by iteratively selecting sets of *configurations* for each code to be evaluated (executed) on the target system (3). Each configuration corresponds to an instantiation of a transformation skeleton's parameters.

In the end, the optimizer derives a Pareto set consisting of the best configurations found so far. These configurations are passed on to the *runtime backend* (4), which generates a specialized code version for each of these configurations as part of a single, multi-versioned executable (5). The runtime system can then execute one of the configurations of the Pareto set according to the current requirements of e.g. the system or the user (6).

4.3.2 Optimizers

The main search engine of for this work, described in previous work [65], is called RS-GDE3 and aims at computing the Pareto set of code configurations. RS-GDE3 combines an approximation technique from the class of Differential Evolution (DE) [116], a subclass within *Metaheuristics* [46], and a search space reduction mechanism based on *Rough Sets* [95]. The goal of this latter technique is to reduce the search to a small area where RS-GDE3 assumes the location of the optimal configurations, instead of dealing with a very large space of potential configurations which would be far too time-consuming to explore.

RS-GDE3, depicted in Algorithm 4.1, can be configured via three parameters: the size of code configurations $|C|$, with $|C| > 3$; the *differential weight* DW, with $0 \leq DW \leq 2$; and the *crossover probability* CR, with $0 \leq CR \leq 1$. These parameters are usually set in a pre-tuning phase, may affect the performance of the algorithm and the quality of the results, and many works deal with the issue of good parameter settings, such as [96]. Our choices for these parameters are shown in Table 4.2.

The main idea of DE methods is to generate new code configurations by considering transformations of old configurations that proved to be good candidates. In detail, the algorithm works as follows: A set C of initially randomly generated code configurations is created. Within this set C, each configuration $c_x \in C$ is defined by a vector $c_x = c_{x,1}, \ldots, c_{N,t}$ of t tunable parameters, and the algorithm iteratively generates new values for them. For every configuration c_x in C, three (as in the original example presented by Storn and Price [116]) of the best performing configurations in C are identified, denoted here by c_1, c_2, and c_3, with $c_1 \neq c_2 \neq c_3 \neq c_x$. The crossover probability CR decides whether a new code configuration c' is generated. If so, it is computed using a *differential weight* DW and the three configurations c_1, c_2, and c_3 previously chosen. The tunable parameters of the new configuration c' reside within the search space B and in the vicinity of c_1, c_2, and c_3. This newly generated configuration c' then replaces the worst configuration in C. In addition, if the values of the tunable parameters for this code version are close to the limits defining the search space B, this search space is updated accordingly (shown as *updateSearchSpace(*B, c'*)* in Algorithm 4.1). This method was successfully applied to an optimization problem with two conflicting objectives in [65], whereas we apply it for the first time to three objectives in this work. However, RS-GDE3 is a true multi-objective optimizer that can handle an arbitrary number of objectives within the scope of Pareto optimality.

Algorithm 4.1 Generating a new configuration in DE.

```
1: Input: c_x, c_1, c_2, c_3 four different configurations, B the current search space
2: Output: c'
3: index ← floor(rand() · |c_x|) + 1
4: for z = 1 .. |c_x| do
5:     if rand() < CR or z = index then
6:         c'_z ← c_{1,z} + DW · (c_{2,z} − c_{3,z})
7:     else
8:         c'_z ← c_{x,z}
9:     end if
10: end for
11: return updateSearchSpace(B, c')
```

In addition to RS-GDE3, the Insieme compiler includes two other search engines, which are used for comparison, based on a hierarchical and a random search technique. The hierarchical search evaluates points on an equidistant grid defined over each tunable parameter. The random search generates a set of code configurations by randomly setting the values of each tunable parameter.

4.4 Experiment Design

4.4.1 Objectives

In this work we try to optimize shared-memory parallel programs for three objectives and investigate the trade-offs between them:

- *execution time*,
- *resource usage*, and
- *energy consumption.*

Execution time is inherently an objective of interest, as providing results within the shortest possible time is desirable for most programs. In terms of the software model presented in Section 2.2, it is defined as

$$t = \max_{r \in \mathcal{R}}(\tau_{\text{end}}(r)) - \min_{r \in \mathcal{R}}(\tau_{\text{start}}(r)). \tag{4.1}$$

We furthermore include *resource usage*, denoted by

$$ru = |\mathcal{U}_{\text{core}}| \cdot t_p \tag{4.2}$$

with $\mathcal{U}_{\text{core}}$ being the set of cores involved in executing the program and t_p denoting the parallel execution time, as an objective to reflect computing costs. Most economic cost models that focus on computational resources, such as the ones used by cloud providers, are based on CPU hours [7]. Similarly, many academic computing centers base their accounting on CPU hours even if users are not charged. Hence, we believe that resource usage (reflecting computing costs – economic or otherwise) is an important optimization goal for parallel applications.

As a third objective of interest we consider *energy consumption*, defined as

$$E = \sum_{u \in \mathcal{U}_{\text{cpu}}} E_u \tag{4.3}$$

Table 4.1: Code characteristics.

Code	mm	dsyrk	jacobi-2d	3d-stencil	n-body
Problem Size	1200^2	1200^2	10000^2	600^3	500000
Computation	$\mathcal{O}(N^3)$	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
Memory	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N^3)$	$\mathcal{O}(N)$
Tile Sizes	$(1\text{–}600)^3$	$(1\text{–}600)^3$	$(1\text{–}5000)^2$	$(1\text{–}300)^2$	1–1000, 1–500000
No. of Cores	1–32				
CPU Freq. (GHz)	1.2–2.7 + Turbo Boost				
Total No. of Configurations	$1.11 \cdot 10^{11}$	$1.11 \cdot 10^{11}$	$1.28 \cdot 10^{10}$	$4.61 \cdot 10^{7}$	$2.56 \cdot 10^{11}$

where $\mathcal{U}_{\text{cpu}}$ is the set of CPUs that hold cores participating in the computation and E_u is the measured energy consumption of a single CPU. Reducing energy consumption is of interest to both HPC center operators and users (as future cost models might include energy consumption due to its increasing workload dependence). The CPU is the largest contributor of the overall energy consumption of a non-accelerated HPC node that can also be influenced the most by the workload executed [112]. Hence, we focus our energy optimization efforts on this component (see Section 4.4.2 for details on the method used to obtain measurements).

For brevity, we refer to execution time only as *time* and to energy consumption as *energy* throughout the rest of the paper.

4.4.2 Benchmarks and Target Platform

Our benchmarks consist of a matrix multiplication kernel (*mm*, using an ijk loop order), a BLAS-3 linear algebra kernel (*dsyrk*, computing $B = A * A^T + B$), two stencil codes (*jacobi-2d* and a generic 3x3x3 *3d-stencil*) and an implementation of an *n-body* simulation. Except for the *mm* and *dsyrk* codes, all of them exhibit distinct computation and memory complexities as listed in Table 4.1 and hence considerably different memory reuse and access patterns. Furthermore, although identically categorized in terms of complexity, the memory access patterns of *mm* and *dsyrk* are very different since the (on-the-fly) transposition of A eliminates the unaligned matrix access conducted within the *mm* code. Table 4.1 also lists the tunable parameters and their ranges for each code.

The target platform is a quad-socket ($|\mathcal{U}_{\text{CPU}}| = 4$) shared-memory parallel computer equipped with Intel Xeon E5-4650 Sandy Bridge EP processors [28]. The platform is the same as presented in Example 2.1, with each CPU offering eight

cores ($|\mathcal{U}_{\text{core}}| = 8$ per CPU). Each core features private L1 and L2 caches of 64 and 256 KB each in addition to the CPU-wide shared L3 cache of 20 MB. The system provides 128 GB of main memory, uses a Linux operating system with a 3.5.0 kernel and our backend compiler is GCC 4.6.3. Hyper-Threading was not used in any of our experiments.

Since frequency scaling is one of the tunable parameters as listed in Section 4.3, we provide further detail to the target platform's DVFS capabilities. The hardware supports DVFS with $\mathcal{F} = \{1.2 \text{ GHz}, \ldots, 2.7 \text{ GHz}\}$, additionally operating at up to 3.3 GHz when using Turbo Boost. All of these frequencies are coupled with voltages well above the threshold voltage (see Section 2.1.1), however since each $f \in \mathcal{F}$ has a non-modifiable voltage v assigned, we do not report the full tuple $(f, v) \in \mathcal{P}$ but only refer to f for brevity. Also, the DVFS capabilities of the target platform are limited to one $f \in \mathcal{F}$ for all cores per CPU. Therefore, from the perspective of the hardware model defined in Section 2.1, the system holds one DVFS domain per CPU, or formally $\mathcal{D}_{\text{DVFS}} \in \mathcal{U}_{\text{cpu}}$.

We rely on the Intel RAPL interface for providing energy data. On our target platform, RAPL offers energy estimations with a resolution of 15.3 microjoules at a rate of 1 KHz for the entire CPU package, i.e. a per-CPU energy measurement domain as exemplified in Example 2.4 or $\mathcal{D}_{\text{E}} \in \mathcal{U}_{\text{cpu}}$. Recent related work showed RAPL to be accurate enough for purposes such as ours [31, 63, 54]. It should be noted that we use RAPL due to its wide availability, however the Insieme compiler can use any energy measurement/modeling system that meets the necessary accuracy and resolution requirements.

4.4.3 Configuration of the Optimizers

We run the three optimizers available within the Insieme framework: RS-GDE3, hierarchical search, and random search. The parameters for RS-GDE3 and hierarchical search are described in the following and summarized in Table 4.2. In the case of RS-GDE3, we need to set the size of set C of code configurations (processed by RS-GDE3), the parameters CR and DW required by the differential evolution method, (see Algorithm 4.1), and the termination condition of the algorithm. These values have been determined during a preceding tuning phase over a small set of problems, have an impact on the optimization results and may depend on the target architecture. As termination condition, RS-GDE3 stops when it does not generate a better code configuration for a specific number of consecutive iterations chosen by the user (set to 5 for this work).

For the hierarchical search only the sampling grid needs to be defined. It depends

Table 4.2: Optimizers' Parameter Setting.

Algorithm	Parameters
RS-GDE3	$\|C\| = 30$ $CR = 0.5$ $DW = 0.5$
Hierarchical Search	**2D Tiling Problems** 21 values for each tiling parameter 6 different numbers of cores 6 different frequencies
	3D Tiling Problems 8 values for each tiling parameter 6 different numbers of cores 6 different frequencies
Random Search	15000 configurations total uniform probability distribution

on the number of tunable parameters and defines the total number of configurations to be evaluated. We have configured the hierarchical search with an equidistant grid such that at least 15000 different configurations are examined. For generating the grid we only need to specify how many equidistant values we consider for every tunable parameter (note that for the number of cores, we only select powers of 2, or formally $|\mathcal{U}_{\text{core}}| = 2^x$ with $0 \leq x \leq 5$) for the given target platform.

Finally, for the random search, we need to specify the number of configurations to be examined (also 15000 for this work) and the probability distribution to be used (uniform probability distribution for this work).

4.4.4 Comparison Criteria

To systematically compare different search-based optimization strategies we use two different metrics: (1) the *efficiency* of each strategy, and (2) the *quality* of the configuration set obtained.

Efficiency.

Since we deal with iterative compilation, it is worth evaluating the effort required to obtain a solution. To address this, we count the total number of configurations evaluated by the auto-tuners for obtaining the solution. We denote this number as N. Additionally, as our optimization includes time and energy, we also introduce two

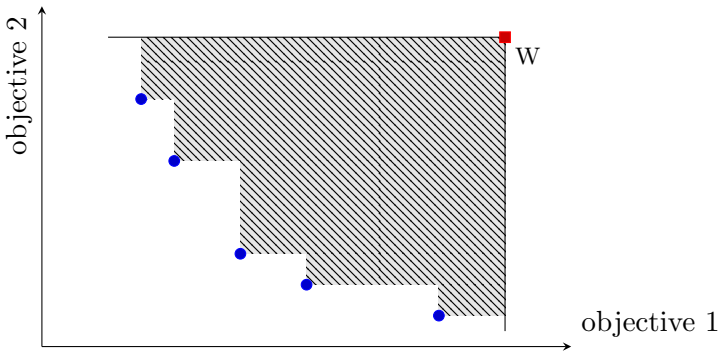


Figure 4.2: Two-dimensional example of a hypervolume $V(C_{\text{fin}})$ of a set C_{fin} of trade-off configurations (●) and a hypothetical worst-case configuration (■).

metrics to examine how RS-GDE3 performs with respect to these two objectives; we call this *time-to-solution* and *energy-to-solution*. They denote the amount of time or energy required by the three search methods to arrive at the final configuration set C_{fin}.

Quality.

Assessing the quality of a configuration which optimizes only one objective can be achieved by simply analyzing its value in that objective, e.g. the lower the value the better the configuration (assuming a minimization problem). However, comparing configurations of a multi-objective optimization problem is more complex since it requires comparing sets –the computed trade-offs– instead of single values. The *hypervolume* $V(C_{\text{fin}})$ of a set of non-dominated configurations C_{fin} is a metric proposed in [142] that solves this problem. It consists of the normalized volume –an area in case of a dual-objective problem– containing configurations that are worse than those contained in C_{fin}, as illustrated by Figure 4.2 for an arbitrary bi-objective problem. In other words, for any configuration enclosed by that volume there is a configuration in C_{fin} with better values for all the considered objectives. Naturally, the larger the hypervolume the better the quality of the configurations in C_{fin}. The largest hypervolume value ($V(C_{\text{fin}}) = 1$) belongs to the utopia point (unattainable optimal configuration), i.e. the point consisting of the optimum value for each criterion.

We also propose another metric to evaluate the quality of C_{fin}: the freedom in selection. The metric aims at quantifying how many different high quality configurations a technique exposes to the user. Simply using $|C_{\text{fin}}|$ to measure this does not completely address the problem: e.g. a configuration set obtained by strategy A could contain a lot of points dominated by the single point computed with strategy

B. For this reason, we also employ $|C_{\text{fin}}|'$, denoting the relative amount of configurations which are not dominated by the configurations computed by any other of the auto-tuners used. Hence, the higher the percentage, the higher the quality of the configurations contained within C_{fin}.

Since random search and RS-GDE3 are stochastic algorithms, they may produce different results in different runs. Therefore, the results of a single run are not sufficient for a meaningful comparison. In our evaluation we use the arithmetic means $\overline{N}$, $\overline{|C_{\text{fin}}|}$, $\overline{|C_{\text{fin}}|'}$, and $\overline{V(C_{\text{fin}})}$, derived over 5 runs, as directly comparable substitutes.

4.5 Experimental Results

4.5.1 RS-GDE3 Evaluation

Table 4.3 gives an overview of the performance of RS-GDE3 compared to hierarchical and random search with respect to the three considered metrics. It shows that RS-GDE3 needs only 5–12% of the number of evaluations compared to the hierarchical and random search strategies to provide configurations that dominate between 77% and 100% of the configurations offered by the other two. In addition, the configuration sets offered by RS-GDE3 span larger hypervolumes than the configuration sets provided by hierarchical and random search.

Table 4.4 illustrates the efficiency of RS-GDE3 compared to hierarchical and random search regarding the time and energy required to obtain a final configuration set. Beyond the already low number of evaluations compared to hierarchical and random search, RS-GDE3 spends disproportionally less time and energy for finding the final configuration set since it quickly converges on good configurations. Hence, only 0.7–7.2% of the time and 1.2–8% of the energy are required by RS-GDE3 compared to hierarchical and random search. It should be noted that the optimization problem cannot be simplified by sequentially optimizing parameters (e.g. finding an optimal tile size first and then tuning the number of cores), as the optimal choices for these settings have been shown to be inter-dependent in related work [65] as well as in our experiments. The comparison also shows that random search seems to be more efficient than hierarchical search. This may be caused by comparatively bad configurations that are found at the edges and corners of our search space – very specific points that are rarely evaluated by random search using a uniform probability distribution. Examples for bad configuration parameters include the minimum number of threads (causing long execution times) and maximum number of threads (for codes that do not scale well up to the maximum machine size), large tiling

Table 4.3: Performance comparison of the different evaluated algorithms.

		mm	dsyrk	jacobi-2d	3d-stencil	n-body
Hierarchical	N	18432	18432	15876	15876	15876
	$\lvert C_{\text{fin}}\rvert$	18	21	31	30	26
	$\lvert C_{\text{fin}}\rvert'$	2%	5%	78%	22%	0%
	$V(C_{\text{fin}})$	0.00	0.00	0.69	0.75	0.50
Random	$\overline{N}$	15000	15000	15000	15000	15000
	$\overline{\lvert C_{\text{fin}}\rvert}$	4.4	2.2	17.2	24.8	30
	$\overline{\lvert C_{\text{fin}}\rvert'}$	0%	11%	5%	60%	17%
	$\overline{V(C_{\text{fin}})}$	0.33	0.17	0.55	0.61	0.70
RS-GDE3	$\overline{N}$	956.2	1149.6	1243.6	981.4	1801.4
	$\overline{\lvert C_{\text{fin}}\rvert}$	23.4	24.8	29.8	28.2	29.6
	$\overline{\lvert C_{\text{fin}}\rvert'}$	98%	98%	75%	77%	87%
	$\overline{V(C_{\text{fin}})}$	0.48	0.31	0.76	0.76	0.77

Table 4.4: Time-to-solution and energy-to-solution of RS-GDE3 in comparison to hierarchical and random search.

	Hierarchical Search		Random Search	
Benchmark	Time	Energy	Time	Energy
mm	0.71%	1.18%	2.41%	2.30%
dsyrk	1.49%	2.43%	5.08%	4.95%
jacobi-2d	3.85%	4.05%	7.15%	4.59%
3d-stencil	2.78%	4.62%	6.20%	5.98%
n-body	2.58%	5.21%	6.93%	8.03%

parameters that do not improve cache behavior, or the minimum DVFS setting of $f = 1.2$ GHz that causes the highest execution time among all DVFS settings.

4.5.2 Energy-Time Trade-off as a Function of Resource Usage

Related work has already shown the existence of a trade-off between time and power consumption [103]. It is easily explained by different levels of CPU usage: faster configurations commonly use a higher number of cores, naturally demanding a higher power budget. Additionally, power states as defined in Definition 2.8 usually require voltage v and frequency f to be changed in unison, and as Equation (2.4) of Section 2.1.1 shows, voltage v appears on the function computing power P_{dynamic} as a power of two, whereas the clock frequency f affects time at most in a linear fashion.

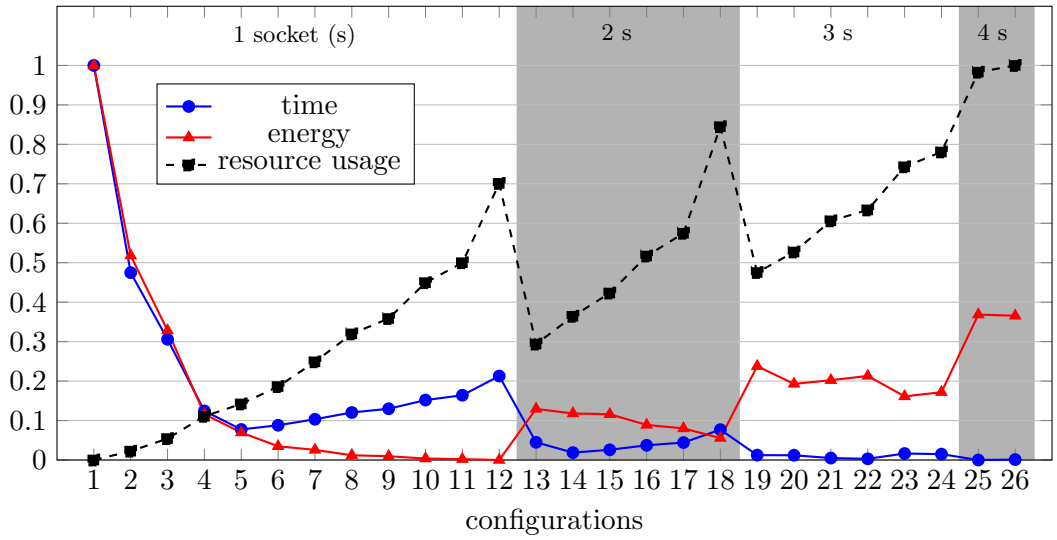


Figure 4.3: RS-GDE3 computed trade-offs among time, energy and resource usage for *mm*.

However, trade-offs between time and energy have been less studied in literature and are more difficult to obtain/explain since energy also depends on time and hence f. Thus, any optimization promising a trade-off between time and energy must increase or decrease power consumption disproportionally high compared to the decrease or increase in time. Our experiments show that the trade-off between time and energy varies with the resource usage and can expose different behaviors. In the rest of this section, we analyze these results and describe which parameters/situations are responsible for such trade-offs.

For the sake of clarity, we summarize our results using a graphical representation as illustrated by Figure 4.3. It shows the time, energy, and resource usage behavior of the set of code configurations computed by RS-GDE3 for *mm*. These configurations, listed in Table 4.5, are first ranked according to the number of sockets used, and configurations using the same number of sockets are further sorted by increasing resource usage. For clarity, configurations that use the same number of sockets are presented on the same background color and the number of sockets is given on the top. We will first discuss the *mm* case and present our findings, with the results of the remaining codes following thereafter.

Two different parts can be observed in the figure: a part where time and energy are highly positively correlated, and a second one indicating a trade-off between the two. The first part corresponds to configurations using a single CPU socket. As a consequence, we structure our discussion in two blocks: the *single-socket* and the

multi-socket case.

The single-socket case.

The results show that the configurations using only one socket can be further divided into a subset where reducing time also reduces the energy, and a subset where reducing time increases the energy. Without loss of generality we focus our discussion on the example of *mm* (Figure 4.3). When taking resource usage into consideration, we observe that time and energy are highly correlated when resource usage is low; however, this only holds until the resource usage reaches a critical point (configuration no. 5 in Figure 4.3), when both, energy and time, become conflicting objectives. For instance, energy can be further reduced from that point onwards while time increases.

A detailed analysis of the computed configurations (listed in Table 4.5) reveals that they use almost identical tile sizes. These values correspond to an optimal (local or global) tile size configuration found by the auto-tuner. Thus, once this optimal tile size configuration has been found, there are only two tunable parameters influencing the behavior of a code: the number of cores $|\mathcal{U}_{\text{core}}|$ and the clock frequency f.

Due to our sorting, the left-most configuration in Figure 4.3 is the one with the lowest resource usage ($|\mathcal{U}_{\text{core}}| = 1$, at the highest frequency). From this point, increasing the number of used cores reduces the time, and at the same time also the energy. The reason for this behavior can be explained with the power consumption breakdown of the CPU: using a single core requires most off-core entities of a socket to be active, such as the last level cache or the memory controller. Generally, increasing the number of used cores does not require providing additional power to activate those shared hardware units. Hence, doubling the number of used cores for example does not usually require double the power. Thus, as both time and power per used core decrease, the overall energy is also reduced. In fact, our experiments show that configurations no. 1–5 in Figure 4.3, where time and energy do not conflict, only differ in the number of used cores. Note that this holds only for scalable codes such as the ones used in our experiments. If a code does not scale sufficiently, parallelization may lead to a disproportionally low decrease in time compared to the increase in power, and the overall energy will increase as well. Since we target HPC codes, we assume scalability for the rest of the analysis. Our first observation can then be stated as follows:

1. Assuming scalable codes, parallelism is a way of reducing both time and energy when using a single socket computing system if the other parameters are kept invariable.

Table 4.5: Details of all *mm* configurations depicted in Figure 4.3.

Conf. No	Tile Size A	Tile Size B	Tile Size C	No. of Cores	CPU Freq. (GHz)
1	37	248	6	1	2.7
2	30	248	6	2	2.7
3	24	248	6	3	2.7
4	31	248	6	6	2.7
5	30	236	6	8	2.7
6	30	248	6	8	2.7
7	30	248	6	8	2.5
8	30	248	6	8	2.3
9	30	248	6	8	2.2
10	30	248	6	8	2.0
11	30	248	6	8	1.9
12	30	248	6	8	1.6
13	21	248	6	12	2.7
14	18	248	6	16	2.7
15	30	248	6	16	2.6
16	18	248	6	16	2.3
17	30	248	6	16	2.2
18	32	248	6	16	1.7
19	31	248	6	19	2.7
20	25	248	6	20	2.6
21	21	248	6	23	2.7
22	30	248	6	23	2.3
23	15	248	6	24	2.7
24	24	248	6	24	2.3
25	21	248	5	32	2.7
26	24	248	6	32	2.7

The second way of modifying the behavior with regard to the left-most configuration is via tuning the frequency parameter f. Lowering f —despite possibly decreasing the energy— increases time. The results of RS-GDE3 show that frequency tuning leads to dominated configurations if it is applied before fully exploiting parallelism. The reason explaining this is very simple. For every other configuration, the optimizer finds a configuration with increased parallelism reducing the time and obtaining a higher energy reduction than by using lower frequencies. Our second observation can be stated then as:

2. In a single-socket scenario, parallelism allows for higher rates of energy reduction than frequency tuning and, in addition, reduces time.

Once the maximum number of cores has been reached, the auto-tuner exploits frequency tuning. These configurations correspond to the second part of the graph, where energy and time are conflicting objectives. As follows from our previous discussion, decreasing the time is no longer possible since parallelism has already been exploited and all cores are working at their maximum frequencies. Decreasing the frequency will naturally increase the execution time but energy reductions can be achieved, caused by the cube root rule [38]: the power consumption of a CPU scales cubically as long as its voltage changes with the frequency in a correlated fashion; however, the performance of a code usually scales at most linearly with the CPU clock frequency. Hence, a trade-off between time and energy is formed and continues up to the energy-optimal frequency setting. This energy-optimal setting is workload-dependent and was found to be around $f = 1.5$ GHz on our target platform by our auto-tuner, as lower frequencies show an increase in energy (because the CPU voltage v cannot be scaled down accordingly by the hardware). Thus, as lower frequencies would worsen all three objectives, such configurations are rejected by the optimizer. Our third observation in this case is:

3. When parallelism has been already exploited, energy can still be further reduced by the sake of slightly increasing time, via applying frequency tuning.

The multi-socket case.

Again, without loss of generality we focus on the results depicted by Figure 4.3. According to the results illustrated in that graph, moving to a configuration using an increased number of sockets has been successfully exploited by the auto-tuner. In such situations, RS-GDE3 has always found a configuration which reduces the time compared to configurations using a lower number of cores (see for example the first configurations using two, three, or four sockets in Figure 4.3). However, this jump to a higher number of sockets always comes with an increase in energy. Thus, our

observation (1) in the previous section does not hold in the case of using multiple sockets due to the required energy to operate additional sockets. This fact allows us to state our fourth observation:

4. Multiple sockets can be exploited to decrease the execution time of an application but not to further reduce its energy.

Our experiments also reveal that, when using more than one socket, the number of cores leading to optimal trade-off configurations does not gradually increase as in the single socket case, but almost instantly reaches the maximum number. This results in our fifth observation:

5. Optimal trade-off configurations using more than one socket span over the maximum number of available cores.

We also observe that the energy can be reduced by the sake of increasing the time. This situation corresponds to observation (3), where the auto-tuner reduced the frequency for energy savings. Therefore, that observation also applies to the case of configurations involving several sockets at a full utilization level.

Figure 4.4 shows that our observations and findings also hold for the remaining target problems *3d-stencil*, *n-body*, *dsyrk*, and *jacobi-2d*. It should be noted that RS-GDE3 computed configurations that use up to four sockets for all problems except for *jacobi-2d*. This is explained by an average scaling behavior of the *jacobi-2d* code, which reaches its minimal execution time by using 10 cores instead of the maximum of 32. The remaining four codes scale well on our target hardware.

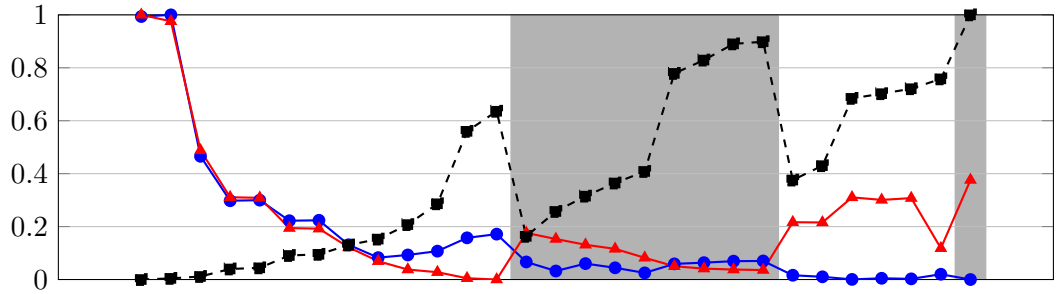
4.5.3 Impact of Turbo Boost

In addition to the results presented thus far, we investigated whether Intel's Turbo Boost feature [25] might have any effect on our observations. Turbo Boost (TB) is a mechanism provided by Intel to increase the performance of applications with a limited degree of parallelism, by defining the maximum clock frequency $f_{\max}$ to be inversely proportional to the numbers of cores in use, i.e.

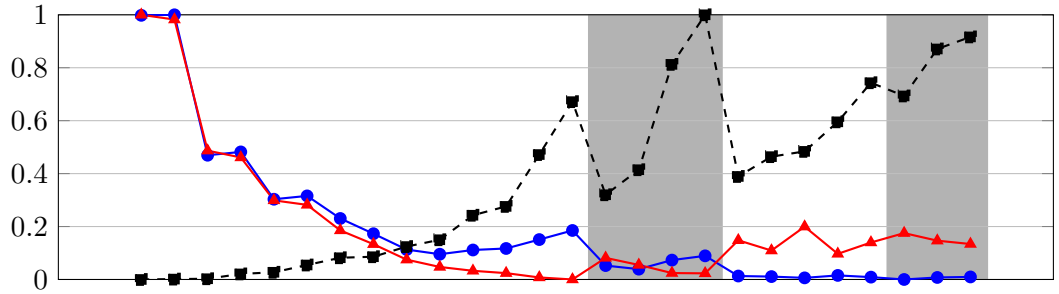
$$f_{\max,\text{cpu}} \propto \frac{1}{|\mathcal{U}_{\text{core,cpu}}|} \tag{4.4}$$

where $\mathcal{U}_{\text{core,cpu}}$ is the set of all cores of a CPU. As its operation depends on run-time properties that are difficult to predict, such as the current thermal and power budget of the CPU [107], TB is often disabled in related work. However, we believe that such hardware characteristics can be exploited by auto-tuning, and therefore, we enabled it for all our experiments.

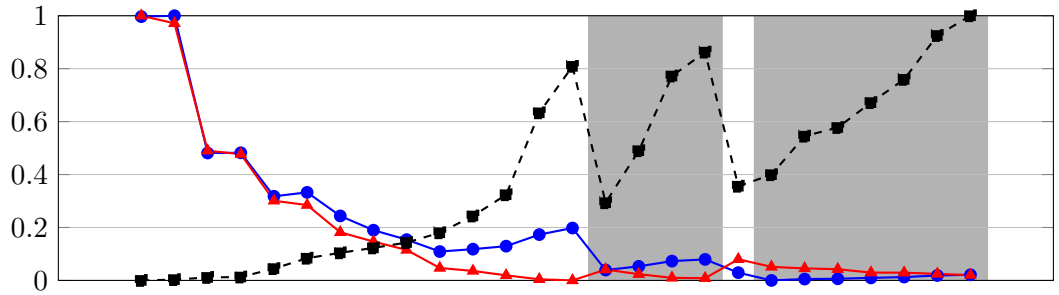
To check whether the observations discussed in the previous section depend on



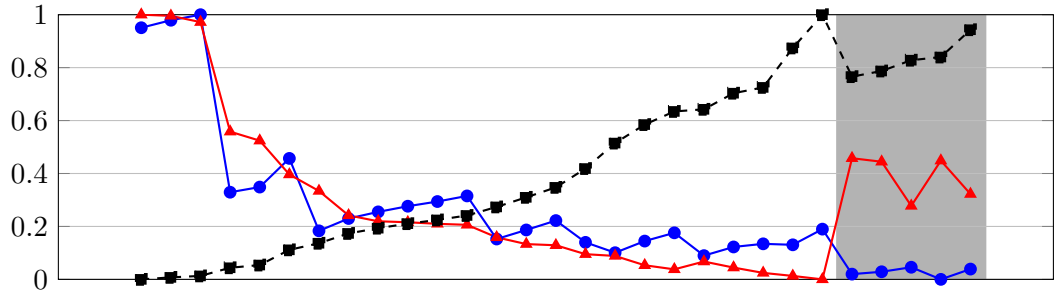
(a) *3d-stencil*



(b) *n-body*

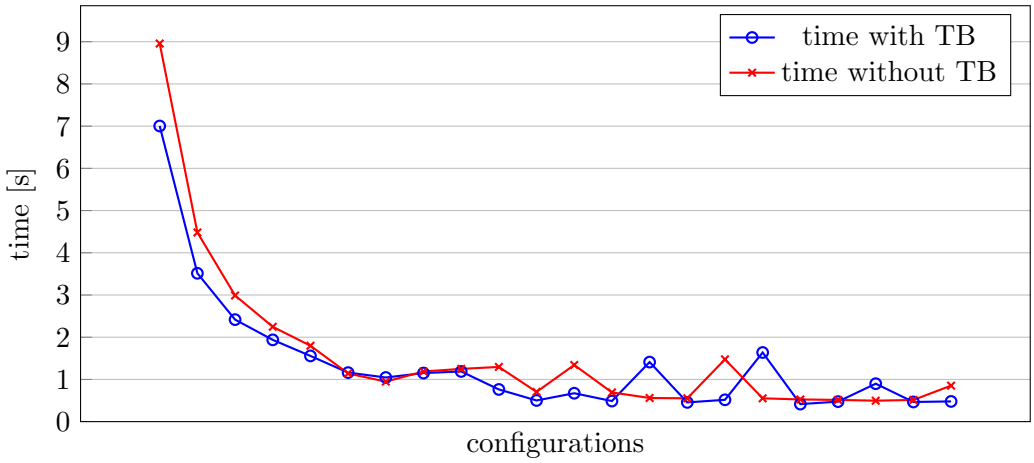


(c) *dsyrk*

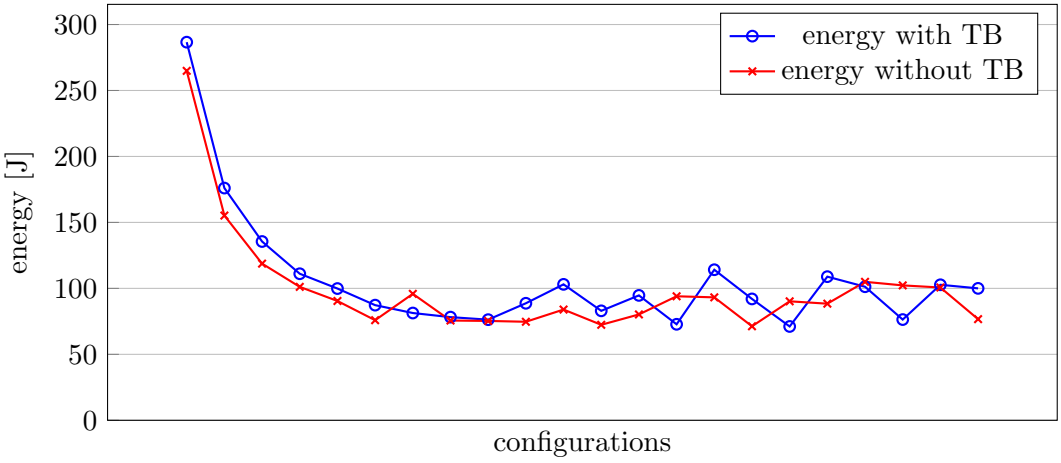


(d) *jacobi-2d*

Figure 4.4: RS-GDE3 computed trade-offs among time (—●—), energy (—▲—) and resource usage (- ■ -) for *3d-stencil*, *n-body*, *dsyrk*, and *jacobi-2d*.



(a) execution time



(b) energy consumption

Figure 4.5: Sample Pareto sets of single RS-GDE3 runs for the *n-body* code with Turbo Boost enabled and disabled.

the use of TB, we also ran RS-GDE3 with the TB option disabled and found all our observations to be still valid. Nevertheless, RS-GDE3 generates additional configurations with TB, allowing for a larger trade-off between time and energy for low numbers of cores and hence extending the objective space. Figure 4.5a and Figure 4.5b illustrate this effect. They show time and energy for the *n-body* code with the data sorted in ascending order of resource usage. Due to the sorting, configurations on the left show a relatively large difference between time and energy, as the effect of TB is stronger with fewer cores in use. As the number of cores increases, the effect diminishes.

However, it should be noted that TB might lead to dominated trade-off configurations in a multi-socket scenario, due to sockets working at different frequencies if their numbers of active cores do not match.

4.5.4 Evaluation of RS-GDE3 for Dual-Objective Optimization

Previous sections showed the potential of our RS-GDE3 method for three-objective auto-tuning. The aim of this section is to empirically evaluate RS-GDE3 for dual-objective optimization. Although RS-GDE3 has been already applied to optimize execution time and resource usage in [65], here we extend the analysis comparing RS-GDE3 with NSGA-II [32], the most popular algorithm for multi-objective optimization. Figures 4.6 and 4.7 illustrate the Pareto fronts computed by RS-GDE3 and NSGA-II when they are applied to optimize execution time and resource usage, and execution time and energy for *mm*; notice that we have always included execution time in our experiments since minimizing it is still the main objective in HPC. The figure shows that the Pareto fronts computed by RS-GDE3 cover a wider range of possible objective values in both cases, therefore offering a larger trade-off. Additionally, for each solution included in the Pareto fronts computed by NSGA-II, there is a solution computed by RS-GDE3 which dominates it (i.e. it is faster and exhibits lower resource usage or it is faster and requires less energy). Thus RS-GDE3 clearly delivers better results than NSGA-II.

4.5.5 Tiling Effects

One of the parameters tuned by RS-GDE3 are loop tile sizes. Examining the results more closely reveals points that seem to provide small trade-offs between time and energy caused by tile size changes while keeping the same number of cores and CPU frequency setting. The first three energy-time pairs of configurations in Figure 4.4b and Figure 4.4c show this characteristic. Between the two configurations of a pair, only the tile size was modified. Upon further investigation we found this difference

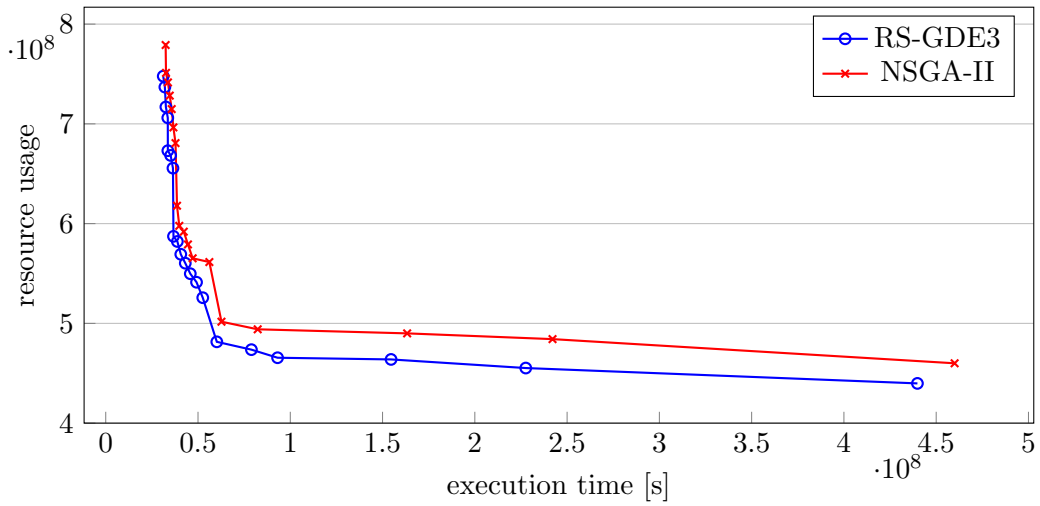


Figure 4.6: Sample Pareto fronts obtained by RS-GDE3 and NSGA-II for *mm* when optimizing for execution time and resource usage.

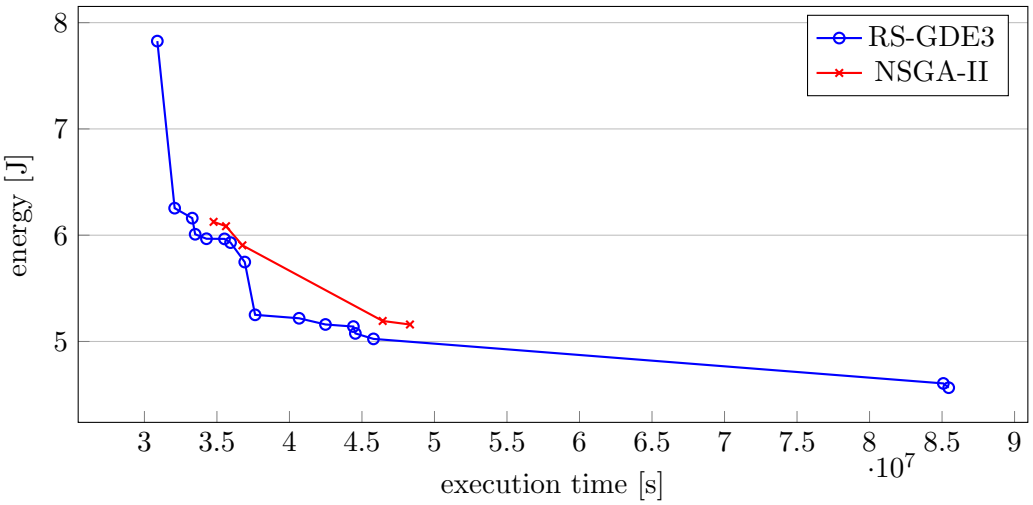


Figure 4.7: Sample Pareto fronts obtained by RS-GDE3 and NSGA-II for *mm* when optimizing for execution time and energy.

to be statistically insignificant and assume causes such as inaccuracies of the energy measurement system. Another possible reason is external load on the remaining but usually idle cores of the same socket. While this would show little impact on the execution time of our codes, it would fully contribute to the energy which is only measured on a per-socket-basis. For configurations that use all cores on a socket, there would also be a correlated impact on the execution time, hence not leading to a presumed trade-off.

4.6 Summary

In this work, we have shown the application of a multi-objective auto-tuner which optimizes for three conflicting criteria: execution time, resource usage and energy consumption. We compared RS-GDE3 with a hierarchical and a random search and showed that RS-GDE3 requires at least 93% less time and 92% less energy to obtain solutions of equal or higher quality in a benchmark composed of five computationally intensive codes. We identified the complex relationships between the three objectives and the effect of our tunable parameters on them. Our results have been outlined with clear observations that can be used to guide the development of auto-tuners and code optimization.

Possible future extensions of this work include investigating more sophisticated ways of dealing with hardware characteristics such as Turbo Boost or considering additional objectives. In addition, the search space could be extended by adding more tunable knobs such as scheduling strategies and topology-aware thread/core affinity mappings.

Chapter 5

Significance-driven Optimization of Code Execution

This chapter presents work on significance-driven code execution and near-threshold voltage computation, published under the title *On the Potential of Significance-Driven Execution for Energy-Aware HPC*, see [52]. It investigates attributing parts of iterative solvers (statements or data) with varying susceptibility to faults, justifying executing parts of the solver on unreliable hardware at reduced energy consumption. My contribution for this work was implementing the fault simulation and experimental framework, conducting measurements, data analysis, and investigating a partial fault protection method.

5.1 Introduction

While high performance is still the main objective in HPC, many considerations have been raised recently that require including power and energy consumption of hardware and software in the evaluation of innovative methods and technologies. The motivational reasons are diverse, ranging from infrastructural limits such as the 20 MW power limit proposed by the US Department of Energy to financial or environmental concerns [131].

As a result, hardware industry has shifted its focus to include power and energy minimization into their designs. The results of these efforts are evident by features such as DVFS or power and clock gating.

DVFS has been an efficient tool to reduce power consumption. In cases when a CPU core is not fully utilized, frequency and subsequently voltage are reduced to save power with limited impact on performance. However, the increased voltage margins,

that are required with the shrinking of transistors, put a limit on how much we can scale voltage. These constraints do not allow to operate hardware at the minimum energy consumption point, which is in the sub-threshold area below the nominal transistor threshold voltage (see Section 2.1.1). Operating in the sub-threshold voltage area gives rise to increased variation, timing errors and performance degradation that would be unacceptable for HPC applications. However, operating with supply voltage slightly above the threshold voltage yields power consumption gains similar to those with sub-threshold voltage operation, with less performance degradation due to less aggressive scaling of frequency. This is also referred to as *near-threshold voltage* (NTV) operation. On the downside, operation in NTV does not entirely mitigate timing errors of transistors due to parametric variation, i.e. the deviation of design parameters due to imperfection of manufacturing methods. Karpuzcu et al. [69] propose operation at NTV employing multiple frequency domains to cope with variability and provide reliable hardware in order to cope with parametric variations. They suggest that operation in NTV will incur 10–50× power reduction with subsequent frequency reduction by 2–5×. To cope with this frequency-induced performance degradation they use parallelism, fitting hundreds of cores in the available power budget.

Methods for tolerating errors in hardware have been studied in the past [37, 60]. This resulted in several solutions at different levels of the design stack, in both hardware and software. However, these solutions often come with non-negligible performance and energy penalties. As an alternative, the shift to an approximate computing —also known as *significance-based computing*— paradigm has been recently proposed [11, 69, 78, 110]. Approximate computing tries to trade reliability for energy consumption. It allows components to operate in an unreliable state by aggressive voltage scaling, assuming that software can cope with the timing errors that will occur in transistors with higher probability. The objective is to reduce energy consumption by using NTV and avoid the cost of fault-tolerant mechanisms.

In this work, we are trying to utilize the potential for power and energy reduction that *near-theshold voltage computing* (NTC) promises combined with *significant-based computing*. We investigate the effects of operating hardware outside its standard reliability specifications on iterative HPC codes, incurring both computational errors as well as reductions in energy consumption. We show that codes can be analyzed in terms of their significance, describing their susceptibility to faults with respect to their convergence behavior. Using the Jacobi method as an example, we show that there are iterative HPC codes that can naturally deal with many computational errors, at the cost of increased iterations to reach convergence. We

also investigate scenarios where we distinguish between *significant* and *insignificant* parts of Jacobi and execute them selectively on *reliable* or *unreliable* hardware, respectively. We consider parts of the algorithm that are more resilient to errors as insignificant, whereas parts in which errors increase the execution time substantially are marked as significant. This distinction helps us to minimize the performance overhead due to errors and utilize NTV optimally.

We show that, on our target platform, we can achieve 65% energy gains for a parallel version of Jacobi running at NTV compared to a serial version at super-threshold voltage (i.e. normal, reliable operation) along with time savings of 43%, when we execute with 20% of the super-threshold frequency.

Section 5.2 discusses related work relevant to our research. The notion of significance is introduced in Section 5.3. We will describe our methodology and experiment setup in Section 5.4 and analyze and illustrate our results in Section 5.5. Finally, Section 5.6 will conclude and provide an outlook for future research.

5.2 Related Work

Research in NTC has attracted considerable interest. It is a direct effect of industry's strive to keep up with Moore's law while coping with thermal and power limits. Prior research in NTC [34, 68, 69, 70] investigates both hardware and software solutions to cope with the entailed proneness to faults and identify energy saving possibilities. However, this research either does not explore the effect of unreliability on unprotected codes or confines its exploration to probabilistic applications that can afford direct interventions to their convergence criterion and computation discarding [78]. Similarly, there are many works that explore the influence of errors on individual hardware units of processors, without further exploration of their implications on software [109].

Software methods for improving error resilience include checkpointing for failed tasks [60] or replication to identify silent data corruption [37]. Among others, Hoemmen and Heroux investigate iterative methods for their fault tolerance [57] and Elliott et al. quantify the error of single bit flips in progressing iterations of Jacobi [35]. However, these works do not investigate the impact of fault recovery on energy consumption or how fault resilience can be leveraged to reduce it.

Recently, there has been interest in exploiting approximate computing to build fault resilient systems. Leem et al. [78] build a system of few reliable and many unreliable cores. The system executes the control-intensive part of the application —which is highly fault intolerant— on reliable cores and the fault-tolerant compute-

intensive part of the application on relaxed-reliability cores. This scheme achieves 90% or better accuracy of the output of applications even for 2×10^4 errors per second per core. Agarwal et al. [2], Misailovic et al. [87] and Rinard et al. [106] propose a static analysis-based technique to reduce the number of iterations in a loop without compromising correctness. In the same context, Baek et al. [11] provide a framework where the programmer specifies the functions and loops they want to approximate and the desired loss of Quality of Service (QoS). Then, the program is transformed to meet the QoS degradation target. Rinard et al. [105] propose to discard some tasks of the application and produce new computations that execute only a subset of the tasks of the original computation. Sampson et al. [110] propose a technique to distinguish the data types that need precise computation from the ones that can be approximated. They guarantee that the approximate instructions will never crash the program but only reduce power consumption.

5.3 Significance

We motivate the notion of code *significance*, that different parts of an application (i.e. code regions of Section 2.2) show different susceptibility to errors in terms of the change in the end result. This applies to a code region's statements as well as its data and gives rise to considering partial protection methods, employed only where and when necessary. This distinction, coupled with the prospect of NTC, creates the opportunity for saving significant amounts of power by running non-significant parts of the computation on unreliable hardware (i.e. hardware units of Section 2.1) in a near-threshold operating mode.

We want to illustrate the applicability of significance classification on iterative solvers and their resilience in the presence of faults. Iterative solvers operate on repeatedly updating the solution of a system of equations until it reaches a desired level of accuracy. Errors occurring in these algorithms can be gracefully mitigated at the cost of an increased number of iterations to reach convergence. As a result, these applications are suitable candidates for trading time-to-convergence for lower energy consumption.

In correspondence with Section 2.2, we want to attribute significance to program statements and the data they operate on. We select the weighted Jacobi method as a representative use case in order to study the resilience to errors in the broader class of iterative numerical applications. Jacobi solves the system $A \times X = B$ for a

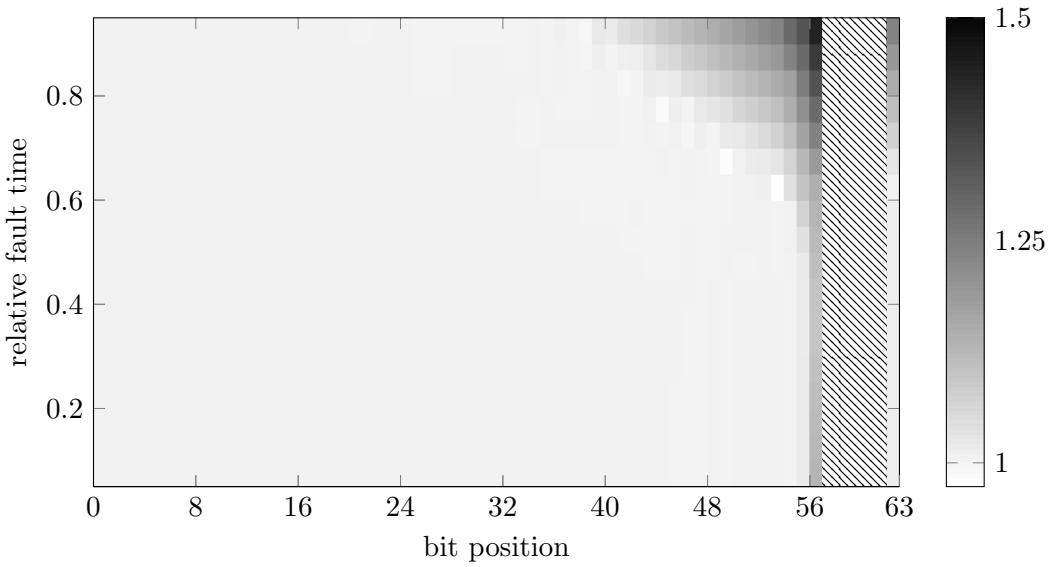


Figure 5.1: Relative time overhead of Jacobi for faults in A at various iterations, averaged over all matrix positions, for all bit positions, with a problem size of $N = 1000$. The hatched bar denotes divergence.

diagonally dominant matrix A, i.e.

$$|A_{x,x}| \geq \sum_{y \neq x} |A_{x,y}|. \tag{5.1}$$

It starts with an initial approximation of the solution, X^0, and in each step updates the estimation for the solution, according to

$$X_x^{(z+1)} = \omega \left(\frac{1}{A_{x,x}} B_x - \sum_{y \neq x} (A_{x,y} \cdot X_y^z) \right) + (1 - \omega) \cdot X_x^z. \tag{5.2}$$

The algorithm iterates until the convergence condition $\|A \times X - B\| \leq$ limit is satisfied, and is guaranteed to converge if A is strictly diagonally dominant, i.e.

$$|A_{x,x}| > \sum_{y \neq x} |A_{x,y}|. \tag{5.3}$$

To demonstrate the applicability of significance to Jacobi, Figure 5.1 presents the effect of a single bit flip fault happening in matrix A at various iterations of Equation (5.2) of an otherwise fault-free Jacobi run. It shows the relative overhead of Jacobi (i.e. the number of additional iterations) required to reach the convergence

limit compared to a fault-free run. Generally, Jacobi exhibits a logarithmic convergence rate and later iterations are more significant due to the overhead required to recover from a fault. Furthermore, because the achieved residual for later iterations is lower than for earlier iterations (due to the better X that has been computed), later iterations also show higher sensitivity to faults. Both these factors render program statements in late Jacobi iterations more significant than in early iterations [35]. This motivates the potential application of partial protection or recovery mechanisms, for later Jacobi iterations only.

In addition, Figure 5.1 illustrates that Jacobi is able to cope well with flips happening in lower bit positions, as they cause little to no overhead. This can be attributed to the high precision of double-precision floating point numbers. Flips happening in the high bits of the exponent for elements of A however can have two possible outcomes, depending on their position and the error they introduce. If the flip causes a positive error in the floating point number and happens aside the diagonal, i.e. in any $A_{x,y}$ with $x \neq y$, there is a risk of violating Jacobi's convergence condition of strict diagonal dominance for A, Equation (5.3) (analogously for negative errors on the diagonal, or $A_{x,y}$ with $x = y$). These cases manifest themselves as the solid bar shape in Figure 5.1 for bits 57–62. For the majority of cases that violate this condition Jacobi does not converge and ends up with an residual of either *infinity* (*Inf*) or *not a number* (*NaN*), depending on the operations involved. Overall, Figure 5.1 shows that for most bit positions there is no protection or recovery necessary, except for a few high bits of the exponent that justify mitigation techniques.

In addition to Jacobi's varying significance depending on the progress of the algorithm, significance can also vary depending on the data that is exposed to a fault (A, B or X), as well as the position within that data. As an example, Figure 5.2 presents the relative overhead of injecting a fault on the diagonal and offside the diagonal of iteration matrix A (for illustrative clarity, we choose the problem size to be 10×10). The fact that elements on the diagonal show lower impact (and hence lower significance) can be attributed to the combination of two reasons. First, these elements are used in a division operation whereas the others are used in a multiplication (see Equation (5.2)). Second, we use a uniform distribution to randomly initialize the elements of A and B, that leads to the majority of numbers being positive and greater than 1. Hence, a multiplication operation tends to increase any effect a fault might have, whereas a division generally reduces it. For this reason, our experiment results presented in Section 5.5 involving matrix A are based on sampling positions both on and aside the diagonal and computing a weighted average for the entire matrix.

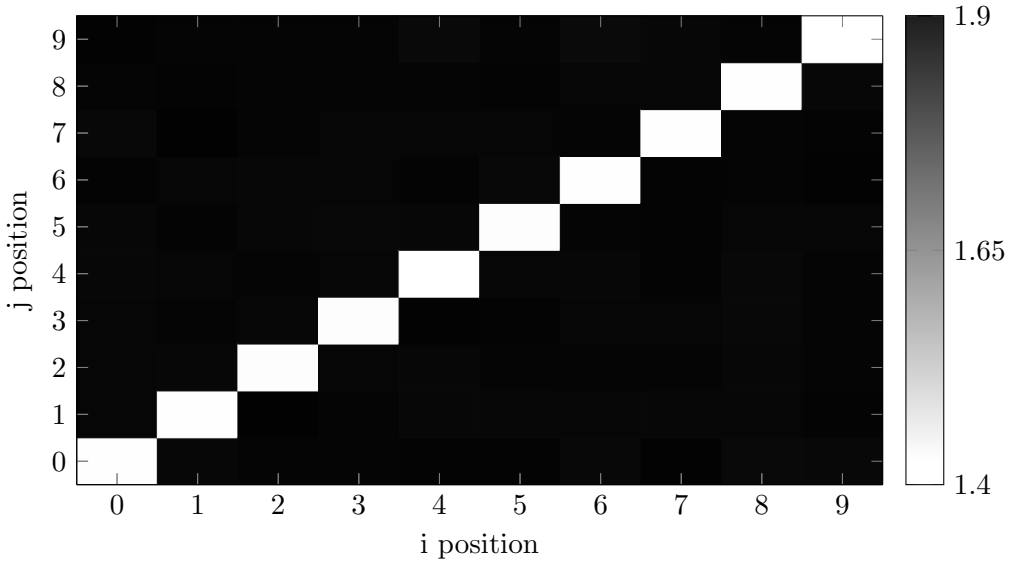


Figure 5.2: Relative time overhead of Jacobi for faults in A at all matrix positions, averaged over all bit positions, with a problem size of $N = 10$.

It should be noted that the weighting between diagonal vs. aside elements naturally changes with the matrix dimensions. As a consequence, the overall significance of A is also partially dependent on the input data size.

Detecting significance can be difficult for large applications. It is an attribute specific to the algorithm and input data, and hence input from programmers can be indispensable for a system that provides support for significance-based computing. Such a system would rely on the programmer's knowledge about the algorithm, who would denote e.g. which parts can be executed unreliably. Large pieces of software often compose from smaller mathematical kernels whose tolerance in faults has been extensively studied. Such studies could be used by programmers to mark code regions as significant or non-significant, without having to perform extensive profiling. However, there is ongoing research for algorithmic detection of the significance of code based on profiling (e.g. automatic differentiation [134]). This approach studies the sensitivity of code blocks, monitoring the range of the output of a code block after perturbation applied to the input. A code block is then considered to be more sensitive to errors, the larger the range of possible output values is. Nevertheless, automatically and efficiently detecting code significance is still an open research area.

Additionally, the design of a system for significance-based computing should provide fallbacks for applications that cannot afford unreliability in their execution. In order for these applications to be able to benefit from NTC, the system must employ

software or hardware fault recovery mechanisms.

The goal of this work is to demonstrate the existence of a trade-off between energy consumption and convergence time of iterative solvers in the presence of faults, and to analyze the properties of this trade-off.

5.4 Methodology

This section describes the fault model of our work. Moreover, it elaborates the power and energy effects that we expect from operating hardware unreliably and provides details about the hardware and our measurement methods.

5.4.1 Fault Model

Following common practice in related work [109], we categorize faults as follows:

no impact The fault has no effect on the application. This is the case for faults that happen in unused parts of hardware, such as hardware registers that are written after the fault occurred before they are read again, or parts of hardware not used by the application altogether. Analogously, faults can happen in parts of the application that are not executed or used (e.g. control flow branches that are not taken).

data corruption

silent The fault is only detectable with knowledge about the application, e.g. using an application-specific assertive check *"this computation must yield positive numbers"*.

non-silent The fault is detectable without knowledge about the application. Examples are faults that cause *infinity* (inf) or *not a number* (NaN) in computations that are expected to yield ordinary floating point numbers.

looping Faults that cause the application to loop, which might not be detectable (c.f. halting problem [132]).

other The fault represents an immediate, unrecoverable failure in the execution of the application. This category includes illegal instructions, segmentation faults, etc. Note that while segmentation faults are caused by accessing invalid memory addresses, these addresses are not considered data as per category *data corruption*.

Of these fault classes we consider silent data corruption (SDC) faults since they are the most insidious in high performance computing. Signaling errors such as *Inf*, or *NaN* or application crashes due to illegal instructions are comparatively easily detectable. Also, looping might be identified by detecting fixed points in the iteration data of an application or constraints on the execution time of code regions. In contrast, SDCs can cause graceful exits with possibly wrong results, making them particularly important to be dealt with.

SDCs can be categorized further as persistent or non-persistent. Persistent faults occur at the source of the data in question, i.e. if the data is read multiple times it will exhibit the same deviation each time. Non-persistent faults on the other hand are faults in temporary copies of data that are only used once (e.g. faults happening directly in execution units or registers).

We consider persistent faults, mappable to faults happening in CPU data caches that are read multiple times and might also be written back to main memory. We do not account for errors in machine code in instruction caches, because these can lead to non-recoverable errors. We assume that instruction caches are employed with protection mechanisms.

5.4.2 Energy Savings Through Unreliability

We explore execution schemes that deliberately compromise the reliability of processors, using NTV operation, for achieving power and energy savings in HPC codes [34]. The main issue of today's CMOS technology is that, while the number of transistors keeps increasing in accordance with Moore's law, the power supply and heat dissipation requirements for a fully loaded system (i.e. all or most transistors switching at high rates) are not sustainable under normal conditions. The solution to this problem is to operate the system at NTV. Karpuzcu et al. [69] suggest that power savings between 10× and 50× are possible with NTV, albeit with a 5× to 10× reduction in clock frequency (required by the increase in transistor switching time, also *circuit delay* [34], at NTV). Under these assumptions, a processor would consume 2× to 5× less energy per operation with NTV, compared to above-threshold voltage operation. Given a fixed power budget, a system design could replace few cores operating in the above-threshold region with many cores operating in the NTV region. A similar strategy of trading each reliable core with many unreliable NTV cores could be applied to achieve a fixed performance target. Addressing this performance concern, via parallelism, is fundamental, since we aim for application of NTV in HPC.

5.4.3 Experiment Setup

The experimental testbed used for testing our method consists of an HPC node equipped with four Intel Xeon E5-4650 Sandy Bridge EP processors [28]. Each CPU offers 8 cores with 32 KB and 256 KB of private caches each, and a processor-wide shared cache of 20 MB. The system runs a 3.5.0 Linux kernel and we used gcc 4.8.2 for compilation.

Our workload —a C implementation of Jacobi— is parallelized using OpenMP. The problem size $N = 1000$ was chosen such that the entire data resides in the last-level cache to minimize main memory interaction not covered in our energy measurement scope (described below), while still being large enough to ensure reasonable run times with regard to our measurements. As described in Section 3.2.3, time measurements were done via x86's *rdtsc* instruction and we used RAPL for obtaining energy consumption information of the entire CPU, resulting in a per-CPU energy measurement domain as exemplified in Example 2.4 of Section 2.1.3, or $\mathcal{D}_{\mathrm{E}} \in \mathcal{U}_{\mathrm{cpu}}$. To achieve consistent energy readings, the target hardware was warmed up for an ample amount of time before taking measurements.

Since our target hardware system only provides fixed tuples of voltage and frequency $(v, f) \in \mathcal{P}$ (see Definition 2.8) that the user can only select among but not change arbitrarily (as would be the case with processors such as the one described in Example 2.3), our target hardware system only allows reliable operation. For this reason, we have to simulate unreliable operation resulting in power and energy savings, as well as faults. Using the processor described in Example 2.3, which can be set to operate at arbitrary combinations of $v \in \mathcal{V}$ and $f \in \mathcal{F}$, is not an option for this work, as its voltage and frequency domains are too coarse-grained and would lead to instability in the entire system, preventing us from specifically investigating silent data corruption faults.

The simulation of power and energy savings can be achieved by correcting the energy consumption data obtained via RAPL with regard to the observations of near-threshold computing as discussed in Section 5.4.2. Moreover, to be able to simulate an arbitrary number of reliable or unreliable cores on non-configurable, commodity multi-core hardware, we need to take care when processing RAPL data as it includes off-core entities that might be oversized or not necessarily present in some cases (i.e. ring bus for a single core). To that end, we profiled the CPUs of the target platform with regard to their power consumption for all numbers of cores in a weak scaling experiment with Jacobi. Figure 5.3 shows the results of this endeavor. It indicates that the power consumption increases linearly with the number of cores with an offset for off-core entities of 7.1 Watts, which will be removed in subsequent

data analysis to provide a fair comparison between arbitrary numbers of reliable or unreliable cores. The figure also shows a different maximum power consumption for the two CPU samples (44.3 vs. 39.6 Watts), that can be explained by differences e.g. in supply voltage or the temperature.

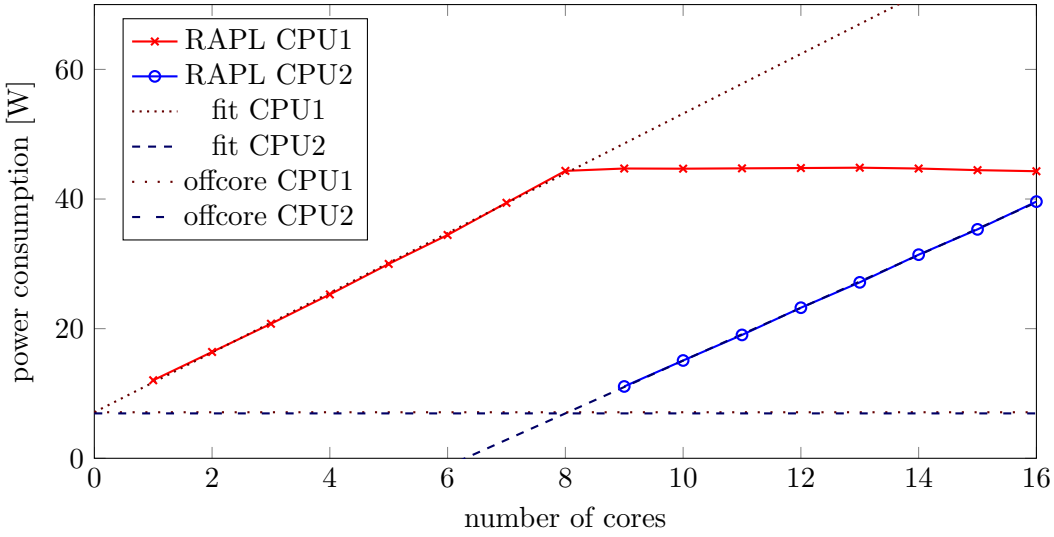


Figure 5.3: Power consumption per number of cores on two Intel Xeon E5-4650 for a weakly scaling Jacobi run as measured by RAPL, and offcore amount as inferred via linear fitting.

Furthermore, our target hardware platform also forces us to simulate faults in software. Algorithm 5.1 illustrates the experimental setup of our fault simulation. We inject persistent faults (modeled via fault function δ) represented by bit flips in the original data (matrix A, vectors B and X) at a range of bit positions of double precision floating point numbers prior to the computation of a Jacobi iteration. This simulates bit flips happening in the data caches of CPUs, that are accessed frequently during the computation. The implementation of the fault simulation is based on binary operators applied to the respective element in an inlined function, causing only negligible performance overhead compared to the overall execution time of a Jacobi iteration.

We assume a single bit flip per overall execution of Jacobi (i.e. multiple iterations), since related work indicates that the effects of multiple faults will lead to similar observations [10] and because it reduces simulation complexity. Hence, an *experiment*, which can be seen as an entity within a parameter study, is defined by

- the data component in which the fault occurs (in the case of Jacobi matrix A, or vectors X or B);

Algorithm 5.1 Simplified illustration of experimental fault simulation.

```
1:  A ← rand()                                          ▷ Matrix
2:  B ← rand()                                          ▷ Vector
3:  X ← 0                                               ▷ Solution
4:  residual ← 0                                        ▷ Solution residual
5:  count ← 0                                           ▷ Iteration count
6:  n ← 0                                               ▷ Temporal position of fault
7:  m ← ?                                               ▷ Temporal position of switch to reliable mode
8:  for n = 0 to m do
9:      do
10:         if count = n then                           } Fault injection
11:             A, B, X ← δ(A, B, X)                    } Fault injection
12:         end if                                      } Fault injection
13:         X ← jacobi(A, B, X)                         ▷ OpenMP implementation
14:         residual ← comp_residual(A, B, X)
15:         count ← count + 1
16:     while residual > 10^5 ∧ count ≤ limit
17: end for
```

- the bit position j at which a flip occurs;
- the Jacobi iteration m when the switch from unreliable to reliable mode occurs, with $1 < m < M$ and M denoting the total number of iterations of a fault-free Jacobi run; and
- the Jacobi iteration n, before the execution of which the fault occurs, with $1 \leq n < m$.

To minimize simulation time, we do not inject faults at every possible element of the vectors and matrices of Jacobi with a problem size of $N = 1000$, but perform representative sampling (e.g. elements both on and aside the diagonal of a matrix) with respect to the algorithm. Furthermore we employ a convergence limit of a factor of 10. This means that we consider a faulty Jacobi run as not converging if it takes more than 10× the number of iterations of a correct Jacobi run for the same input data set.

5.4.4 IEEE 754 Double-precision Floating-point Format

The data type in use for storing A, X and B in our Jacobi implementation is the default double-precision floating-point type of the C programming language, *double*. The analysis of our results in Section 5.5 is based on the bit positions within the

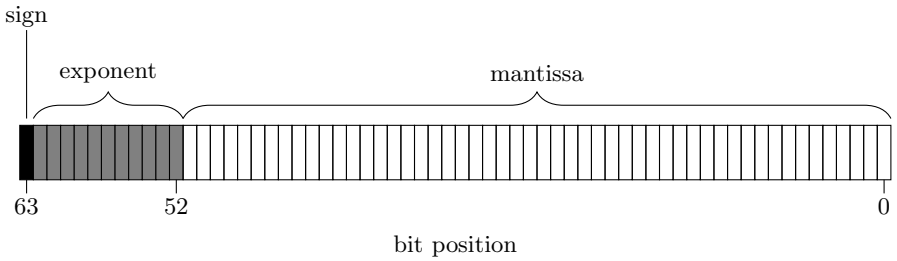


Figure 5.4: The IEEE 754 binary64 format

IEEE 754 [62] binary representation of these numbers, the *binary64* format. Elliot at al. already provide a detailed discussion regarding the effects of bit flips in this representation in [35]. Nevertheless, for clearness, we feel it is necessary to include a brief description of the binary64 format and elaborate on the magnitude of errors introduced by bit flips, dependent on the position of the bit.

Figure 5.4 shows the binary layout of the widely-used *binary64* format. The first 52 bits (positions 0–51) correspond to the mantissa, the following 11 bits (positions 52–62) are used for the exponent and bit 63 denotes the sign. Furthermore, within the mantissa and the exponent, their lowest bits are the least significant ones. The decimal value x of a floating-point number is then computed by the formula

$$x = (-1)^{sign} \cdot \left(1 + \sum_{j=0}^{51} \left(\mu_j \cdot 2^{j-52}\right) \cdot 2^{e-1023}\right), \tag{5.4}$$

where $sign$ denotes the sign bit, μ_j the j-th bit of the mantissa and $e - 1023$ the exponent (stored with a bias of 1023). Hence, the altered floating point number resulting from a single bit flip in position j' can be expressed as

$$x' = \begin{cases} x \pm 2^{j'-52} \cdot 2^{e-1023} & \text{flip in mantissa,} \quad (5.5a) \\ x \cdot 2^{\pm 2^{j'}} & \text{flip in exponent,} \quad (5.5b) \\ -x & \text{flip in sign.} \quad (5.5c) \end{cases}$$

We will evaluate the effect of these perturbations on the energy consumption and execution time of Jacobi in Section 5.5.

5.5 Results

In this section we compare executing Jacobi in parallel on unreliable hardware at near-threshold voltage (NTV) to sequential and parallel versions of Jacobi executed

on reliable hardware at super-threshold voltage. Our results present three cases:

Parallel unreliable vs. sequential reliable First, we execute Jacobi at NTV in parallel throughout its entire execution (i.e. all iterations) and analyze and discuss the energy savings that can be gained compared to a sequential run at super-threshold voltage. Since we are dealing with an HPC code, we will also investigate the performance impact of operating at NTV.

Parallel unreliable vs. parallel reliable Second, this analysis is repeated when comparing to a parallel execution of Jacobi.

Significance-dependent reliability switching Third, we explore the possibility of switching from NTV to super-threshold voltage for later iterations, motivated by our discussion in Section 5.3. We investigate whether later iterations of Jacobi are significant enough to justify the energy and performance expense compared to operating at NTV. This could create a potential trade-off, since executing late iterations at super-threshold voltage also prevents late faults and thereby saves convergence overhead.

Given the absence of documentation on the clock frequency impairment that results from near-threshold computing, we consider two extreme cases found in literature: a frequency reduction by a factor of 5 (i.e. 20% of the nominal frequency, denoted by f=0.2, best case) and a reduction by a factor of 10 (i.e. 10% of the nominal frequency, denoted by f=0.1, worst case), as discussed in Section 5.4.2.

The results illustrate the significance of matrix A. It is the biggest component of Jacobi in terms of memory consumption, $\mathcal{O}(N^2)$ compared to $\mathcal{O}(N)$ for B and X, and therefore presumably shows higher probability to be affected by faults than data segments with smaller footprints. Hence, we inject exactly one error in matrix A of Jacobi under NTV execution using the process outlined in Section 5.4.2. Furthermore, we present the worst case regarding the iteration before which a fault can happen, i.e. the last unreliably executed one. All results presented are averages over 50 random input data sets for statistical soundness, with an overall variance of 10^{-5} for the relative overhead of the number of iterations for fault-injected Jacobi runs.

5.5.1 Sequential Reliable Jacobi

First, we investigate replacing a single, reliable core by multiple unreliable cores (16, in case of our target platform) to execute Jacobi under the same power envelope, as supported by NTC in literature (per-core power reductions of 10×–50×). Hence, we assume their maximum power consumption to be equal. Figure 5.5 illustrates the

results of such a series of experiments, where in each experiment a fault happens in a different bit position. It shows the relative energy and time savings over a sequential, reliable run of Jacobi for all possible bit positions where faults may happen.

The results show that the effects of bit flip faults on energy and time may be categorized as follows:

A no observable loss in energy or time,

B observable loss in energy or time,

C divergence.

Moreover, this classification coincides with the bit position that is flipped within an IEEE 754 double-precision floating-point number. Faults happening in bit positions 0–32 can be categorized as class **A** since they show no effect on energy savings. This can be contributed to both the resilience of the Jacobi method to faults with small magnitudes, as well as the overall high precision of double-precision floating point numbers. Note that bits 0–32 are part of the mantissa and that a bit flip in these positions can affect normalized floating-point numbers by at most $2^{-20} \cdot 2^{e-1023}$, as illustrated by Equation (5.5). As a result, energy savings of 31% and 65% are possible for a 10× (f=0.1) and 5× (f=0.2) frequency reduction respectively.

Bit positions 33–54 and 63 are classified as class **B**, with higher bit positions up to 54 showing a higher impact on energy and time. The maximum possible floating-point error for this class is $\pm 2^{-1}$ for the mantissa (see Equation (5.5), bit 51) and a factor of $2^{\pm 2^{54}}$ for the exponent (see Equation (5.5), bit 54). As such, energy savings are reduced to e.g. 48% for f=0.2 in the worst case. Bit 63 is not part of the exponent but holds the sign, and as such a flip in this position induces an absolute error of $2x$ (see Equation (5.5)) for any floating point number x, resulting in average energy savings of 52%. class **B** warrants protection mechanisms if the user wishes to control the performance penalty incurred by NTV execution.

The missing data points at bit positions 55–62 are a member of class **C** since they are the highest significant bits of the exponent of a double-precision floating point number (induced errors between $2^{\pm 2^{55}}$ and $2^{\pm 2^{62}}$). For our setup, flips in any of these positions aside the diagonal of matrix A cause violations in Jacobi's convergence criterion, lead to divergence and have Jacobi break eventually with a non-silent *Inf* or *NaN* in most cases (see Section 5.3), which are easily detectable. Therefore, these bit positions should be protected in any case.

The correlation of time and energy savings in our results is a direct consequence of both our constant workload (i.e. arguably leading to constant power consumption)

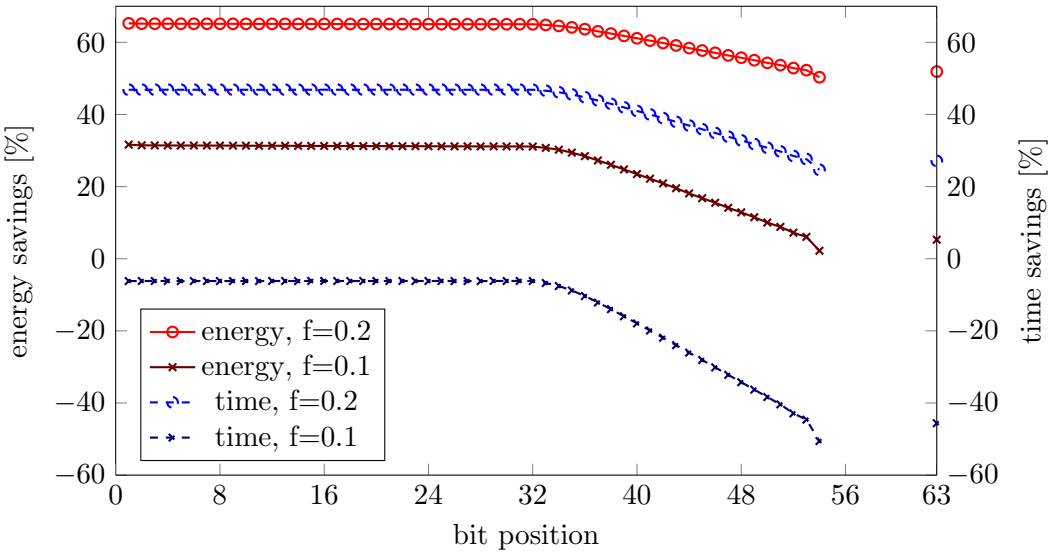


Figure 5.5: Relative energy and time savings of a parallel Jacobi run on 16 unreliable cores compared running on a single, reliable one. The missing data at bits 55–62 denotes divergence.

and the fact that we assume the same power budget for the unreliable cores and the reliable one.

5.5.2 Parallel Reliable Jacobi

Our second experiment compares parallel execution of the Jacobi code at NTV against a parallel run at super-threshold voltage. The results of this comparison, depicted by Figure 5.6, show an identical classification of bit positions compared to our previous experiment. Nevertheless, one should note the lower performance compared to the previous scenario, attributed to the frequency reduction of unreliable hardware operating at NTV by a factor of 5 to 10, as well as Jacobi's sub-linear parallel scaling behavior. Therefore, for class **A** faults, performance losses between 413% and 925% are visible. Second, energy savings increase slightly (up to 35% and 67% respectively). This is expected due to the more energy-expensive super-threshold voltage setup, since Jacobi does not scale linearly with increasing numbers of cores. As a result, the (linear) increase in power consumption is not fully compensated by a reduction in run time, hence leading to a higher energy consumption. In turn, the relative energy savings of the NTV execution increase.

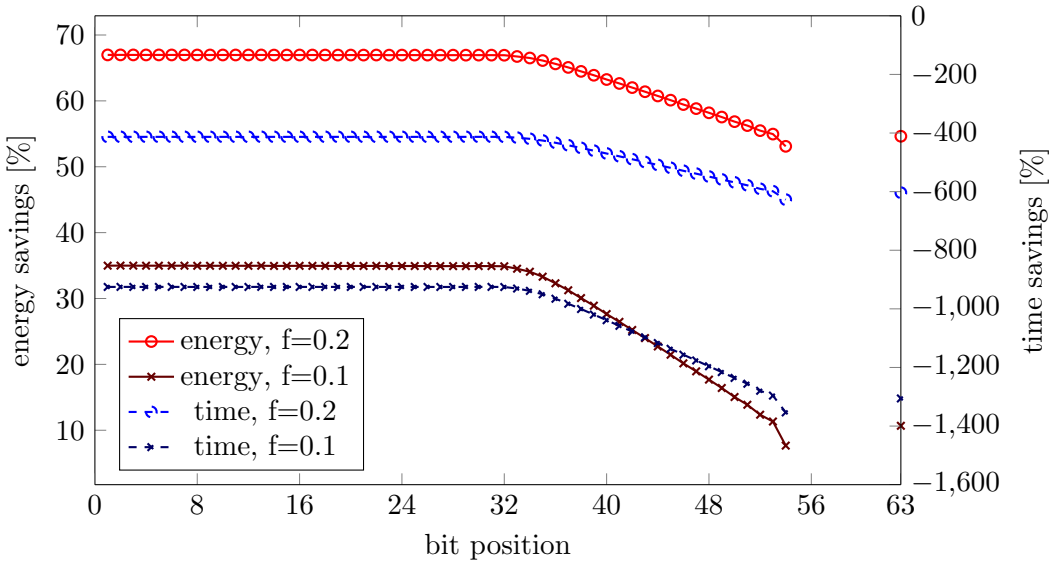


Figure 5.6: Relative energy and time savings of a parallel run of Jacobi on 16 unreliable cores compared to a parallel run on 16 reliable cores. The missing data at bits 55–62 denotes divergence.

5.5.3 Significance-dependent Reliability Switching

In our third scenario, we investigate whether a fraction of the last iterations of Jacobi are significant enough to justify running them reliably at super-threshold voltage, and if so, when a switch from parallel execution at NTV to sequential execution at super-threshold voltage should occur. On one hand, switching to sequential execution increases run time and energy consumption. On the other hand, running at super-threshold voltage prevents faults and guarantees convergence, without necessitating recovery iterations.

To that end, we run experiments where we switch from NTV execution to execution with super-threshold voltage at three points during a Jacobi run: 75%, 85% and 95% through completion. The intuition for this choice of switching points is Figure 5.1, where we observe that Jacobi experiences a significant slowdown when faults happen past the upper quartile of iterations. The energy consumption of each of these adaptive execution schemes is depicted in Figure 5.7. We show results for f=0.2, representing the best case in terms of performance impact of NTV. Switching at a late point in time shows the highest savings for class **A**, as bit flips in this class have little to no effect on Jacobi and do not justify the energy expense of running at super-threshold voltage. Hence, while later Jacobi iterations are more significant, the data shows this increase in significance to be too low to warrant protection.

However, the switching point coupled with the increased significance of later iterations affects the classification of bit positions. Switching later implies that faults in lower bit positions will have higher impact, since they happen in iterations with higher significance for convergence. As a result, switching at the 75% mark results in bit positions 0–47 to be categorized as class **A**, while the same class includes only bits 0–35 when switching at the 95% mark. This naturally changes the lower bit boundary of class **B** accordingly. However, it should be noted that it does not affect class **C**. If a bit flip in matrix A leads to divergence of Jacobi, it will always do so, regardless of when it happens. Overall, Figure 5.7 shows that switching at any of these three points in time does not pay off if the objective is to minimize energy consumption. The best strategy is to switch as late as possible (in our case at 95%), however all adaptive executions are outperformed by executing all iterations at NTV (65% energy savings vs. 43% for switching at 95%), illustrated by Figure 5.5.

Figure 5.8 shows execution time with adaptive execution, leading to similar observations for the classification of flipped bits and their impact compared to energy consumption. However, the best strategy from an execution time perspective depends on the position of the bit flip, which affects the impact of late class **B** faults. For example, when a flip happens at bit position 48, switching at the 75% mark yields a relative time loss of 44%, while the time loss is 60% when switching at the 85% mark and 78% when switching at the 95% mark. Furthermore, it is evident that switching at the 85% mark or earlier already yields performance losses due to the time spent in sequential execution.

Overall, our results lead us to conclude that while Jacobi does indeed show an increase in significance for later iterations, this increase is generally too small — within the boundaries of our experimental setup— to justify switching from parallel execution at NTV to sequential execution at super-threshold voltage.

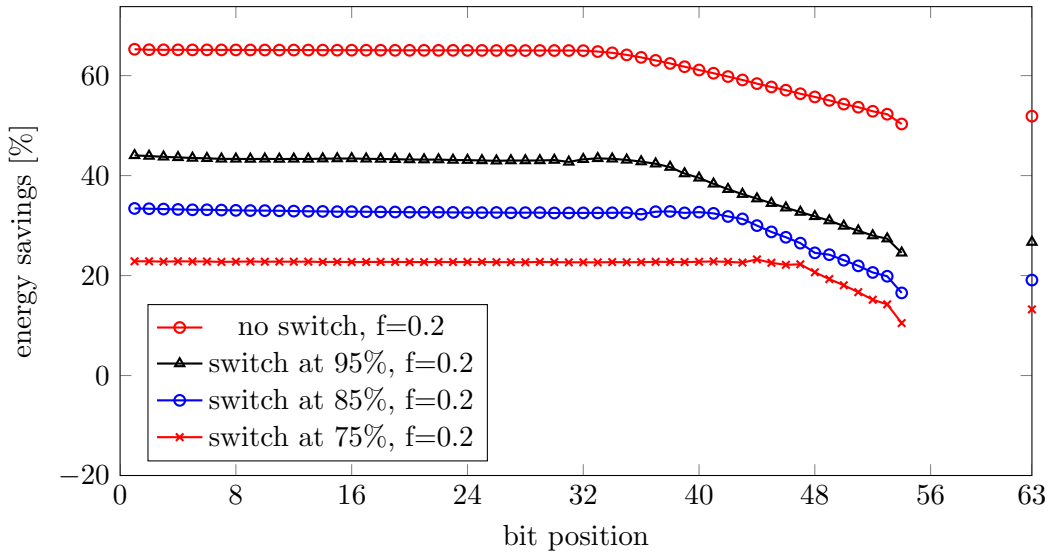


Figure 5.7: Relative energy savings of an adaptive reliable/unreliable run of Jacobi on 16 cores compared to a reliable, sequential one, for switching to reliable hardware at 75%, 85% and 95% run time. The missing data at bits 55–62 denotes divergence.

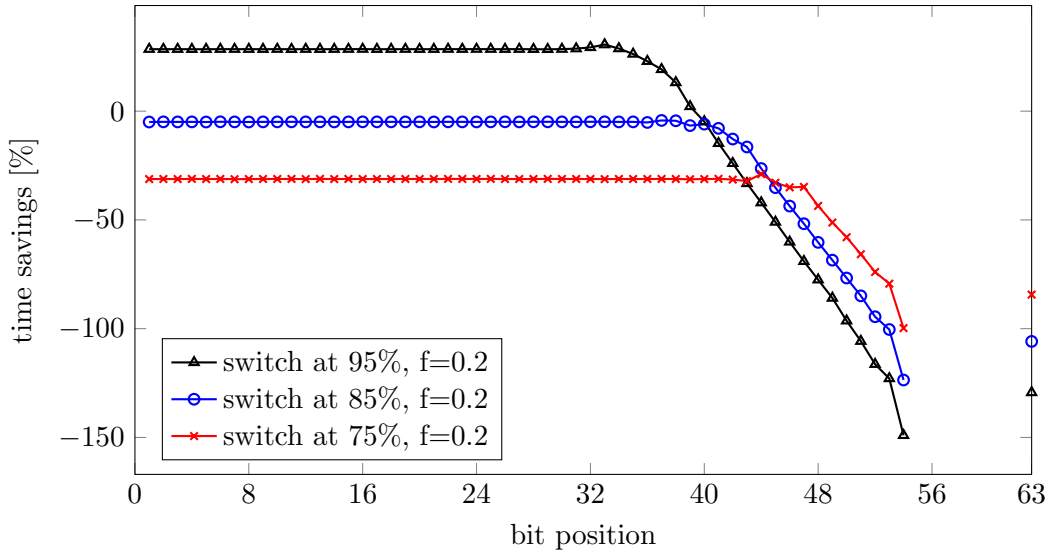


Figure 5.8: Relative time savings of a hybrid reliable/unreliable run of Jacobi on 16 cores compared to a reliable, sequential one, for switching to reliable hardware at 75%, 85% and 95% run time. The missing data at bits 55–62 denotes divergence.

5.6 Summary

In this work we explored the applicability and effect of near-threshold voltage (NTV) computation to a representative HPC code. We have shown that it can be a viable means of reducing the energy consumption, and that performance impairments caused by NTV can be mitigated via parallelism. We presented the notion of *significance-driven execution*, attributing varying significance to parts of a code or data and thereby deciding on whether they are a candidate for NTV computation or not. Our results show potential energy savings between 35% and 67%, depending on the use case. As such, significance-driven execution and NTV are a viable method of reducing the energy consumption in HPC environments without compromising correctness or performance. Future research opportunities include a detailed analysis of the effect of different degrees of parallelism, protection mechanisms for intolerable faults as identified in Section 5.5, and investigating and comparing the significance of additional iterative HPC codes. Additionally, ways of automatically determining the significance of code regions within compiler frameworks such as the Insieme compiler [65] could be explored.

Chapter 6

Compiler-assisted Execution Time and Energy Modeling

This chapter presents work on compiler-assisted analytical time and energy modeling of distributed memory parallel programs, to be submitted under the title *Automatic Compiler-assisted Performance and Energy Modeling for Message-Passing Parallel Programs* to Euro-Par 2017[1]. Using the Insieme compiler for analyzing distributed memory parallel programs, static information about these programs is derived. This static information is parametrized with regard to the input problem size and the machine size. Additionally, a single execution using the Insieme runtime system provides run-time data regarding execution time and energy consumption. Combining these static and dynamic data, models predicting execution time and energy consumption for larger problem and machine sizes are generated, and verified against measurements on two distributed memory parallel computers.

6.1 Introduction

Optimization of message-passing parallel programs is a wide-spread topic of interest and research, whether done manually by developers or automatically by tools such as compilers or runtime systems. Many modern optimization approaches are iterative in nature (e.g. auto-tuners [53]) and require the execution of a possibly large number of solution candidates to observe their non-functional behavior. Performance prediction is a useful technology to reduce the effort in the search for effective code optimizations.

To be automatic in nature, many prediction approaches —whether incorporat-

[1] http://europar2017.usc.es/

ing static, analytical information or purely observation-based— rely on a series of program executions to train their models. Related work shows that the number of training executions generally correlates with the lack of available static information. In this chapter, we introduce a novel prediction tool that incorporates compiler knowledge about the target application and compiler transformations to reduce model training overhead to a single program execution. We consider a large class of iterative message passing parallel programs that follow the bulk synchronous parallel (BSP) and single program multiple data (SPMD) models.

We use the Insieme compiler and runtime system presented in Chapter 3 to analyze an input program's source code. It extracts static information from the source code such as the structure and boundaries of loops, or message sizes of communication primitives. Insieme then invokes a single execution of the input program for a small problem and machine size on a target architecture. The runtime data of this program execution together with the static analysis information are then used to generate a parametrized model that can predict the execution time and energy consumption for larger problem and machine sizes on the given target architecture. The model can be ported to a new target architecture with a single execution of the input program. Additionally, only a handful of offline-measurable hardware parameters are required that specify cache properties, as well as bandwidths and latencies for the memory hierarchy and network.

The major contributions of this work are:

- a definition and automatic localization of target code regions that matches a large number of distributed memory parallel programs,

- a new method for the automatic generation of parametrized performance models for execution time and energy consumption that is problem and machine size sensitive based on compiler analysis and a single program execution, and

- evaluation, analysis and validation of this model for several target applications and a wide set of problem and target machine sizes on two hardware architectures.

The chapter is structured as follows: Section 6.2 lists related work and compares it to our method, which is described in Section 6.3. We detail on our evaluation methodology and experimental setup in Section 6.4 and provide results and model output analysis in Section 6.5. Finally, Section 6.6 concludes and provides an outlook for future work.

6.2 Related Work

There is a plethora of related works in the field of non-functional parameter prediction. While there are many ways of categorizing them, we focus on key aspects relevant to our work. First there are a number of automatic approaches that require little to no user action for obtaining predictions. The *COMPASS* framework by Lee et al [77] based on the *ASPEN* language by Spafford et al [113] is among the most prominent. Implemented in OpenARC, this tool offers execution time prediction using a domain-specific language. While ASPEN is capable of modeling message passing programs, to the best of our knowledge, they have not explored their prediction for energy metrics.

Bhattacharyya and Höfler presented *PEMOGEN*, an LLVM-based prediction tool for Fortran parallel programs in [18, 19], which features online modeling with the aim of reducing storage costs. Using a regression approach, they require a series of training executions, whereas we only need a single training execution. Also, the authors only demonstrated execution time prediction. Comparing their storage cost goal, our approach indeed stores static information and training data persistently, however this comprises profile information only at a very small storage footprint (a few hundred kilobytes for all input programs used in this work). Furthermore, the design of our model does not prohibit online prediction.

Hammer et al developed a semi-automatic tool called *Kerncraft* [55] which offers performance bottleneck predictions in shared memory environments. However, to simplify analysis, their acceptable input programs are limited to simple stencil and streaming kernels that are written in a subset of C99. In contrast to that, we support a larger set of message passing programs and pose no restrictions on the C input accepted by the Insieme compiler.

PALM [117] offers analytically-derived execution time predictions, however it relies on user annotations for describing the model and important parameters.

Although some of these works may be capable of providing energy predictions, to the best of our knowledge, none of the above have explored this. A recent work using *ExaSAT* [133] includes very limited energy concerns in its considerations, however lacking actual measured energy consumption data. Contrary to that, there are a number of models that focus more strongly on energy prediction, whether build empirically in an analytical fashion [93, 104], using regression models [93] or neural networks [130]. However, none of these provide execution time and energy predictions for message passing programs in a fully automatic fashion without any user directives except for specifying the problem sizes and machine sizes to be explored.

Moreover, there are a number of works that provide predictive models although

specifically tailored to aid in aspects such as task aggregation [79], or resilience and reliability [86, 141].

Finally, there are a number of invaluable pen-and-paper model works that do not aim at predicting the execution time or energy consumption for code regions or full programs, provide a characterization of hardware behavior under certain conditions. Examples are *Roofline*, which has been applied to execution time [140] and energy consumption [26], or the *ECM* model [115]. We use their concepts as the foundation for our proposed method as mentioned throughout this work.

6.3 Model

In order to properly motivate the presented model, we first need to establish several goals that we aim for. As discussed in Section 6.1, one of the possible applications of our prediction method are iterative methods such as auto-tuners. To suit this use case, we need to consider both the overhead of our method and its accuracy. To that end, our model should fulfill the following goals:

1 low resource consumption (i.e. time, energy, storage) in training,

2 low resource consumption in deployment and evaluation, and

3 automatically applicable to a range of iterative distributed memory parallel applications without requiring any user interaction.

Goal **1** can be achieved by shifting the requirements of large training data sets to static analysis performed during compilation. The compiler can inspect the input program with regard to properties such as the structure of loops and communication points, loop iteration counts, or message buffer sizes. This allows us to reduce the number of training runs otherwise needed to e.g. obtain supporting points for regression models. Also, having a compiler examine the code structure automatically removes the need for any user directives. We meet Goal **2** by building analytical, parametrized models that only require a few constants to be filled in for evaluation. Finally, to satisfy Goal **3** we apply our approach to a range of input applications and validate all predictions with measurements.

6.3.1 Method

Our approach targets specific code regions of distributed memory parallel programs, and we rely on static analysis within the compiler for region identification and subsequent modeling. Based on the software model presented in Section 2.2, we define a code region as

$$
\begin{aligned}
s &::= exp \\
&\mid \texttt{for(}\ var\ \texttt{=}\ exp\ \texttt{..}\ exp\ \texttt{:}\ exp\ \texttt{)}\ s \\
&\mid \texttt{g(}\ exp\texttt{,}\ exp\texttt{,}\ \ldots\texttt{,}\ exp\ \texttt{)} \\
&\mid \texttt{\{}\ s\texttt{;}\ s\texttt{;}\ \ldots\texttt{;}\ s\ \texttt{\}} \\
exp &::= \texttt{acc(}\ exp\texttt{,}\ \texttt{\{}\ exp\texttt{,}\ \ldots\texttt{,}\ exp\texttt{\}, \{}\ exp\texttt{,}\ \ldots\texttt{,}\ exp\ \texttt{\})} \\
&\mid var \\
&\mid num \\
var &::= char\ var \mid char \\
char &::= \texttt{a-z} \mid \texttt{A-Z} \\
num &::= \texttt{1-9}\ num \mid \texttt{0-9}
\end{aligned}
$$

where s is a statement, *exp* an expression, *var* a variable, and *acc* an accessor function. For completeness, *char* denotes an identifier and *num* a numeric literal.

This grammar allows us to form code regions that consist of loops (*for*) with an iterator variable and fixed (but not necessarily known) lower and upper bounds and step expressions, external function calls (*g()*) and compound statements ($\{\ldots\}$). We choose this selection of statements since for loops and external function calls (the definition of which is unknown to the compiler) are good candidates for high resource consumption, whereas compound statements are included for composition. We furthermore include an accessor function *acc* to be able to model array and pointer subscript expressions, which takes as arguments first a base expression (i.e. the pointer or array) and two lists of index expressions, the first for read operations and the second for write operations. This accessor function aids us in identifying loops with steady-state cache properties (further detailed in Section 6.3.4). Note that while the grammar does not prohibit write access to the loop iterator variable inside the loop body, our approach requires the number of iterations to be immutable upon execution of the first iteration.

6.3.2 Automatic Region and Parameter Detection

For our approach to be fully automatic, we need to identify code regions of interest without user interaction. In addition, contrary to other approaches that rely on user directives to identify significant code properties such as loop boundaries, we want to automatically obtain the properties described in Section 6.3.1 at compile-time in parametrized form. Figure 6.1 illustrates the overall architecture and workflow, based on the Insieme compiler and runtime system already presented in Chapter 3.

We will briefly outline this architecture, with each step elaborated on in detail in the remainder of this section. An input program is loaded by the compiler and

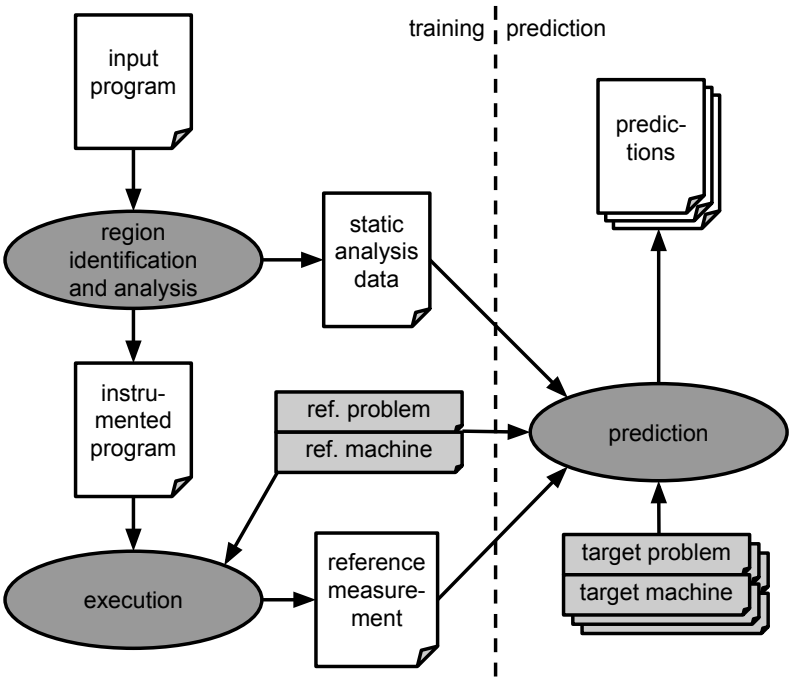


Figure 6.1: Workflow of our prediction approach.

target regions are automatically identified and static analysis information is derived. The program is then instrumented to be executed on the target machine using a small reference problem size and reference machine size. The collected reference measurement data is handed to the predictor in combination with the static analysis information and the reference problem and machine sizes, enabling it to predict the measured metrics for larger problem sizes and machine sizes.

We use a pattern-based approach [66] to search an input program for target code regions, which are loop nests that meet the following criteria:

- all loops in the nest have fixed (but not necessarily known) lower and upper bounds and increments, and
- there are communication primitives anywhere within the loop nest.

Communication primitives are external function calls, identified by their signature (function identifier and parameter types). This allows easy porting of our region detection to support arbitrary communication libraries. For this work, we explore MPI blocking and non-blocking communication primitives. For the latter, we capture both the non-blocking communication operation itself, as well as the corresponding wait function call (identified by its matching `MPI_Request*` argument) and treat

```
1  for(int n=1; n<nsteps-1; ++n) {
2
3    if(rank<size-1) {MPI_Send(...); MPI_Recv(...);}
4    if(rank>0) {MPI_Recv(...), MPI_Send(...);}
5
6    for(int x=1; x<gridsize-1; ++x) {
7      for(int y=1; y<gridsize-1; ++y) {
8        grid[n%2][x][y]=(grid[(n-1)%2][x-1][y] +
9                         grid[(n-1)%2][x+1][y] +
10                        grid[(n-1)%2][x][y-1] +
11                        grid[(n-1)%2][x][y+1])/4.0;
12   }
13 }
```

```
1  for(n=1..nsteps:1) {
2    MPI_Send(rank<size-1, ...);
3    MPI_Recv(rank<size-1, ...);
4    MPI_Recv(rank>0, ...);
5    MPI_Send(rank>0, ...);
6    for(x = 0..gridsize:1) {
7      for(y = 0..gridsize:1) {
8        acc(grid, {((n-1)%2,x-1,y),((n-1)%2,x+1,y),
9            ((n-1)%2,x,y-1),((n-1)%2,x,y+1)}, {});
10       acc(grid, {}, {(n%2,x,y)})
11     }
12   }
13 }
```

Figure 6.2: A simplified version of a *jacobi* stencil input code shown in C (top) and in our model representation (bottom).

them as a single unit. Since communication primitives are frequently nested inside a conditional statement (e.g. to enforce border limits for neighbor exchange patterns), we prepend the conditional expression to the list of parameters of g. If it is not enclosed in a conditional, this first parameter is set to true. We also automatically extract arithmetic formulas for source or target buffers of communication operations as well as the type and size of the data transmitted, since all of this information is encoded in a function's signature. For loops, we similarly extract the lower and upper bounds as well as the increment. Figure 6.2 shows a simple code example in both C and the textual representation of our model.

6.3.3 Automatic Parameter Extraction

We want to offer a prediction tool that does not require the users to alter their work flow compared to running their input program, or depend on any user directives beyond providing problem size and machine size program parameters. To achieve this, the prediction tool is intended to be a wrapper for an input program and requires relating the parameters described in Section 6.3.2 to the user's input program parameters. This is done using a *compiler integrated* analysis framework. This work-in-progress framework provides various data-flow analyses (DFA) and is based on the *constraint-based analysis* approach described in [91]. It is capable of inter-procedural analyses, which is essential for fulfilling Goal **3**.

Identified parameters such as loop boundaries, loop step expressions, and communication buffer sizes can be modeled by the analysis framework as arithmetic formulas. Such an arithmetic formula is constructed by traversing the input code from a parameter back to its declaration and keeping track of applied operations and operands. This is done automatically by the compiler without any user interaction required, and yields arithmetic expressions that compute e.g. loop boundaries for given input program parameters.

Note that only basic integer arithmetic is supported at the moment (i.e. addition, subtraction, multiplication, modulo, and division if the remainder evaluates to zero), as this is sufficient for our use case of evaluating expressions that compute e.g. grid slice sizes, the MPI ranks of neighbors, and similar parameters.

6.3.4 Execution Time Prediction

Our method can be extended to a number of metrics, however, without loss of generality, we present predictors for execution time in this section and energy consumption in Section 6.3.5.

Algorithm 6.1 Statement identification and selection.

1: $R \leftarrow$ identify all target code regions (loop nest with communication primitives inside)
2: $I \leftarrow \emptyset$ ▷ statements to be instrumented, measured, predicted
3: **for** $r \in R$ **do**
4: $C \leftarrow$ identify all communication primitives in r
5: **for** $c \in C$ **do**
6: $L_c \leftarrow$ all loop nests appearing in the same compound statement as c
7: $I \leftarrow I \cup \{c\} \cup L_c$
8: **end for**
9: **end for**

Algorithm 6.2 Prediction algorithm.

1: $R \leftarrow$ all target code regions as identified by Algorithm 6.1
2: $I \leftarrow$ all instrumented statements as identified by Algorithm 6.1
3: $\text{prediction}_i \leftarrow 0$ for all $i \in I$
4: $\text{prediction}_r \leftarrow 0$ for all $r \in R$ ▷ final output of prediction
5: **for** $i \in I$ **do** ▷ training phase
6: instrument statement i
7: **end for**
8: perform single execution of input program for reference problem and machine size
9: **for** $i \in I$ **do** ▷ prediction phase
10: $T_{i,\text{comp}} \leftarrow$ assume ideal scaling using reference and target loop iteration counts
11: $T_{i,\text{mem}} \leftarrow$ replace communication primitives, reduce time loop iteration count, predict cache misses
12: $T_{i,\text{comm}} \leftarrow$ build communication graph, evaluate with hardware model
13: $T_{i,\text{all}} \leftarrow$ combine individual predictions using γ and ϕ
14: **end for**
15: **for** $r \in R$ **do**
16: $\text{prediction}_r \leftarrow p(r)$
17: **end for**

To predict a metric for a target code region, we predict this metric for selected individual statements and then aggregate the metric data to obtain values for the entire code region. Algorithm 6.1 elaborates on the specific selection process of such individual statements. Once these statements are identified, we predict execution time as outlined by Algorithm 6.2, whose separate steps will be described subsequently. Our execution time prediction follows the idea of the Roofline model [140] and ECM [55] in the sense that we attribute individual busy times for all hardware components defined in Section 2.1 and interpret these results as the lower bound of

the overall execution time:

$$T_{\text{all}} = \gamma\big(\phi(T_{\text{comp}}, T_{\text{mem}}), T_{\text{comm}}\big)$$
$$T_{\text{mem}} = \phi(T_{\text{cache}_0}, T_{\text{cache}_1}, \ldots, T_{\text{RAM}})$$

where γ and ϕ are aggregation functions with $\phi = \max$ for execution time prediction and $\gamma = \sum$ in case of blocking communication and $\gamma = \max$ in case of non-blocking communication. T_{comp} represents the time a *core* is busy with computation, T_{mem} is the active time of the memory hierarchy (which, for this use case, includes *caches* and *RAM*) and T_{comm} is the amount of time for which messages are exchanged. Throughout the remainder of this section, we will detail on the prediction each of these metrics to determine T_{all}.

Computation time

We start with predicting T_{comp}. For this we employ a two-stage approach: First, we collect a single measurement — so-called *reference measurement* — for the previously selected statements in our target code region for a small problem and machine size on a given machine architecture. This is necessary since our work is based on a source-to-source compiler, and therefore we lack any knowledge about to-binary compiler optimizations (e.g. vectorization) or hardware computing speeds. Example 2.6 in Section 2.1.3 has shown the large impact these aspects can have on the execution time. Combining this reference measurement with static analysis information from the compiler, we can extrapolate the behavior of these statements for previously unseen target problem and machine sizes. Knowing all loop iteration counts for both the reference and the target problem and machine sizes (obtained via the analysis described in Section 6.3.3), we can compute T_{comp} assuming ideal scaling and hence equal memory hierarchy or communication contention.

Memory hierarchy time

Only predicting T_{comp} is insufficient, as this naturally ignores e.g. increased memory hierarchy traffic for larger problem sizes or increased per-rank cache sizes for larger machines. Hence, we require a prediction of T_{mem}. Since Goal **3** is to cover as many applications as possible, relying on analytical approaches such as ECM [115] is impractical since they are confined to subsets of our code regions by either limiting the acceptable input structure (e.g. only perfect loop nests) or syntax (e.g. basic array subscripts). For this reason, we would like to employ ready-to-use cache simulators to obtain cache miss data and combine these with measured cache band-

width information to arrive at T_{mem}. Nevertheless, while cache simulators do not have the aforementioned limitations of analytical models, they are usually orders of magnitude slower at producing results, which requires us to reduce the overhead for practical use.

For this reason, we introduce a compiler analysis component. First, let $\mathbb{S}$ be the set of all statements, then

$$\sigma : \mathbb{S} \to \mathcal{P}(\mathbb{S})$$

is a function that, for a given statement s, returns a set holding all statements contained within s. Second, let s be a statement, then

$$\text{isTimeLoop} : \mathbb{S} \to \mathit{Bool}$$

$$\text{isTimeLoop}(s) =$$

$$\begin{cases} \text{true}, & \text{if } s = \text{for}(x = \mathit{lower}..\mathit{upper} : \mathit{step})\ s' \\ & \text{and } \ \text{acc}(\_, R, W) \notin \sigma(s'), x \in R \cup W \\ \text{false}, & \text{otherwise} \end{cases}$$

is a function which identifies a for loop as a *time loop* if the loop iterator does not appear in any accessor function (i.e. does not occur in any array or pointer subscript) within the body of the loop. Our hypothesis is that, first, most HPC codes do have such loops that iteratively compute e.g. physical processes over time. Second, we further hypothesize these loops to have foreseeable effects on the cache behavior of target code regions, since the first iteration warms up the CPU caches after which the system is in a steady state (barring any noise from the OS). Therefore, it should be possible to simulate only the first couple of iterations and extrapolate to their full number assuming a simple linear relationship. Hence, for a time loop with $n > 1$ iterations, the overall misses can be approximated by

$$\sum_{x=0}^{n} M_x \approx (n-1) \cdot \frac{\sum_{x=1}^{n-k} M_x}{n-k-1} + M_0$$

where M_x denotes the number of cache misses induced by loop iteration x, and k is the number of omitted loop iterations. If our assumptions hold, we can use the compiler to transform our target code regions to reduce the number of iterations of previously identified time loops and yet obtain well-approximated cache miss counts. However, there is a trade-off to be managed between large k (causing less overhead

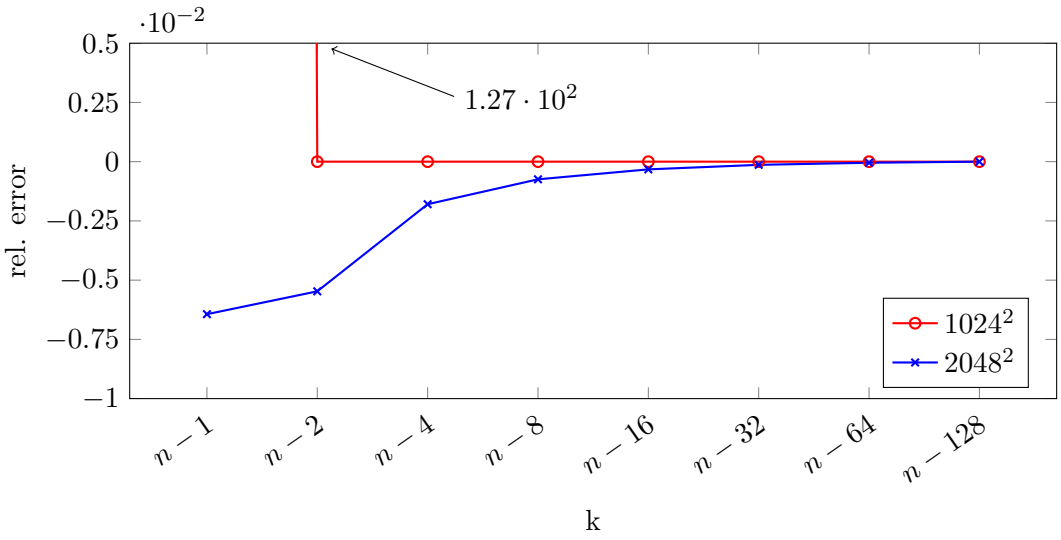


Figure 6.3: Relative L3 cache miss prediction error for *jacobi* for $n = 128$ and decreasing k for two selected problem sizes. The cache was set to be 20 MB in size with a line size of 64 bytes and a 20-way associativity, representative of modern Intel architectures.

at the price of lower accuracy) and small k (yielding higher accuracy at the expense of more overhead).

Figure 6.3 shows the results of evaluating this hypothesis with cachegrind [90] for decreasing k for the *jacobi* implementation shown in Figure 6.2. We simulated an L3 cache of 20 MB, a line size of 64 bytes and a 20-way associativity, representative for modern Intel architectures (e.g. an Intel Xeon E5-4650 processor [28]), and chose two problem sizes such that the smaller one fits in the cache whereas the larger one does not. Figure 6.3 confirms our expectations that using only a single iteration of the time loop ($k = n - 1$) is not sufficient for problem sizes that fit in the cache (yielding a relative error of $1.27 \cdot 10^2$), however increasing the number of iterations beyond 2 ($k = n - 2$) does not amortize the increase in overhead, with a relative error of already only $-5.47 \cdot 10^{-3}$. Since we verified similar behavior for all our input programs, we always choose $k = n - 2$ for the work presented here, which allows a substantial reduction in simulation overhead.

However, there are additional opportunities for decreasing the overhead. Since our input programs follow the BSP/SPMD models, we can simulate a single MPI rank and extrapolate to the full size of the target machine, effectively reducing the CPU time of our simulation by a factor of the target machine size. Because it is impossible to execute arbitrary MPI programs with a single rank, the communi-

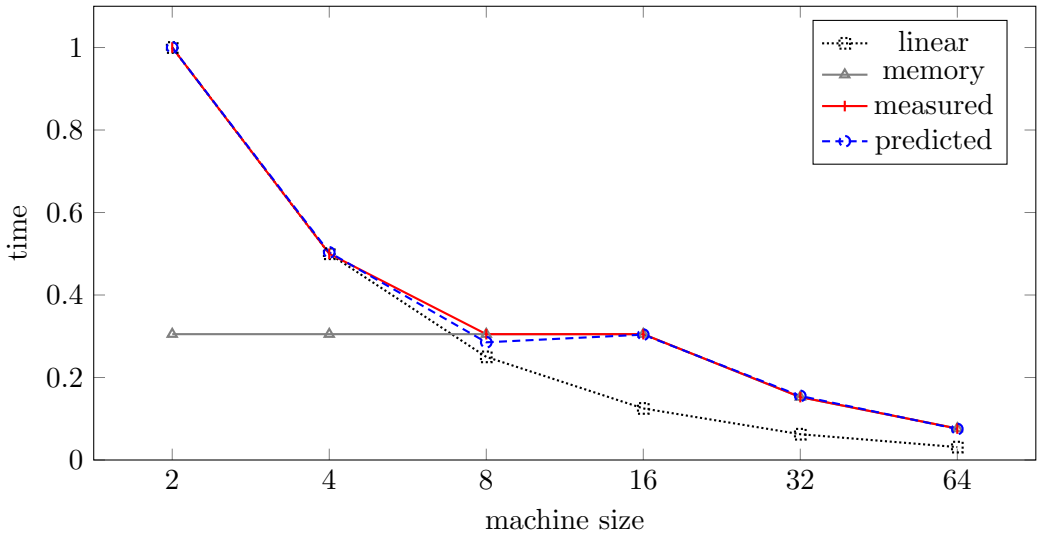


Figure 6.4: Execution time prediction breakdown of the *jacobi* input program for $N = 32768$ for increasing machine sizes.

cation primitives require special treatment. Ideally, we would like to remove the communication while keeping the cache behavior of the primitives and therefore of the surrounding code region (e.g. buffer reuse for computation and communication). For this reason, we define a transformation function to replace them as follows — for brevity we only show the semantics of send and receive primitives:

$$\text{transformCommPrimitive} : \mathbb{S} \to \mathbb{S}$$

$$\text{transformCommPrimitive}(s) =$$

$$\begin{cases} \text{for}(x = 0..size : 1)\ \ \text{acc}(buffer, \{x\}, \{\}), & \text{if } s = g(\_, buffer, size, \_, \_, \_, \_) \\ & \text{and } g = \text{MPI\_Send} \\ \text{for}(x = 0..size : 1)\ \ \text{acc}(buffer, \{\}, \{x\}), & \text{if } s = g(\_, buffer, size, \_, \_, \_, \_, \_) \\ & \text{and } g = \text{MPI\_Recv} \\ \vdots & \\ s, & \text{otherwise} \end{cases}$$

All communication primitives are replaced by transformCommPrimitive with corresponding linear reads and writes of *buffer* as they are expected occur in the actual library calls.

Furthermore, since our input programs follow the SPMD model and therefore are homogeneous in their workload but lack application data sharing (contrary to e.g. OpenMP programs), we hypothesize that shared cache effects can be emulated by reducing the available shared cache size per core by the factor of the number of cores sharing this cache and participating in the computation.

Figure 6.4 confirms this theory, as the predicted memory hierarchy time of the target code region matches the actual execution time when the code region is memory bound (machine sizes 8–64). It should be noted that one could also think of simply executing the transformed application on the target hardware instead of executing it in a cache simulator for performance reasons. However, first this removes the possibility of simulating the effect of reduced cache sizes. Second, hardware-specific performance counters would be required to ascertain e.g. the number of cache misses, whereas we only rely on information that can be obtained automatically via the `cpuid` instruction (cache levels, cache sizes, line sizes, associativities). Moreover, using a cache simulator allows us to evaluate our model for multiple target problem and machine sizes in parallel without the risk of measurement perturbation, increasing model prediction throughput. The compiler also confines cache simulation to the target code regions by inserting control statements that start the cache simulation only prior to execution of the first target regions, and exit the input program after the last one. The final cache miss data is then combined with cache and memory bandwidth information from the model defined in Section 2.1 (obtained via offline measurements, once per target architecture) to compute T_{mem}.

Communication time

Finally, we require a prediction for T_{comm}. Let m be the size of a message sent from x to y, then (x, y, m) is a tuple denoting this message transfer. We can automatically derive these parameters as described in Section 6.3.2 from communication primitives as x, y and m are all encoded in g. Note that we require x, y, and m to depend only on integer operations as described in Section 6.3.3, program parameters such as the problem size, and `MPI_comm_rank` and `MPI_comm_size`. We require our input programs to match the BSP model with regard to communication adhering to supersteps (i.e. an iteration of the time loop) and perform the same communication pattern each iteration. Then, we can evaluate the arithmetic formulas for x, y and m for each rank x, which are then examined for widely used communication patterns such as neighbor exchange. Note that collective operations can also be handled assuming their communication pattern is known for a given message size and rank size [56]. We use this information in combination with bandwidth and

latency information of the hardware model defined in Section 2.1 and a given rank-core mapping policy to derive the data transfer time T_{comm}.

Aggregation

At this point, we predicted T_{comp}, T_{mem}, T_{comm} and hence T_{all} for individual statements. To get predictions for the entire target code region, we require a recursive aggregation function. Let A be the type of the metric to be predicted (in our case execution time or energy consumption) and *eval* an evaluation function for arithmetic expressions. Then

$$\text{get} : \mathbb{S} \to A, \quad p : \mathbb{S} \to A, \quad \text{data} \in A$$

$$\text{get}(s) = \begin{cases} \text{data}, & \text{if data available} \\ p(S), & \text{otherwise} \end{cases}$$

$$p(s) = \begin{cases} \text{get}(s') \cdot \text{eval}(\frac{upper - lower}{step}), & \text{if } s = \text{for}(x = lower..upper : s)\ s' \\ \sum_{x=0}^{n} \text{get}(s_x), & \text{if } s = \{s_0, s_1, \ldots, s_n\} \\ 0 & \text{otherwise} \end{cases}$$

are functions retrieving and aggregating data for loops and compound statements, returning the identity element (0 for this work) for all other cases.

6.3.5 Energy Prediction

Energy prediction can be done in a similar fashion to the one described in Section 6.3.4, except that we use energy measurements as the reference for our extrapolation, and we always choose $\phi = \gamma = \sum$ to express each hardware component's contribution to the overall energy consumption (contrary to busy times of hardware components, which can overlap). Hence, we assume the following relationships between energy (E), power (P) and execution time (t):

$$E_{\text{all}} = \sum \Big(\sum (E_{\text{computational}}, E_{\text{memory}}), E_{\text{communication}} \Big)$$

while for each hardware component x we consider

$$E_u = \overline{P}_{x,\text{idle}} \cdot t_{x,\text{idle}} + \overline{P}_{x,\text{load}} \cdot t_{x,\text{load}}$$
$$\overline{P}^{\text{ref}}_{x,\text{load}} = \frac{E^{\text{ref}}_{x,\text{load}}}{T^{\text{ref}}_{x,\text{load}}}$$
$$t_{x,\text{idle}} = \begin{cases} t_{\text{all}} - t_x, & \text{if } t_x < t_{\text{all}} \\ 0, & \text{otherwise} \end{cases}$$

where $\overline{P}_{x,\text{idle}}$ is the average idle power consumption of hardware unit u (to be measured offline, once per target architecture), $T_{u,\text{load}}$ is obtained via prediction as described in Section 6.3.4, and $T^{\text{ref}}_{u,\text{load}}$ and $E^{\text{ref}}_{u,\text{load}}$ are execution time and energy consumption as measured by our single reference measurement. Note that this implies that the reference measurement (and hence chosen problem size and machine size) is representative of larger problem sizes or machine sizes in terms of hardware component usage. We then predict $\overline{P}^{\text{target}}_u$ of a specific target hardware unit u of reference computer $\mathcal{M}$ and a target computer $\mathcal{M}'$ as

$$\overline{P}^{\text{target}}_u = \frac{\overline{P}^{\text{ref}}_u}{|\{x \in \mathcal{M} | x = u\}|} \cdot |\{x' \in \mathcal{M}' | x' = u\}|.$$

Using the aggregation formula of Section 6.3.4, we are able to predict E_{all} for a given target code region.

6.4 Experimental Setup

We implemented a prototype of our described work as part of the Insieme Compiler and Runtime System presented in Chapter 3. The compiler provides all needed facilities required to analyze an input program's source code, transform it, and generate an instrumented version. The runtime system then executes and measures the instrumented application and feeds back all measured data to the compiler. Our input codes are C/MPI distributed memory parallel applications, the model presented in Section 6.3 — like the compiler — is implemented in C++. To increase prediction throughput of the model, our reference implementation relies on `std::async` and allows the simultaneous prediction of multiple problem and machine sizes.

The experimental testbed used for our experiments consists of two distributed memory machines named *ortlerSandy* and *ortlerIvy*, Table 6.1 lists their characteristics. The CPU clock frequency was fixed as listed in Table 6.1, and Hyperthreading was disabled on all machines. The nodes are connected via a dedicated Gigabit

Table 6.1: Machine characteristics.

property	ortlerSandy	ortlerIvy
nodes	4	4
CPUs per node	4x E5-4650 2.7 GHz	2x E5-2690 v2 3.0 GHz
cores per node	32	30
cache sizes	priv.: 32 KB, 256 KB, shared: 20 MB	priv.: 32 KB, 256 KB, shared: 25 MB
RAM	256 GB	128 GB
OS	CentOS 6.7, 2.6.32-573	CentOS 6.5, 2.6.32-431
compiler	gcc 5.1 -O3	gcc 5.1 -O3
MPI	Open MPI 1.10.2	Open MPI 1.10.2

Table 6.2: Input programs, and properties.

program	description	comp.	memory.
cg	conjugate gradient	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
homb	laplace solver	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
jacobi	2d jacobi solver	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
mm_ijk	matrix multiplication, ijk loop order	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
mm_ikj	matrix multiplication, ikj loop order	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
shs	simple hyperbolic solver	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
stencil3d	generic 3x3x3 3d stencil	$\mathcal{O}(N^3)$	$\mathcal{O}(N^3)$

Ethernet network. We enforced process binding to cores, with a mapping that uses at least 2 nodes with one core each (machine size 2), and then increasing first the number of cores on sockets already in use before employing new sockets. Similarly, new nodes are only added when all current sockets are fully utilized.

Measurements were obtained via x86's *rdtsc* assembler instruction for execution time and Intel's *RAPL* interface for energy consumption. The latter offers a data resolution of 15.3 microjoules and time resolution of 1 millisecond, and related work has shown it to be accurate enough for our purpose [54]. However, since it only captures CPU packages, we present energy prediction results of the CPUs. The cache simulator in use for cache miss prediction is cachegrind 3.11 [90].

A selection of input programs for our work, their basic properties as well as tested problem sizes are listed in Tables 6.2 and 6.3. *cg* is an iterative conjugate gradient solver, *homb* [61] the Hybrid OpenMP MPI Benchmark, *jacobi* a two-dimensional jacobi solver, *shs* [8] the Simple Hyperbolic Solver, computing the compressible Navier-Stokes equation on a two-dimensional domain, and *stencil3d* a generic 3x3x3

Table 6.3: Input program problem sizes.

program	iter.	problem sizes (S: ortlerSandy, I: ortlerIvy)
cg	1000	S: 1024 2048 3072 4096 5120 6144 7168 8192 I: 1040 2080 3120 4160 5200 6240 7280 8320
homb	100	S: 1024 2048 4096 8192 16384 32768 I: 960 1920 3840 7680 15360 30720
jacobi	128	S: 1024 2048 4096 8192 16384 32768 I: 960 1920 3840 7680 15360 30720
mm_ijk	50	S: 448 640 960 1280 1600 1920 2240 2560 2880 I: 400 600 800 1000 1200 1400 1600 1800 2000
mm_ikj	50	S: 448 640 960 1280 1600 1920 2240 2560 2880 I: 400 600 800 1000 1200 1400 1600 1800 2000
shs	40	S: 128 256 512 1024 2048 4096 I: 80 160 320 640 1280 2560
stencil3d	100	S: 128 256 384 512 640 I: 160 240 320 400 480

3d stencil. To show the wide applicability of our model, we also include two matrix multiplication kernels, *mm_ijk* and *mm_ikj*, which exhibit substantially different cache behavior. While both perform a matrix-matrix multiplication, *mm_ikj* uses an i-k-j loop order that eliminates costly column-wise array traversal. All of these codes are written in C and rely on MPI for parallelism (*homb* also offers OpenMP parallelism, which we disabled for our experiments).

To validate the model, we predict execution time and energy consumption for a number of target problem and machine size parameter combinations and also measure the same parameter combinations. To minimize any inaccuracy caused by external load such as the operating system, all reported measurement data represents the median over 5 runs. For predicted data however, since our model is fully deterministic, a single run is sufficient. We furthermore evaluate the accuracy of the model by computing the normalized root-mean-square error (NRMSE) and the coefficient of determination (R^2).

6.5 Results

To illustrate and analyze the performance of the generated models, we will focus on the first of our input programs, a simple two-dimensional Jacobi implementation (*jacobi*), since it is well-studied and shows all our considered aspects of modeling distributed memory parallel applications. Subsequently, we will show results for all other input programs to demonstrate the general applicability of our approach.

Figure 6.5 presents the results of predicting execution time for *jacobi* with a reference problem size of 1024 and a reference machine size of 2 (two nodes with

Table 6.4: Overall model errors for *ortlerSandy*.

	time		energy	
code	NRMSE	R^2	NRMSE	R^2
cg	0.018	0.978	0.038	0.928
homb	0.035	0.955	0.047	0.929
jacobi	0.020	0.980	0.068	0.905
mm_ijk	0.044	0.929	0.080	0.812
mm_ikj	0.025	0.980	0.053	0.931
shs	0.082	0.945	0.068	0.947
stencil3d	0.053	0.940	0.092	0.839
mean	0.040	0.950	0.064	0.899

one core each, as per our mapping policy detailed in Section 6.4) on *ortlerSandy*. The shading denotes the relative error of predicting T_{comp}, $\max(T_{\text{comp}}, T_{\text{mem}})$, and $\sum(\max(T_{\text{comp}}, T_{\text{mem}}), T_{\text{comm}})$ compared to actual measurements, and illustrates the incremental increase in accuracy for each prediction step added. The shapes indicate a mainly memory-hierarchy-bound program, with prediction of only T_{comp} yielding a mean relative error of 0.69. The memory contention is evident by the visible column-like separation of machine sizes 2–4, 8 and 16–64, due to the fact that *jacobi* is already memory bound at machine size 8 (as also indicated by Figure 6.4) for problem sizes larger or equal to 4096 (resulting in a working set of 32 MB for 20 MB of L3 cache on *ortlerSandy*). Including the prediction of T_{mem} already substantially improves accuracy for these cases, lowering the overall mean error to 0.25. However, a number of cases with small problem sizes but large machine sizes are naturally not predicted properly, as communication time contributes a major part of T_{all} here, due to the large number of messages exchanged and short T_{comp} for small workloads shared among many cores. Including T_{comm} in our prediction also covers these cases, lowering the overall mean error to 0.06.

Tables 6.4 and 6.5 present the overall results for both time and energy for all our input programs on both *ortlerSandy* and *ortlerIvy*. As the data illustrates, our prediction generally achieves higher accuracy across all input programs for execution time (mean R^2 of 0.95) compared to energy consumption (mean R^2 of 0.90). This is a result of the mapping of our σ and ϕ functions, partially described by the differences between Roofline [140] and ECM [115]. When predicting the execution time and thus aggregating the maximum over multiple sub-predictions, only the largest element directly impacts the result, provided the relative order of the sub-predictions is correct. Contrary to that, energy consumption is aggregated as the sum over all sub-predictions, requiring high-quality predictions for all of them for

Table 6.5: Overall model errors for *ortlerIvy*.

	time		energy	
code	NRMSE	R^2	NRMSE	R^2
cg	0.025	0.984	0.028	0.979
homb	0.008	0.996	0.045	0.947
jacobi	0.014	0.995	0.091	0.849
mm_ijk	0.068	0.856	0.078	0.840
mm_ikj	0.031	0.975	0.076	0.893
shs	0.060	0.971	0.058	0.932
stencil3d	0.016	0.995	0.064	0.921
mean	0.032	0.967	0.063	0.909

high overall accuracy.

The highest error case for our method is *mm_ijk*, explained by its expensive column-wise matrix traversal. The results for *mm_ikj* confirms this, performing better without this expensive traversal. While the cache simulator in use, *cachegrind*, is likely more precise than many analytical models, it is limited to assuming idealized caches without noise, only considers two cache levels and a hard-coded LRU replacement policy, and lacks advanced knowledge about the complex issue of hardware prefetching. Furthermore, the results on *ortlerIvy* do not always correlate with the ones obtained on *ortlerSandy*. Apart from different memory controller speeds and CPU clock frequencies, the former holds CPUs with 25 MB of L3 cache with a 20-way associativity — a parameter combination that results in set sizes other than powers of two, which *cachegrind* cannot simulate. To overcome the aforementioned limitations, we predict cache misses for the first and last level cache, and as a fallback switch to 25-way associativity whenever required. Overall, our method achieves a mean NRMSE of 0.036 for execution time and 0.064 for energy consumption across all benchmarks and both architectures.

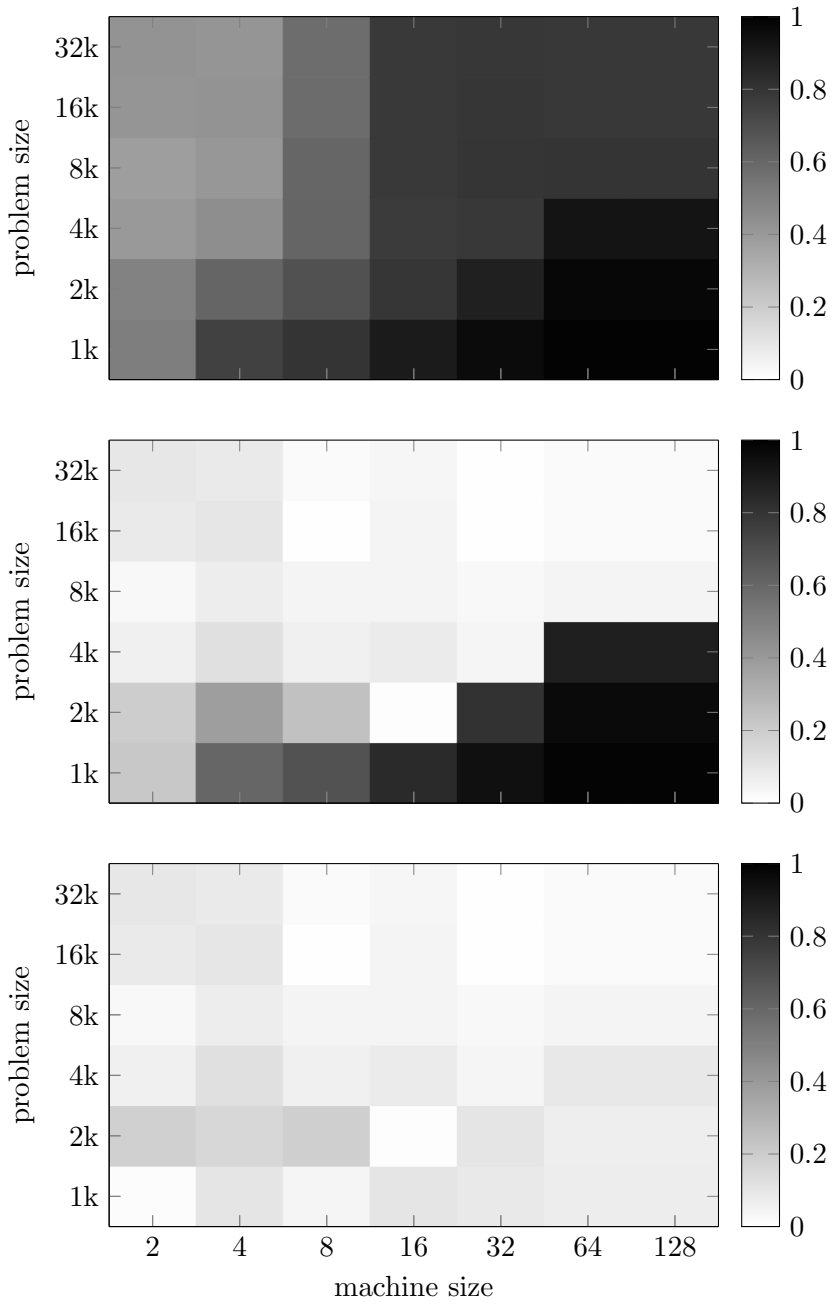


Figure 6.5: Relative error of the execution time prediction of the *jacobi* application for increasing problem and machine sizes on *ortlerSandy* for T_{comp} (top), $\max(T_{\text{comp}}, T_{\text{mem}})$ (center), and $\sum(\max(T_{\text{comp}}, T_{\text{mem}}), T_{\text{comm}})$ (bottom).

6.6 Summary

In this work we have presented a novel compiler-based prediction tool that automatically generates models for execution time and energy for a large set of message passing parallel programs. We introduced a code region definition that matches the structure of these programs, and illustrated the benefits of using compiler analysis for deriving analytical models and minimizing the overhead of model generation. We demonstrated that a single reference execution per input program and target architecture is sufficient for training our models. Our reference implementation showed the validity of our model, with a mean coefficient of determination of 0.93 over 7 input programs. Future work includes examining the prediction accuracy with regard to varying the reference problem and machine sizes, improving cache miss prediction or using more sophisticated network models that include network contention, extending the model for derived data types, and exploring additional hardware architectures and additional input programs.

Chapter 7

Conclusion

7.1 Contributions

In this thesis, non-functional properties of parallel hardware and software were established, and discussed. A formal model has been presented, describing the fundamentals regarding execution time and energy in HPC environments. This model was derived from benchmarking, analysis, and modeling of both commodity hardware (published under the title *Modeling CPU Energy Consumption of HPC Applications on the IBM POWER7* [51]) as well as experimental prototype hardware (published under the title *Performance Analysis and Benchmarking of the Intel SCC* [50]), both of which have shown the applicability of the model. Furthermore, we established non-functional metrics on top of this model, by defining a common, generic specification for all metrics that allows the introduction of new, user-specified metrics and custom aggregation policies.

In addition, we saw how to analyze, model, and optimize these metrics as well as their trade-offs at the example of execution time and energy consumption, and what further knowledge and guidelines to gain from all these endeavors. The problems described in this thesis are all tackled from a compiler perspective, which distinguishes this work from most state-of-the-art research that instead deals with non-functional parameter analysis and optimization purely from the perspective of scheduling and resource management. The amount of competitive work is further reduced when considering energy, the main focus of all efforts described in this thesis, as the majority of literature in high performance computing solely discusses execution time concerns.

For the purpose of the research described in this thesis and practical use, the formal model was implemented via the instrumentation and measurement framework

for the Insieme Compiler and Runtime System. It enables researchers to automatically identify (parallel) code regions of interest, instrument and measure them for any number of user-defined metrics on parallel hardware architectures and transfer the gathered information back to the compiler for further use such as auto-tuning. This instrumentation and measurement system has been used for the specific contributions detailed in Chapters 4 to 6, and outlined below.

Multi-Objective Auto-Tuning

Although related work regarding multi-objective optimization in HPC has grown over the past two decades, only few works apply true multi-objective optimization that captures the trade-off between conflicting objectives, and most of them are runtime approaches. Moreover, to the best of our knowledge, none of them has been applied to optimize for more than two objectives.

Contrary to that, we established three metrics of interest to the HPC community, execution time, resource usage, and energy consumption. Furthermore, we described an optimizing algorithm, RS-GDE3, capable of identifying the trade-offs between these three objectives using the concept of Pareto optimality. It showed the capabilities of the Insieme compiler and runtime system framework when tackling the problem of auto-tuning involving three types of tunable parameters: compiler-side source code transformations (in this case loop tiling); hardware parameters such as frequency and voltage scaling of CPUs; and runtime system parameters such as the degree of parallelism. The output of the RS-GDE3 algorithm was used to derive observations that can serve as guidelines when executing parallel programs in resource-restricted HPC environments. This work was presented in Chapter 4 and published under the title *Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage* [53].

Significance-driven Optimization of Code Execution

Near-threshold voltage (NTV) computation targets saving energy by operating hardware closer to the transistor threshold voltage than super-threshold methods such as ordinary DVFS. Related work has shown this to be a viable means of saving power and also energy, however the entailed increase in fault probabilities has hindered its deployment in HPC. While there are works that try to safely apply this technology by employing resilience methods to ensure proper code execution, the majority does not investigate the effect of faults on unprotected codes. Furthermore, the energy saving potential via unreliable code execution of naturally converging solvers remained unexplored.

We introduced the concept of code significance of HPC codes at the example of an iterative solver for linear equations, and showed that code regions and their data can be attributed with varying significance regarding their sensitivity to faults. This sensitivity was examined with respect to a widely-used floating point number representation of IEEE 754, binary64. In addition, we have seen the concept of operating hardware at NTV with the prospect of large reductions at energy consumption while incurring a non-negligible impact on performance and an increased probability of faults. Combining the aspects of code significance and NTV, the effect of executing iterative solvers on unreliable hardware was examined. The results showed that iterative solvers can compensate for NTV-induced faults via increased convergence time. Additionally, the results illustrate that parallelism can be a viable means of mitigating the performance impact of NTV, thus leading to overall energy savings between 35% and 67% without compromising program correctness. This work was discussed in Chapter 5 and published under the title *On the Potential of Significance-Driven Execution for Energy-Aware HPC* [52].

Compiler-assisted Execution Time and Energy-Modeling

Literature has shown predictive modeling to reduce the overhead of optimization methods that open large parameter search spaces. Many of them employ stochastic methods based on run-time parameters, and rely on repeated search space sampling for generating and training their models. Others rely on the user to describe the model or point to important program parameters. However, very few works incorporate static information obtained automatically in order to reduce this model generation and training overhead, or to provide parametrized models with respect to the problem size or machine size.

By contrast, we presented a novel energy prediction approach for distributed memory parallel programs, incorporating compiler knowledge in order to reduce prediction model generation overhead to a single target program execution. The method produces analytical models for execution time and energy consumption which are parametrized regarding the machine configuration and problem size. It automatically identifies key statements in the target program such as loops and communication points, and derives a relationship between their parameters, the input parameters of the program and the machine configuration via data flow analysis. The resulting model is trained with a single execution and can then extrapolate the execution time and energy consumption of larger machine sizes and problem sizes. Results showed that while reducing the number of training executions of the target program to a single one, the accuracy achieved can match state-of-the-art research that requires

multiple executions to train stochastic models. This work was discussed in Chapter 6 and submitted to Euro-Par 2017 under the title *Automatic Compiler-assisted Performance and Energy Modeling for Message-Passing Parallel Programs.*

7.2 Future Work

Throughout the course of this work, many additional open research issues arose. These include:

Automatic code significance The work of Chapter 5 could be extended to allow the compiler to automatically determine the significance of code regions and their data, and influence run-time decisions such as scheduling or trading hardware reliability for energy costs. There is some research in the field of automatic quantification of such metrics, however they lack the power of a compiler and rely on black-box approaches such as automatic differentiation [134]. A compiler could inspect the content of code regions, examine operation kinds and data types, and compute significance automatically to guide the process of executing code reliably or unreliably in order to save energy.

Additional auto-tuning parameters Chapter 4 is based the tuning parameters loop tile sizes, DVFS and degree of parallelism. However, there are a great many possibilities to extend this search space and even further optimize parallel programs. Examples are thread-core affinity policies, additional source code transformations, smaller DVFS domains (e.g. per-core), or scheduling on heterogeneous architectures.

Topology-aware search space reduction The tuning parameter of thread-core affinity policies poses a large combinatorial problem space. Hence, if explored, additional means of search space reduction may be required. Establishing equivalence classes that hold similar policies with respect to the hardware topology might prove beneficial. An example is the order in which software threads are assigned to CPU cores of the same CPU, that share the same caches. Changing this order is likely to have no measurable effect, but the investigation of topology equivalence classes is an open problem and might also depend on the data access pattern and communication pattern of the application to be tuned.

Auto-tuning with mixed search space dimension types The search space presented in Chapter 4 consists of several dimensions, all of which form continuous functions with parameters ranging from 0 or 1 to an upper limit. However,

there are parameters, such as thread-core affinity policies, which do not satisfy this property but instead only form e.g. a partial order. Efficiently navigating search spaces consisting of continuous and non-continuous search space dimensions with auto-tuners is an open issue.

Combine auto-tuner with prediction The energy prediction method in Chapter 6 might be used as input for the auto-tuning approach described in Chapter 4 when optimizing distributed memory programs. Given that the degree of parallelism is a tunable parameter, the model could predict time and energy for various machine sizes, thereby reducing the number of evaluation of the iterative compilation search, which poses the vast majority of overhead of this method.

Appendices

List of Symbols

Symbol	Description	Introduction
$(\ldots,\ldots)$	Control flow edge	Page 20
$[\ldots,\ldots]$	Communication edge	Page 21
a	Clocks per instruction	Page 16
$\mathcal{A}$	Sequential program	Page 20
$\mathcal{A}_p$	Parallel program	Page 21
b	Bandwidth	Page 17
d	Data type	Page 16
$\mathcal{D}$	Set of domains	Page 15
ϵ	Efficiency	Page 23
E	Energy	Page 8
$\mathcal{E}$	Set of sequential control flow edges	Page 20
$\mathcal{E}_e$	Set of entry edges	Page 22
$\mathcal{E}_p$	Set of parallel control flow edges	Page 21
$\mathcal{E}_c$	Set of communication edges	Page 21
$\mathcal{E}_x$	Set of exit edges	Page 22
f	Frequency setting	Page 15
$\mathcal{F}$	Set of frequency settings	Page 15
i	Instruction type	Page 16
l	Hardware link	Page 10
$\mathcal{L}$	Set of hardware links	Page 10
$\mathcal{M}$	Parallel Computer	Page 10
$\mathbb{N}^+$	Set of natural numbers excluding 0	Page 22
$\propto$	Proportionality relation	Page 56
p	Power state	Page 15
$\mathcal{P}$	Set of power states	Page 15
P	Power	Page 8
P_{avg}	Average power	Page 8

Symbol	Description	Introduction
ϕ	Function returning the average power	Page 23
$\mathcal{R}$	Parallel code region	Page 22
ru	Resource usage	Page 45
s	Sequential statement	Page 20
$\mathcal{S}$	Set of sequential statements	Page 20
s_p	Statement expressing parallelism	Page 21
$\mathcal{S}_p$	Set of statements expressing parallelism	Page 21
s_e	Entry statement	Page 22
s_x	Exit statement	Page 22
$\mathcal{S}_e$	Set of entry statements	Page 22
$\mathcal{S}_x$	Set of exit statements	Page 22
σ	Speedup	Page 23
t	Time	Page 8
t_{wall}	Wall time	Page 23
t_{cpu}	Resource usage or CPU time	Page 23
τ	Function returning time	Page 23
u	Hardware unit	Page 10
$\mathcal{U}$	Set of hardware units	Page 10
v	Voltage setting	Page 15
$\mathcal{V}$	Set of voltage settings	Page 15
w	Instruction width	Page 16
x, y, z	Temporary variables or iterators as needed	

List of Figures

List of Tables

List of Definitions

List of Examples

List of Algorithms

Bibliography

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical report, Massachusetts Institute of Technology, 2009.

[3] Ferdinando Alessi, Peter Thoman, Giorgis Georgakoudis, Thomas Fahringer, and Dimitrios S. Nikolopoulos. *Application-Level Energy Awareness for OpenMP*, pages 219–232. Springer International Publishing, Cham, 2015.

[4] Saman Amarasinghe. Petabricks: A language and compiler based on autotuning. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 3–3, New York, NY, USA, 2011. ACM.

[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. *SIGPLAN Not.*, 44(6):38–49, June 2009.

[6] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA, 2014. ACM.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica,

and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28, 2009.

[8] Lukas Arnold. IBM BG/P workshop. http://www.training.prace-ri.eu/uploads/tx%5Fpracetmo/BlueGeneP.pdf, Oct 2009. Accessed: Oct 7, 2016.

[9] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[10] Fatemeh Ayatolahi, Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. A study of the impact of single bit-flip and double bit-flip errors on program execution. In *Proceedings of the 32Nd International Conference on Computer Safety, Reliability, and Security - Volume 8153*, SAFECOMP 2013, pages 265–276, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[11] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Not.*, 45(6):198–209, June 2010.

[12] R. Jacob Baker. *CMOS Circuit Design, Layout, and Simulation.* Wiley-IEEE Press, 3rd edition, 2010.

[13] Prasanna Balaprakash, Ananta Tiwari, and Stefan M. Wild. Multi Objective Optimization of HPC Kernels for Performance, Power, and Energy. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation: 4th International Workshop, PMBS 2013, Denver, CO, USA, November 18, 2013. Revised Selected Papers*, pages 239–260, Cham, 2014. Springer International Publishing.

[14] Nikhil Bansal, Ho-Leung Chan, Kirk Pruhs, and Dmitriy Katz. Improved bounds for speed scaling in devices obeying the cube-root rule. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I*, ICALP '09, pages 144–155, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[16] S. Benedict, R. S. Rejitha, P. Gschwandtner, R. Prodan, and T. Fahringer. Energy prediction of openmp applications using random forest modeling approach. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 1251–1260, May 2015.

[17] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, May 2012.

[18] Arnamoy Bhattacharyya and Torsten Hoefler. PEMOGEN: Automatic adaptive performance modeling during program runtime. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 393–404, New York, NY, USA, 2014. ACM.

[19] Arnamoy Bhattacharyya, Grzegorz Kwasniewski, and Torsten Hoefler. Using compiler techniques to improve automatic performance modeling. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 468–479, Washington, DC, USA, 2015. IEEE Computer Society.

[20] M. Boersma, M. Kroner, C. Layer, P. Leber, S. M. Muller, and K. Schelm. The POWER7 binary floating-point unit. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 87–91, July 2011.

[21] Luigi Brochard, Raj Panda, and Sid Vemuganti. Optimizing performance and energy of HPC applications on POWER7. *Computer Science - Research and Development*, 25(3):135–140, 2010.

[22] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. *SIGARCH Computer Architecture News*, 28(2):83–94, May 2000.

[23] Steve Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 400–405, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[24] Yair Censor. Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization*, 4(1):41–59, 1977.

[25] James Charles, Preet Jassi, Narayan S. Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the Intel Core i7 Turbo Boost Feature. In *Proceed-*

ings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), IISWC '09, pages 188–197, Washington, DC, USA, 2009. IEEE Computer Society.

[26] Jee Whan Choi, Daniel Bedard, Robert Fowler, and Richard Vuduc. A roofline model of energy. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 661–672, Washington, DC, USA, 2013. IEEE Computer Society.

[27] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society.

[28] Intel Corporation. Intel Xeon Processor E5-4650 Specifications. `http://ark.intel.com/products/64622/Intel-Xeon-Processor-E5-4650-20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI`, 2012. Accessed: November 4th, 2016.

[29] Intel Corporation. Intel Xeon Processor E5-1600, E5-2600, and E5-4600 v3 Product Families, Volume 1 of 2, Electrical Datasheet. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v3-datasheet-vol-1.pdf`, Jun 2015. Accessed: November 4th, 2016.

[30] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[31] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: Memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194, Aug 2010.

[32] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Trans. Evol. Comp*, 6(2):182–197, April 2002.

[33] Yong Dong, Juan Chen, Xuejun Yang, Lin Deng, and Xuemeng Zhang. Energy-oriented openmp parallel loop scheduling. In *Proceedings of the 2008 IEEE*

International Symposium on Parallel and Distributed Processing with Applications, ISPA '08, pages 162–169, Washington, DC, USA, 2008. IEEE Computer Society.

[34] Ronald G Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010.

[35] J Elliot, F Müller, Miroslav Stoyanov, and Clayton Webster. Quantifying the Impact of Single Bit Flips on Floating Point Arithmetic. Technical report, Tech. Rep. ORNL/TM-2013/282, Oak Ridge National Laboratory, One Bethel Valley Road, Oak Ridge, TN, 2013. 6, 9, 2013.

[36] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[37] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[38] Michael J. Flynn, Patrick Hung, and Kevin W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, 19(4):11–22, July 1999.

[39] Inc. Free Software Foundation. Gcc, the gnu compiler collection. `https://gcc.gnu.org/`, 2016.

[40] Vincent W. Freeh and David K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 164–173, New York, NY, USA, 2005. ACM.

[41] Vincent W. Freeh, David K. Lowenthal, Feng Pan, Nandini Kappiah, Rob Springer, Barry L. Rountree, and Mark E. Femal. Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):835–848, June 2007.

[42] Matteo Frigo. A fast fourier transform compiler. *SIGPLAN Not.*, 34(5):169–180, May 1999.

[43] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming*, 39(3):296–327, 2011.

[44] Grigori Fursin, Abdul Wahid Memon, Christophe Guillon, and Anton Lokhmotov. Collective mind, part II: towards performance- and cost-aware software engineering as a natural science. *18th International Workshop on Compilers for Parallel Computing (CPC15)*, 2015.

[45] R. Ge, X. Feng, S. Song, H. C. Chang, D. Li, and K. W. Cameron. Powerpack: Energy Profiling and Analysis of High-Performance Systems and Applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, May 2010.

[46] Fred W Glover and Gary A Kochenberger. *Handbook of metaheuristics*, volume 57. Springer Science & Business Media, 2006.

[47] R. Gonzalez, B. M. Gordon, and M. A. Horowitz. Supply and Threshold Voltage Scaling for Low Power CMOS. *IEEE Journal of Solid-State Circuits*, 32(8):1210–1216, Aug 1997.

[48] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, Sep 1996.

[49] Ivan Grasso, Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic problem size sensitive task partitioning on heterogeneous parallel systems. *SIGPLAN Not.*, 48(8):281–282, February 2013.

[50] P. Gschwandtner, T. Fahringer, and R. Prodan. Performance Analysis and Benchmarking of the Intel SCC. In *2011 IEEE International Conference on Cluster Computing*, pages 139–149. IEEE, Sept 2011.

[51] P. Gschwandtner, M. Knobloch, B. Mohr, D. Pleiter, and T. Fahringer. Modeling cpu energy consumption of hpc applications on the ibm power7. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 536–543, Feb 2014.

[52] Philipp Gschwandtner, Charalampos Chalios, Dimitrios S. Nikolopoulos, Hans Vandierendonck, and Thomas Fahringer. On the Potential of Significance-driven execution for energy-aware hpc. *Computer Science - Research and Development*, 30(2):197–206, 2015.

[53] Philipp Gschwandtner, Juan J. Durillo, and Thomas Fahringer. Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage. In *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 87–98. Springer International Publishing, 2014.

[54] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Perform. Eval. Rev.*, 40(3), January 2012.

[55] Julian Hammer, Georg Hager, Jan Eitzinger, and Gerhard Wellein. Automatic loop kernel analysis and performance modeling with kerncraft. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '15, pages 4:1–4:11, New York, NY, USA, 2015. ACM.

[56] Torsten Hoefler and Dmitry Moor. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Supercomputing frontiers and innovations*, 1(2):58–75, 2014.

[57] Mark Hoemmen and Michael Heroux. Fault-tolerant iterative methods via selective reliability. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society*, volume 3, page 9, 2011.

[58] Kenneth Hoste and Lieven Eeckhout. Cole: Compiler optimization level exploration. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 165–174, New York, NY, USA, 2008. ACM.

[59] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. Van Der Wijngaart. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *IEEE Journal of Solid-State Circuits*, 46(1):173–183, Jan 2011.

[60] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open mpi. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[61] Maxwell Lipford Hutchinson. Hybrid OpenMP MPI Benchmark. https://sourceforge.net/projects/homb/, Apr 2013. Accessed: Oct 7, 2016.

[62] *IEEE standard for binary floating-point arithmetic.* New York, 1985. Note: Standard 754–1985.

[63] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B Part 2*, Jun 2013. Accessed: November 4th, 2016.

[64] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. INSPIRE: The insieme parallel intermediate representation. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 7–18, Piscataway, NJ, USA, 2013. IEEE Press.

[65] Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 10:1–10:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[66] Herbert Jordan, Peter Thoman, and Thomas Fahringer. *A High-Level IR Transformation System*, pages 647–656. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[67] Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. Power7: IBM's next-generation server processor. *IEEE Micro*, 30(2):7–15, March 2010.

[68] U. R. Karpuzcu, I. Akturk, and N. S. Kim. Accordion: Toward soft near-threshold voltage computing. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 72–83, Feb 2014.

[69] Ulya Karpuzcu, Nam Sung Kim, and Josep Torrellas. Coping with parametric variation at near-threshold voltages. *IEEE Micro*, 33(4):6–14, July 2013.

[70] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (NTV) design: Opportunities and challenges. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1153–1158, New York, NY, USA, 2012. ACM.

[71] Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. *Automatic Data Layout Optimizations for GPUs*, pages 263–274. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[72] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 149–160, New York, NY, USA, 2013. ACM.

[73] A. Kohler, M. Radetzki, P. Gschwandtner, and T. Fahringer. Low-latency collectives for the intel scc. In *2012 IEEE International Conference on Cluster Computing*, pages 346–354, Sept 2012.

[74] Ted Kubaska. The SCC Programmer's Guide. Technical report, Intel Labs, 2010.

[75] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Trans. Archit. Code Optim.*, 6(1):1:1–1:36, April 2009.

[76] James H. Laros, III, Kevin T. Pedretti, Suzanne M. Kelly, Wei Shu, and Courtenay T. Vaughan. Energy based performance tuning for large scale high performance computing systems. In *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pages 6:1–6:10, San Diego, CA, USA, 2012. Society for Computer Simulation International.

[77] Seyong Lee, Jeremy S. Meredith, and Jeffrey S. Vetter. COMPASS: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 405–414, New York, NY, USA, 2015. ACM.

[78] Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, and Subhasish Mitra. Ersa: Error resilient system architecture for probabilistic applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1560–1565, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[79] D. Li, D. S. Nikolopoulos, K. Cameron, B. R. de Supinski, and M. Schulz. Power-aware mpi task aggregation prediction for high-end computing systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

[80] Dong Li, Bronis R. de Supinski, Martin Schulz, Dimitrios S. Nikolopoulos, and Kirk W. Cameron. Strategies for energy-efficient resource management of hybrid programming models. *IEEE Trans. Parallel Distrib. Syst.*, 24(1):144–157, January 2013.

[81] Charles Lively, Xingfu Wu, Valerie Taylor, Shirley Moore, Hung-Ching Chang, and Kirk Cameron. Energy and performance characteristics of different parallel implementations of scientific applications on multicore systems. *Int. J. High Perform. Comput. Appl.*, 25(3):342–350, August 2011.

[82] Lianjie Luo, Yang Chen, Chengyong Wu, Shun Long, and Grigori Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. *3rd Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART09)*, 2014.

[83] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core scc processor: The programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[84] John D McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, pages 19–25, 1995.

[85] S. McFarling. Program optimization for instruction caches. *SIGARCH Comput. Archit. News*, 17(2):183–191, April 1989.

[86] Bryan Mills, Taieb Znati, Rami Melhem, Kurt B. Ferreira, and Ryan E. Grant. Energy consumption of resilience mechanisms in large scale systems. In *Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '14, pages 528–535, Washington, DC, USA, 2014. IEEE Computer Society.

[87] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the 32Nd ACM/IEEE International*

Conference on Software Engineering - Volume 1, ICSE '10, pages 25–34, New York, NY, USA, 2010. ACM.

[88] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, pages 7–10, 1999.

[89] Frank Mueller. Pthreads library interface. Technical report, Florida State University, 1993.

[90] Nicholas Nethercote. *Dynamic binary analysis and instrumentation.* PhD thesis, PhD thesis, University of Cambridge, 2004.

[91] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Publishing Company, Incorporated, 2010.

[92] University of Innsbruck. Insieme compiler project. `http://www.insieme-compiler.org`, 2016.

[93] S. Pakin and M. Lang. Energy modeling of supercomputers and large-scale scientific applications. In *2013 International Green Computing Conference Proceedings*, pages 1–6, June 2013.

[94] Maurizio Palesi and Tony Givargis. Multi-objective design space exploration using genetic algorithms. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pages 67–72, New York, NY, USA, 2002. ACM.

[95] Zdzisław Pawlak. Rough sets. *International Journal of Computer & Information Sciences*, 11(5):341–356, 1982.

[96] Magnus Erik Hvass Pedersen. *Tuning & simplifying heuristical optimization.* PhD thesis, University of Southampton, 2010.

[97] Joshua Peraza, Ananta Tiwari, Michael Laurenzano, Laura Carrington, and Allan Snavely. Pmac's green queue: A framework for selecting energy optimal dvfs configurations in large scale mpi applications. *Concurr. Comput. : Pract. Exper.*, 28(2):211–231, February 2016.

[98] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. In

Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pages 431–444, New York, NY, USA, 2013. ACM.

[99] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 144–156, Washington, DC, USA, 2007. IEEE Computer Society.

[100] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb 2005.

[101] Mohammed Rahman, Louis-Noël Pouchet, and P. Sadayappan. Neural network assisted tile size selection. In *International Workshop on Automatic Performance Tuning (IWAPT2010)*. Springer, 2010.

[102] Shah Faizur Rahman, Jichi Guo, and Qing Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 107–116, New York, NY, USA, 2011. ACM.

[103] Shah Mohammad Faizur Rahman, Jichi Guo, Akshatha Bhat, Carlos Garcia, Majedul Haque Sujon, Qing Yi, Chunhua Liao, and Daniel Quinlan. Studying the impact of application-level optimizations on the power consumption of multi-core architectures. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 123–132, New York, NY, USA, 2012. ACM.

[104] Thomas Rauber and Gudula Rünger. Modeling the energy consumption for concurrent executions of parallel tasks. In *Proceedings of the 14th Communications and Networking Symposium*, CNS '11, pages 11–18, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[105] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 324–334, New York, NY, USA, 2006. ACM.

[106] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. *SIGPLAN Not.*, 45(10):806–821, October 2010.

[107] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *Micro, IEEE*, 32(2), 2012.

[108] J. T. Russell and M. F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*, pages 328–333, Oct 1998.

[109] Giacinto P. Saggese, Nicholas J. Wang, Zbigniew T. Kalbarczyk, Sanjay J. Patel, and Ravishankar K. Iyer. An experimental study of soft errors in microprocessors. *IEEE Micro*, 25(6):30–39, November 2005.

[110] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164–174, June 2011.

[111] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. Ibm power7 multicore server processor. *IBM J. Res. Dev.*, 55(3):191–219, May 2011.

[112] Shuaiwen Song, Rong Ge, Xizhou Feng, and Kirk W. Cameron. Energy profiling and analysis of the hpc challenge benchmarks. *Int. J. High Perform. Comput. Appl.*, 23(3):265–276, August 2009.

[113] Kyle L. Spafford and Jeffrey S. Vetter. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[114] Robert Springer, David K. Lowenthal, Barry Rountree, and Vincent W. Freeh. Minimizing execution time in mpi programs on an energy-constrained, power-scalable cluster. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 230–238, New York, NY, USA, 2006. ACM.

[115] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 207–216, New York, NY, USA, 2015. ACM.

[116] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11(4):341–359, December 1997.

[117] Nathan R. Tallent and Adolfy Hoisie. Palm: Easing the burden of analytical performance modeling. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 221–230, New York, NY, USA, 2014. ACM.

[118] Insieme Developer Team. Insieme source code repository. `https://github.com/insieme/insieme`, 2016.

[119] S. Thakkur and T. Huff. Internet Streaming SIMD Extensions. *Computer*, 32(12):26–34, Dec 1999.

[120] P. Thoman, P. Gschwandtner, and T. Fahringer. On the quality of implementation of the c++11 thread support library. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 94–98, March 2015.

[121] Peter Thoman. *Insieme-RS: A Compiler-supported Parallel Runtime System.* PhD thesis, University of Innsbruck, Innrain 52, 6020 Innsbruck, Austria, 2013.

[122] Peter Thoman, Herbert Jordan, and Thomas Fahringer. Adaptive granularity control in task parallel programs using multiversioning. In *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pages 164–177, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[123] Peter Thoman, Herbert Jordan, and Thomas Fahringer. *Adaptive Granularity Control in Task Parallel Programs Using Multiversioning*, pages 164–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[124] Peter Thoman, Herbert Jordan, and Thomas Fahringer. Compiler multiversioning for automatic task granularity control. *Concurrency and Computation: Practice and Experience*, 26(14):2367–2385, 2014.

[125] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. *Automatic OpenCL Device Characterization: Guiding Optimized Kernel Design*, pages 438–452. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[126] Peter Thoman, Stefan Moosbrugger, and Thomas Fahringer. *Optimizing Task Parallelism with Library-Semantics-Aware Compilation*, pages 237–249. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[127] Peter Thoman, Hans Moritsch, and Thomas Fahringer. *Topology-Aware OpenMP Process Scheduling*, pages 96–108. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[128] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[129] Ananta Tiwari, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. *Auto-tuning for Energy Usage in Scientific Applications*, pages 178–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[130] Ananta Tiwari, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. Modeling power and energy usage of hpc kernels. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 990–998, Washington, DC, USA, 2012. IEEE Computer Society.

[131] Matthew Tolentino and Kirk W. Cameron. The optimist, the pessimist, and the global race to exascale in 20 megawatts. *Computer*, 45(1):95–97, January 2012.

[132] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society*, s2-43(1):544–546, 1938.

[133] Didem Unat, Cy Chan, Weiqun Zhang, Samuel Williams, John Bachan, John Bell, and John Shalf. Exasat: An exascale co-design tool for performance modeling. *Int. J. High Perform. Comput. Appl.*, 29(2):209–232, May 2015.

[134] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. Openad/f: A modular open-source tool for automatic differentiation of fortran codes. *ACM Trans. Math. Softw.*, 34(4):18:1–18:36, July 2008.

[135] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005.

[136] Vincent M. Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. Measuring energy and power with PAPI. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 262–268, Washington, DC, USA, 2012. IEEE Computer Society.

[137] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[138] Stefan Widerin. Insieme Runtime System for Windows. Master's thesis, University of Innsbruck, Innrain 52, 6020 Innsbruck, Austria, Oct 2013.

[139] Maurice V. Wilkes. The memory gap and the future of high performance memories. *SIGARCH Comput. Archit. News*, 29(1):2–7, March 2001.

[140] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[141] Z. Zheng, L. Yu, and Z. Lan. Reliability-aware speedup models for parallel applications with coordinated checkpointing/restart. *IEEE Transactions on Computers*, 64(5):1402–1415, May 2015.

[142] E. Zitzler and L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, 3(4), 1999.