Habilitation à diriger des recherches

université PARIS-SACLAY

High Performance Computing Code Optimizations: Tuning Performance and Accuracy

Habilitation à diriger des recherches de l'Université Paris-Saclay

présentée et soutenue à ...., le ...., par

Pablo DE OLIVEIRA CASTRO HERRERO

Composition du jury

Prénom NOM Titre, Affiliation	Rapporteur ou Rapportrice
Prénom NOM Titre, Affiliation	Rapporteur ou Rapportrice
Prénom NOM Titre, Affiliation	Rapporteur ou Rapportrice
Prénom NOM Titre, Affiliation	Examinateur ou Examinatrice
Prénom NOM Titre, Affiliation	Examinateur ou Examinatrice

Contents

1 Introduction

Contents

Computer simulation implements mathematical models to approximate the behavior of a system. It is often used when a closed-form analytical solution is not available, and it has become a widely used approach to gather knowledge in scientific fields such as physics, chemistry, biology, climatology, and social sciences.

There is a large variety of computer models. In physical sciences, models are usually derived from a set of differential equations that are discretized. The simulation operates either at the particle level, computing step-by-step interactions between a group of bodies (e.g. atoms, planets), or at the field level, simulating the evolution of a continuous field (e.g. fluid, wave). Other models include agent-based, discrete event, and stochastic simulations.

To motivate the need for computer models, we consider the classical n-body problem that models the interaction of n particles through gravitational forces. It plays a central role in understanding the motion of celestial objects. The masses, initial positions, and velocities of the particles are given. Applying Newton's law of gravitation to each particle produces a system of equations describing the motion of the particles. For $n = 2$, the system was solved by Bernoulli in 1734. In Bernoulli solution, the positions of the particles can be written as simple equations that directly depend on the time, the masses, and the initial conditions. For $n = 3$, the problem is harder: particular cases were explicitly solved by Euler and Lagrange but no general solution was known in the 18th century. In 1909, Sundman gave an analytical solution of the three-body problem as an infinite series. Unfortunately, the series converges so slowly, that it cannot be practically computed. Moreover, the form of Sundman solution does not bring insights on the particles' motion [84] because it can only be expressed in a non-absolute referential. Therefore, for $n \geq 3$, the n-body problem cannot be practically solved through classical methods. On the other hand, with a computer simulation we can effectively compute the motion of the

particles. When n is small, finite difference methods can numerically integrate the differential equations produced by Newton's law. Approximated methods that simplify the computation of forces for distant particles, such as Barnes-Hut or Fast Multipole Methods, scale to a larger number of particles.

Computer simulations have developed in parallel with the digital supercomputers required to execute them. Since the beginning of the field of high performance computing (HPC) after World War II, there has been a rapid increase in computing resources and simulation complexity.

ENIAC, completed in 1945, can be considered the first supercomputer and its primary use was performing ballistic computations. ENIAC could compute around 500 floating point operations per second (Flop/s) with vacuum tube logic. In the sixties, various companies such as Control Data Corporation, IBM, DEC, and others developed transistor supercomputers that were used in research centers and business. For example, the CDC 6600 supercomputer was built in 1964 and achieved 500 kFlop/s. In the seventies and eighties, Cray Research pushed the industry forwards with vector processors that were designed to operate efficiently on large single dimension arrays. One such supercomputer, the Cray 1 achieved 133 MFlop/s in 1976. In the nineties, supercomputers started incorporating thousand of processors. Departing from the vector SIMD (Single Instruction Multiple Data) model, architectures such as the Intel Paragon had up to 3 680 processors executing different instructions for a total computing power of 143.4 GFlop/s.

When this manuscript was written in the spring of 2022, the fastest supercomputer, Fugaku, had more than 7 million cores and achieved a peak performance of 537 PFlop/s, simulating large-scale phenomena. When simulating the n-body problem, Fukagu integrates 1.45 trillion particles per second [96]. However, these formidable simulations have a high energy cost: Fukagu requires 29MW, the average energy needed to power 23 000 homes in France.

Recently, machine learning models have generated much interest, particularly with deep neural networks (DNN) development. DNN have achieved impressive results; for example, the natural language GPT-3 models generates English text that is difficult to distinguish from text written by humans [20]. This feat comes at a high price: GPT-3 has 175 billion parameters and required 3 640 PFlop/s-days for training. As DNN complexity spikes, many [12, 153, 176] challenge their energy and environmental cost.

Reducing the cost, both in time and energy, of computer simulations is critical. This requires careful optimization of the programs and the architecture but also of the compiler and runtime. These optimizations are complex since they involve trade-offs: in particular, the precision of the model should be sufficient to provide scientific insights but as low as possible to save energy and computation time. Since joining the computer science department at University of Versailles Saint-Quentin in 2010, my research focused on optimizing HPC programs.

1.1 Optimizing high performance computing

Simulation programs are usually large, with thousands or even millions lines of code. HPC relies on complex heterogeneous architectures with massive concurrency at different execution levels, a deep memory hierarchy, and dedicated interconnect networks. This inherent complexity generates an optimization space

composed of many factors such as the chosen architecture, the algorithmic variant used, the floating-point precision and format, the compiler optimization passes and their parametrization, the number of threads, the thread placement, the thread affinity, and many others.

When optimizing an HPC system, one must consider different objective functions such as the time and energy spent and satisfy multiple constraints such as the amount of memory available or the target accuracy. Exhaustively measuring the different factor combinations for each program part is too costly. I have proposed various methods to automate and accelerate the exploration of the HPC design search space to deal with this inherent complexity.

A first approach, used both for auto-tuning and performance characterization, reduces the number of factor combinations measured in the search space. In my research, I have mixed traditional design of experiment techniques with adaptive sampling techniques that explore first the *interesting* [146] parts of the HPC search space. Defining what constitutes an interesting region for exploration is at the core of this approach and will be discussed in chapter 3. The exploration is used to train a surrogate performance model that acts as a proxy for the original HPC system to understand performance bottlenecks and find optimal design points.

A different and orthogonal perspective is to break down programs into elementary pieces called *codelets* [45]. This provides multiple advantages. Instead of optimizing a monolithic application, one can optimize the standalone codelets individually. Moreover, because the codelets capture elementary computation building blocks, there is redundancy among them. Keeping a single copy among a group of similar codelets also reduces the search space. Similarities between codelets can be studied both statically, for example, when two code fragments perform a similar operation, and dynamically, when two invocations of the same codelet in time operate on a similar data set. With Mihail Popov, Ph.D. student co-advised by William Jalby and myself, Chadi Akel, and other contributors, we developed a compiler-based tool, CERE [148], that automatically breaks-down large applications into standalone codelets.

The two approaches above are orthogonal: different parts of the code or different phases in the code can benefit from different auto-tuning factors. Our optimization search space is formed by the Cartesian product of the studied codelets and the various factors considered.

In both approaches, the key insight is exploiting similarities in the search space to reduce its size. The similarities can appear at the level of factors producing the same performance response and at the level of codelets. Both approaches can be used together to reduce the optimization cost in the factor dimensions and the codelet dimension. We apply these techniques in different domains, such as optimizing a seismic imaging proto-application [146] or reducing simulation time for hardware-software co-design [162, 132].

1.2 Accuracy of simulation programs

Many simulations deal with physical quantities that are represented by real numbers. Due to finite memory, algorithms usually manipulate an approximated representation of real numbers. One such approximation is the fixed-point representation, which keeps a fixed number of digits in the integer and fractional

part. While simple to manipulate, it is not a good fit for programs using values across different scales. For these programs, floating-point (FP) numbers are preferred. Inspired by the usual scientific notation, they represent a number as a mantissa multiplied by a scaling factor.

Until the eighties, computer manufacturers used different FP representations. Nowadays, all general-purpose chips use the well-established IEEE 754 standard for Floating-Point Arithmetic. The first version of the standard was published in 1985 and has since been regularly revised and updated. Most codes usually default to either the binary32 (`float`) or the binary64 (`double`) representations defined in IEEE 754.

Still, some applications could profit from smaller precisions to reduce power and computation time. For example, DNN models do not need the full precision provided by binary64 or binary32. Using a much lower precision for parameters drastically reduces the storage and computation cost while preserving good accuracy. This *quantization* has been extensively studied [69] in recent years and has become a central optimization of DNN models. Driven by these developments, new reduced FP formats such as BF16 or FP16 are used in some GPU, FPGA, and dedicated hardware accelerators for machine learning such as the TPU.

Because FP numbers operate on a limited precision, not all reals can be represented as FP numbers causing representation errors. During computations, inexact operations require rounding and introduce small round-off errors, which can accumulate over time. A third source of inaccuracy is called *catastrophic cancellation*. When subtracting two FP values that are very close, the result is renormalized to a smaller exponent. The renormalization can increase the magnitude of previous rounding errors in the last digits of the mantissa.

Round-off and catastrophic cancellation errors make FP arithmetic non-associative: accumulation of numerical errors depends on the order of operations. Because HPC architectures and optimizations rely on massive concurrency at the level of threads and the level of instructions, many optimizations can affect the order of operations and, therefore, the accuracy of the computations.

Therefore, verifying the accuracy of FP computations is paramount in optimizing HPC for at least two reasons,

1. Because parallelization, vectorization, and compiler optimizations change the accuracy of computations; it is important to verify numerical accuracy at the same time to assess the faithfulness of the computer model.

2. Reducing the precision of FP computation is a significant optimization. Removing bits from the mantissa lowers the memory required and reduces the communication and computation cost for FP numbers.

Both points highlight the trade-off between the precision of computer models and their computation cost that was previously mentioned.

Numerical accuracy is usually measured through the error between the actual computation and a reference value, such as the exact mathematical solution or a measure obtained by experimentation. For many complex programs or intermediate computations, a reference value is not known beforehand and might be hard to obtain. Fortunately, it is still possible to evaluate the sensitivity of the solution to numerical errors through stochastic methods.

In 2015, in collaboration with Eric Petit (UVSQ and later Intel Corporation) and Christophe Denis (ENS Paris-Saclay), we published the initial version of Verificarlo [150]. Verificarlo was later extended by Yohan Chatelain, Ph.D. student co-advised by William Jalby and myself, and other contributors. Verificarlo is an open-source tool that helps verify and optimize numerical accuracy in complex programs. Verificarlo includes different FP backends that simulate the effect of numerical errors and the impact of using lower precision. For example, one such backend implements Monte Carlo arithmetic (MCA) [187] that models imprecise computations as random variables and estimates the number of significant digits through repeated stochastic executions.

Verificarlo has been used to:

- Pinpoint, and provide insights to fix the numerical instabilities in large simulations such as Density Functional Theory quantum mechanical modeling [31], neuroimaging pipelines [108] or structure simulations [183].
- Explore the trade-off between performance and accuracy in iterative simulations [30] and mathematical libraries [21].

This applied research has been nourished by the exploration of its theoretical underpinnings, and in particular MCA. In [183] we have proposed a probabilistic definition for the number of significant digits and derived statistical confidence intervals for both Gaussian and arbitrary MCA distributions. El-Mehdi El-Arar, who is doing his Ph.D. at UVSQ, co-advised by Devan Sohier and myself, has started studying the bias and variance of MCA computations, uncovering interesting properties of stochastic rounding leading to less numerical errors and faster convergence in some application domains, corroborating previous results in the literature.

1.3 Discussion

After having introduced the two main research directions of my work, this section discusses the choices and idiosyncrasies that have guided these efforts.

The importance of opening software research tools In computer science, the importance of engineering is often dismissed, and implementation is deemed less important than theory. Because of this point of view and the pressure to quickly publish results, research software is not always polished, published, or maintained. Research papers are sometimes published without the accompanying software programs and experiment artifacts.

On the contrary, the computer software produced during research contributes to the field since it helps reproduce results. Moreover, other researchers can use well-designed software as a stepping stone for improving the work or producing new results. Reproducibility of results is one of the pillars of the scientific method; to fully reproduce results based on a computer simulation publishing the software code is essential [155].

I have published the software tools developed in my research as open-source projects. Some of them, such as Verificarlo, have become active open-source tools used in academia and industry and received various community contributions. Through its flexible design, one can play with and quantify the effects of

alternative FP computation model. It has been used in three Ph.D. thesis as an experimental framework.

Working at the level of the compiler Developing an HPC program involves a complex technological stack encompassing physical and mathematical modeling, discretization, implementation on a high-level programming language, compilation, runtimes, and execution on particular hardware. At each level of this stack, opportunities for optimization arise. Often optimizing HPC applications involves a collaboration between experts at the different levels of the stack.

I have often taken the compiler as the starting point in my research projects. Nevertheless, this is a compromise, and there is no single correct choice: other optimization approaches target either the high-level languages or the low-level assembly successfully. Working at the compiler level provides an interesting perspective, though, because it bridges the high-level and low-level ends of the HPC stack. Both CERE and Verificarlo build upon the LLVM compiler project. By operating at the compiler Intermediate Representation (IR) level they remain architecture agnostic within some limits. Also, at the compiler level, we still retain high-level information about the parallel regions, the data types, and the source code, which we can exploit. We will continue this discussion in more depth in chapters 2 and 7.

Dealing with HPC complexity In the early days of HPC, one could estimate the performance of a program by counting the number of assembly operations and accounting for their respective latency. Such a formula to estimate the performance of a program from its static characteristics is called an analytical performance model. When applicable, analytical performance models are faster than measurement or simulation.

Nowadays, analytical models are still powerful and have their place, such as the roofline model [143] to identify performance bottlenecks or the reuse distance to predict cache behavior. Unfortunately, with the growth of hardware and software complexity, it is challenging to build faithful analytical performance models. Modern micro-architectures aggressively exploit parallelism: multiple memory operations and computations are in flight simultaneously. Memory architectures are deep and have many layers of storage and cache. HPC programs are increasingly complex and depend on multiple software layers, including external libraries, runtimes, and the underlying operating system. They also exploit concurrency at many levels: SIMD, threads, processes, and some will offload computations to an accelerator. All these factors can affect the performance of the program. Because the exact order of the operations and the internal state of the computing and memory nodes is unknown, the performance of programs becomes non-deterministic [2] and cannot be completely captured with an analytical performance model. Moreover, the performance of a program can be affected by external factors such as network latency or synchronization between nodes. When measuring the performance of a program, it is thus important to control the influence of external factors and the machine's initial state. Despite these precautions, multiple executions of the same program will give slightly different measures. Capturing the non-deterministic nature of computer performance requires applying statistical analysis to the performance

measures.

When verifying numerical accuracy, one is also confronted with the complexity of HPC programs. For well-known algorithms, bounds of the numerical error can be proved mathematically. Exact methods have also been proposed to derive error bounds automatically. One well-established exact method for deriving error bounds is Interval Arithmetic [136], in which each real value in the algorithm is replaced by an interval that contains all the possible values of the computation. Because intervals are conservative, they tend to become overly large when the algorithm or control flow is complex. In general, for complex HPC programs of thousands of code lines, deriving such an analysis is intractable and gives pessimistic bounds. In these cases, statistic techniques based on Monte Carlo arithmetic can help understand and optimize complex HPC programs. They are not a substitute for analytical or exact methods since they are usually dataset dependent and do not provide deterministic bounds on accuracy. Nevertheless, they provide a complementary tool that can bridge the complexity gap.

1.4 Outline of the manuscript

The first part of the manuscript presents our contributions to optimizing HPC applications. We focus on reducing the cost of exploring the HPC optimization space either through breaking applications into small *codelets* or adaptively sampling the performance search space.

In the second part of the manuscript, using alternative floating-point models, such as Monte Carlo arithmetic, we explore the compromise between numerical precision and performance. A probabilistic definition of the number of significant digits is introduced and used to estimate the accuracy of a computation. Verificarlo is applied to pinpoint numerical bugs in large HPC codes and assert the impact of reducing the precision in parts of the code. We discuss new perspectives, in particular algorithms where using a stochastic rounding mode improves error bounds.

Finally, we examine the challenge of reducing the power consumption of HPC. In particular, we advocate for sobriety in our usage of computing resources. Computer simulation offers unique possibilities for advancing research, yet we should be conscious of its energy cost. Instead of always reaching for more complex simulations, we should find the right fit for our problem.

Part I

Reducing the search space for HPC optimizations

2 Codelet extractor and replayer

Contents

This chapter includes contributions from the following publications and software projects:

- Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. "Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.1 (2015), p. 6
- Mihail Popov, Chadi Akel, Florent Conti, William Jalby, and Pablo de Oliveira Castro. "PCERE: Fine-grained parallel benchmark decomposition for scalability prediction." In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE. 2015, pp. 1151–1160
- Mihail Popov, Chadi Akel, Yohan Chatelain, William Jalby, and Pablo de Oliveira Castro. "Piecewise holistic autotuning of parallel programs with CERE." in: *Concurrency and Computation: Practice and Experience* (2017), e4190. ISSN: 1532-0634. DOI: 10.1002/cpe.4190. URL: http://dx.doi.org/10.1002/cpe.4190
- Pablo de Oliveira Castro, Mihail Popov, Chadi Akel, and Yohan Chatelain. *benchmark-subsetting/cere: CERE v0.3.1 release*. Version v0.3.1. Nov. 2018. DOI: 10.5281/zenodo.5910793. URL: https://doi.org/10.5281/zenodo.5910793

Performance evaluation for optimization, system benchmarking, or compiler evaluation is time and resource consuming. The expensive cost limits the number of iterations engineers can perform in a given budget. Different approaches

to overcome this limitation have been proposed by the community, such as analytical models [88, 127], machine learning [28, 159], checkpoint-restart [80], or simulations [178]. An interesting and versatile approach is code isolation [120, 158, 123, 1]. Code isolation reduces benchmarking cost and allows piecewise optimization of an application.

This chapter presents Codelet Extractor and REplayer (CERE), an open source framework for code isolation. CERE finds and extracts the hotspots of an application as isolated fragments of code, called *codelets*. Codelets can be modified, compiled, run, and measured independently of the original application. CERE [148] is available at `http://benchmark-subsetting.github.io/cere` under the GNU Lesser General Public License. Many people have contributed to CERE by reporting problems, suggesting various improvements or writing actual code. In particular regarding the code development, Chadi Akel (during an internship at UVSQ and later as a research engineer) and myself wrote the initial version of CERE. Mihail Popov, during his Ph.D. at UVSQ, significantly extended CERE to work on parallel OpenMP regions [161, 162]. Yohan Chatelain, during an internship at UVSQ, improved the memory tracer.

2.1 Code isolation

Usually, in scientific applications, the hotspots represent a small fraction of the total source lines [111]. Code isolation finds and extracts the hotspots of an application as standalone fragments of code, called *codelets*. Codelets can be compiled and replayed independently of the original application. For each codelet, the isolation process captures the memory working set and the relevant machine state such as the cache content to achieve realistic replays.

Breaking an application into independent codelets provides multiple benefits. Executing isolated codelets instead of whole applications is faster and enables piecewise evaluation and optimization of an application. Indeed, different codelets may expose different performance bottlenecks and react differently to optimizations. With code isolation they can be individually modified to evaluate the payoff of new optimizations and tune performance at a fine-grain level.

Effective code isolation raises multiple challenges. First, to be practical, isolation must support many programming languages, applications, and optimizations. Second, codelets should be replayable on a variety of target architectures. Third, to achieve accurate performance measures, the memory working set, cache state, NUMA state and thread placement must be captured and restored before each replay.

CERE is an IR level code extractor framework based on LLVM that targets representative OpenMP parallel regions or loop nests. Figure 2.1 presents the full CERE pipeline. CERE takes as an input the source files of an application or a benchmark suite. All the languages supported by the LLVM front-ends (C, C++, Fortran, D, etc.) are accepted. The source loops or parallel regions are outlined and instrumented with profiling probes to identify the application hotspots. We filter out short loops that contribute less than 1% to the execution time. Different invocations of the same codelet may have different performance behaviors, which depend on the working set and cache state of each invocation. We use a clustering algorithm, presented in section 3.1.2 on page 37 that

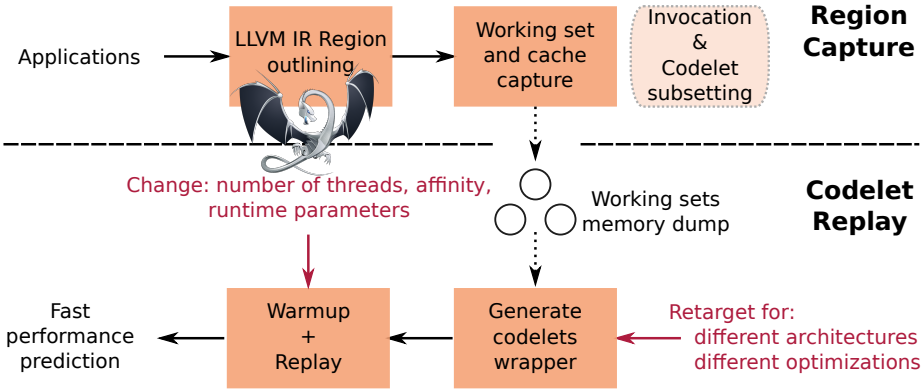


Figure 2.1: CERE workflow. Applications are partitioned into a set of codelets, which may be pruned using different criteria. A set of representative invocations are selected and captured. The codelets can then be replayed with different options and on different targets.

analyzes the performance trace of each codelet to find a representative subset of invocations. The memory, cache, thread and NUMA state of each selected invocation is then captured and dumped to disk. The output of this process is a set of representative codelets and invocations, which can be redistributed, recompiled and replayed on different systems and architectures. The codelet set can be used as a proxy for original application in optimization or benchmarking studies.

2.2 Related work

In this section, we review previous works on codelet extraction, working set capture, cache capture, and practical applications of codelets.

2.2.1 Codelet extraction

A first challenge when isolating codelets is choosing the right granularity. Two possibilities have been proposed in previous works: assembly isolation and source code isolation. CERE explores a new level: Intermediate Representation (IR) isolation.

Assembly isolation [178, 179, 114] extracts codelets as blocks of assembly instructions. Simpoint [178] successfully speeds up architecture simulation by sampling a limited number of assembly codelets. Yet assembly isolation is not practical for performance tuning because the assembly code cannot be recompiled with different performance flags or easily retargeted to a new architecture. The extraction software is tied to a specific instruction set architecture. It is also difficult to map assembly codelets to source code regions. However, this approach is language agnostic and resilient to the compiler effect: what you extract is what is executed.

Source code isolation [1, 120, 158, 123] is portable and can be easily used to tune compiler options [102] or to select the best architecture [149]. Furthermore, because extraction occurs at source level, before compiler transformations, the

performance information gathered during replay can be easily mapped to the source high-level constructs. Unfortunately, source code isolation requires a specific parser and extraction process for each language. Therefore, supporting multiple languages is extremely costly because writing a robust extraction pass for complex languages, such as C++, is technically challenging. Finally, one must ensure that the source level extraction process does not alter the performance behavior of the original hotspot. Indeed, some transformations used during source isolation may hinder compiler optimization passes [1, 123].

In our work, we explore code isolation at the IR level which provides a good trade-off between assembly and source code isolation. We choose to target the LLVM [118] compiler IR. CERE extraction is therefore tied to the LLVM compiler, but it supports all LLVM front-ends and back-ends with no extra engineering cost. Extracting codelets at the IR level is much simpler than at the source code level which requires parsing complex input languages. It also facilitates the process of instrumenting the code, capturing the memory and outlining the codelet thanks to the powerful integrated flow analysis passes as detailed in section 2.3.

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

2.2.2 Memory capture and cache warm up

Memory capture raises two challenges: capturing the codelet working set, and the cache hot set.

Memory capture Before replaying a codelet, the memory state from the original execution must be restored. This ensures that the replayed execution will be equivalent to the original one even considering data dependent branching code.

Multiple techniques exist to checkpoint the original memory state. Code Isolator [120] analyzes the static data flow of the original application to determine which data structures need to be captured. This method produces small dumps because only the required data are captured, but cannot deal with pointer aliasing. Astex [158] captures the convex hulls of the memory accesses. However, it does not preserve the data layout information and does not remap the memory at the same addresses during replay. Therefore,pointer based structures such as linked lists are not supported.

Codelet Finder [25, 1] takes a full snapshot of the original application address space. A full memory dump is very large but handles the pointer aliasing problem since the full memory is recorded. It also preserves the relative alignment and offsets among data structures. Nevertheless, a full snapshot of the application memory for each codelet can be prohibitive in terms of memory and replay time.

In CERE, we propose a page level granularity snapshot. Using the memory protection mechanism we capture the memory pages containing the working set. During replay, we remap this set of pages at their original addresses. This

ensures that the dump remains small and fast. Furthermore, the replay works even with complex pointer aliasing, because the memory layout is preserved.

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

Usual techniques [105] mitigate cold start bias by modeling the warm up effects during a window of time preceding the region of interest. Multiple heuristics [36, 83] have been proposed to optimally determine the window's size.

Two main approaches have been proposed in the literature for cache state warm up in code isolation. The first approach is to warm up the cache by running a few executions of the codelet itself [158, 25, 1]. The rationale is that the hotspots forming the codelets are loop based and thus can be warmed up by their own previous iterations. This heuristic proves to be efficient in many cases [149, 1]. The second, more accurate approach, warms up the cache by replaying the history of the memory accesses in a simulator [178] or using a warm up routine [120]. These techniques require tracing memory accesses which is costly and incurs significant slowdowns [67].

We propose two warm up approaches. The first is an optimistic warm up strategy that preloads the whole working set into the cache. The second, is a page memory tracing technique, which warms the cache by replaying memory access history at the memory page granularity.

2.2.3 Capturing parallel regions

Extending code isolation techniques to multithreaded simulations is difficult because of the threads interactions. Wenisch et al. [199] and Van Biesbrouck et al. [192] both propose techniques to accelerate multithreaded simulations. They both build their model under the assumption that each thread is independent. Therefore, they do not support explicit threads synchronization.

Perelman [157] applies the SimPoint [179] methodology to parallel applications using instruction-based sampling. However, Carlson et al. [26] and Ardestani et al. [8] both show that instructions are not a good proxy for execution time in multithreaded programs. Instead, they propose a time based sampling method.

Carlson et al. [27] propose BarrierPoint, a sampling methodology which detects globally synchronizing barriers in multi-threaded applications. BarrierPoint estimates total application execution time through detailed simulation of the most representative inter-barrier regions. Regions representativeness is defined with micro-architecture independent information and data signatures. BarrierPoint achieves an average speed up of 24.7 $\times$ over the NPB and Parsec benchmarks with an average error of 0.9%. The speedup is similar to the one achieved by CERE on parallel applications, yet the accuracy of BarrierPoint is better. Nevertheless, accuracies are not directly comparable since BarrierPoint measures accuracy on a functional simulator whereas PCERE measures accuracy on real hardware.

Proposed techniques for sampling parallel applications are similar to our work in that they extract representative phases from applications and allow accurate replay. Nevertheless, all proposed techniques must be used in a simulator, whereas our method is more versatile since it produces IR codelets that can be recompiled and run both on simulators and on real hardware. Another key difference is that BarrierPoint and the other sampling techniques do not allow changing the number of threads at replay. Each thread configuration requires a separate capture. Therefore, unlike CERE, they cannot be easily used to evaluate parallel scalability.

2.2.4 Comparison to related work

Our work builds upon two different lines of research: code isolation and sampled simulation of programs. Both share the same objective: accelerating performance evaluation of programs. Code isolation extracts pieces of a program as standalone codelets whereas sampled simulation uses a hardware simulator to replay a small set of representatives phases in a program.

A first set of papers [120, 158, 123, 1] study code isolation. Lee and Hall [120] introduce the concept of code isolation for debugging and iterative performance tuning. Their tool, Code Isolator, leverages the Stanford SUIF compiler to outline and generate codelets. They use Code Isolator on a finite element application, LS-DYNA, to quickly evaluate the L1 cache misses of the hotspots. Petit, Papaure, and Bodin [158] and Liao et al. [123] use code isolation for automatic kernel tuning and specialization. Akel et al. [1] evaluate the Codelet Finder tool, developed by Caps Entreprises [25] and study under which conditions codelets preserve the performance characteristics of the original programs.

Sampled simulation identifies and clusters similar program phases to reduce simulation time. Lafage and Seznec [114] propose a method to find slices of a program that are representative for data cache simulation. It uses hierarchical clustering on two metrics: memory spatial locality and memory temporal locality. SimPoint [178, 179] identifies similar program phases by comparing Basic Block Vectors (BBV). Phases are samples of 100M instructions. Simpoint reduces simulation time by removing repeated phases. BBV are program dependent, therefore SimPoint cannot use representatives of one program to predict another. Eeckhout, Sampson, and Calder [54] extend SimPoint by matching inter-application phases using microarchitecture-independent features. SimPoint and its extensions are similar to our work in that they extract representative phases from an application. But SimPoint must be used in a simulator, whereas our method is more versatile since the IR codelets can be recompiled and retargeted and run both on simulators and on real hardware.

Table 2.1 compares the features of the main code isolation tools on multiple criteria. First we compare the supported input languages, the isolation level and the support of indirect memory accesses. Second we consider if the tool allows replay on real hardware or is tied to a simulator. Finally, we examine whether the tool attempts to reduce the capture size, the number of working sets, or the number of representative codelets.

	CERE	Code Isolator	Astex	C. Finder	SimPoint
Support					
Language	C(++), Fortran, etc.	Fortran	C Fortran	C(++) Fortran	assembly
Extraction	IR	source	source	source	assembly
Indirections	yes	no	no	yes	yes
Replay					
Simulator	yes	yes	yes	yes	yes
Hardware	yes	yes	yes	yes	no
Reduction					
Capture size	reduced	reduced	reduced	full	-
Working set	yes	manual	manual	manual	yes
Codelet	yes (cf. chapter 3)	no	no	no	yes

Table 2.1: Feature comparison of code isolation tools.

2.3 CERE: Codelet Extractor and REplayer

2.3.1 IR Capture and Replay Overview

CERE (Codelet Extractor and REplayer) targets the LLVM Intermediate Representation. IR provides multiple advantages over source or assembly code isolation techniques as discussed in section 2.2.1.

Because it operates at the IR level, CERE can use any LLVM front-ends. For example CERE has been tested on all NAS and SPEC 2006 FP programs. While C and C++ benchmarks used the Clang front-end, Fortran programs used the GCC gfortran front-end through the dragonegg plugin [172]. CERE also works on less mainstream languages. For example CERE successfully extracts codelets from D [3] applications compiled with the LLVM D front-end, LDC.

Table 2.2 presents CERE's capture and replay process of a selected codelet. In Step 1, the input program is compiled to LLVM IR.

In Step 2, the region to be captured is outlined in a separate function using the CodeExtractor LLVM pass. CodeExtractor does a flow analysis to detect all the live-in and live-out dependencies of the region to extract [134]. This pass simplifies the codelet extraction process, since it extracts the region code in its own function. The codelet region is outlined in a new function. Finally, CodeExtractor inserts a call to the outlined function in the original code. The dependencies are preserved by passing the live-in and live-out values through function arguments. CodeExtractor is also the starting point for our portable memory capture mechanism discussed in section 2.3.5.

Step 3 generates the instrumented binary for memory capture. It inserts special calls to our capture library before and after the outlined region in the original application. The calls are used to trigger the memory and cache warm up state captures, described in sections 2.3.5 and 2.3.6. The instrumented binary execution generates a set of dump files that can be used during replay to restore the memory state and to warm up the caches. The aim is to ensure that the replay context closely mimics the original execution context.

Step 4 is the replay mechanism. It generates a wrapper to directly call the outlined region. This wrapper performs important steps to restore the original execution environment, such as variable cloning, cache and memory restoration.

#	Step	Output

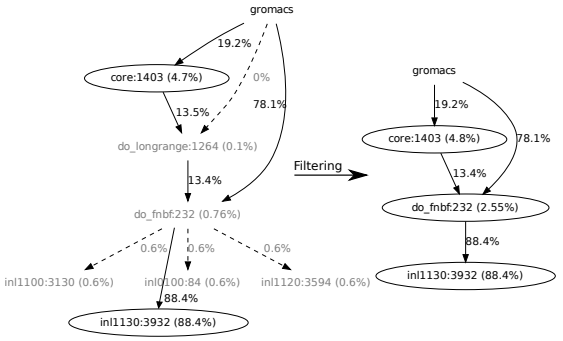
1	**Front-end:** Transform the C, C++, Fortran, D input program into LLVM Intermediate Representation (uses Clang, dragonegg, or LDC).	`original:`
		`%0 = load i32* %i, align 4`
		`%1 = load i32* %s.addr, align 4`
		`%cmp = icmp slt i32 %0, %1`
		`br i1 %cmp, ; loop branch here`
		`label %for.body,`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

2	**Outline:** Outline the region to extract. Flow analysis is used to compute all live-in and live-out values which are passed as arguments. (see section 2.3.4)	`define internal void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr) {`
		`%0 = load i32* %i, align 4`
		`...`
		`ret void }`
		`original:`
		`call void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr)`

3	**Capture:** Insert calls to CERE capture library. Run the instrumented binary to capture the runtime state. (see sections 2.3.5 and 2.3.6)	`define internal void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr) {`
		`call void @start_capture(i32* %i,`
		`i32* %s.addr, i32** %a.addr)`
		`%0 = load i32* %i, align 4`
		`...`
		`call void @end_capture()`
		`ret void }`

4	**Replay:** Generate minimal replay wrapper that calls the outlined region. Compile and run replay possibly with new optimization options or on a different architecture. (see section 2.3.7)	`define i32 @main(i32 %argc, i8** %argv){`
		`; Allocate clone variables`
		`%i = alloca i32`
		`%s.addr = alloca i32`
		`%a.addr = alloca i32*`
		`; Restore arguments and memory`
		`call void @restore(...)`
		`; Call outlined region`
		`call void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr`
		`; Anti-deadcode for live-out values`
		`call void @antideadcode(i32* %i)}`

Table 2.2: Codelet capture and replay main steps.



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

Figure 2.2: *(left)* CERE call graph, before and after filtering, for SPEC 2006 gromacs. Each node represents a captured codelet. The percentage inside the node is the codelet's self time. Edges represent calls to other codelets, the edge percentage is the time spent in calls to those nested codelets. *(right)* Replay percentage error of gromacs codelets using Working Set warmup.

The replay IR code can be compiled with different optimization flags to find the best performance configuration. Or it can be compiled with different back-ends to evaluate the performance on multiple targets. The replay process is detailed in section 2.3.7.

2.3.2 Application partitioning

To find interesting codelets for performance optimization, CERE concentrates on the application hotspots. In scientific applications, performance is mainly concentrated on loops. Therefore, CERE considers all the loops of the original program as potential candidates to be extracted as codelets. For OpenMP applications, CERE considers instead the parallel regions. Then, CERE profiles the candidate loops or parallel regions and keeps the ones significantly contributing to the total execution time.

CERE provides two profiler modes. First, a low overhead sampling profiler based on the Google Performance Tools library [73]. Second, an instrumentation profiler, which is slower but more precise.

Despite our efforts to restore the original execution environment through warm up code reinlining, and variable cloning (see sections 2.3.6 and 2.3.7), the codelet replay sometimes does not match the original loop performance. Clearly, those *ill-behaved* codelets cannot be used as a performance proxy in benchmarking or optimization studies. Therefore, CERE runs a sanity check where it replays and profiles each codelet to ensure that only valid codelets are returned to the user. The tolerated discrepancy threshold can be configured. Its sensitivity is presented on figure 2.3. We consider that codelets match the original performance when the replay error is under 15%.

After collecting profile data, CERE produces an annotated call graph such as the graph in figure 2.2. This call graph is then pruned by removing regions contributing for less than 1% to the total execution time. Furthermore, if an *ill-behaved* codelet is detected, CERE also removes it from the call graph. When removing a region from the call graph, we propagate its *self time* to its parent codelets. In our example, the time from the three `inl` removed regions is propa-

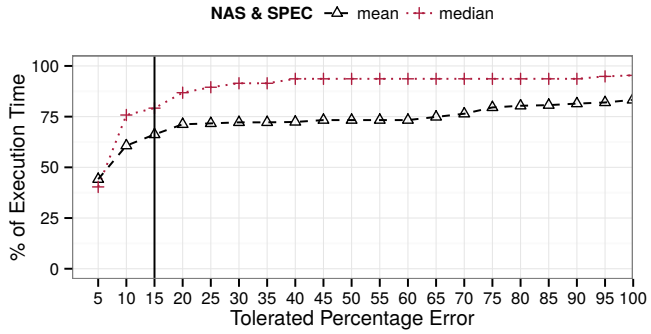


Figure 2.3: Mean and median captured execution time as a function of the tolerated replay error on the NAS and SPEC 2006 FP benchmarks. The mean is lower than the median due to the IO-intensive and short kernel benchmarks described in section 2.4, which skew the distribution.

gated to their caller `do_fnbf`. In the example of figure 2.2, since all the codelets match the original execution time, none would be removed.

The above selection algorithm extracts all the well-behaved codelets whose contribution to the program execution time is over a given threshold. To trade coverage for replay time, for example, when using codelets to accelerate system benchmarking, the user wants the minimal set of codelets that can be quickly replayed while simultaneously capturing the application performance accurately. For this purpose, CERE includes a codelet *selector* that uses integer linear programming to find an optimal codelet set. It is similar to the tuning selection algorithm proposed by Pan and Eigenmann [152]. In the example in figure 2.2 it would drop codelets `do_fnbf` and `core`, losing less than 7.35% coverage but significantly reducing the replay cost.

2.3.3 Partitioning OpenMP programs

In OpenMP programs, the application concurrency is described through a set of compiler directives and library calls. For instance, a parallel region can be declared using the directive `#pragma omp parallel`. Figure 2.4 shows a simple C OpenMP program where each thread prints its thread identifier.

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

Figure 2.4 shows how LLVM outlines the region code in a `microtask` function. `kmpc fork`, an OpenMP Runtime library function, spawns a pool of threads. Then, every thread runs the outlined `microtask` function which describes the region parallel work.

Multithreaded execution is a well known source of nondeterminism: race conditions and synchronization delays between threads may change the order of the operations from one execution to the next. In particular, when multiple

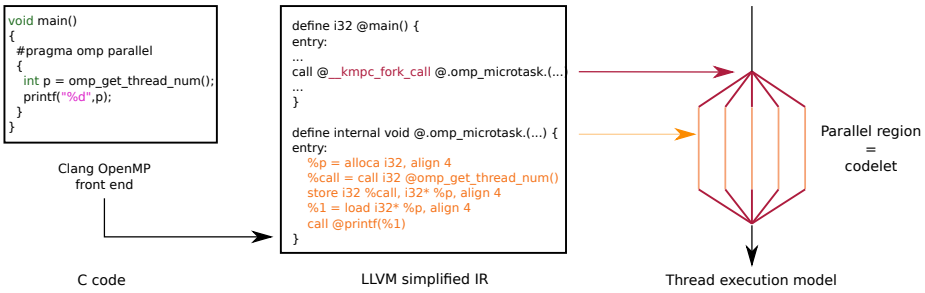


Figure 2.4: Clang outlines each C parallel region as an independent IR function:`omp_microtask`. The call to `kmpc_fork` spawns a pool of threads that runs the outlined microtask.

threads are running, each one may be executing a different region of code. This makes it difficult to isolate a particular region of code.

To avoid thread nondeterministic issues, parallel codelets start at the beginning of a parallel region and finish at the end of the region. Indeed, the beginning of an OpenMP region is a global synchronization point where all threads positions in the program are known. Capturing codelets at the start of the region has another advantage: it enables changing the number of threads at replay. Indeed, the capture happens just before the call to `kmpc fork` that decides how many threads are spawned.

Lock support OpenMP uses futex (fast userspace mutexes) calls to implement the lock support on Linux. Each futex requires a kernel space wait queue. System calls are used to request operations on the wait queue from user space. Our memory capture only saves the user space process memory, therefore it does not preserve the state of the futex wait queue.

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

2.3.4 Codelet checkpoint-restart strategy

Traditional checkpoint techniques [51] can save the state of a program at any given point. A full dump of the memory and of the register banks including the program counter allows to restart the program after capture. Yet, this approach requires that the replayed code keeps the same code layout and uses exactly the same registers as during the capture. Traditional checkpointing is therefore not suited to test compiler optimizations which may remap registers or change code layout. Also, it limits codelet portability to architectures sharing the same Application Binary Interface (ABI) and register layout.

Codelet based piecewise iterative optimization and architecture selection require a portable checkpoint-restart strategy. The outlining pass (Step 2 in Table 2.2) wraps and isolates the region of interest inside a separate function. Because the region now follows a function call, we can guarantee that the accessed

data is either in memory or is passed as arguments to the outlined function.

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

Codelet portability has been extensively tested and works across six different Intel CPU generations (Atom, Core 2 Duo, Nehalem, Sandy Bridge, Ivy Bridge, and Haswell) running various 64-bit Linux distributions on the NAS and SPEC codelets. Codelet portability has also been tested between different ARMv8 architectures (ARM big.LITTLE and ThunderX2). A codelet repository for ARMv8 is available [145].

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

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

2.3.5 Capturing the memory

CERE captures codelet's working sets by intercepting accesses to the memory pages. First, CERE guarantees that all the memory locations accessed by the original program are dumped. Second, because only the touched pages are saved, the memory dump is the smallest page-granularity over-approximation. Therefore, it can be easily stored and distributed.

Figure 2.5 shows the memory dump process. First, all the memory pages of the process are protected and a special segmentation fault handler is set. Each time a protected page is accessed, a segmentation fault occurs and triggers the handler. The handler dumps the touched memory page to disk and unprotects it before continuing the original program execution.

It is important to protect all newly allocated memory. If memory is allocated but returned to the user unprotected, the tracer misses the access to the memory segment. We catch all calls to the memory allocation library, such as malloc, realloc, or memalign using the `LD_PRELOAD` mechanism. However, some special memory sections must not be protected, such as the pages containing the code of

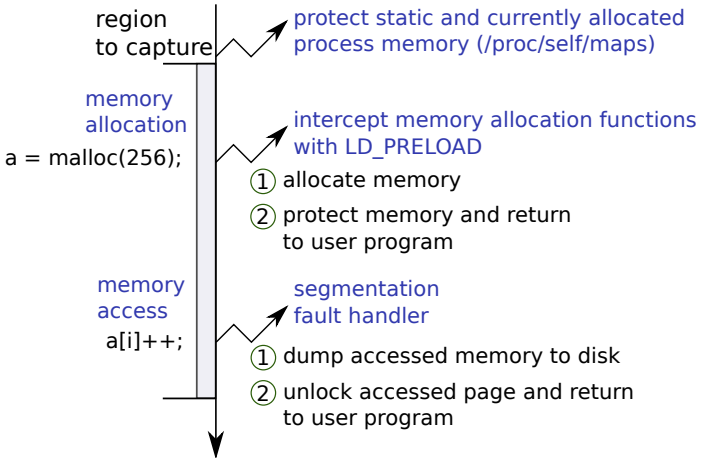


Figure 2.5: The memory dump process operates at page granularity. Each page accessed is dumped by intercepting the first touch using memory protection support.

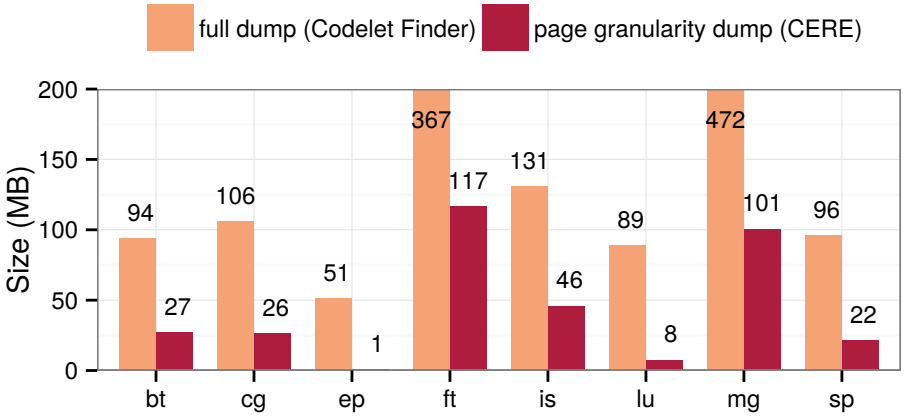


Figure 2.6: Comparison between the page capture and full dump size on NAS.A benchmarks. CERE page granularity dump only contains the pages accessed by a codelet. Therefore, it is much smaller than a full memory dump.

the tracing library and the segmentation fault handler itself. Therefore CERE carefully avoids protecting its own pages and system specific memory segments.

Figure 2.6 compares the average dump size for the NAS benchmark codelets for two techniques: CERE's page granularity dump and Codelet Finder's full dump. As can be seen, the page granularity dump is 3 to 51 times smaller than a full dump. With this technique CERE extracts light portable codelets from industrial application with large working sets.

CERE captures the memory state at a page level granularity by only saving touched pages. After protecting the whole target application memory, the OS raises a signal when a memory page is touched. This signal is caught by CERE's handlers which unprotect and dump the memory page associated with the signal.

NUMA Aware Warm up Due to the node local first touch policy in Linux, a page is mapped to the first thread accessing it, and therefore to the corresponding NUMA domain. To guarantee that the codelet replay has the same behavior as the original region, we must ensure that pages are mapped to the same NUMA domains as they have been in the original run. The problem is that pages are not necessarily bound to the same NUMA domains across the different thread affinities. For instance, scatter runtime strategy maximizes the number of NUMA domains while compact minimizes it.

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

Multithreaded capture The memory must not be modified while we attempt to protect it. In multithreaded capture, threads have to be stopped before the tracing thread protects memory pages. Otherwise, CERE can attempt to protect a segment which is no longer valid. Indeed, a race condition exists: a thread may protect a page which has since been deallocated by another running thread. To avoid the issue the tracing thread should be able to stop all the other threads.

A first solution would be to send a `SIGSTOP` signal to the other threads. Yet, sending `SIGSTOP` to a process stops all its threads, including the thread which actually sent the signal. This will not work since the tracing thread will also be stopped hanging the capture. A second solution would be to use functionalities provided by the Oracle Solaris OS `thr_suspend` and `thr_continue` or Window OS `ResumeThread` and `SuspendThread` which allow stopping and restarting each individual thread. Unfortunately, the POSIX standard does not implement these functions. It is possible to simulate stop/restart functionalities by using a shared mutex, but this requires knowing the spawned threads in advance and modifying their code to include calls to the mutex. Since the CERE capture library does not make assumptions about the underlying program, we cannot apply this solution.

To address this challenge, we leverage the `ptrace` mechanism. Ptrace is a system call which allows a process called `tracer` to monitor another process called `tracee`. Tracer can examine and change the tracee's memory and registers. To follow a thread, the tracer must attach it with ptrace. Since this

command is per thread, the tracer must attach each thread of the tracee. (see block `Attach all threads` in Fig. 2.7). So the capturing process and the application respectively act as tracer and tracee.

When a signal is delivered to the tracee, the kernel stops the process and sends the signal to the tracer. The ptrace API provides a mechanism called `signal injection and suppression`: the tracer can choose to inject or suppress the signal. If the signal is injected, it is sent to the tracee. If the signal is suppressed, it is lost and the tracee remains stopped. We use this mechanism to capture the signal `SIGSEGV` raised when a thread touches a protected page. Protecting or unprotecting tracee's memory pages cannot be done from the tracer since a process can only modify its own memory.

The code for dumping and unprotecting a page is injected by the tracer in the tracee memory with ptrace. Then, the tracer resumes the tracee to execute the injected code, a `SIGTRAP` call at the end of the injected payload returns the focus to the tracer. (See block `Memory capture` Fig. 2.7)

Figure 2.7 details the capture which is composed of four successive phases:

1. `Attach all threads:` the tracer attaches tracee threads with the ptrace attach command. Then it sends a SIGSTOP to each tracee to stop it. The tracer checks that the SIGSTOP has been received for each tracee. Once all the tracee threads are stopped the tracer is ready for the second phase.

2. `Memory Protection Mechanism:` the tracer protects the whole memory of the tracees by injecting a protecting assembly payload and restarts the threads. If a thread was already stopped before receiving the tracer SIGSTOP, the queued SIGSTOP signal must be cleared at restart to avoid a deadlock.

3. `NUMA first touch and page trace` starts once all the memory is protected. It captures the `tid` of the first thread to touch each page. It also keeps a trace of the most recently touched pages that is used to warm up the cache state at replay.

4. `Memory capture` starts when the region to capture is reached. CERE reprotects the whole memory and starts executing the region. It dumps all touched pages that are accessed until the region ends.

2.3.6 Capturing the cache state

We address the problem of cache warm up for codelet replay previously discussed in section 2.2.2. CERE includes three warm up strategies: *Cold*, *Working Set*, and *Page Trace*.

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

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

The *Page Trace* strategy mitigates cold start bias by replaying a memory trace at a page level granularity. It is less accurate than a full memory trace

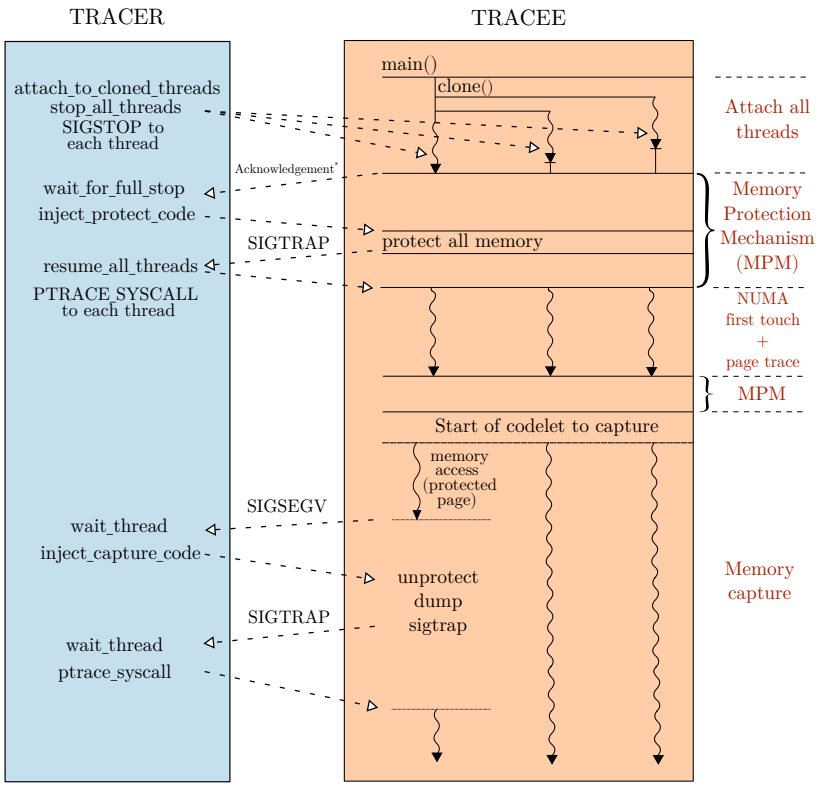


Figure 2.7: Multithreaded capture mechanism with ptrace. Capture is split into two distinct processes : the CERE capturing method as the tracer the studied application as the tracee.

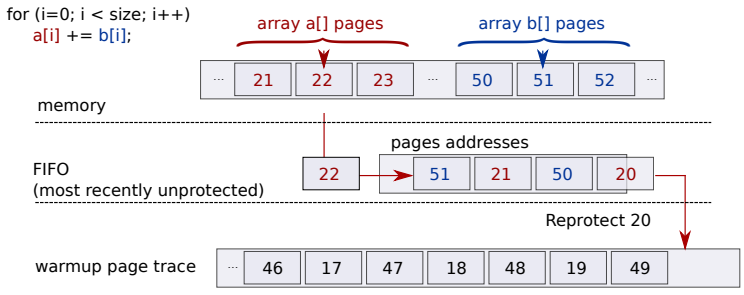


Figure 2.8: Cache page tracer on a simple codelet adding two arrays. Each page access is logged. Recently unprotected pages are kept in a FIFO with N slots (here $N = 4$). Once evicted from the FIFO, the pages are protected again.

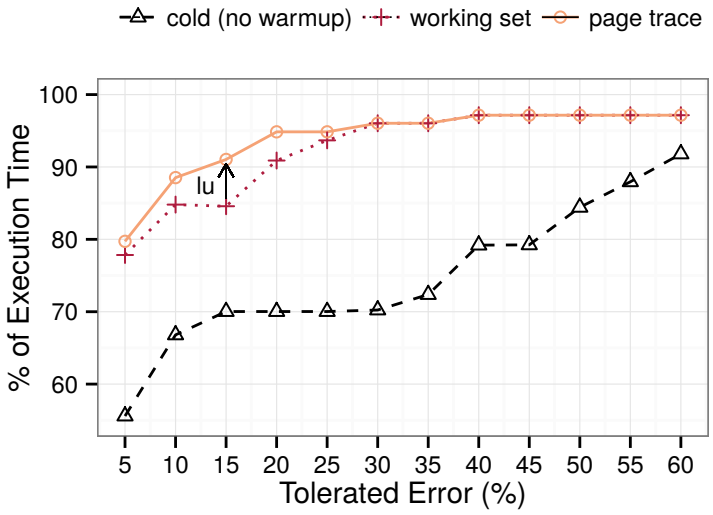


Figure 2.9: Comparison of the three cache warm up techniques included in CERE on NAS codelets. The plot shows the percentage of execution time as a function of the replay error. Page Trace and Working Set warm up achieve the best results. Page Trace is more accurate than Working Set on the LU benchmark.

	CERE	ATOM 3.25	PIN 1.71	Dyninst 4.0
cg.a	19.4	98.82	222.67	896.86
ft.a	24.1	44.22	127.64	1054.70
lu.a	62.4	80.72	153.46	>>301.4
mg.a	8.9	107.69	168.61	989.53
sp.a	73.2	67.56	93.04	>>203.66

Figure 2.10: CERE capture overhead. We measure the slowdown of a full capture run against the original application run. (The overhead takes into account the cost of writing the memory dumps and logs to disk and of tracing the memory accesses during the whole execution.). We compare to the overhead of other memory tracing tools as reported by Gao et al. [67]. Gao et al. did not measure bt, is, and ep.

warm up, but much faster. It provides a good trade-off between cost of codelet capture and replay accuracy. The technique is similar to the page tracing technique in [23].

Our page tracer is implemented on top of the memory dump process described in section 2.3.5: all the memory pages are protected, and a special segmentation fault handler intercepts accesses to memory. The difference is that unlike the memory dump in which only the first touch to a page is important, the page tracer should capture all the memory accesses to a page.

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

To reduce the cost of the technique, we keep the most N recently accessed pages unprotected. The tracer uses a FIFO to track the recently accessed pages. Each time an access to a page is detected, the page is unprotected and added to the FIFO. The oldest page is popped from the FIFO, reprotected, and added to the page access log. Figure 2.8 illustrates this approach on a codelet that adds two arrays together.

If a codelet simultaneously accesses less than N separate memory streams, the FIFO ensures that a page remains unprotected for all the consecutive streamed accesses. Assuming a stride-one access, the page tracer handler is only invoked every 4096 byte (for 4K pages). Therefore, we choose N higher than the number of separate memory streams accessed by most loops. Kashnikov et al. [102] show that most application loops use less than 16 simultaneous streams. In our experiments we choose $N = 64$. Nevertheless, by keeping the most recent N pages unprotected, our trace is less accurate. In the code of figure 2.8 for instance, each cell of array `a` is accessed twice (it is first read then written to), but the page tracer only sees the first access. When interpreting the trace, one must keep this inaccuracy in mind: the trace presents which pages were accessed but neither how many times nor the precise ordering.

Figure 2.9 compares the three warm up techniques implemented in CERE on the NAS benchmarks. The *tolerated error* is the maximum percentage difference between the original execution time and the replayed execution time. The plot shows the percentage of execution time of NAS codelets replayed with an error smaller than the *tolerated error*. For example, if we use the *Cold* strategy, 70% of the execution time can be replayed with an error under 15%.

We observe that the *Working Set* and *Page Trace* strategies significantly improve the replay accuracy. On the NAS codelets, the *Page Trace* strategy is slightly better than the *Working Set* one. The improvement comes from LU codelets whose irregular accesses are better captured by the *Page Trace* warm up.

Figure 2.10 shows the overhead of the capture run for the NAS benchmarks. For each benchmark we compute the slowdown between the original run-time and a full capture run. This measure includes initialization of the capture library, writing the dumped pages and the memory trace logs to disk for all the codelets in the application. `IS` is particularly slow because one of its codelets accesses memory randomly. This rapidly fills the pages FIFO and slows down the tracer. Figure 2.10 also compares CERE capture cost with the overhead of other memory tracing tools. CERE overhead is similar to ATOM 3.25 overhead and lower than PIN 1.71 and Dyninst 4.0 overhead.

2.3.7 Replay

Once the memory and cache state are captured, a codelet can be replayed. Because codelet replay is fast, it is useful to quickly evaluate the impact of moving to a different architecture or changing compiler optimizations.

To replay a codelet, CERE generates the special wrapper shown in Step 4 of Table 2.2. First, it allocates clone variables for the input and output flow dependencies to the outlined region. Second, it restores memory and cache state. Finally, it calls the outlined region.

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

A second remark is that the outlining compilation pass dereferences the input and output dependencies. By passing the variables by reference, it is easy to preserve the values modifications during the codelet execution. This is a classic technique in code outliners [120, 123] which has the unfortunate side effect of disabling many compiler optimizations. In many codelets, dereferencing makes codelet replay slower and therefore unfit to be used as performance proxies of the original code. We solve this problem in three steps. First, we tag each dereferenced pointer with the IR attribute *NoAlias* which informs LLVM that the dereferenced pointer is not aliased. This is known because the extra dereference is created by CERE outliner and used only once during replay. Second, we tag the outlined function itself with the attribute *AlwaysInline* which forces LLVM to reinline the function in the replay wrapper. Third, LLVM alias analysis optimization pass removes the extra layer of dereference. In section 2.4 the effect of reinlining and marking cloned variables as NoAlias are measured. These two techniques improve replay accuracy in eleven applications without degrading the other benchmarks

One could think that the outlining step is unnecessary since it is reverted later on by LLVM inliner pass. But as explained in section 2.3.4, the outlining step guarantees that CERE finds a safe checkpoint to capture the context just before a procedure call.

Once the replay wrapper is generated, it is compiled and possibly optimized depending on the optimization flags selected by the user. To generate the final replay binary, CERE uses a custom link script, that reserves the virtual memory segments occupied by the working set pages during the memory capture. This step is needed so that CERE can preserve the original memory layout.

2.4 Evaluation of CERE

We evaluate CERE capture coverage and replay accuracy on the NAS 3.0 serial benchmarks and the SPEC 2006 FP benchmarks. All the benchmarks in both test-suites are used in our evaluation, therefore CERE was tested on twenty-six different benchmarks in total. NAS benchmarks were tested on class A and B data sets. SPEC benchmarks were tested on *ref* data sets.

The experiments were performed on the machines described in table 2.3.

	Atom	Core 2	Nehalem	Sandy B.	Ivy B.	Haswell
CPU	D510	E7500	L5609	E31240	i7-3770	i7-4770
Frequency (GHz)	1.66	2.93	1.86	3.30	3.40	3.40
Cores	2	2	4	4	4	4
L1 cache (KB)	2×56	2×64	4×64	4×64	4×64	4×64
L2 cache (KB)	2×512	3 MB	4×256	4×256	4×256	4×256
L3 cache (MB)	-	-	12	8	8	8
Ram (GB)	4	4	8	6	16	16

Table 2.3: Test architectures.

They belong to six different Intel CPU generations (Atom, Core 2 Duo, Nehalem, Sandy Bridge, Ivy Bridge, and Haswell) and possess quite distinct memory hierarchies. These machines were selected to validate that CERE replay process is portable across architectures.

We compared the replay times of the NAS codelets with memory captures done on Core 2 and Haswell and observed no difference. We conclude that the architecture used for capturing the memory has no significant impact on replay accuracy. Yet for completeness, the reader should note that the final memory capture dumps used in the following experiments were performed on the Core 2 machine for the NAS benchmarks and on Haswell for the SPEC benchmarks.

The experiments were performed with CERE v0.1.0. C and C++ benchmarks were compiled with Clang 3.3 and Fortran benchmarks were compiled with GCC 4.6 through dragonegg.

As discussed in section 2.3.2, we consider that a codelet is accurately replayed if its replay performance is within 15% of the original execution time.

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

Figure 2.11 shows for the NAS and SPEC 2006 FP benchmarks the percentage of execution time captured by codelets and the percentage of execution time that could be accurately replayed. On average, the extracted codelets cover 97.3% of the execution time in NAS and 76.6% in SPEC.

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

NAS codelets were replayed in two different architectures to show that CERE reliably supports multiple architectures. The small differences in coverage between Haswell and Core 2 are due to the changes in contribution of codelets to the execution time, for example CG spends relatively more time on I/Os on Haswell architecture.

SPEC FP results are evaluated on the Haswell architecture. Eleven out of eighteen benchmarks have high coverage and replay accuracy, over 75%. Here is a list of the problems affecting the seven remaining benchmarks:

sphinx3, wrf, povray, and calculix have low coverage because most of the time is spent in I/O operations. The current version of CERE does not capture

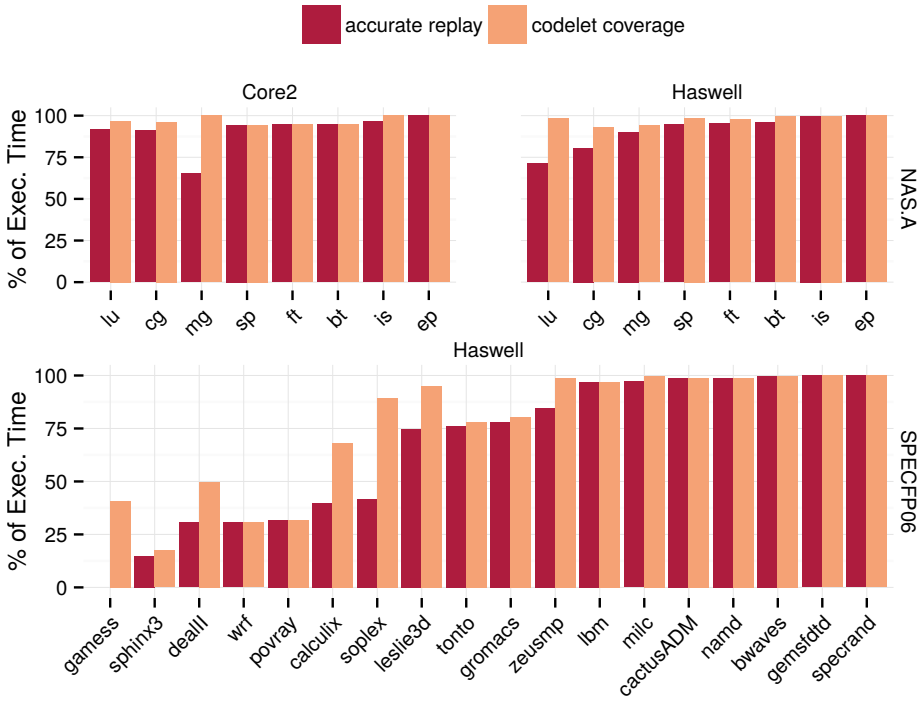


Figure 2.11: Evaluation of CERE on NAS and SPEC FP 2006. The Coverage is the percentage of the execution time captured by codelets. The Accurate Replay is the percentage of execution time replayed with an error less than 15%.

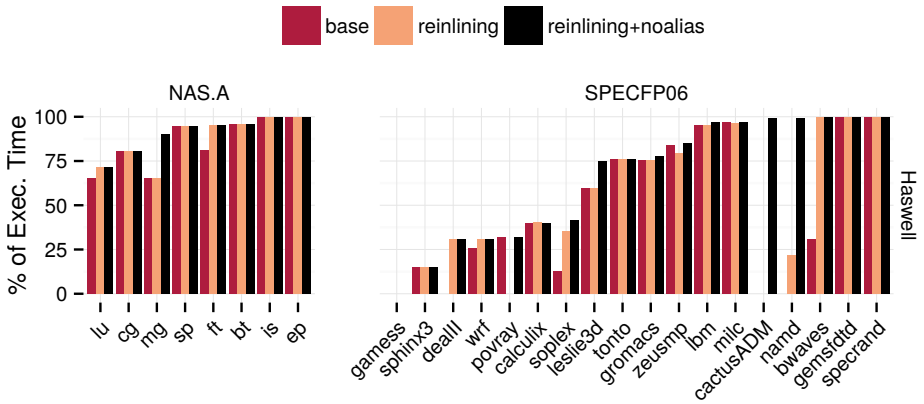


Figure 2.12: Percentage of execution time accurately replayed (error $< 15\%$) on the NAS and SPEC FP benchmarks with different replay configurations. Reinlining and explicitly marking cloned variables as NoAlias improve replay accuracy in eleven benchmarks.

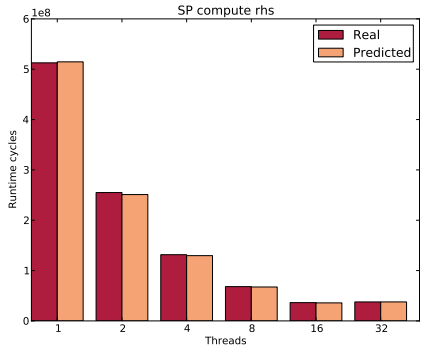


Figure 2.13: Real vs. PCERE execution time predictions on Sandy Bridge for the SP compute rhs codelet

codelets performing I/O because the dump does not preserve file descriptors state. However, 100% of captured codelets match.

gamess and dealII have low coverage because most of the performance is spent in loops taking less than 2000 cycles, which were not considered.

gamess has low matching because the only remaining codelet, covering 40% of the execution time, is not accurately replayed. It is due to a warm up bug which is being investigated.

calculix has low matching because of a borderline codelet `isortii` which has a replay error of 16% but accounts for 10% of the running time. It is a sort function which is very sensitive to warm up effects.

soplex has low matching because CERE fails, due to a capture bug, to replay its main codelet covering 47.4% of the execution time.

Figure 2.12 shows that the reinlining and NoAlias-tagging performed during the replay compilation pass are beneficial in 11 benchmarks. Overall CERE coverage and accuracy are high in both NAS and SPEC benchmarks, showing that CERE codelets can be efficiently used as performance proxies for many applications.

CERE allows capturing OpenMP multithreaded programs and replay them with a different number of threads. Popov et al. [161] shows that CERE replay accuracy on NAS parallel benchmarks is high. As an example, in figure 2.13 we compare the real and predicted execution time on SP `compute rhs` parallel codelet.

CERE has higher replay accuracy than the state of the art code isolator tool, Codelet Finder. On NAS, Codelet Finder accurately replays 69% [1] of the execution time, whereas CERE replays 90.9%. On SPEC, Codelet Finder has very low replay accuracy or fails to extract codelets for many benchmarks (the 2013 version of Codelet Finder hangs on gamess, gromacs, cactus, calculix, tonto, specrand, and wrf), whereas CERE accurately replays 66.3% of the SPEC execution time.

CERE includes a report generator that automatically captures the execution traces, selects representative invocations and computes coverage and replay accuracy of a given set of benchmarks. The user clicks on any captured codelet in the call graph to see its invocation clustering and replay accuracy

statistics. The reports for all NAS and SPECG benchmarks can be viewed at http://benchmark-subsetting.github.io/cere/.

2.5 Conclusion

CERE is an LLVM based Codelet Extractor and Replay framework. It finds and extracts the hotspots of an application as *codelets*. Codelets can be modified, compiled, run, and measured independently of the original application. Code isolation reduces benchmarking cost and allows piecewise optimization of large HPC applications.

In this chapter we have focused on the technical foundations of CERE and its capture and replay mechanisms. We have demonstrated that CERE codelets are a good proxy for HPC optimization since they accurately replay 66.3% of the SPEC benchmarks.

In section 3.1 of the next chapter, we will show that similarities across codelets can be taken advantage of. By grouping codelets into a small number of similar classes, we further reduce the cost of HPC optimization.

3 Reducing HPC search space

Contents

This chapter includes contributions from the following publications:

- Pablo de Oliveira Castro, Eric Petit, Asma Farjallah, and William Jalby. "Adaptive Sampling for Performance Characterization of Application Kernels." In: *Concurrency and Computation: Practice and Experience* (2013). ISSN: 1532-0634. DOI: 10.1002/cpe.3097
- Pablo de Oliveira Castro, Eric Petit, Jean Christophe Beyler, and William Jalby. "ASK: Adaptive Sampling Kit for Performance Characterization." In: *Euro-Par 2012 Parallel Processing - 18th International Conference*. Ed. by Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis. Vol. 7484. Lecture Notes in Computer Science. Springer, 2012, pp. 89–101. ISBN: 978-3-642-32819-0
- Pablo de Oliveira Castro, Yuriy Kashnikov, Chadi Akel, Mihail Popov, and William Jalby. "Fine-grained Benchmark Subsetting for System Selection." In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM. 2014, p. 132

An accurate HPC performance model captures all interactions among the system's elements such as: multiple cores with an out-of-order dispatch or complex memory hierarchies. Building analytical models is increasingly difficult with the complexity growth of current architectures. Instead, one often empirically measures the performance by running a benchmark application under different settings, which we will call *factors*.

This benchmarking can be expensive. First, HPC applications have often large code bases and long running times; so measuring a single run can be expensive. Moreover, as the number of considered factors grows – architecture, cache levels, problem size, number of threads, thread mappings, and access patterns – the size of the design space explodes and exhaustively measuring

each combination of factors becomes infeasible. To mitigate the cost of the design space exploration, we propose two complementary approaches in this chapter.

Section 3.1 shows how breaking an application into codelets using the CERE framework presented in chapter 2 can speedup this process. The key idea is to minimize redundancy inside an application or benchmark suite. Instead of measuring the whole application; we select a minimal set of codelets that capture its performance behavior and use it as a proxy for doing the measures.

Section 3.2 presents a second approach for reducing the factor combinatorial explosion. We sample only a limited number of factor combinations. From the sampled points, we build a surrogate performance model to study, predict, and improve architecture and application performance. The factors combinations are chosen carefully through various adaptive sampling strategies that minimize the model error while keeping the number of measures small.

3.1 Exploiting codelet similarities

In this section, we exploit similarities between codelets to find a minimal set of benchmarks that faithfully capture the original HPC application behavior.

We address two sources of redundancy:

- **Multiple invocations:** Codelets that are repeatedly invoked with the same context in an application lifetime, have the same running time for each invocation. In applications where a single codelet is called thousands of times, measuring only a few invocations achieves significant gains in benchmarking time.

- **Similar computation kernels:** Benchmark suites share many similar codelets: simple ones, like set-to-zero or memory copy loops, and more complex ones, like Single-precision real Alpha X Plus Y (SAXPY) loops. There is no need to measure multiple copies of the same code.

Our hypothesis is that codelets from the same cluster share similar performance features and should react in the same way to architecture change or optimizations. For example, memory-bound codelets will benefit from faster caches, whereas highly vectorized codelets will benefit from wider vectors. Therefore, by measuring a single representative per cluster we can extrapolate the performance of all its siblings.

Using this method we achieve significant speedups in benchmarking time. First, because only one benchmark per cluster is executed, and second, because it uses fewer invocations than the original application.

3.1.1 Background on benchmark reduction methods

A first set of papers study similarities among programs, in order to uncover hidden redundancies or predict performance. Vandierendonck and Bosschere [193] analyze SPEC CPU 2000 execution time. They group applications according to their performance bottlenecks, showing that SPEC CPU 2000 contains redundant benchmarks. Hoste *et al.* [92, 91, 90] use microarchitecture-independent metrics to build a performance database that is used to predict performance of

new programs. Phansalkar *et al.* [160] use a similar approach with hardware performance counters and statistical methods to analyze the redundancy of the SPEC CPU 2006 benchmark suite. Compared to the whole benchmark suite, 6 integer programs and 8 floating point programs capture the weighted average speedup with an error of 10% and 12% respectively. Bienia *et al.* [14] study redundancy between SPLASH-2 and PARSEC applications with statistical and machine learning methods They use execution-driven simulation with the Pin tool to characterize program's workloads and collect a large set of metrics. They use Principal Components Analysis to improve the original feature space and hierarchical clustering to find redundancies between applications.

A second set of papers discusses benchmark reduction methods for speeding simulation time. Citron *et al.* [34] survey 173 papers from ISCA, Micro, and HPCA conferences and criticize methods that only use a subset of applications from the SPEC CPU benchmark suite. They demonstrate how projecting performance of a subset on the whole benchmark suite can bias speedups and yield incorrect conclusions. Contrary to existing approaches, our subsetting method does not remove entire applications from a benchmark suite, but only removes redundant fragments. Despite reducing the total time required for the benchmark suite evaluation, our method keeps the important performance information from the whole suite.

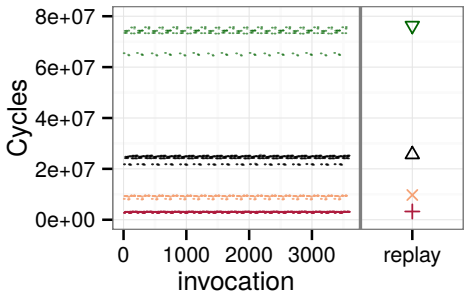
Lafage and Seznec [114] propose a method to find slices of a program that are representative for data cache simulation. It uses hierarchical clustering on two metrics: memory spatial locality and memory temporal locality. SimPoint [178, 179] is a tool which identifies similar program phases by comparing Basic Block Vectors (BBV). Phases are samples of 100M instructions. BBV are program dependent, therefore SimPoint cannot use representatives of one program to predict another. In constrast, our method can take advantage of similarities across different applications. We show in Section 3.1.6 that exploiting inter-applications redundancies reduces the number of representatives while preserving accuracy. Eeckhout *et al.* [54] extend SimPoint by matching inter-application phases using microarchitecture-independent features. SimPoint and its extensions are similar to our work in that they extract representative phases from an application. But unlike CERE, SimPoint can only be used within a hardware simulator.

3.1.2 Clustering invocations of the same codelet

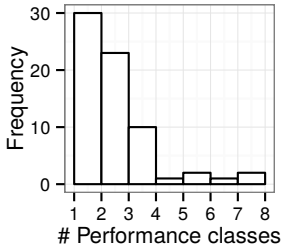
Inside an application, the same codelet may be called multiple times. In many codes two invocations of the same codelet may have different execution times. This is due to the different working sets or initial conditions.

For example, the codelet `make_ft@shell2.F90:1133` extracted from `tonto` is one of the steps of a specialized Fast Fourier Transformation. In the original application, this loop is called 3587 times with different workloads. Figure 3.1a shows its execution trace. A cluster analysis of the invocations reveal that they can be sorted into 4 performance behaviors, which are represented with different colors in the figure. Other codelets such as `flux_lam@flux.f:58` extracted from `bwaves`, have a constant workload size but the first invocation is slower because of cache warmup effects.

To accurately replay a codelet, we must capture each different invocation state. When the number of invocations is high, this process becomes costly



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



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

Figure 3.1: Working set reduction: (a) A clustering analysis of tonto's trace detects four different performance behaviors depending on the workload. The initial 3587 invocations are captured with only four representative replays. (b) Most of the NAS codelets can be captured with less than four representative working sets.

both in time and space. Fortunately, applications exhibit some regularity; and most of the time the invocations can be reduced to a few representative classes. Figure 3.1b shows the distribution of the number of different classes across all NAS benchmarks. One can note that most of the codelets can be captured with less than 4 representatives.

To automatically detect the performance classes and generate a set of representative captures, we use CLARA clustering algorithm [103] which can cope with large traces with more than 10^9 invocations found in some of our benchmarks.

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

3.1.3 Clustering codelets with the same performance behavior

The previous section showed how different invocations of the same codelet can be clustered in a small set of classes. A second reduction in the number of replays can be achieved by detecting and exploiting similar and repeated computation patterns. Benchmark suites and applications naturally contain redundant computation patterns across different benchmarks. For instance, two linear algebra solvers, despite using different algorithms, will share common computation patterns such as vector copy loops, inner products, or matrix-vector multiplications.

The method presented in figure 3.2 detects repeated computation patterns, and keeeps only one representative copy of each, reducing a suite of benchmarks to an essential set of micro-benchmarks. Because only duplicated patterns are removed, the important performance features of the original benchmarks are preserved.

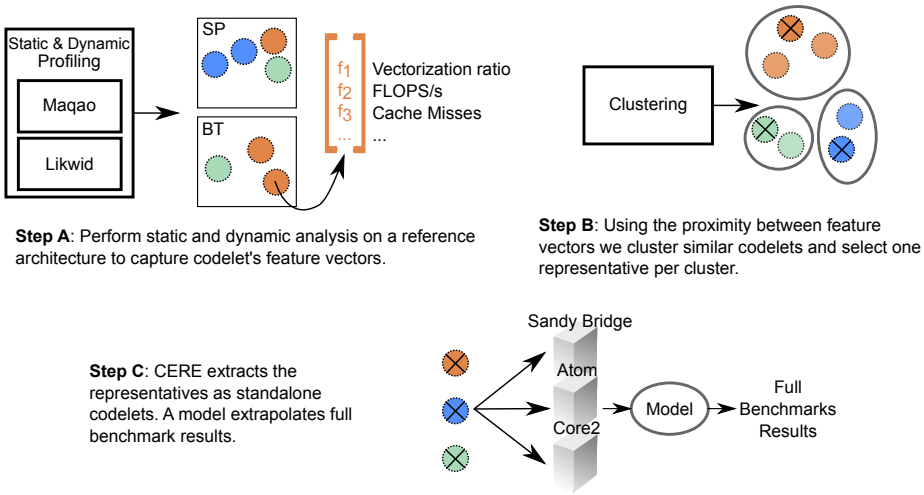


Figure 3.2: Overview of the benchmark reduction method for execution time prediction. The reduction method can be extended to support compiler or optimization evaluation.

Step A statically analyzes and profiles each codelet on an architecture chosen as a reference. Each codelet is tagged with a feature vector containing static and dynamic features.

Step B groups codelets sharing similar feature vectors into clusters. The feature vector is used as a performance signature to detect similar codelets.

Step C selects a representative for each cluster and extracts it as a standalone CERE codelet.

Performance Features To detect similar codelets we measure performance features, both static and dynamic. Static features are useful to evaluate the assembly code quality and to detect performance problems specific to the microarchitecture. MAQAO static loop analysis [50, 102] provides a set of static metrics for the innermost binary loops. Examples of such metrics are the size of the loop, the pressure on dispatch ports, the number of used registers, the type of instructions. To compute these metrics, MAQAO disassembles and analyzes the binary. Some MAQAO metrics, provide a lower bound on performance by assuming that all the memory access hit the L1 cache. For benchmarks with datasets larger than L1, those metrics are less relevant. Hardware counters help us to overcome this issue and deal with dynamic hazards.

To characterize the dynamic behavior of codelets, we use a set of metrics provided by the Likwid tool [189]. Likwid measures hardware performance counters and derives a set of dynamic performance metrics such as execution time, cache misses, floating point instructions per second, or memory bandwidth.

MAQAO and Likwid produce 76 individual features. Irrelevant features add noise that degrades the clustering and the prediction accuracy. Therefore, it is important to wisely select features, keeping only those that adequately represent program behavior and improve prediction.

Evaluating the 2^{76} combinations of features is too costly. To find a good set of features in a reasonable time, we use genetic algorithms [200]. Genetic

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

Table 3.1: Feature set to estimate performance similarity of codelets.

algorithms (GA) start with a population of randomly generated individuals. In our case, each individual represents a candidate feature set. This population evolves towards an optimal solution by recombining the best individuals with crossover and mutation operators [201].

To evaluate individuals, we consider the average prediction error of Numerical Recipes [163] benchmarks on two architectures: Atom and Sandy Bridge. Best individuals should have a high prediction accuracy on both architectures but with a low number of representatives. To achieve this objective we choose the fitness function: $\max(error\_atom, error\_sandybridge) \times K$ where K is the number of clusters. We intentionally leave Core 2 and NAS benchmarks out of the training process to fairly evaluate how our feature set fares on new architectures and new benchmarks.

We perform 100 GA iterations for a population size of 1000, and a mutation probability of 0.01. The Genetic algorithm converges to the optimal feature set presented in Table 3.1 by generation 47. The four selected dynamic features are computed using eight performance events that can be precisely measured in a single run without multiplexing.

Clustering The feature vectors are the codelets performance signatures. Features are normalized to have unit variance and to be centered. Normalization ensures that features have equal weight when computing a distance between two feature vectors. To cluster similar codelets, we use hierarchical clustering with Ward's criterion [198].

Prediction Model Codelets from the same cluster share the same features and should react in the same way to architecture or compiler changes. For example, memory-bound codelets will benefit from faster caches, whereas highly vectorized codelets will benefit from wider vectors. Therefore, by measuring a single representative per cluster we can extrapolate the performance of all its siblings.

Once the performance classes are identified, CERE selects one representative invocation per class. CERE selects the invocation closest to the medoid

of the cluster, in other words, the invocation most closely matching the median performance of all the invocations inside the cluster. When replaying the benchmark, CERE extrapolates the full benchmark performance by weighting each representative replay time according to the contribution of its performance class in the original execution. The full prediction model is detailed in [149].

3.1.4 Clustering evaluation

Table 2.3 on page 30 presents the machines used for experiments. They belong to four different Intel CPU generations (Nehalem, Core 2 Duo, Atom, and Sandy Bridge) and possess quite distinct memory hierarchies. All the codelets are profiled on Nehalem, the reference architecture, at step A. The representatives are benchmarked on each target architectures (Atom, Sandy Bridge, and Core 2) at step C.

We evaluate our method using two criteria: the prediction error and the benchmarking reduction factor. For each codelet, the prediction error is the difference between predicted and real measurements, computed as a percentage. The benchmarking reduction factor is the ratio between the execution times of the representatives and the full benchmark suite. These two metrics are tied. In practice, the more clusters we add, the more we decrease the prediction error. However, by adding more clusters we also increase the number of representatives and therefore the benchmarking time.

We use two benchmarks suites: 28 Numerical Recipes (NR) [163] and the NAS SER benchmarks [10]. NR codes are simple but cover a large spectrum of algorithms. They were used as a training set to find the feature set presented in 3.1.3. The NAS benchmarks produce 67 codelets. They are run with CLASS B datasets. The NAS benchmarks are used to validate that the feature set trained on NR can be successfully applied to more complex benchmarks and other architectures. In our initial paper [149], codelets were extracted using Codelet Finder; later [45] experiments were replicated with CERE codelets. The differences in results between the two experiments were marginal.

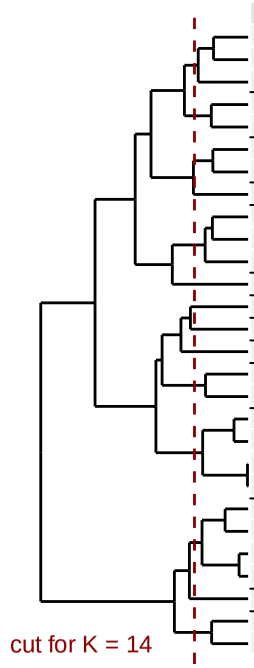
3.1.5 Numerical Recipes evaluation

This section evaluates the clustering on NR codelets performed with the feature set described in table 3.1. Table 3.2 shows a 14-group clustering built on the reference architecture.

Despite depending on the reference architecture, our feature set is closely related to architecture-independent features [92, 91]. For example, codelets can use scalar instructions (S), vector instructions (V), or a mix of both (V + S). We manually analyzed the vectorization of the codelets, *Vec.*, and compared it to the vectorization ratio, *Vec. %*, reported by MAQAO. They are highly correlated.

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

We note that many clusters are formed of codelets with similar *computation patterns*. For example, cluster 10 gathers codelets that divide elements in a vector, cluster 11 codelets that perform a reduced sum, cluster 14 codelets that



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

Table 3.2: NR clustering with 14 clusters and speedups on Atom. The dendrogram on the left shows the hierarchical clustering of the codelets. The height of a dendrogram node is proportional to the distance between the codelets it joins. The dashed line shows the dendrogram cut that produces 14 clusters. The table on the right gives for each codelet: the cluster number C, the Computation Pattern, the Stride, the Vectorization, and the Speedup on Atom s. The speedup of the selected representative is emphasized with angle brackets.

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

Table 3.3: Prediction errors on Numerical Recipes with 14 and 24 clusters.

compute element-wise multiplications on vectors or columns. The two "Dense Matrix x vector product" codelets have been separated because they use different floating point precision.

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

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

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

We evaluate the prediction error using the 14 clusters' representatives on Atom and Sandy Bridge. Table 3.3 summarizes the prediction errors. The overall accuracy of the prediction is good. Nevertheless, the NR were used during the feature selection training. The feature set was selected to minimize prediction accuracy. Next section validates our method on a different set of benchmarks and one new architecture not used during training.

3.1.6 Subsetting the NAS benchmark suite

In this section, we reuse the feature set trained on NR benchmarks and validate our benchmark reduction method on the NAS SER suite. We also evaluate a new target architecture Core 2.

Codelet performance prediction Figure 3.3 shows the predicted and real execution times on Sandy Bridge. The boxes gather the codelets by application. The applications may contain codelets coming from different clusters with different speedups. The execution time on Sandy Bridge is predicted with a

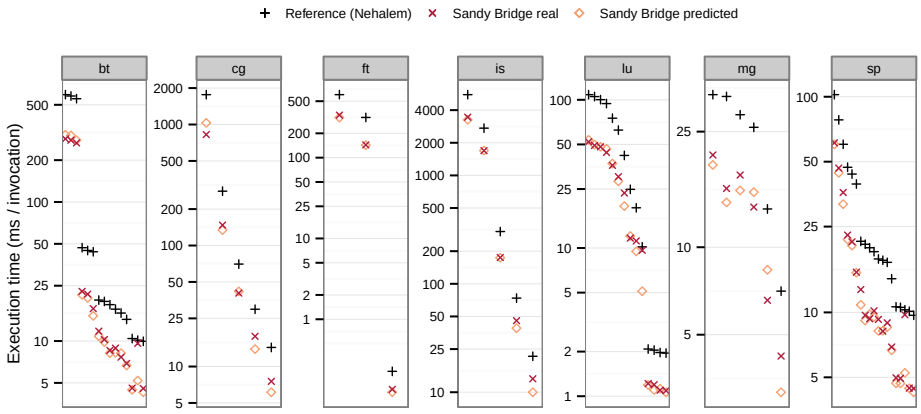


Figure 3.3: Predicted and Real execution times on Sandy Bridge compared to the Nehalem reference execution. Each box presents the codelets extracted from one of the NAS applications. Only three codelets in BT, LU, and SP are mispredicted.

median error of 5.8%. The error mainly comes from short-lived codelets (less than 10 ms per invocation) which are more affected by measurement errors such as instrumentation overhead. Codelets are faster on Sandy Bridge than on the reference. It is not surprising as Sandy Bridge frequency is almost twice the reference one. The median prediction error is 8% for Atom and 3.9% for Core 2.

Evaluating the feature-guided clustering To evaluate the quality of our feature-guided clustering, we compare it to 1000 random clusterings. In Figure 3.4, we make K, the number of clusters, vary from 1 to 24. For each value of K, we generate 1000 random partitionings into K clusters. We compute the prediction error for each partitioning. The proposed feature-guided clustering is most of the time close or better than the best random clustering. Our choice of features and clustering yields competitive results.

3.1.7 Codelets as proxies for faster performance studies

Codelets can be used as reduced benchmarks for performance studies when testing multiple architectures. Selecting the best computing system for a set of applications is a costly process which requires benchmarking the applications on the different systems. We propose to reduce the benchmarking cost by extracting a set of representative CERE codelets capturing the performance characteristics of the original applications.

By clustering similar codelets, eighteen representative codelets were selected and extracted using CERE. Then they were replayed in three different architectures: Atom, Core 2, and Sandy Bridge (see table 2.3).

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

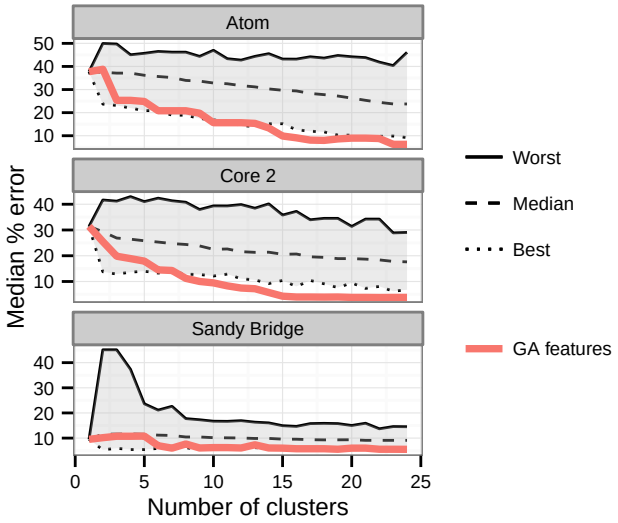


Figure 3.4: Genetic-Algorithm feature clustering compared to random clustering. For each number of clusters, from 2 to 24, 1000 random clusters are evaluated. Clustering with our GA feature set is consistently close or better than the best random clustering (out of 1000).

	CERE speedup	
Warmup mode	Working Set	Page Trace
Core 2	× 30.5	× 9.9
Atom	× 46.6	× 10.7
Sandy Bridge	× 18.3	× 7.3

Table 3.4: Benchmarking acceleration by replaying only the representatives. CERE replays are 7.3× to 46.6× faster than running the whole NAS.B suite.

Figure 3.5 compares the performance predicted using CERE replays to the real performance measured by running the full benchmark suite. The performance predictions are very close, but CERE replays are 7.3× to 46.6× cheaper than running the full benchmarks.

Table 3.4 details the benchmark reduction cost achieved by only replaying the selected representative codelets. We observe that the *Working Set* warmup is much faster than the *Page Trace* warmup that has the overhead of replaying the memory access history.

The benchmarking reduction comes from two factors. First, representatives are benchmarked during a small number of invocations. Second, by clustering the codelets, only the representatives have to be measured.

The data and code used are available as an IPython Notebook that allows to reproduce our experiments. The notebook can be accessed at `http://benchmark-subsetting.github.io/fgbs/`.

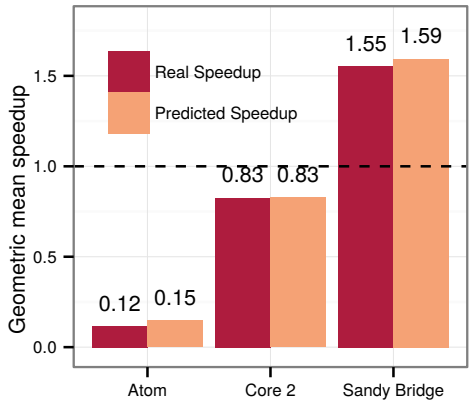


Figure 3.5: NAS geometric mean speedup on three architectures. Baseline is a NAS run on Nehalem compiled with `icc 12.1.0 -O3 -xsse4.2`. The predicted speedup is computed by using the replay performance of eighteen CERE representative codelets using *Working Set* warmup.

3.2 Adaptive sampling the performance design space

Adaptive sampling is a complementary approach to codelets for reducing the cost of HPC performance characterization and optimization. Instead of sampling the whole search space, we adaptively choose a small number of measures to build a performance model.

Samples should be chosen with care to faithfully represent the whole design space. A good sampling strategy should capture the performance accurately with the minimal number of samples. This section presents the Adaptive Sampling Kit (ASK), an open source framework that gathers state-of-the-art sampling strategies and surrogate models for HPC performance characterization.

The two fundamental elements of a sampling pipeline are the sampling strategy and the surrogate model.

1. The sampling strategy decides what combinations of the design space should be explored.

2. The surrogate model extrapolates from the sampled combinations a prediction on the full design space.

To reduce the number of samples, ASK implements an original adaptive sampling strategy, Hierarchical Variance Sampling (HVS) that concentrates exploration in the most irregular regions of the design space.

The user provides ASK with a description of the design space parameters. Then, ASK automatically selects the points that should be sampled, measure their response, and returns a model that predicts the performance of any given set of parameters. ASK also provides reporting and model validation modules to assess the quality of the sampling and ease the experimental setup exploration for performance characterization.

3.2.1 Background on sampling strategies

There are two kinds of sampling strategies: space filling designs and adaptive sampling. Space filling designs select a fixed number of samples with sensible statistical properties such as uniformly covering the space or avoiding clusters. For instance, Latin Hyper Cube designs [185] are built by dividing each dimension into equal sized intervals. Points are selected so the projection of the design on any dimension contains exactly one sample per interval. Maximin designs [98] maximize the minimum distance between any pair of samples; therefore spreading the samples over the entire experimental space. Finally, low discrepancy sequences [49] choose samples with low discrepancy: given an arbitrary region of the design space, the number of samples inside this region is almost proportional to the region's size. By construction, the sequences uniformly distribute points in space. These designs are often better than Random Sampling, which may clump samples together [180].

Space filling designs choose all points in one single draw before starting the experiment. Instead, adaptive sampling strategies iteratively adjust the sampling grid to the complexity of the design space. By observing already measured samples, they identify the most irregular regions of the design space. Further samples are drawn preferentially from the irregular regions, which are harder to explore.

The definition of irregular regions varies depending on the sampling strategy. Variance-reduction strategies focus the sampling in regions with high variance. The rationale is: irregular regions require more measurements to be accurately modeled. Query-by-Committee strategies build a committee of models trained with different parameters and compare the committee's predictions on all the candidate samples. Selected samples are the ones where the committee's models disagree the most. Adaptive Multiple Additive Regression Trees (AMART) [121] is a recent Query-by-Committee approach based on Generalized Boosted Models (GBM) [165], it selects non-clustered samples with maximal disagreement. Another recent approach by Gramacy et al. [79] combines the Tree Gaussian Process (TGP) [78] model with adaptive sampling strategies [35]. For an extensive review of adaptive sampling strategies please refer to Settles [177].

The Surrogate Modeling Toolbox (SUMO) [74] offers a Matlab toolbox building surrogate models for computer experiments. SUMO's execution flow is similar to ASK's: both allow configuring the model and sampling strategy to fully automate an experiment plan. SUMO focuses mainly on building and controlling surrogate models, offering a large set of models. It contains algorithms for optimizing model parameters, validating the models, and helping users choose a model. A recent approach, LOLA-Voronoi, is included, which finds trade-offs between uniformly exploring the space and concentrating on nonlinear regions of the space [40]. SUMO is open source but restricted to academic use and depends on the proprietary Matlab toolbox.

ASK specifically targets adaptive sampling for performance characterization, unlike SUMO. It includes recent state-of-the-art approaches that were successfully applied to computer experiments [79] and performance characterization [121]. Simpson et al. [182] show one must consider different trade-offs when choosing a sampling strategy: affinity with the surrogate model or studied response, accuracy, or cost of predicting new samples. Therefore, ASK comes with a large set of approaches to cover different sampling scenarios in-

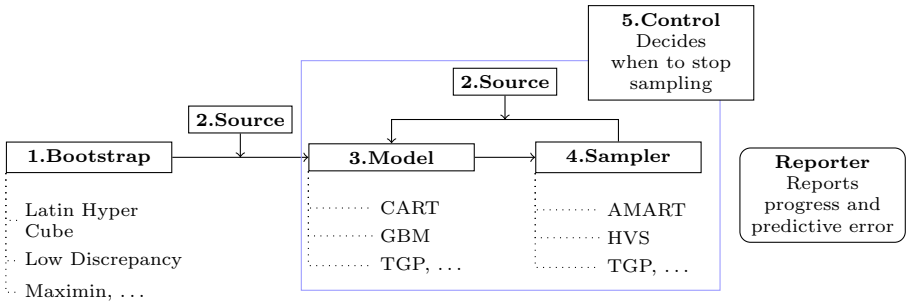


Figure 3.6: ASK pipeline

cluding Latin Hyper Cube designs, Maximin designs, Low discrepancy designs, AMART, and TGP. Additionally, ASK includes a new approach, Hierarchical Variance Sampling (HVS).

3.2.2 ASK Architecture

Choosing an adequate sampling strategy is not simple: for best results one must carefully consider the interaction between the sampling strategy and the surrogate model [182]. Many implementations of sampling strategies are available, but they all use different configurations and interfaces. Therefore, building and refining sampling strategies is difficult. ASK addresses this problem by providing a common interface to these different strategies and models. Designed around a modular architecture, ASK facilitates building complex sampling pipelines.

When running an experiment, ASK follows the pipeline presented in Figure 3.6:

1. A *bootstrap* module selects an initial batch of points. ASK provides standard bootstrap modules for the space filling designs described in Section 3.2.1: Latin Hyper Cube, Low Discrepancy, Maximin, and Random.

2. A *source* module, usually provided by the user, receives a list of requested points. The source module computes the actual measurements for the requested factors and returns the response.

3. A *model* module builds a surrogate model for the experiment on the sampled points. Currently ASK provides CART [18], GBM [165, 65], and TGP [79] models.

4. A *sampler* module iteratively selects a new set of points to measure. Some sampler modules are simple and do not depend on the surrogate model. For instance, the `random` sampler selects a random combination of factors and the `latin` sampler iteratively augments an initial Latin Hyper Cube design. Other sampler modules are more complex and base their decisions on the surrogate model.

5. A *control* module decides when the sampling process ends. ASK includes two basic strategies: stopping when it has sampled a predefined amount of points or stopping when the accuracy improvement stays under a given threshold for a number of iterations.

From the user perspective, setting up an ASK experiment is a three-step process. First, the range and type of each factor is described by writing an experiment configuration file in the JavaScript Object Notation (JSON) format. ASK accepts real, integer, or categorical factors. Then, users write a *source* wrapper around their measuring setup. The interface is straightforward: the wrapper receives a combination of factors to measure and returns their response. Finally, users choose which bootstrap, model, sampler, control, and reporter modules to execute. Module configuration is also done through the configuration file. ASK provides fallback default values if parameters are omitted from the configuration. An excerpt of a configuration with two factors and the hierarchical sampler module follows:

```
"factors": [{"name": "image-size",
             "type": "integer",
             "range": {"min": 0, "max": 600}},
            {"name": "stencil-size",
             "type": "categorical",
             "values": ["small", "medium", "large"]}],
"modules": {"sampler": {"executable": "sampler/HVS",
                        "params": {"nsamples":50}}}
```

Editing the configuration file quickly replaces any part of the ASK experiment pipeline with a different module. For example, by replacing `sampler/HVS` with `sampler/latin` the user replays the same experiment with the same parameters but using a Latin Hyper Cube sampler instead of Hierarchical Variance Sampling. All the modules have clearly defined interfaces and are organized to follow the *separation of concerns* principle [94]. This organization allows the user to quickly integrate custom made modules to the ASK pipeline.

3.2.3 Hierarchical Variance Sampling

Many adaptive learning strategies are susceptible to bias because the sampler makes incorrect decisions based on an incomplete view of the design space. For instance, the sampler may ignore a region although it contains big variations because previous samplings missed the variations.

To mitigate the problem, ASK includes the new Hierarchical Variance Sampling, HVS. HVS reduces the sampling bias using confidence intervals that correct the variance estimation. HVS partitions the exploration space into regions and measures the variance of each region. A statistical correction depending on the number of samples is applied to obtain an upper bound of the variance. Further samples are then selected proportionally to the upper bound and size of each region. By using a confidence upper bound on the variance, the sampler is less greedy in its exploration and is less likely to overlook interesting regions. In others words, the sampler is less likely to ignore a region until the number of sampled points is enough to confidently decide the region has low variance.

HVS is similar to Dasgupta et al. [42] proposing a hierarchical approach for classification tasks using confidence bounds to reduce the sampling bias. The Dasgupta et al. approach is only applicable to classification tasks with a binary or discrete response unlike HVS.

To divide the design space into regions, HVS uses the Classification and Regression Trees (CART) partition algorithm [18]. The splitting point is chosen to reduce the weighted residual sum of squares in each region. After the recursive partitioning, CART prunes the tree to optimize cross validation error. The

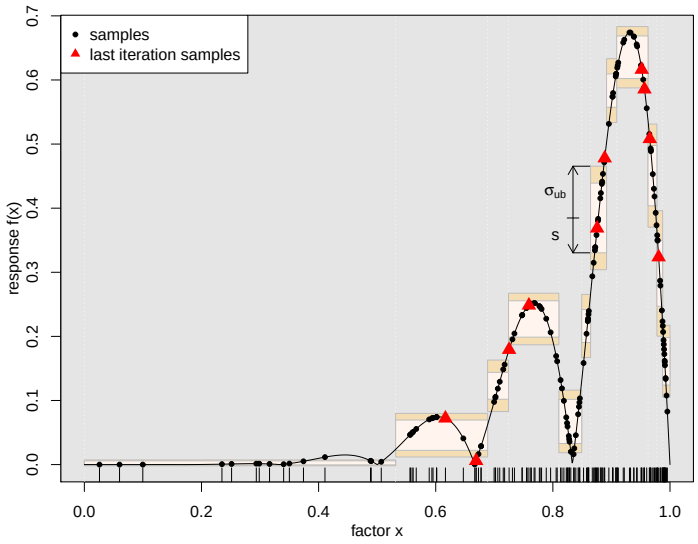


Figure 3.7: HVS on a synthetic 1D benchmark after fifteen drawings of ten samples each. The true response, $f(x) = x^5|sin(6.\pi.x)|$, is the solid line. CART partitions the factor dimension into intervals, represented by the boxes horizontal extension. For each interval, the estimated standard deviation, s, is in a light color and the upper bound of the standard deviation, σ_{ub}, is dark. HVS selects more samples in the irregular regions.

result of a CART partitioning is shown in Figure 3.7 where each box depicts a region.

After partitioning, HVS samples the most problematic regions and ignores the ones with low variance. The sampler only knows the empiric variance s^2 that depends on previous sampling decisions; to reduce bias HVS derives an upper bound of the true variance σ^2. Assuming a close to normal region's distribution, HVS computes an upper bound of the true variance σ^2 satisfying $\sigma^2 < \frac{(n-1)s^2}{\chi^2_{1-\alpha/2,n-1}} = \sigma^2_{ub}$ with a $1-\alpha$ confidence[1]. To account for the sampling uncertainty HVS uses the corrected upper bound accounting for the number of samples drawn.

For each region, Figure 3.7 plots the estimated standard deviation s, light colored, and upper-bound σ_{ub}, dark colored. As shown in Figure 3.7, samples are selected proportionally to the variance upper bound multiplied by the size of the region. New samples, marked as triangles, are chosen inside the largest boxes. HVS selects few samples in the $[0, 0.5]$ region, which has a flat profile.

If the goal of the sampling is to reduce the absolute error of the model, then the HVS strategy is adequate because it concentrates on high-variance regions. On the other hand, if the goal is to reduce the relative error of the model, it is better to concentrate on regions with high relative variance, $\frac{s^2}{\bar{x}^2}$. HVSrelative is an alternate version of HVS using relative variance with an appropriate confidence interval [133].

3.2.4 GBM model and HVS sampler interactions

Fitting a model is a trade-off between the model complexity and the accuracy. When tuning the model, it is important to take into account the design space and the sampling strategies. This section studies the interactions between the GBM model and the HVS sampler.

Traditional regression trees approaches, such as CART [18], partition the design space into regions and fit a constant or linear model to each region. The partitioning is represented by a tree where each leaf corresponds to one of the regions in the partitioning. GBM improves over CART by combining the predictive power of many individual trees. GBM models the response as a function $f(\mathbf{x})$ where $\mathbf{x}$ is the input vector of factors. The function f is defined as a linear combination of regression trees. Each regression tree models the interactions among a subset of factors.

GBM improves the accuracy of f by minimizing a loss function, such as $\frac{1}{n}\sum_{i=1}^{n}(y_i - f(\mathbf{x_i}))^2$, which computes for every point i, the mean squared error between the measured response y_i and the prediction $f(\mathbf{x_i})$. This formulation implies that every $\mathbf{x_i}$ has the same effect on the final model. Therefore, if the space is not uniformly explored, regions with few samples will be underrepresented in the loss function.

To avoid this effect, ASK takes advantage of the GBM weights feature [165], which replaces the above loss function by $\sum_{i=1}^{n} w_i.(y_i - f(\mathbf{x_i}))^2$ where w_i is the weight of point i. To remove the sampling bias introduced by the adaptive strategy we set all weights in region r to $w_r = \frac{s_r}{n_r}$, where s_r is the proportion of the total design space occupied by region r and n_r is the proportion of samples in region r.

[1] $1-\alpha = 0.9$ confidence bound is default in our experiments. χ is the Chi distribution.

3.2.5 Experimental validation

This section evaluates ASK sampling strategies on the performance study of 2D cross-shaped stencils of varying size on a parallel SMP. A wide range of scientific applications use stencils: for instance, Jacobi computation [43, 190] uses a 2×2 stencil and high-order finite-difference calculations [52] use a 6×6 stencil.

Later in section 4.3, we show how ASK has been used in tandem with Codelets to optimize a finite-difference time-domain (FDTD) kernel extracted from an industrial seismic imaging code.

The experiments were performed on two Nehalem architectures: an 8-core dual-socket Xeon E5620 at 2.40GHz with 24GB of RAM and a 32-core four-socket Xeon X7550 at 2.00GHz with 128GB of RAM. The OpenMP mapping policy was set to `Scatter`. All the benchmarks were compiled with ICC 12.0.0 version.

All studied sampling strategies use random seeds, which can slightly change the predictive error achieved by different ASK runs. Therefore, the median error, among nine different runs, is reported when comparing strategies. The strategies were called with the following set of parameters.

Samples All the strategies sampled in batches of fifty points per iteration.

Bootstrapping All the strategies were bootstrapped with samples from the same Latin Hyper Cube design, except Random, which was bootstrapped with a batch of random points.

Surrogate Model Tuning accurately the model parameters is important to get accurate performance predictions. The TGP strategy uses the tgpllm model with its default parameters and the adaptive sampling setup described in section 3.6 of [78]. The other strategies used GBM [165]. The detailed parameterization of the models for each experiment is described in our paper [146].

HVS, HVSrelative used a confidence bound of $1 - \alpha = 0.9$.

In the studied stencil code, Figure 3.8a, five factors can be tuned – X and $Y \in \{1, 2, 4, 8, 16\}$ the horizontal and vertical sizes of the stencil, $N \in [64, 2048]$ the number of rows of the matrix, $M \in [64, 2048]$ the number of columns of the matrix, and $T \in [1, 32]$ the number of threads.

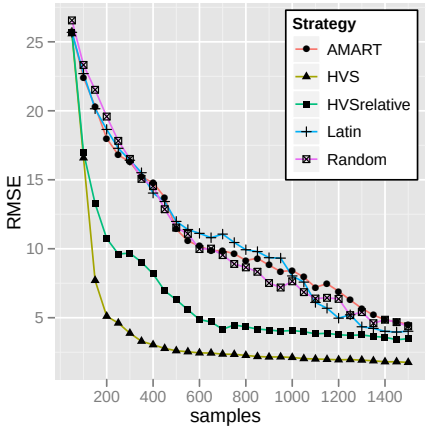
Given the large parameter space, exhaustive exploration is infeasible, but is possible using ASK's adaptive sampling strategies. The error was evaluated on an independent test set of 25 600 points.

The sampling strategies' accuracy is measured in terms of RMSE, in Figure 3.8b. Here only the 32-core results are examined because the sampling strategies' accuracy was similar for both the 8 and 32-core architectures. For RMSE, HVS outperforms all other strategies both in quality of the final model, 1.76 RMSE, and speed of convergence.

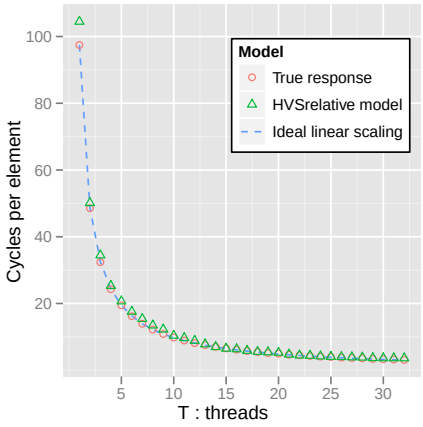
Figure 3.8c shows the performance prediction for HVS and HVSrelative on the $X \times 16$ stencils. Each square represents a unique (X, Y, T) configuration. Inside each square the performance is plotted depending on the matrix size $N \times M$. For example the outlined left-bottom square plots the performance predicted by HVS for a 1×16 stencil with one thread.

```
#pragma omp parallel for
for(i=Y; i<N-Y; i++)
  for(j=X; j<M-X; j++) {
    for(k=j-X; k<=j+X; k++)
      out[i][j] += in[i][k];
    for(l=i-Y; l<=i+Y; l++)
      out[i][j] += in[l][j];
  }
```

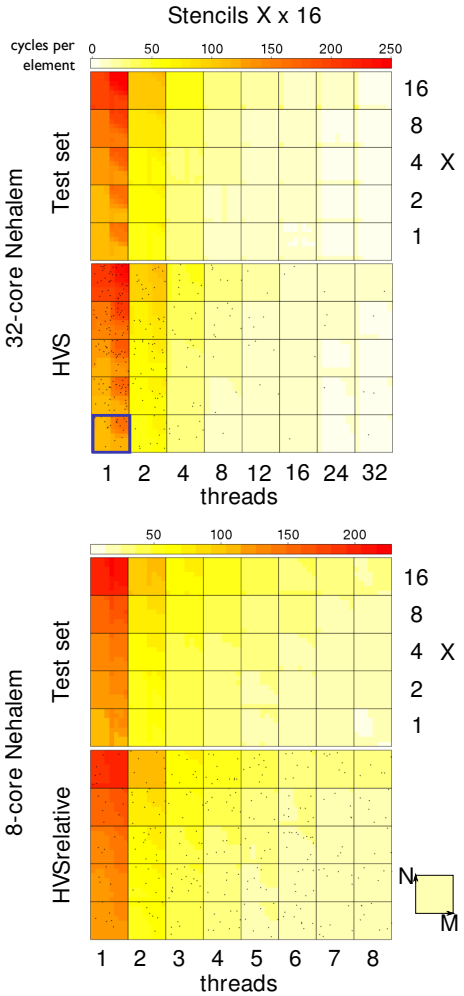
(a) Stencil code evaluated



(b) Error curves for the exploration on 32 cores. The median among nine runs of each strategy was taken to remove random seed effects.



(d) Scalability for the 8×8 stencil on a $1\,000 \times 1\,000$ matrix.



(c) Stencil $X \times 16$ predicted performance vs. test set performance in cycles per element.

Figure 3.8: Stencil experiments

The kernel is slowest for high Y stencils whose column order accesses stress the cache. In comparison, X stencil's size impact on performance is negligible. Performance degrades for large matrices, as shown from the darker top-right corners of each square, probably because the matrices exceed the L2 cache capacity. Therefore, choosing an adequate blocking factor should improve the performance. On ASK HVS' 32-core model the mean speed-up obtained by varying the matrix size is 1.65. Using a small matrix with a high number of threads is detrimental because the cost of synchronization predominates.

On both 8 and 32-core targets the stencil code scales linearly up to, respectively, 8 and 32 threads. As an example, the scalability of the application was studied on the 8×8 stencil on a $1\,000 \times 1\,000$ matrix. Figure 3.8d shows the performance per number of threads predicted by the HVSrelative strategy. The prediction follows the measured true response. The matrix sizes explored fit into the socket's 18Mb L3 cache, additionnaly the 2D stencil benefits from data reuse, which explains the strong scaling.

Measuring the whole design space would take centuries whereas ASK adaptive sampling took less than an hour of experiment time with a 1.76 RMSE.

3.3 Conclusion

This chapter presents two complementary methodologies significantly reducing benchmarking time:

- Detecting similar codelets within an application or across different applications, to reduce a full benchmark suite to a small set of representatives microbenchmarks, much faster to measure.
- Adaptively sampling the performance design space with a small number of points and using a surrogate model for performance characterization or optimization.

Chapter 4 applies these techniques on different HPC use-cases.

4 Optimizing HPC applications

Contents

This chapter includes contributions from the following publications:

- Mihail Popov, Chadi Akel, Yohan Chatelain, William Jalby, and Pablo de Oliveira Castro. "Piecewise holistic autotuning of parallel programs with CERE." in: *Concurrency and Computation: Practice and Experience* (2017), e4190. ISSN: 1532-0634. DOI: 10.1002/cpe.4190. URL: http://dx.doi.org/10.1002/cpe.4190
- Pablo de Oliveira Castro, Eric Petit, Asma Farjallah, and William Jalby. "Adaptive Sampling for Performance Characterization of Application Kernels." In: *Concurrency and Computation: Practice and Experience* (2013). ISSN: 1532-0634. DOI: 10.1002/cpe.3097

We have presented various techniques for reducing the HPC optimization search space. We demonstrate them in three applications: auto-tuning thread affinity and compiler passes, finding optimal thread mapping in an heterogeneous ARM big.LITTLE architecture, and optimizing a seismic imaging proto-application.

4.1 Auto-tuning thread affinity and compiler passes with codelets

Achieving full efficiency on a given architecture requires fine tuning parameters such as the degree of parallelism, thread placement, or compiler optimization. Runtime and compiler standard parameter levels (such as `-O3` compiler flag or scatter thread placement) achieve good-enough performance across most of the

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

Table 4.1: Execution time in megacycles of SP parallel regions across different thread affinities with `-O3` optimization. For n threads, we consider three affinities: scatter s_n, compact c_n, and hyperthread h_n. Executing SP with the c8 affinity provides an overall speedup of 1.71× over the standard (s16).

codes and architectures. But they can miss specific optimization opportunities since they are calibrated to work well on a large panel of applications and architectures.

There are different approaches to tuning parameters. Iterative compilation [109] is a well-known automated search method for solving the compiler optimization pass ordering problem. The idea is to apply successive compiler transformations to a program and to evaluate them by executing the resulting code. Similar execution-driven studies [131] explore the efficiency of thread placement strategies. A common method to accelerate the compiler tuning process is to guide the search exploration through machine learning techniques such as Genetic Algorithms (GA) [113, 89, 38, 66].

A common point of these execution-driven studies is that they require a full program evaluation and execution to quantify the impact of a single parameter value. Also, as regions of code do not benefit from the same parameters, an evaluation of the full program is not able to determine the optimal parameters for each region.

We propose to auto-tune parameters at the *codelet* level. Instead of evaluating parameters on the whole application, we separately evaluate them on each codelet. This makes the exploration faster and tailored to each code region.

We consider the Scalar Penta-diagonal solver (SP) from the C version of the NPB 3.0 OpenMP benchmarks [161]. CERE starts by profiling SP and automatically selects representative OpenMP regions to tune. `xsolve`, `ysolve`, `zsolve`, and `rhs` are chosen and cover 93% of SP execution time. These regions are extracted as codelets and tuned on three factors: thread number, thread placement, and LLVM compiler passes. Once satisfying parameters are found, CERE produces an hybrid application where each region uses the best found parameters.

We separate the optimization into two parts: first, we will tune thread number and affinity mapping. Second, we will optimize compiler optimizations. Combining all the parameters explored produces an exploration space of 1800 points, which gives an insight into how costly it is to tune multiple parameters simultaneously.

Figure 4.1 shows the performance of two SP parallel regions across this ex-

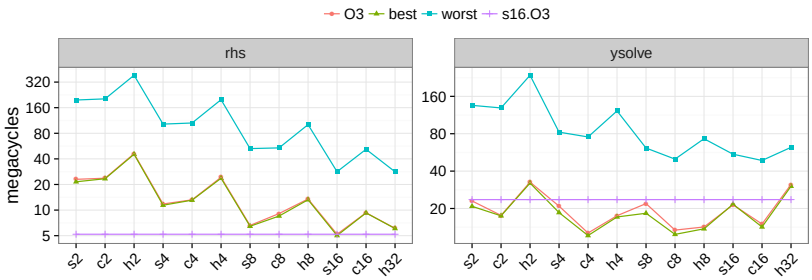


Figure 4.1: Tuning exploration for two SP regions. For each affinity, we plot the best, worst, and `-O3` optimization sequences. Custom optimization beats `-O3` for `s2`,`s4`, and `s8` on `ysolve`.

ploration space. We notice a strong interaction between the compiler and the thread parameters as they both significantly impact the performances. Moreover, the best parameters are different for the two regions: scatter placement is best for `rhs` while compact benefits `ysolve`.

4.1.1 Thread number and affinity optimization

We explore the interactions between 12 thread configurations combining different number of threads and affinity mappings including scatter, compact, and hyperthread. Scatter distributes the threads as evenly as possible across the entire system. The opposite strategy, compact, assigns the threads to the cores as closely as possible. Hyperthread acts like compact but binds multiple threads to the same physical core to take advantage of virtual cores.

Custom parameters outperform the standard `16 threads scatter s16 -O3` on SP. Table 4.1 shows the performance of different thread affinities compiled with `-O3`. The best custom thread affinity `0;1;2;3;4;5;6;7` (single NUMA socket) achieves a speedup of 1.71× over the standard `16 threads scatter` (two NUMA sockets).

4.1.2 Compiler pass optimization

We complete this study by exploring LLVM optimization sequences generated by random sub-samplings of the standard `-O3` optimization set. CERE makes it possible, through codelet replay, to independently explore each region. Moreover, thanks to CERE clustering of identical invocations, it is possible to quickly evaluate the impact of each configuration by using only a few datasets.

Figure 4.2 shows how CERE clustering can accelerate the compiler pass optimization. CERE estimation of the impact of each compiler pass configuration closely matches the performance measured in the original application. `ysolve` has 400 invocations with a similar performance behavior that CERE clusters as a single performance class. Since CERE only executes one representative invocation, tuning the region is 149× cheaper with a codelet than running the full SP benchmark.

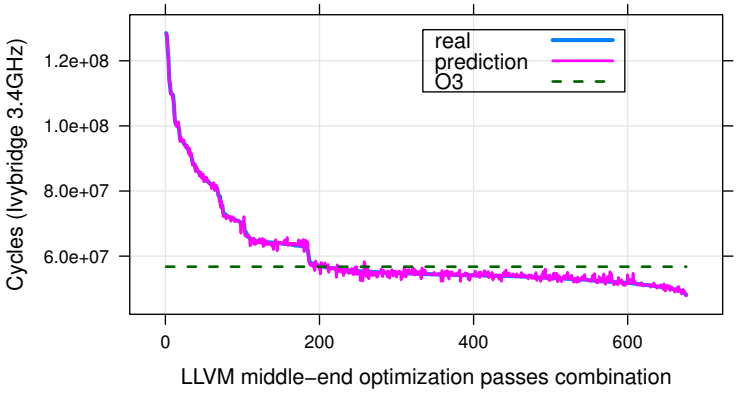


Figure 4.2: NAS SP `ysolve` region execution time across 1000 schedules of random compiler combinations passes based on `O3`. Compilation sequences are sorted according to their original execution time. We remove compilation sequences that produce the same binary. `real` and `predicted` respectively represents the original execution time and the codelet prediction based on representative invocations replay. The codelet faithfully predicts the region execution time across the different compiler optimizations.

We systematically apply this exploration of compiler optimization sequences on the best single NUMA configuration found above. `xsolve` and `ysolve` work best at the default `-O2 level`, but a custom best sequence is found for `zsolve` and `rhs`. Figure 4.3 shows the performance of each region compiled with the default optimization and the best custom sequences. No single sequence is the best for all regions. CERE hybrid compilation produces a binary where each region is compiled using its best sequence, achieving a speedup that cannot be reproduced using traditional monolithic compilation.

CERE evaluates thread affinities and compiler optimizations on SP, respectively 5.84× and 4.52× times faster than a full application evaluation while keeping a low average error of 2.33%. CERE autotuning achieves a 0.82× performance speedup over the standard parameters levels.

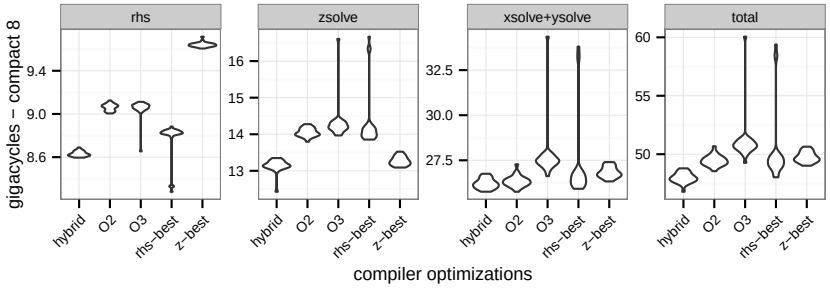


Figure 4.3: Violin plot execution time of SP regions using best NUMA affinity. Measures were performed 31 times to ensure reproducibility. When measuring total execution time, hybrid compilation outperforms all other optimization levels, since each region uses the best available optimization sequence.

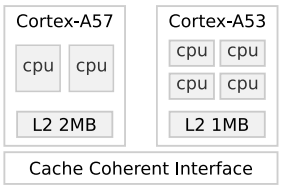


Figure 4.4: Juno big.LITTLE ARM architecture

4.2 Exploring runtime parameters in heterogeneous architectures

Tuning through CERE codelets is flexible and can be extended to new domains. In this section, we consider the problem of mapping and tuning a parallel application over a heterogeneous architecture such as the ARM big.LITTLE Juno development board. This study was done within the Mont-Blanc 3 European research project, which explored architecture designs for an ARM high-performance computer. The architecture, as shown in Fig. 4.4, combines two *clusters*: one *big* dual-core A57 and one *little* A53 quad-core. Each cluster has its own L2 cache, but they can share data through a cache-coherency interface.

Mapping a parallel application to this architecture is challenging because of the compute imbalance between the A53 and A57 and the difficulty in estimating the communication cost between clusters. Through CERE codelets, one can quickly test different mapping and scheduling strategies to find the best configuration. We demonstrate this by exploring PARSEC Blackscholes configurations on the Juno board.

PARSEC Blackscholes computes option pricing by solving a Partial Differential Equation. PARSEC OpenMP implementation is embarrassingly parallel except for the data initialization phase. We use CERE to capture the main parallel region found at Blackscholes.m4.cpp line 368.

We systematically explored the available thread affinities. The horizontal axis of Figure 4.5 shows the fourteen considered mappings: the first four are homogeneous executions on the A53; the next two are homogeneous executions on the A57; the final eight are heterogeneous mappings. We used the OpenMP default static scheduling strategy, which divides loops iterations into equally sized chunks across the different threads. The time of each execution was measured using the `cntvct_el0` cycles register [9]. To ensure that CERE estimates were correct, we validated each run execution time against the execution time of the original benchmark.

Running Blackscholes on the fourteen configurations took 1.42 seconds when using the CERE codelet and 60.53 seconds when running the original benchmark. Despite this significant speedup, the CERE estimates are very accurate across all configurations.

When a homogeneous cluster is used, either A53 or A57, we see that Blackscholes linearly scales as expected from an embarrassingly parallel benchmark. On the other side, performances on heterogeneous configurations are limited by the work imbalance. Let us consider heterogeneous mappings with three threads. We observe similar performance when using two A53 cores or two A57 cores. Since the workload is equally divided across the different cores, performance

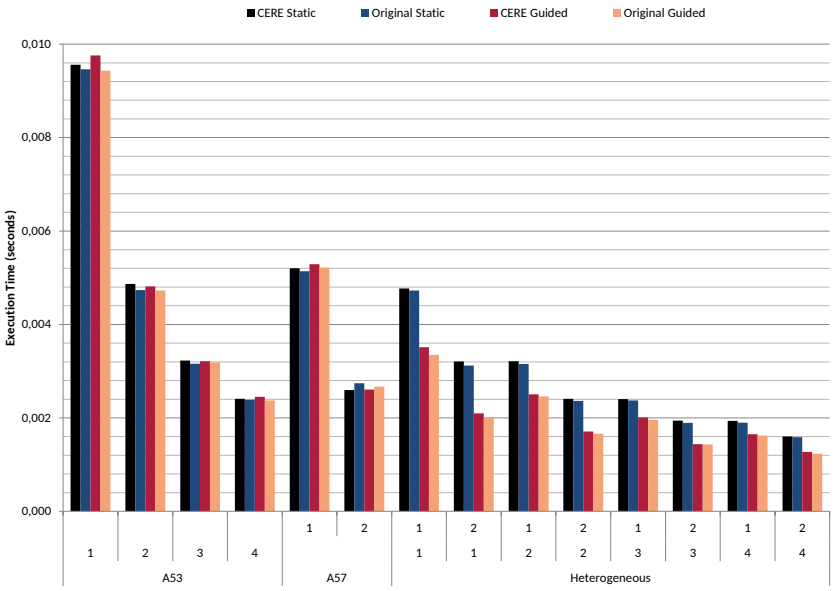


Figure 4.5: Execution time of PARSEC Blackscholes 64K in a Juno board. The horizontal axis shows the thread mapping across the four A53 and two A57 cores. The numbers in the first (respectively second) line are the number of threads mapped on A57 (respectively A53) cores. The first two categories consider homogeneous mappings, and the last category considers heterogeneous mappings. For each mapping, we both consider static and guided scheduling strategies. For each scheduling, we validate the time estimated by CERE to the time measured in the original benchmark.

is limited by the workload running on the A53 cluster. Similarly, A53 cores limit performance across the other heterogeneous mappings with two and four threads.

To take advantage of the A53 cores, we propose to switch the scheduling policy for loop iterations from static to the OpenMP guided. Instead of equally dividing the loop iterations across the cores, the guided policy considers a work queue of loop iterations grouped into chunks. When a thread finishes its chunk, it retrieves the next chunk from the top of the queue. While this OpenMP policy improves work-balancing by considering the target processors, it also introduces an overhead. To partially reduce this overhead while preserving work-balancing, guided starts with large chunks and reduces them through the execution. Figure 4.5 demonstrates the benefits of the guided scheduling over the default OpenMP policy. Using guided scheduling achieves a speedup of 1.29× compared to the default policy over the best mapping strategy. CERE remains faithful to the original executions across the different guided mappings while quickly finding the best scheduling strategy.

This study on the big.LITTLE ARM architecture demonstrates that CERE autotuning capabilities can easily be applied to problems involving heterogeneous architectures by simultaneously considering different number of threads, thread mapping strategies, and scheduling polices. Within the Mont-Blanc project, codelets were also used to accelerate GEM5 architectural simulations.

4.3 Optimizing a Seismic proto-application

In this section, we consider the optimization of a seismic proto-application which was developed by Asma Farjallah in her Ph.D. thesis [57].

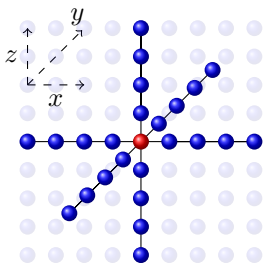
Reverse Time Migration (RTM) [11] is a commonly used application for seismic imaging applications. The first step in RTM consists of building synthetic seismograms based on a model of the crust. This process is called seismic modeling. The second step is a retro-propagation of the waves recorded during exploration campaigns. These waves are the reflection of vibrations sent into the earth. The seismic model is adjusted using the collected data in order to get the actual image of the subsurface. For geophysical imaging, one can solve the wave equation numerically using three distinct approaches: spectral method, strong formulation, and weak formulation [197].

Here we consider the strong formulation of the partial differential equation, which offers a good trade-off between image accuracy and computational cost. We focus on a finite-difference time-domain (FDTD) implementation in an isotropic medium. The equation governing the wave propagation in such a medium is,

$$\frac{1}{c^2}\frac{\partial^2 U}{\partial t^2} = \Delta U,$$

where U designates the wave field and c the velocity, constant in our case. In a finite-difference approach, each field value is updated using a combination of its neighboring values. Figure 4.6 shows the discretization of the wave equation. The number of neighbors in each direction defines the spatial order of the stencil. Our implementation uses the centered explicit stencil shown in figure 4.6. In the following, p denotes the half order of the stencil. For example, in figure 4.6,

$2p$ is equal to 8, which corresponds to the order of the Taylor expansion in each dimension of the space.



$$U^{t+1}_{i,j,k} = 2U^{t}_{i,j,k} - U^{t-1}_{i,j,k} + c^2\Delta t^2 \left\{ \frac{1}{\Delta x^2} \sum_{m=-p}^{p} b_m U^{t}_{i+m,j,k} + \frac{1}{\Delta y^2} \sum_{m=-p}^{p} b_m U^{t}_{i,j+m,k} + \frac{1}{\Delta z^2} \sum_{m=-p}^{p} b_m U^{t}_{i,j,k+m} \right\}$$

Figure 4.6: Discretization of the pressure field U using a stencil of order $2p$ in space and 2 in time. Δt and $(\Delta x, \Delta y, \Delta z)$ designate discretization steps in time and space respectively. b_m are constant coefficients weighting neighboring values that are the result of the Taylor expansion. A 3D centered stencil of order 8 ($p = 4$) is used.

4.3.1 Codelet extraction

We applied codelet extraction on the seismic proto-application. We focused on the dominant codelet, FDTD, which takes 91.1% of the original execution time. This FDTD codelet performs the stencil computation described in the previous section and is called 3000000 in the original proto-application.

Using the clustering codelet-similarity analysis described in section 3.1.3 on page 38 we found that FDTD codelet is very close to two NAS codelets: `BT/rhs.f:266-311` and `SP/rhs.f:275-320`. The three codelets have a very similar computation pattern: a three nested loop with a Jacobi stencil computation. Our feature set captures this similarity and gathers the three codelets in the same cluster.

In the FDTD cluster, the selected representative is the codelet from BT `BT/rhs.f:266-311`. Reusing our previously built codelet model (cf. section 3.1.6), we predicted FDTD on different target architectures without any additional benchmarking cost. We estimate the performance of FDTD on different architectures with the following errors: Atom, 7%, Core 2, 3%, and Sandy Bridge, 10%.

We also applied the compiler pass search presented in section 4.1.2 on the FDTD codelet which provided a speed-up of 1.11× over the `-O3` baseline.

4.3.2 Optimizing the FDTD codelet

Performance optimization of stencil computations is widely studied by the community [43]. Common optimizations target data locality through spatial and

temporal cache blocking, loop tiling, and padding. Most of these optimizations are machine-dependent and need to be manually tuned, for instance, by selecting the best blocking size and padding width. Performance modeling helps the tuning process by exploring the performance trade-offs in large parameter spaces.

Using ASK (cf. section 3.2 on page 46), we explore the following parameters of the FDTD kernel:

- the grid size (X, Y, Z) where each dimension is independently selected in the range $[768 : 1536]$ by steps of 128,
- the half stencil order p in $[1 : 8]$,
- the number of threads in $\{4, 8, 16, 32\}$,
- the number of blocks (NX, NY, NZ) respectively on X direction in $\{1, 2, 4, 8, 16\}$, Y and Z directions in $\{4, 16, 32, 64, 128\}$,
- the variant of the algorithm. We considerer two variants, `isotropic` and `isotropic-split`, that will be described later.

The possible factor combinations amount to more than 2.7 million. In order to evaluate the error, we measure the real response of 3225 randomly selected points, which represent more than 60 hours of experiments on the 32-core machine. Since the test set is small compared to the design space, confidence intervals of the prediction error are computed using 1000 ordinary bootstrap iterations [55].

Using ASK, a GBM performance model of the kernel is built using the HVS 800 points sampling. Figure 4.7 shows the RMSE and mean percentage error for the three sampling strategies. The experiment is stopped after 16 sampling steps to show ASK capability of building a performance model with a limited number of samples. The most accurate model is built using HVS and GBM with a final mean error of 7.71% and an RMSE error of 4.14.

The GBM model offers a useful feature that allows sorting the model factors by their relative influence. Figure 4.8 shows the relative influence of the different factors considered in our experiment. The relative influence is computed using the method proposed by Friedman in [65] and determines how much a given variable affects the response in the GBM model. Dominant factors of performance are the order of the stencil, the code variant, number of blocks on X and Y, and the number of threads. Figure 4.8 gives an insight into the parameters to consider in priority to enhance performance. The following paragraphs give more details on the way these parameters affect performance. The metric used is the number of cycles required per lattice update.

The Variant Influence We consider two variants of the FDTD implementation. The first variant, `isotropic` computes the Laplacian in a triple nested loop. In the second variant, `isotropic-split`, the inner loop is split into p smaller loops. Each split loop corresponds to one of the b_p factored blocks. Figure 4.9 shows that the number of cycles per update increases for both variants as the stencil order increases. Yet, for high stencil orders with $2.p > 10$, the `isotropic` variant is significantly more costly than the `isotropic-split` variant. The goal of loop splitting is to lower the register pressure and the number

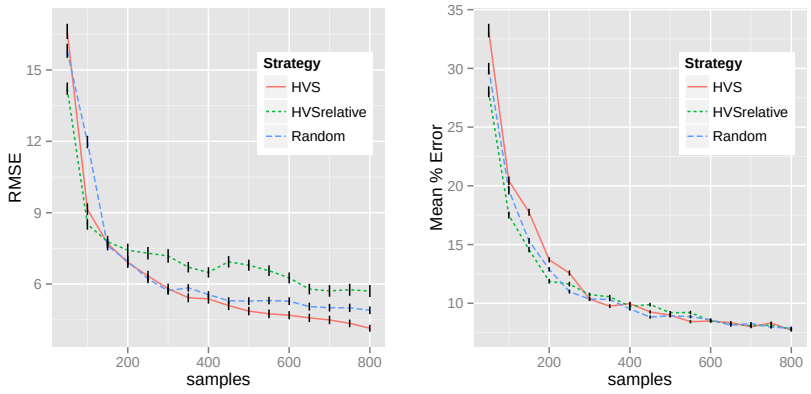


Figure 4.7: Root mean square error and Mean percentage error for the three tested strategies in the FDTD case study. The error is evaluated against a randomly selected test set of 3225 points. The vertical black lines show the bootstrap confidence intervals.

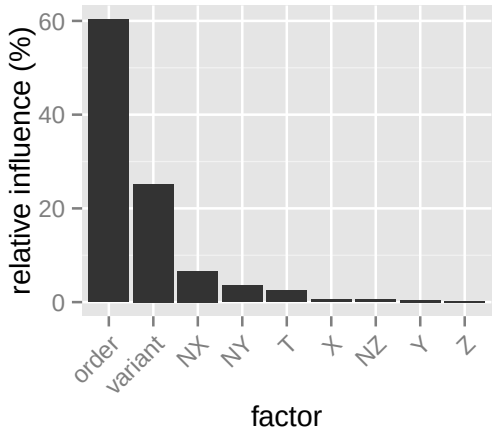


Figure 4.8: Relative influence of input factors in the FDTD kernel. The influence determines how much a factor affects the response. For the GBM model, it is computed as described in [65].

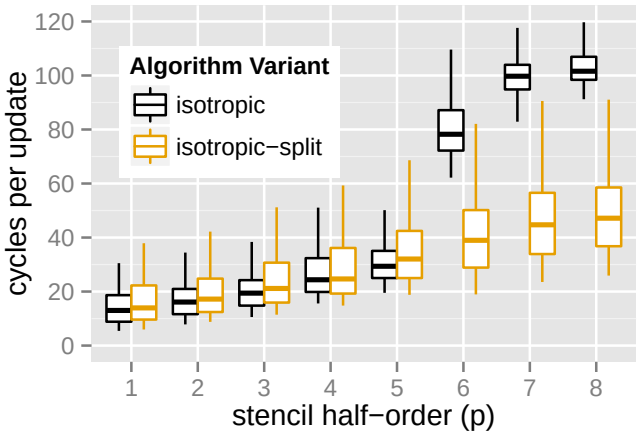


Figure 4.9: Performance of the `isotropic` and `isotropic-split` code variants. For p larger than five, the isotropic-split version is significantly faster.

of concurrent memory streams. It is particularly effective on large loop bodies, such as the ones needed by high-order stencils.

The Blocking Influence This section studies the impact of the spatial cache blocking on performance in the `isotropic-split` variant with $p = 4$. Results are similar for other configurations. Cache blocking aims to increase data locality by reusing data before eviction from the cache. The stencil computation can be blocked across the three dimensions.

Figure 4.10 illustrates the impact of blocking on the innermost dimension X. The grid is tiled with blocks: increasing the number of blocks reduces each block size. Performance deteriorates as the number of blocks on dimension X increases. Indeed, for small block sizes on X, the hardware prefetcher streams additional data, which are evicted from the cache before being used. Therefore, using small X block sizes result in an increase in memory traffic. The number of blocks in the X dimension should be kept low.

On the other hand, the outer Y and Z loops should be blocked to make the data working set of all the threads fit the cache. Figure 4.11 shows the correlation between the number of Y blocks and the number of threads. With many threads, configurations with a high number of Y blocks are the best. All the threads in the same socket share the same Last Level Cache (LLC). As the number of threads increases, the cache budget per thread decreases, requiring smaller block sizes. Blocking across Z also helps, but the pay-off is smaller since it exposes less data reuse. Similar conclusions can be found in other studies on performance optimization of stencil computations, such as [140, 166].

Scalability This section studies the strong scalability of a stencil of order 8 ($p = 4$) with the best variant and blocking parameters determined in the previous analysis. The selected parameters are `isotropic-split`, $p = 4$, $NX = 1$, $NY = 128$, and $NZ = 32$.

Figure 4.12 shows the scalability for two different grid sizes.

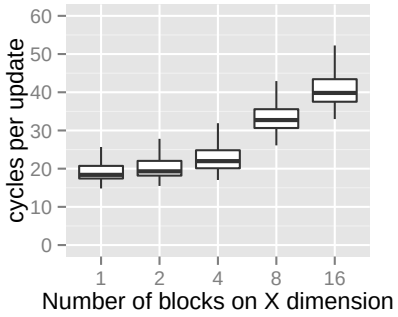


Figure 4.10: Performance of different X blocking configurations. The size of the blocks is inversely proportional to the number of blocks. Large block sizes across X exhibit the best performance.

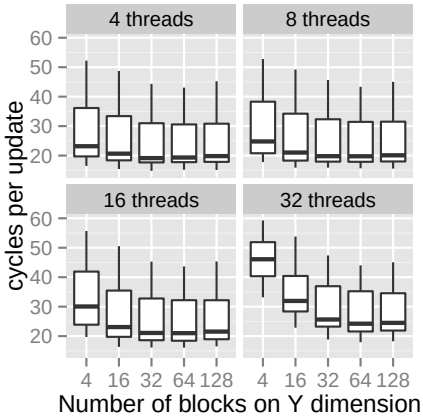


Figure 4.11: Performance of different Y blocking configurations. For a high number of threads, reducing the Y block size improves performance.

The kernel scales well up to 16 threads. ASK pinpoints a potential scalability problem: at 32 threads the speedup is around 26. Since the results are extrapolated from the model, we verify this result by direct measurement. Despite small local discrepancies, the model predicts the general trend.

4.4 Conclusion

This chapter concludes the first part of the manuscript by applying the previously presented techniques to accelerate HPC optimization studies to different use-cases.

First, we presented how CERE tunes thread number, affinity, and compiler optimizations at the same time on a scalar penta-diagonal solver from the NPB 3.0 OpenMP benchmarks. We also showed that CERE efficiently explores the thread mapping on an Aarch64 big.LITTLE heterogeneous architecture. Both of these use-cases are detailed in our paper [162] which applies the methodology systematically to all the NPB 3.0 benchmarks and presents additional experiments and details.

Second, we showed how CERE and adaptive sampling ease the study and optimization of a seismic proto-application. The model generated by ASK allowed us to tune the variant and the blocking factor, and detect a scalability problem in the FDTD kernel.

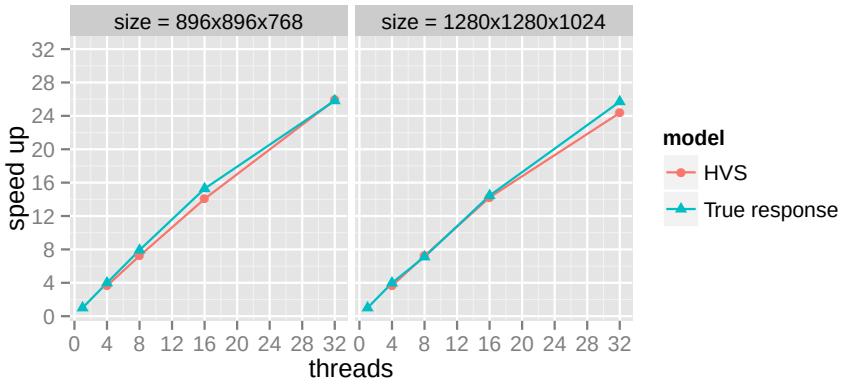


Figure 4.12: Scalability for the `isotropic-split` implementation with half order $p = 4$, $NX = 1$, $NY = 128$, and $NZ = 32$. The speedup is computed using the single thread performance with the same parameters.

Part II

Accuracy and performance trade-offs

5 Monte Carlo Arithmetic

Contents

This chapter includes contributions from the following publications:

- Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. "Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic." In: *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*. 2016, pp. 55–62. DOI: 10.1109/ARITH.2016.31. URL: http://dx.doi.org/10.1109/ARITH.2016.31
- Devan Sohier, Pablo de Oliveira Castro, François Févotte, Bruno Lathuilière, Eric Petit, and Olivier Jamond. "Confidence Intervals for Stochastic Arithmetic." In: *ACM Transactions Mathematical Software* 47.2 (Apr. 2021). ISSN: 0098-3500. DOI: 10.1145/3432184. URL: https://doi.org/10.1145/3432184

Modern computers use the IEEE-754 standard for implementing floating point (FP) operations. Each FP operand is represented with a limited precision. Single precision numbers have 23 bits in the fractional part and double precision numbers have 52 bits in the fractional part. This limited precision may cause numerical errors [86] such as absorption or catastrophic cancellation which can result in loss of significant bits in the result. Floating Point computations are used in many critical fields such as structure, combustion, astrophysics or finance simulations. For simple algorithms, such as summation, bounds of the numerical error have been derived mathematically [86].

One useful metric to evaluate the numerical accuracy of a computation is the number of significant digits which measures the relative numerical error by counting the number of accurate digits in the FP mantissa against a reference.

Unfortunately for many complex programs or intermediate computations, an exact reference value is not known beforehand. To overcome this problem, we turn to Monte Carlo arithmetic (MCA), a stochastic method introduced by Stott Parker [187], that simulates the effect of numerical errors and directly estimates the number of significant digits.

In MCA numerical errors are modeled by introducing random perturbations at each FP operation turning the outputs of a simulation code into realizations of a random variable. Performing a statistical analysis of a set of sampled outputs allows to stochastically approximate the impact of numerical errors on the code results.

5.1 Background on automatic numerical error analysis

Automatic methods for deriving bounds on round-off errors can be loosely categorized into two categories: exact methods and approximate methods.

Exact methods give a conservative and proven bound on the error of a computation. One well established exact method for deriving error bounds is Interval Arithmetic [136], in which each real value in the algorithm is replaced by an interval that contains all the possible values of the computation. The operations are redefined to handle intervals operands and guarantee that the resulting interval provide rigorous bounds on the computation. Multiple software frameworks [169, 170, 164] for interval arithmetic have been released. Interval arithmetic have been applied to derive error bounds and optimize numerical methods [135, 101], linear algebra [82], and physical simulation [48]. Because intervals are conservative, they tend to become overly large when the algorithm or control flow is complex. It is possible to refine the analysis by considering a union of interval subdivisions [85] or more sophisticate objects such as zonotopes [76] or affine arithmetic [70]. Instead of relying on intervals, FPTaylor [184] uses a Taylor expansion of a computation with error terms to find the maximum numerical error. Finally, floating point proof assistants [44, 15, 16] can derive semi-automatic certified proofs on floating point errors on small programs. However, the use of such methods is not always tractable, especially for complex programs with data dependent control paths. In that case the solution often rapidly diverges to dramatic overestimation of the error and therefore does not bring useful information [128].

On the other hand, approximate methods do not provide deterministic bounds on the numerical error and depend on the input data, but are able to efficiently analyze large and complex programs, such as found in industrial codebases. A first set of methods estimate the numerical error by comparing the IEEE-754 result to a computation performed in higher-precision. FpDebug [13] uses shadow memory for detecting accuracy problems with Valgrind by computing in higher precision. Similarly, Herbgrind [171] is another tool based on Valgrind that can automatically localize floating-points errors and find the causes of inaccuracies by tracking operations dependencies.

A second set of approximate methods simulate errors as random variables. The CESTAC [196, 33] method models round-off errors by randomly rounding FP values upwards or downwards equiprobably. CESTAC is implemented in the

CADNA [116] library which synchronously computes three CESTAC orbits for each FP value to estimate statistically the numerical error. Verrou [59, 194] is an open-source floating point diagnostics tool based on CESTAC and MCA. It is based on Valgrind [138] to transparently intercept floating point operations at runtime and replace them by their random rounding counterpart. The interposition at runtime allows to address large and complex-code applications with no intervention of the end-user.

CESTAC, while a pioneer method in automatic numerical estimation, has some shortcomings for the statistical estimation of errors as discussed in section 5.3.1. An alternative stochastic approach is Monte Carlo Arithmetic (MCA), introduced by Stott Parker[187], which is the main topic of this chapter.

5.2 IEEE-754 floating-point arithmetic

Normalized floating-point numbers $\mathcal{F} \subset \mathbb{R}$ form a subset of the real numbers, the elements of which can be written as $x = \pm m \times \beta^{e-p}$, where β is the basis, $\beta^{p-1} \leq m < \beta^p$ is an integer mantissa, $e \in \mathbb{Z}$ is the exponent, and p is the working precision. In this chapter, we restrict ourselves to normalized floating point numbers, excluding special cases of the IEEE-754 norm such as ± 0, NaNs, $\pm\infty$, and denormals.

When a program is run on an IEEE-754 compliant processor, the result of each floating-point operation $y \circ z$ is replaced by a rounded value. For example, the default rounding mode, round$(y \circ z)$, rounds to the nearest representable, tying to an even mantissa.

Definition 5.2.1. For $x \in \mathbb{R}$, Upward rounding $\lceil x \rceil$ and downward rounding $\lfloor x \rfloor$ are defined by:

$$\lceil x \rceil = \min\{y \in \mathcal{F} : y \geq x\}, \quad \lfloor x \rfloor = \max\{y \in \mathcal{F} : y \leq x\},$$

clearly, $\lfloor x \rfloor \leq x \leq \lceil x \rceil$, with equalities if and only if $x \in \mathcal{F}$.

Let us assume that x is a real that is not representable: $x \in \mathbb{R}/\mathcal{F}$. The distance between the two floating-point numbers enclosing x is $\epsilon(x) = \lceil x \rceil - \lfloor x \rfloor = \beta^{e-p}$.

Lemma 5.2.1. *$\beta^{p-e}\lfloor x \rfloor = \lfloor\!\lfloor \beta^{p-e} x \rfloor\!\rfloor$. where $\lfloor\!\lfloor x \rfloor\!\rfloor$ is the integer part of x.*

Proof. If $x \in \mathcal{F}$, then $\beta^{p-e}x \in \mathbb{Z}$ and $\lfloor x \rfloor = x$, the result follows. If $x \in \mathbb{R} - \mathcal{F}$, then $\beta^{p-e}\lfloor x \rfloor, \beta^{p-e}\lceil x \rceil \in \mathbb{Z}$, and $\lfloor x \rfloor < x < \lceil x \rceil$, then $\beta^{p-e}\lfloor x \rfloor < \beta^{p-e}x < \beta^{p-e}\lceil x \rceil$. We thus have

$$\beta^{p-e}\lfloor x \rfloor \leq \lfloor\!\lfloor \beta^{p-e} x \rfloor\!\rfloor < \beta^{p-e}\lceil x \rceil.$$

Since $\lceil x \rceil - \lfloor x \rfloor = \beta^{e-p}$, then $\beta^{p-e}\lceil x \rceil - \beta^{p-e}\lfloor x \rfloor = 1$ and

$$\beta^{p-e}\lfloor x \rfloor \leq \lfloor\!\lfloor \beta^{p-e} x \rfloor\!\rfloor < \beta^{p-e}\lfloor x \rfloor + 1.$$

□

Definition 5.2.2. The fraction of $\epsilon(x)$ rounded away for $x \notin \mathcal{F}$, as shown in figure 5.1, is

$$\theta(x) = \frac{x - \lfloor x \rfloor}{\lceil x \rceil - \lfloor x \rfloor}$$

For $x \in \mathcal{F}$, we define $\theta(x) = 0$.

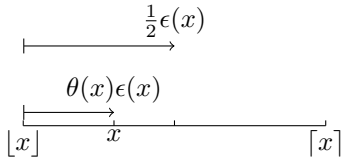


Figure 5.1: $\theta(x)$ is the fraction of $\epsilon(x)$ to be rounded away.

5.3 Stochastic arithmetic

The standard FP error model [86] bounds the error of a real computation $y \circ z = x \in \mathbb{R}$ for the elementary operations $\circ \in \{+, -, *, /\}$ assuming no underflow or overflow with

$$\widehat{x} = \text{round}(y \circ z) = x(1 + \delta)$$

where the relative error $\delta = \frac{\widehat{x}-x}{x}$ satisfies $|\delta| \leq u$ and $u = \frac{1}{2}\beta^{1-p}$ is the unit round off.

The key idea in Stochastic Arithmetic methods, such as MCA or CESTAC, is to replace the error term δ in each operation by a random variable that simulates the rounding errors. A computation is run multiple times in order to produce a set of output results (*i.e.* a set of realizations or samples of the random variable modeling the program output). The samples are then statistically analyzed in order to assess the quality of the result.

Let us denote by $x \in \mathbb{R}$ the quantity computed by a deterministic numerical program. Different values can be defined for this result:

- $\widehat{x} = \text{fl}(x)$ is the value computed with IEEE-754 arithmetic and the default rounding;
- $X_1, X_2, \ldots, X_n$ are the values returned by n runs of the program using stochastic arithmetic. These are seen as n realizations of the same random variable X.

The density of random variable X is unknown, but some of its characteristics can be estimated using n sample values $(X_1, \ldots, X_n)$. In particular:

- the expected value $\mu = E[X]$ can be estimated by the empirical average value of X_i, $\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} X_i$;
- the standard deviation $\sigma = \sqrt{E[(X-\mu)^2]}$ can be estimated by the empirical standard deviation, $\hat{\sigma} = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (X_i - \hat{\mu})^2}$.

5.3.1 CESTAC

CESTAC defines a stochastic arithmetic which randomly rounds FP values upwards or downwards equiprobably:

$$\text{cestac}(x) = \begin{cases} \lfloor x \rfloor & \text{with probability } 1/2 \\ \lceil x \rceil & \text{with probability } 1/2 \end{cases}$$

In CESTAC [195], the average $\hat{\mu}$ of three samples is taken as the computed result, and the analysis then estimates the accuracy of this quantity, seen as an approximation of the real value x.

Definition 5.3.1. The CESTAC number of exact significant bits is defined as the number of bits in common between x and $\hat{\mu}$:

$$s_{\text{CESTAC}} = -\log_2 \left| \frac{x - \hat{\mu}}{x} \right|.$$

In order to estimate the number of exact significant digits, the CESTAC analysis is based on two hypotheses:

1. the distribution X is normal, and
2. the distribution X is centered on the mathematical result $\mu = x$.

Since X is assumed normal, one can derive the following Student t-distribution interval with confidence $(1 - \alpha)$:

$$\mu \in \left[\hat{\mu} \pm \frac{\tau_n \, \hat{\sigma}}{\sqrt{n}} \right],$$

where n is the number of samples, and τ_n is the $1 - \frac{\alpha}{2}$ quantile of the Student distribution with $n - 1$ degrees of freedom.

The maximum error between μ and $\hat{\mu}$ is bounded by this interval for a normal distribution; it follows [122] that an estimated lower bound for the number of exact significant bits is given by

$$s_{\text{CESTAC}} = -\log_2 \left| \frac{\mu - \hat{\mu}}{\mu} \right| \approx -\log_2 \left| \frac{\mu - \hat{\mu}}{\hat{\mu}} \right| \geqslant \underbrace{-\log_2 \left(\frac{\tau_n \, \hat{\sigma}}{\sqrt{n} \, |\hat{\mu}|} \right)}_{\hat{s}_{\text{CESTAC}}}. \tag{5.1}$$

This definition suffers from a few shortcomings. First, the two hypotheses, while reasonable in many cases, do not always hold [32, 101]: Stott Parker shows that the normality assumption of X is not always true [187, p. 49] and that CESTAC is biased and not centered on the real result. The robustness of CESTAC with respect to violations of these hypotheses is discussed in [33].

Second, and more important, the CESTAC definition of the number of significant digits may not necessarily be the most useful for the practitioner. Oftentimes, the objective of the numerical verification process consists in evaluating the precision of the actual IEEE computer arithmetic. CESTAC does not evaluate the number of significant digits of the IEEE result but rather of the average of the CESTAC samples. But in practice, x does not match $\hat{\mu}$.

Last, with this definition, a problem clearly appears when considering the asymptotic behavior of the bound: $\hat{s}_{\text{CESTAC}} \xrightarrow[n \to +\infty]{} +\infty$. Increasing the number of samples arbitrarily increases the number of significant digits computed by CESTAC. On the one hand, this is expected because, according to the definition proposed, any computation is actually infinitely precise when $n \to \infty$ since the strong law of large numbers states that the empirical average is in this case almost surely the expected value. On the other hand, however, this asymptotic case also questions the pertinence of the CESTAC metric for the evaluation of the quality of the results produced by IEEE-754 computations.

5.3.2 Monte Carlo Arithmetic

MCA simulates the effect of different FP precisions by operating at a virtual precision t. To model errors on a FP value x at virtual precision t, it uses the function

$$\text{inexact}(x) = x + \beta^{e-t}\xi,$$

where $e = \lfloor \log_\beta |x| \rfloor + 1$ is the order of magnitude of x and ξ is a uniformly distributed random variable in the range $\left(-\frac{1}{2}, \frac{1}{2}\right)$.

During the MCA run of a given program, the result of each FP operation is replaced by a perturbed computation modeling the losses of accuracy [187, 47, 62]. Three possible expressions can be substituted to $y \circ z$, defining variants of MCA:

1. Random Rounding (RR) only introduces perturbation on the output:
$$\text{round}(\text{inexact}(y \circ z))$$
2. Precision Bounding (PB) only introduces perturbation on the input:
$$\text{round}(\text{inexact}(y) \circ \text{inexact}(z))$$
3. Full MCA (MCA) introduces perturbation on operand(s) and the result:
$$\text{round}(\text{inexact}(\text{inexact}(y) \circ \text{inexact}(z)))$$

Stott Parker shows that in most cases RR mode captures the effect of round-off errors, while PB mode captures the effect of catastrophic cancellations. Using stochastic arithmetic, the result of each FP operation is replaced with a random variable modeling the losses of accuracy resulting from the use of finite-precision FP computations. Since the result of each FP operation in the program is in turn used as input for the following FP operations, it is natural to assume that the outputs of the whole program in stochastic arithmetic are random variables.

Stott Parker explores different distribution choices for ξ. Yet, as he shows, the uniform distribution is preferred because it makes each MCA RR operation unbiased as shown in section 5.4.4.

The magnitude of the ξ is chosen so that the maximum stochastic error matches the maximum IEEE-754 rounding error at virtual precision t.

5.3.3 Estimating the numerical error with MCA

In his study of MCA, Stott Parker proposes a definition for the number of significant digits. He lays this definition on the habits of biology and physics regarding the precision of a measurement: if an MCA-instrumented program is seen as a measurement instrument[1], then the number of significant digits can be defined as the number of digits expected to be found in agreement between successive runs/measurements.

Definition 5.3.2. With the notations defined above, the MCA number of significant bits is defined as

$$s_{\text{MCA}} = -\log_2 \left| \frac{\sigma}{\mu} \right|.$$

[1] In most applications, a measurement is modeled by a random variable following a normal distribution.

This definition, which computes the magnitude of the coefficient of variation, is a form of signal-to-noise ratio: if most random samples share the same first digits, these digits can be considered significant. On the contrary, digits varying randomly among sampled results are considered noise. Another way of giving meaning to this definition is to consider $\mathrm{fl}(x)$ as one possible realization of the random variable X. As such, its distance to μ is characterized by σ. A problem with the MCA definition of significant bits is that it is empirical: the actual meaning of *significance* is not clearly laid, as well as the consequences one can draw from it.

The MCA number of significant bits can be estimated by

$$\hat{s}_{\mathrm{MCA}} = -\log_2 \left| \frac{\hat{\sigma}}{\hat{\mu}} \right|, \tag{5.2}$$

a quantity which can be computed regardless of any hypothesis on the distribution of X. However, since the number of samples n is finite, $\hat{s}_{\mathrm{MCA}}$ is only an approximation of the exact value s_{MCA}. And no confidence interval is provided in order to help choose an appropriate number of samples.

5.3.4 A simple example: Cramer's rule

To illustrate how MCA can estimate the numerical accuracy we use a simple synthetic example proposed by Kahan [100]: solving an ill-conditioned linear system,

$$\begin{pmatrix} 0.2161 & 0.1441 \\ 1.2969 & 0.8648 \end{pmatrix} x = \begin{pmatrix} 0.1440 \\ 0.8642 \end{pmatrix} \tag{5.3}$$

The exact and IEEE `binary64` solutions of equation (5.3) are:

$$x = \begin{pmatrix} 2 \\ -2 \end{pmatrix} \qquad \widehat{x} = \begin{pmatrix} 1.9999999958366637 \\ -1.9999999972244424 \end{pmatrix} \tag{5.4}$$

To keep the example simple, the floating-point solution x_{IEEE} has been obtained by solving the system with the naive C implementation of Cramer's formula in double precision, as shown in listing 5.1.

Code listing 5.1: Solving 2x2 system $a.x = b$ with Cramer's rule

```
1 void solve(const double a[4], const double b[2], double x[2]) {
2   double  det = a[0] * a[3] - a[2] * a[1];
3   double det0 = b[0] * a[3] - b[1] * a[1];
4   double det1 = a[0] * b[1] - a[2] * b[0];
5   x[0] = det0/det;
6   x[1] = det1/det;
7 }
```

The condition number of the above system is approximately 2.5×10^8, therefore we expect to lose at least $\log_2(2.5 \times 10^8) \approx 28$ bits of accuracy or, equivalently, 8 decimal digits. By comparing the IEEE and exact values, we see that indeed the last 8 decimal digits differ. The number of common bits between x and $\widehat{x} = \mathrm{fl}(x)$ is given by

$$s_{\mathrm{IEEE}} = -\log_2 \left| \frac{x - \widehat{x}}{x} \right| \approx \begin{pmatrix} 28.8 \\ 29.4 \end{pmatrix}.$$

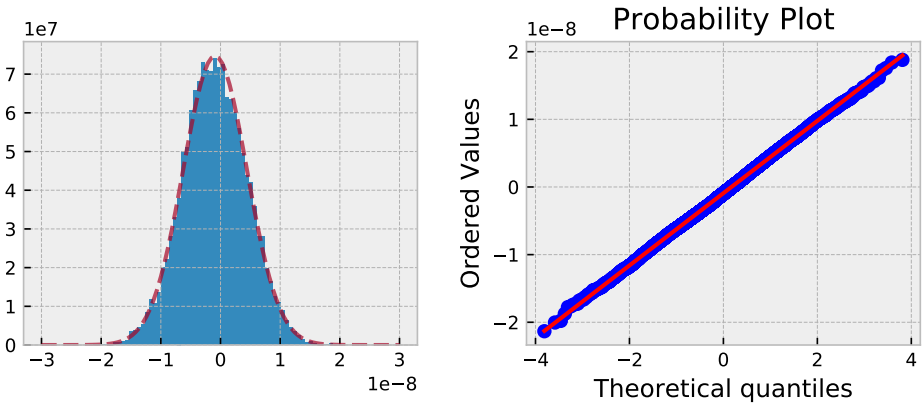


Figure 5.2: Normality of Cramer $X[0]$ sample

Now let us use MCA to estimate the number of significant digits. We compile the above program with verificarlo (cf. chapter 6) which transparently replaces every FP operation by its noisy MCA counterpart. Here a virtual precision of 52 is used to simulate round off errors. Then, we run the produced binary $n = 10\,000$ times and observe the resulting output distribution X.

Both $X[0]$ and $X[1]$ are normal with high Shapiro-Wilk test p-values 73 % and 74 % respectively.[2] Figure 5.2 shows the distribution and quantile-quantile (QQ) plots for $X[0]$, for which the empirical average and standard deviation are given by

$$\hat{\mu} \approx 1.99999999909,$$
$$\hat{\sigma} \approx 5.3427 \times 10^{-9}.$$

Using Stott Parker's formula (5.2) to compute $\hat{s}_{\text{MCA}}$ for $X[0]$, we get a figure close to the expected value 28.8:

$$\hat{s}_{\text{MCA}} = -\log_2 \left| \frac{\hat{\sigma}}{\hat{\mu}} \right| \approx 28.48. \tag{5.5}$$

But how confident are we that $\hat{s}_{\text{MCA}}$ is a good estimate of s_{MCA}? Could we have used a smaller number of samples and still get a reliable estimate of the results quality? Section 5.5 presents a novel probabilistic formulation to get a confidence interval for the number of significant bits with and without assumption of normality that can answer these questions. But before venturing further, we will examine and correct some rounding issues in MCA original definition.

5.4 Choice of the rounding operator in MCA

Stott Parker original definition of MCA RR [187] uses the default IEEE-754 nearest mode to round the inexact computation: for $x \notin \mathcal{F}$ and $p = t$ then MCA RR is defined as $\text{rr}(x) = \text{round}(\text{inexact}(x))$.

[2] Interestingly $X[0]$ fails the Anderson-Darling test, 27 % p-value, due to some anomalies on the tail.

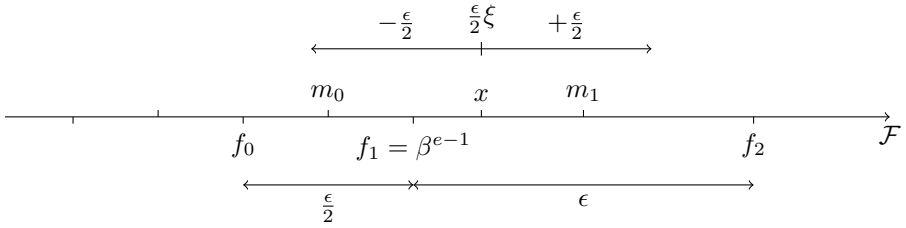


Figure 5.3: Rounding carry-out when x is close to a power of the base.

Using nearest round mode can produce surprising results when $|x|$ is close to β^{e-1}. MCA looses some important properties, in particular it becomes biased. This section studies the problems that arise when defining MCA using round-to-nearest and fixes them by redefining the rounding operator used in MCA.

5.4.1 Problems with nearest rounding in MCA

When $|x|$ is close to β^{e-1}, a carry out can happen during rounding. In that case the exponent of inexact(x) might be smaller by one compared to the exponent of x. Let us examine closely what would happen in that case when using nearest rounding. To simplify the analysis we will consider $x > 0$, but the same reasoning applies for $x < 0$ by reversing the rounding direction.

Let us consider the representable $f_1 = \beta^{e-1}$. The next representable is $f_2 = f_1 + \epsilon$ with $\epsilon = \beta^{e-p}$. The previous representable is $f_0 = f_1 - \frac{\epsilon}{2}$ because the distance between two representables is halved when crossing a power of the base. The midpoints between f_0 and f_1, and between f_1 and f_2, are noted m_0 and m_1 respectively. Figure 5.3 places these three representables and their midpoints on the $\mathcal{F}$ axis.

We recall that

$$\begin{aligned} \text{round}(x - \beta^{e-p}/2) \leq &\text{round}(\text{inexact}(x)) \leq \text{round}(x + \beta^{e-p}/2)) \\ \text{round}(x - \frac{\epsilon}{2}) \leq &\text{round}(\text{inexact}(x)) \leq \text{round}(x + \frac{\epsilon}{2}) \end{aligned}$$

This interval is represented as a double arrow in the top of figure 5.3. An incorrect rounding can only happen if the inexact value is lower than the midpoint m_0

$$\begin{aligned} x - \frac{\epsilon}{2} &< m_0 \\ x &< f_1 - \frac{\epsilon}{4} + \frac{\epsilon}{2} \\ x &< \beta^{e-1}(1 + \beta^{-p-1}) \end{aligned}$$

Therefore, when

$$\beta^{e-1} \leq x < \beta^{e-1}(1 + \beta^{-p-1}) \tag{5.6}$$

random rounding can return f_0 which is neither $\lfloor x \rfloor = f_1$ nor $\lceil x \rceil = f_2$.

We can illustrate the issue with a numerical example taking $p = 3$ and $\beta = 2$, let's choose $x = 2^0(1 + 2^{-5}) = 1.00001$, so $e = 1$, $f_1 = 1.00$, and $f_0 = 0.111$, all floating point numbers noted in binary.

Since $-2^{e-p-1} < 2^{e-p}\xi < 2^{e-p-1}$, -0.000111 is an admissible realization for which $\text{round}(x + 2^{e-p}\xi) = \text{round}(1.00001 - 0.000111) = \text{round}(0.111011) = 0.111 = f_0$. In this example, because x is close to a power of two, $\text{round}(\text{inexact}(x))$ has three possible realizations 0.111, 1.00 and 1.01. This does not match our expectations for a well-behaved stochastic rounding operator since x is not always rounded to one of its closest representables.

5.4.2 MCA bias when using round to nearest

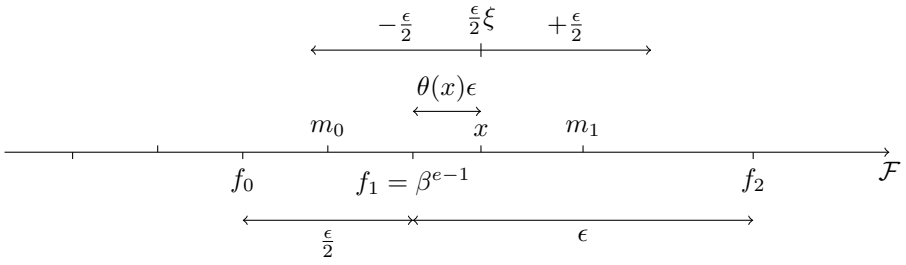
The previous rounding issue also introduces a bias in MCA RR, in other words MCA RR samples are not centered around the exact value x.

Theorem 5.4.1. *When $p = t$ and $\beta^{e-1} \leq x < \beta^{e-1}(1 + \beta^{-p-1})$, MCA RR is biased and*

$$\mathbf{E}(\text{round}(\text{inexact}(x))) = x + \frac{1}{2}\theta(x)\epsilon - \frac{\epsilon}{8}$$

Proof.

$$\begin{aligned}
&\mathbf{E}(\text{round}(\text{inexact}(x + \epsilon\xi))) = \\
&\qquad f_0\,\mathbf{P}(x + \epsilon\xi < m_0) + f_1\,\mathbf{P}(m_0 < x + \epsilon\xi < m_1) + f_2\,\mathbf{P}(m1 < x + \epsilon\xi) \\
&= \left(f_1 - \frac{\epsilon}{2}\right)\mathbf{P}(x+\epsilon\xi < m_0) + f_1\,\mathbf{P}(m_0 < x+\epsilon\xi < m_1) + (f_1 + \epsilon)\,\mathbf{P}(m1 < x+\epsilon\xi)
\end{aligned}$$



Because $x + \epsilon\xi$ is uniformly distributed in $(x - \epsilon/2, x + \epsilon/2)$,

$$\begin{aligned}
&= \left(f_1 - \frac{\epsilon}{2}\right)\left(\frac{\epsilon}{4} - \theta(x)\epsilon\right)\frac{1}{\epsilon} + f_1\left(\frac{\epsilon}{4} + \frac{\epsilon}{2}\right)\frac{1}{\epsilon} + (f_1 + \epsilon)\,\theta(x)\epsilon\frac{1}{\epsilon} \\
&= f_1 - \frac{\epsilon}{8} + \frac{3}{2}\theta(x)\epsilon \qquad \text{and } f_1 = x - \theta(x)\epsilon \\
&= x + \frac{1}{2}\theta(x)\epsilon - \frac{\epsilon}{8}
\end{aligned}$$

□

5.4.3 Redefining the rounding operator

As shown in section 5.4.1, problems arise when inexact changes the exponent of x, in that case the ϵ used during rounding does not match the ϵ used in the inexact function. To fix these issues, we propose to use a consistent ϵ in both operations.

Definition 5.4.1. For any $\beta^{e-1} \leq |x| < \beta^e$, we define the rounding operator for a given $\epsilon = \beta^{e-p}$ as

$$\text{round}_\epsilon(x) = \lfloor x + \epsilon/2 \rfloor = \lfloor x + \beta^{e-p}/2 \rfloor = \beta^{e-p} \lfloor\!\lfloor \beta^{p-e} x + 1/2 \rfloor\!\rfloor$$

The previous definition matches the behavior of round to nearest when $\epsilon = \epsilon(x)$.

Definition 5.4.2. We redefine MCA Random Round (RR) as

$$\text{rr}(x) = \text{round}_{\epsilon(x)}(\text{inexact}(x))$$

- For $x \in \mathcal{F}$, $\text{rr}(x) = x$.
- For $x \notin \mathcal{F}$,

$$\begin{aligned} \text{rr}(x) &= \text{round}_{\epsilon(x)}(x + \beta^{e-t}\xi) \\ &= \lfloor x + \beta^{e-t}\xi + \beta^{e-p}/2 \rfloor \end{aligned}$$

  where $\xi \sim U(-1/2, 1/2)$.

Under this new definition, MCA RR uses a consistent $\epsilon = \epsilon(x)$ for inexact and round. Let us show that this fixes the problems presented in last section. First, we show that for $p = t$, MCA RR can only return one of the two closest representables.

Lemma 5.4.1. $\text{rr}(x) = \text{round}_{\epsilon(x)}(\text{inexact}(x)) \in \{\lfloor x \rfloor, \lceil x \rceil\}$.

Proof. For x representable, the result follows from the definition: $\text{inexact}(x) = x$. Let us consider the case where x is inexact with $\lfloor x \rfloor < x < \lceil x \rceil$ and $e = e_x$.

$$\text{rr}(x) = \text{round}_{\epsilon(x)}(x + \beta^{e-p}\xi)$$

Because $-\frac{1}{2} < \xi < \frac{1}{2}$ and round is monotonous we have,

$$\begin{gathered} \text{round}_{\epsilon(x)}(x - \beta^{e-p}/2) \leq \text{rr}(x) \leq \text{round}_{\epsilon(x)}(x + \beta^{e-p}/2) \\ \beta^{e-p} \lfloor\!\lfloor \beta^{p-e}(x - \beta^{e-p}/2) + 1/2 \rfloor\!\rfloor \leq \text{rr}(x) \leq \beta^{e-p} \lfloor\!\lfloor \beta^{p-e}(x + \beta^{e-p}/2) + 1/2 \rfloor\!\rfloor \\ \beta^{e-p} \lfloor\!\lfloor \beta^{p-e} x \rfloor\!\rfloor \leq \text{rr}(x) \leq \beta^{e-p} \lfloor\!\lfloor \beta^{p-e} x + 1 \rfloor\!\rfloor \\ \lfloor x \rfloor \leq \text{rr}(x) \leq \lceil x \rceil \end{gathered}$$

□

In the following theorem 5.4.2 we revisit a result previously proved by Stott Parker: MCA RR at $p = t$ is equivalent to stochastic rounding [39, 61]. Interestingly, this equivalence is true only under the new definition which uses a consistent ϵ but is not true with the original round to nearest operator of MCA.

Theorem 5.4.2. *When $p = t$,*

$$\text{round}_{\epsilon(x)}(\text{inexact}(x)) = \begin{cases} \lfloor x \rfloor & \textit{with probability } 1 - \theta(x) \\ \lceil x \rceil & \textit{with probability } \theta(x) \end{cases}$$

where $\theta(x) = (x - \lfloor x \rfloor)/(\lceil x \rceil - \lfloor x \rfloor)$.

Proof.

$$\begin{aligned}\rho &= \text{round}_{\epsilon(x)}(\text{inexact}(x)) \\ &= \lfloor x + \beta^{e-p}\xi + \beta^{e-p}/2 \rfloor \\ &= \beta^{e-p} \lfloor\!\lfloor \beta^{p-e}x + \xi + 1/2 \rfloor\!\rfloor \\ &= \lfloor x \rfloor + \beta^{e-p} \lfloor\!\lfloor \theta(x) + \xi + 1/2 \rfloor\!\rfloor\end{aligned}$$

then,

$$\begin{aligned}\mathbf{P}(\rho = \lfloor x \rfloor) &= \mathbf{P}(\lfloor\!\lfloor \theta(x) + \xi + 1/2 \rfloor\!\rfloor = 0) \\ &= \mathbf{P}(-1 - \theta(x) < \xi + 1/2 < 1 - \theta(x)) \\ &= 1 - \theta(x) \qquad \text{because } \epsilon + 1/2 \sim U(0,1)\end{aligned}$$

and because of lemma 5.4.1,

$$\mathbf{P}(\rho = \lceil x \rceil) = 1 - \mathbf{P}(\rho = \lfloor x \rfloor) = \theta(x)$$

□

5.4.4 Bias of MCA RR

We examine the bias of MCA RR when defined using $\text{round}_{\epsilon(x)}$. We consider three cases, $t = p$, $t > p$, and $t < p$.

Theorem 5.4.3. *MCA RR is unbiased when $t = p$:*

$$\mathbf{E}(\text{rr}(x)) = x$$

Proof. When $t = p$,

$$\begin{aligned}\mathbf{E}(\text{round}_{\epsilon(x)}(\text{inexact}(x))) &= \lfloor x \rfloor (1 - \theta(x)) + \lceil x \rceil \theta(x) \\ &= \lfloor x \rfloor + (\lceil x \rceil - \lfloor x \rfloor)\theta(x) \\ &= \lfloor x \rfloor + x - \lfloor x \rfloor = x\end{aligned}$$

□

In verificarlo we use MCA RR to estimate the rounding errors of computer programs, and in that case we usually select $p = t$ which simulates the rounding error at the working precision. In that most usual case theorem 5.4.3 tells us that the resulting samples will be centered around the exact result.

Let us now examine the bias when $t \neq p$. When $t > p$, MCA RR is biased. For example, consider $t = 10$, $p = 3$, and $x = 1.01001$ in base $\beta = 2$.

$$\mathbf{E}(\text{round}_{\epsilon(x)}(\text{inexact}(x))) = \mathbf{E}(\text{round}_{\epsilon(x)}(x + 2^{-10}\xi)) = 1.01 \neq x$$

For $t < p$, Theorem 2 of Stott Parker's manuscript [187, p. 34] states that MCA RR is unbiased; as shown below, the proof does not hold when β is even and x is close to a power of the base.

Theorem 5.4.4. *When $t < p$, MCA RR is biased when β is even and unbiased when β is odd.*

Proof. Let $t < p$.

$$\mathbf{E}(\mathrm{rr}(x)) = \mathbf{E}(\mathrm{round}_{\epsilon(x)}(x + \beta^{e-t}\xi))$$

Since the random variable ξ is uniformly distributed in $(-1/2, 1/2)$, we can rewrite it as

$$\xi = 0.r_1r_2r_3r_4\ldots - 1/2$$

where $r_i \sim \mathcal{B}(1/2)$, a Bernoulli distribution with $p = 1/2$. The r_i form the random fractional part in base β of a FP number in $(0, 1)$. Then,

$$\beta^{e-t}\xi = \beta^{e-t}0.r_1r_2r_3r_4\ldots - \beta^{e-t}/2$$

Since $p - t > 0$, we can shift the mantissa by $p - t$ digits,

$$\beta^{e-t}\xi = \beta^{e-p}(r_1r_2\ldots r_{p-t} + 0.r_{p-t+1}r_{p-t+2}\ldots) - \beta^{e-t}/2$$

Now going back to the expectation,

$$\begin{aligned}
&\mathbf{E}(\mathrm{rr}(x)))\\
&\quad= \mathbf{E}(\mathrm{round}_{\epsilon(x)}(x + \beta^{e-p}(r_1\ldots r_{p-t} + 0.r_{p-t+1}\ldots) - \beta^{e-t}/2))\\
&\quad= \mathbf{E}(\beta^{e-p}\lfloor\!\lfloor\beta^{p-e}x + (r_1\ldots r_{p-t} + 0.r_{p-t+1}\ldots) - \beta^{p-e}\beta^{e-t}/2 + 1/2\rfloor\!\rfloor)\\
&\quad= \mathbf{E}(\beta^{e-p}\lfloor\!\lfloor\beta^{p-e}x + (r_1\ldots r_{p-t} + 0.r_{p-t+1}\ldots) - \beta^{p-t}/2 + 1/2\rfloor\!\rfloor)\\
&\quad= \beta^{e-p}\,\mathbf{E}(r_1\ldots r_{p-t}) + \beta^{e-p}\,\mathbf{E}(\lfloor\!\lfloor\beta^{p-e}x + 0.r_{p-t+1}\ldots - \beta^{p-t}/2 + 1/2\rfloor\!\rfloor))
\end{aligned}$$

Because $r_1r_2\ldots r_{p-t}$ is an uniformly distributed integer in $[0, \beta^{p-t} - 1]$ we have

$$\mathbf{E}(\mathrm{rr}(x))) = \beta^{e-p}(\beta^{p-t} - 1)/2 + \beta^{e-p}\,\mathbf{E}(\lfloor\!\lfloor\beta^{p-e}x + 0.r_{p-t+1}\ldots - \beta^{p-t}/2 + 1/2\rfloor\!\rfloor))$$

First, let us show that when β **is odd** MCA RR is unbiased.

$$\begin{aligned}
&\mathbf{E}(\mathrm{rr}(x))) = \beta^{e-p}(\beta^{p-t} - 1)/2 +\\
&\quad\beta^{e-p}\,\mathbf{E}(\lfloor\!\lfloor\beta^{p-e}x + 0.r_{p-t+1}\ldots - ((\beta^{p-t} - 1)/2 - 1/2) + 1/2\rfloor\!\rfloor))\\
&= \mathbf{E}(\beta^{e-p}\lfloor\!\lfloor\beta^{p-e}x + (0.r_{p-t+1}\ldots - 1/2) + 1/2\rfloor\!\rfloor)) && \text{because } (\beta^{p-t} - 1)/2 \in \mathbb{N}\\
&= \mathbf{E}(\mathrm{round}_{\epsilon(x)}(x + \beta^{e-p}\phi)) && \text{where } \phi \sim U(-1/2, 1/2)\\
&= x && \text{by applying theorem 5.4.3.}
\end{aligned}$$

Second, let us show that when β **is even** MCA RR is biased.

$$\begin{aligned}
&\mathbf{E}(\mathrm{rr}(x)))\\
&\quad= \beta^{e-p}(\beta^{p-t} - 1)/2 + \beta^{e-p}\,\mathbf{E}(\lfloor\!\lfloor\beta^{p-e}x + 0.r_{p-t+1}\ldots - \beta^{p-t}/2 + 1/2\rfloor\!\rfloor))\\
&\quad= \beta^{e-p}((\beta^{p-t} - 1)/2 - \beta^{p-t}/2) + \beta^{e-p}\,\mathbf{E}(\lfloor\!\lfloor\beta^{p-e}x + 0.r_{p-t+1}\ldots + 1/2\rfloor\!\rfloor)\\
&\quad= -\beta^{e-p}/2 + \beta^{e-p}\,\mathbf{E}(\lfloor\!\lfloor(\beta^{p-e}x + 1/2) + (0.r_{p-t+1}\ldots - 1/2) + 1/2\rfloor\!\rfloor)\\
&\quad= -\beta^{e-p}/2 + \mathbf{E}(\mathrm{round}_{\epsilon(x)}((x + \beta^{e-p}/2) + \beta^{e-p}\phi)) \qquad (5.7)\\
&\qquad\text{where } \phi \sim U(-1/2, 1/2).
\end{aligned}$$

Now let us consider two sub-cases depending on the value of x.

First, $x + \beta^{e-p}/2$ and x have the same exponent when

$$\beta^{e-1} \leq |x + \beta^{e-p}/2| < \beta^e$$
$$\beta^{e-1} + \beta^{e-p}/2 \leq |x| < \beta^e - \beta^{e-p}/2.$$

In that case, $\epsilon(x + \beta^{e-p}/2) = \epsilon(x)$ and theorem 5.4.3 applies to equation 5.7 giving

$$\mathbf{E}(\mathrm{rr}(x)) = -\beta^{e-p}/2 + (x + \beta^{e-p}/2) = x$$

Second, when $|x| \geq \beta^e - \beta^{e-p}/2$ or $|x| < \beta^{e-1} + \beta^{e-p}/2$, the result is biased. Let us consider, for example, $\beta^e - \beta^{e-p}/2 < x < \beta^e$, then

$$\mathrm{round}_{\epsilon(x)}((x + \beta^{e-p}/2) + \beta^{e-p}\phi) = \lfloor (x + \beta^{e-p}/2) + \beta^{e-p}\phi + \beta^{e-p}/2 \rfloor$$

We note $y = (x + \beta^{e-p}/2) + \beta^{e-p}\phi + \beta^{e-p}/2$. By considering the range of ϕ,

$$(x + \beta^{e-p}/2) - \beta^{e-p}/2 + \beta^{e-p}/2 < y < (x + \beta^{e-p}/2) + \beta^{e-p}/2 + \beta^{e-p}/2$$
$$x + \beta^{e-p}/2 < y < x + 3 \times \beta^{e-p}/2$$

Now by considering the range of x,

$$\beta^e - \beta^{e-p}/2 + \beta^{e-p}/2 < y < \beta^e + 3 \times \beta^{e-p}/2$$
$$\lfloor \beta^e \rfloor \leq \lfloor y \rfloor \leq \lfloor \beta^e + 3 \times \beta^{e-p}/2 \rfloor$$

But $3 \times \beta^{e-p}/2 < \epsilon(\beta^e) = \beta^{e-p+1}$, so $\lfloor \beta^e + 3 \times \beta^{e-p}/2 \rfloor = \beta^e$. Therefore,

$$\mathbf{E}(\mathrm{round}_{\epsilon(x)}((x + \beta^{e-p}/2) + \beta^{e-p}\phi)) = \beta^e$$

going back to equation 5.7 gives

$$\mathbf{E}(\mathrm{rr}(x)) = -\beta^{e-p}/2 + \beta^e < x$$

so there is a bias when β is even and x is close to a power of the base. □

5.4.5 Numerical evaluation of rounding methods

We will now demonstrate on a numerical example the issues with the original MCA rounding and how they are fixed under the new definition.

Table 5.1 compares MCA RR with $p = t = 24$ with the original rounding and with the corrected one. For each method, we compute 100 000 samples of $\mathrm{rr}(x)$ for $x = 1.0 + 2^{-u}$ with u increasing from 23 to 26. We compute the sampling bias as the difference between the mean of the samples and the value x.

For $u < 26$, both methods are well-behaved, and the sampling average is close to the exact result x.

For $u = 26$, $x = 1 + 2^{-26} < 2^{e-1}(1 + 2^{-p-1})$, from the previous analysis in section 5.4.1 we expect the original MCA definition to fail. This is confirmed numerically as the original definition produces 12522 samples where the result was rounded to f_0. On the contrary, the corrected definition always round to either $\lfloor x \rfloor$ or $\lceil x \rceil$. The sampling bias is two orders of magnitude lower with our corrected computation.

	round to nearest (Stott Parker)				$\text{round}_{\epsilon(x)}(\text{inexact}(x))$		
x	f_0 $1-2^{-24}$	$\lfloor x \rfloor$ 1	$\lceil x \rceil$ $1+2^{-23}$	bias	$\lfloor x \rfloor$ 1	$\lceil x \rceil$ $1+2^{-23}$	bias
$1+2^{-23}$	0	0	100000	0	100000	0	0
$1+2^{-24}$	0	49855	50145	1.7e-10	49974	50026	3.1e-11
$1+2^{-25}$	0	74972	25028	3.3e-11	75072	24928	-8.6e-11
$1+2^{-26}$	12522	75089	12389	7.3e-09	87580	12420	9.54e-11
$1+2^{-27}$	18688	75052	6260	1.1e-08	93723	6277	-3.22e-11

Table 5.1: Comparison of MCA with round to nearest and $round_{\epsilon(x)}$ with $p = t = 24$. 100 000 RR samples were collected, we report the distribution of results and the bias.

5.5 Probabilistic accuracy of a computation

In this section we will define the probabilistic accuracy of a computation. We consider the output of a program performing FP operations as a random variable X. The output is a random variable either because the program is inherently nondeterministic or because we are artificially introducing numerical errors through MCA, or another stochastic arithmetic model.

5.5.1 Choice of a reference value

The accuracy of a result must be defined against a reference value. When a real mathematical result is known, it is a natural choice. If the program is deterministic when executed in IEEE arithmetic, the IEEE result is one straightforward choice for the reference value. If the program is nondeterministic, one can also choose as reference, the empirical average of X. Finally, a third option consists in computing the accuracy against a second random variable Y, which allows computing the accuracy between runs of the same program or allows finding the accuracy between two different programs, such as when comparing two different versions or implementations of an algorithm. We will write the reference value x when it is a constant and Y when it is another random variable.

Four types of studies can be led, depending on whether we are interested in absolute or relative error, and whether we have a reference value. We have reduced the four types of problems to study the random variable Z whose distribution represents the error of a computation in a broad sense.

	reference x	reference Y
absolute precision	$Z = X - x$	$Z = X - Y$
relative precision	$Z = X/x - 1$	$Z = X/Y - 1$

5.5.2 Probabilistic definitions of significant and contributing digits

To define the significance of a digit we use Stott Parker's $\frac{1}{2}$ulsp algorithm [187, p. 19]. The significant bit is at the rightmost position at which the digits differ

by less than one half unit in the last place. That is to say, two values x and y have s significant digits iff

$$\begin{aligned} |x-y| &< \frac{1}{2} \times 2^{e_y - s} = 2^{-s+(e_y-1)} && \text{(scaled absolute error)} \\ |x/y - 1| &< \frac{1}{2} \times 2^{1-s} = 2^{-s} && \text{(relative error)} \end{aligned} \tag{5.8}$$

Without loss of generality, to unify the definition for the relative and scaled absolute cases, in the following sections we assume $e_y = 1$. When working with absolute errors, one should therefore shift the number of digits by $(e_y - 1)$, the normalizing term[3].

The first quantity of interest is the **probability that the result is significant up to a given bit** for a MCA computation. By generalizing equation 5.8 to random variables, we define the probability of the k-th digit being significant as $\mathbf{P}\left(|Z| < 2^{-k}\right)$.

Definition 5.5.1. For a given stochastic computation, the k-th bit of Z is said to be significant with probability p if

$$\mathbf{P}\left(|Z| < 2^{-k}\right) \geqslant p.$$

The number of significant digits in Z with probability p is defined as the largest number $s_{\text{sto}} \in \mathbb{R}$ such that

$$\mathbf{P}\left(|Z| < 2^{-s_{\text{sto}}}\right) \geqslant p.$$

Note that, by definition, if the k-th bit of Z is significant with probability p, then any bit of rank $k' \leqslant k$ is also significant with probability p. In the following, when not otherwise specified, the simple notation s will refer to the s_{sto} notion defined above.

The second quantity we will consider is the **probability that a given bit contributes to the precision of the result**: even if a bit on its left is already wrong, a bit can either improve the result precision, or deteriorate it. As noted in [187, p.45]: *"In other words, in inexact values it can be worthwhile to carry a nontrivial number r of random least significant bits"*. Because the expected result of Z is 0, a bit will improve the accuracy if it is 0 and deteriorate it if it is 1.

Definition 5.5.2. The k-th bit of Z is said to be contributing with probability p if and only if it is 0 with this probability, *i.e.* if and only if

$$\mathbf{P}\left(\left\lfloor 2^k |Z| \right\rfloor \text{ is even}\right) \geqslant p.$$

Now, the k-th bit of Z is 0 if and only if there exists an integer i such that,

$$\begin{aligned} & \left\lfloor 2^k |Z| \right\rfloor = 2i \\ \Leftrightarrow \quad & 2i \leqslant 2^k |Z| < 2i + 1 \\ \Leftrightarrow \quad & 2^{-k}(2i) \leqslant |Z| < 2^{-k}(2i+1). \end{aligned} \tag{5.9}$$

[3] When Y is a random variable, we choose $e_Y = \lfloor \log_2 |E[Y]| \rfloor + 1$.

One should note that the notions of significance and contribution are distinct, but related: if there are s significant bits with probability p, then all bits at ranks $c \leqslant s$ are contributing, with probability p. Indeed,

$$\begin{aligned} & \mathbf{P}\left(|Z| < 2^{-s}\right) \geqslant p \\ \Rightarrow \quad & \forall c \leqslant s,\ \mathbf{P}\left(2^c\,|Z| < 1\right) \geqslant p \\ \Rightarrow \quad & \forall c \leqslant s,\ \mathbf{P}\left(\left\lfloor 2^c\,|Z| \right\rfloor = 2 \times 0\right) \geqslant p. \end{aligned}$$

However, the k-th bit of Z being contributing with probability p does not imply that all bits at ranks $k' < k$ are also contributing[4]. This prevents the definition of such a notion as the number of contributing bits.

In the following, we study these two properties, **significant** and **contributing** bits, under the normality assumption (section 5.5.3) and in the general case (section 5.5.5).

5.5.3 Accuracy under the centered normality hypothesis

In this section we consider that Z is a random variable with normal distribution $\mathcal{N}(0, \sigma)$. In practice, we only know an empirical standard deviation $\hat{\sigma}$, measured over n samples. Because Z is normal, the following confidence interval [174, p. 282] with confidence $1 - \alpha$ based on the χ^2 distribution with $(n-1)$ degrees of freedom is sound [5]:

$$\frac{(n-1)\hat{\sigma}^2}{\chi^2_{\alpha/2}} \leqslant \sigma^2 \leqslant \frac{(n-1)\hat{\sigma}^2}{\chi^2_{1-\alpha/2}}. \tag{5.10}$$

It is important to note that σ is the standard deviation of Z and not of X. For example, when taking a second independent random variable Y as reference, if X and Y both follow a distribution $\mathcal{N}(\mu, \sigma')$, $Z = X - Y$ follows $\mathcal{N}(0, \sqrt{2}\sigma')$.

Significant bits

The theorem below is a more precise restatement of Stott Parker's Theorem 1 [187, p. 23]: *"the difference in the orders of magnitude of the mean μ and the standard deviation σ measures the number of significant digits of X (if $\mu \neq 0$, $\sigma \neq 0$)."* We define the notion of "measuring the number of significant digits" as the estimation of the probability that a given bit is significant at a given confidence level. We then prove that the number of significant bits is given by $-\log_2 \frac{\mu}{\sigma}$ as exposed by Stott Parker (since in a relative precision analysis, $\sigma_Z = \frac{\sigma_X}{x} = \frac{\sigma}{\mu}$ if X is normal and centered at the reference value), but adjusted by a quantity that depends only on the target probability and confidence level.

Theorem 5.5.1. *For a normal centered error distribution $Z \sim \mathcal{N}(0, \sigma)$, the s-th bit is significant with probability*

$$p_s = 2F\left(\frac{2^{-s}}{\sigma}\right) - 1,$$

[4] Although, it is the case for example when Z follows a Gaussian distribution.

[5] This interval is bilateral. If we were only interested in a lower bound for significant and contributing bits we could use the unilateral bound $\sigma^2 \leqslant (n-1)\hat{\sigma}^2/\chi^2_{1-\alpha}$.

with F the cumulative distribution function of the normal distribution with mean 0 and variance 1.

Proof. The probability that the k-th bit is significant is $\mathbf{P}\left[|Z| < 2^{-k}\right] = \mathbf{P}\left[Z < 2^{-k}\right] - \mathbf{P}\left[Z < -2^{-k}\right]$. Now $\mathbf{P}\left[Z < -2^{-k}\right] = 1 - \mathbf{P}\left[Z < 2^{-k}\right]$ by symmetry of the normal distribution, so that $\mathbf{P}\left[|Z| < 2^{-k}\right] = 2\,\mathbf{P}\left[Z < 2^{-k}\right] - 1$. Therefore,

$$\mathbf{P}\left[|Z| < 2^{-k}\right] = 2\,\mathbf{P}\left[\frac{Z}{\sigma} < \frac{2^{-k}}{\sigma}\right] - 1 = 2F\left(\frac{2^{-k}}{\sigma}\right) - 1.$$

□

The number of significant digits with probability p is s such that $2F\left(\frac{2^{-s}}{\sigma}\right) - 1 = p$, *i.e.* $F\left(\frac{2^{-s}}{\sigma}\right) = \frac{p+1}{2} \Leftrightarrow \frac{2^{-s}}{\sigma} = F^{-1}\left(\frac{p+1}{2}\right)$, so that

$$s = -\log_2(\sigma) - \log_2\left(F^{-1}\left(\frac{p+1}{2}\right)\right).$$

The above formula is remarkable because, whatever σ, the confidence interval to reach a given probability is constant and can be computed from a table for F^{-1}. Therefore, one just needs to subtract a fixed number of bits from $-\log_2(\sigma)$ to reach a given probability, as illustrated in figure 5.4.

In practice, only the sampled standard deviation $\hat{\sigma}$ can be measured, but it can be used to bound σ thanks to the χ^2 confidence interval in equation (5.10). This allows computing a sound lower bound $\hat{s}_{\text{CNH}}$ on the number of significant digits in the Centered Normality Hypothesis:

$$s \geqslant \underbrace{-\log_2(\hat{\sigma}) - \underbrace{\left[\frac{1}{2}\log_2\left(\frac{n-1}{\chi^2_{1-\alpha/2}}\right) + \log_2\left(F^{-1}\left(\frac{p+1}{2}\right)\right)\right]}_{\delta_{\text{CNH}}}}_{\hat{s}_{\text{CNH}}}. \tag{5.11}$$

Again, this formula is interesting since $\hat{s}_{\text{CNH}}$ can be determined by just measuring the sample standard deviation $\hat{\sigma}$ and shifting $-\log_2(\hat{\sigma})$ by a value δ_{CNH}, which only depends on a few parameters: the size of the sample n, the confidence $1-\alpha$ and the probability p. This is an improvement over the proposition of [187, p.23] to use a confidence interval on the estimate of μ. Instead, we propose a confidence interval directly on the quantity of interest, namely, the number of significant digits.

Numerical application

Let us consider the $X[0]$ variable from the ill-conditioned Cramer system from section 5.3.4. Statistical tests did not reject the normality hypothesis for $X[0]$. Here we would like to compute the number of significant digits relative to the mean of the sample with a 99 % probability. We consider the relative error, $Z = \frac{X[0]}{\hat{\mu}} - 1 \to \mathcal{N}(0, \sigma)$. Here σ will be estimated from $\hat{\sigma}$ with the χ^2 95 % confidence interval presented in equation (5.10). Computing δ_{CNH} for $n = 10\,000$, $p = 0.99$ and $1 - \alpha = 0.95$, yields $\delta_{\text{CNH}} \approx 1.4$. Recalling the sampled measurements from section 5.3.4, we get $-\log_2(\hat{\sigma}) \approx 28.5$.

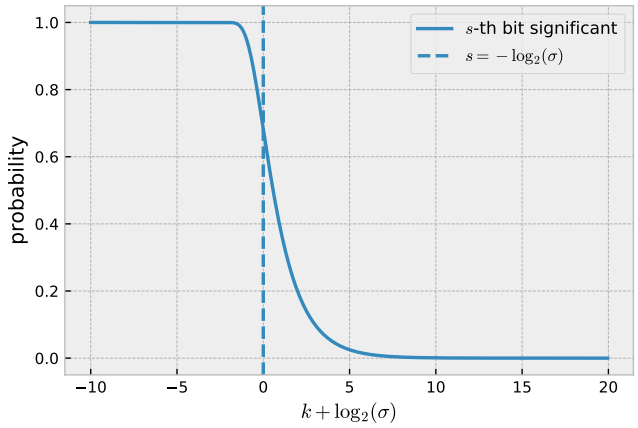


Figure 5.4: Profile of the significant bit curve: when the dashed line is positioned on the $-\log_2 \sigma$ abscissa, the curve corresponds to the probability that the result is significant up to a given bit.

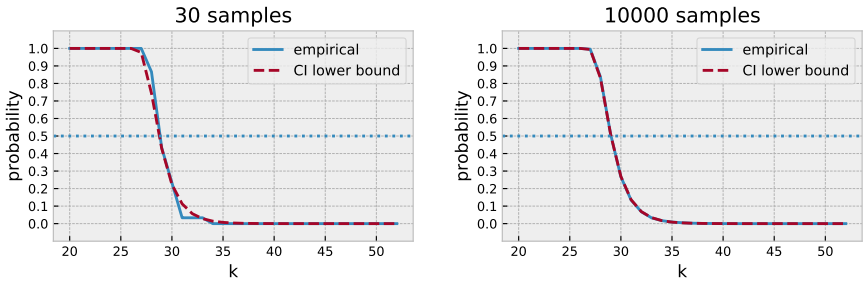


Figure 5.5: Significant bits for Cramer $x[0]$ variable computed under the normal hypothesis using 30 and 10000 samples. The Confidence Interval (CI) lower bound is computed by using the probability of theorem 5.5.1 and bounding σ with a 95% Chi-2 confidence interval.

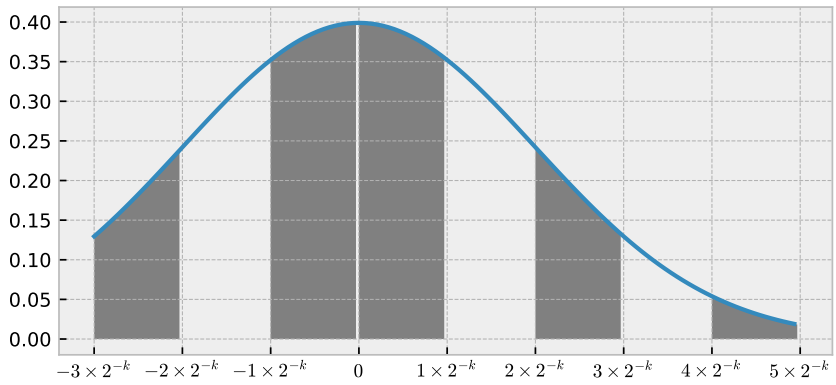


Figure 5.6: Normal curve; the gray zones correspond to the area where the k-th bit contributes to make the result closer to 0 (whatever the preceding digits).

Therefore, at least $28.5 - 1.4 = 27.1$ bits are significant, with probability 99 % at a 95 % confidence level. Figure 5.5 shows that the proposed confidence interval closely matches the empirical probability on the $X[0]$ samples. When the number of samples increases, the confidence interval tightness increases.

Contributing Bits

In the previous section we computed the number of significant bits. Now we are interested in the number of contributing bits: even if a bit is after the last significant digit, it may still contribute partially to the accuracy if it brings the result closer to the reference value.

The theorem below gives an approximation of the number of contributing bits which has the same property as theorem 5.5.1: this approximation computes the number of bits to shift from $-\log_2(\hat{\sigma})$ to obtain the contributing bits based on the same few parameters (sample size n, confidence $1-\alpha$ and probability p), and the shift being independent of $\hat{\sigma}$.

Theorem 5.5.2. *For a normal centered error distribution $Z \sim \mathcal{N}(0, \sigma)$, when $\frac{2^{-c}}{\sigma} \to 0$, the c-th bit contributes to the result accuracy with probability*

$$p_c \sim \frac{2^{-c}}{2\sigma\sqrt{2\pi}} + \frac{1}{2}. \tag{5.12}$$

More precisely, we can bound p_c as follows

$$\frac{1}{2} + \frac{2^{-c}}{2\sigma\sqrt{2\pi}} - \frac{\left(2^{-c}\right)^3\left(4e^{-3/2}+1\right)}{12\sigma^3\sqrt{2\pi}} \leqslant p_c \leqslant \frac{1}{2} + \frac{2^{-c}}{2\sigma\sqrt{2\pi}} + \frac{\left(2^{-c}\right)^3\left(4e^{-3/2}\right)}{12\sigma^3\sqrt{2\pi}} \tag{5.13}$$

Proof. Please refer to our detailed paper [183] for the proof. □

If we wish to keep only bits improving the result with a probability greater than p, then we will keep c contributing bits, with

$$c = -\log_2(\sigma) - \log_2\left(p - \frac{1}{2}\right) - \log_2\left(2\sqrt{2\pi}\right). \tag{5.14}$$

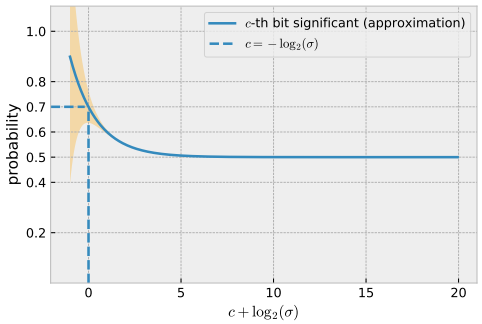


Figure 5.7: Profile of the contribution bit curve: when the dashed line is positioned on the $-\log_2 \sigma$ abscissa, the curve corresponds to the approximation 5.12 of the probability that the bit contributes to the result accuracy. The shaded area represents the bound on the error given by equation 5.13.

As before, this formula can be further refined by replacing σ with $\hat{\sigma}$ and adding a term taking into account the confidence level:

$$c \geqslant \underbrace{-\log_2(\hat{\sigma}) - \left[\frac{1}{2}\log_2\left(\frac{n-1}{\chi^2_{1-\alpha/2}}\right) + \log_2\left(p - \frac{1}{2}\right) + \log_2\left(2\sqrt{2\pi}\right)\right]}_{\hat{c}_{\text{CNH}}}.$$

Figure 5.7 plots the approximation of equation 5.12. We note that for a centered normal distribution the probability of contribution decreases monotonically towards 0.5. Close to 0.5, bits become more and more indistinguishable from random noise since their probability is not affected by the computation.

The approximation of equation 5.12 is tight for $k > -\log_2 \sigma$: in this case, the absolute error of the approximation formula is less than 2 %. The probability of contribution at $k = -\log_2 \sigma$ is 0.7. Therefore, equation 5.14 can be safely used for probabilities less than 0.7. We want to find the limit after which bits are random noise. This limit corresponds to a probability of 0.5 and the approximation is tight for $p < 0.7$.

Numerical Application

Figure 5.8 shows that the approximation proposed in this section tightly estimates the empirical samples in Cramer $x[0]$ example.

If we consider a 51 % threshold for the contribution of the bits we wish to keep, then we should keep $c = -\log_2(\sigma) - \log_2(p - \frac{1}{2}) - \log_2(2\sqrt{2\pi}) = -\log_2(\sigma) + 4.318108$. As in section 5.5.3, we estimate $-\log_2(\sigma)$ with a 95 % Chi-2 confidence interval, and compute $c = 32.8$.

This means that with probability 51% the first 32 bits of the mantissa will round the result towards the correct reference value. After the 34th bit the chances of rounding correctly or incorrectly are even: the noise after the 34th bit is random and does not depend on the computation. Bits 34 onwards can be discarded.

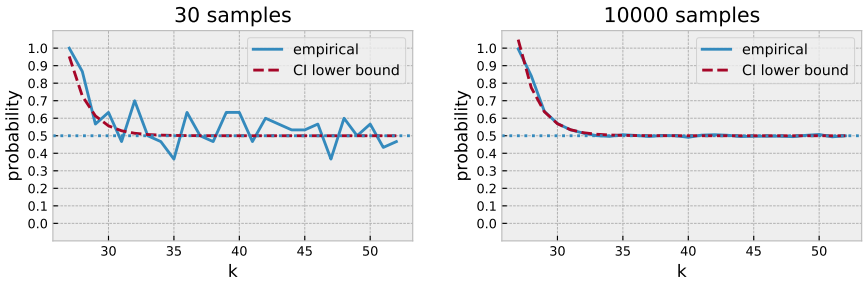


Figure 5.8: Contributing bits for Cramer $x[0]$ variable computed under the normal hypothesis using 30 and 10000 samples with the approximation of equation 5.12.

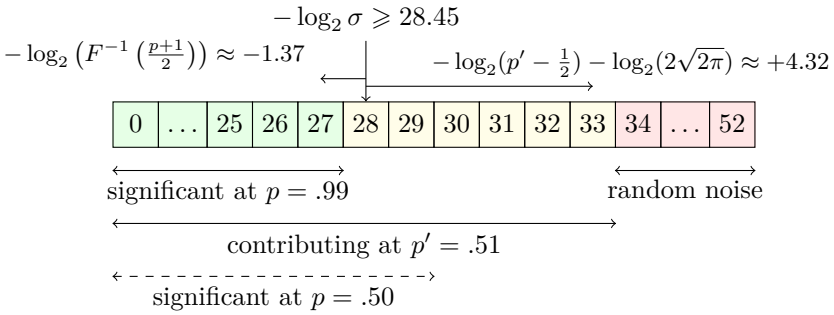


Figure 5.9: Summary of results on the $X[0]$ Cramer example. $28.45 \approx -\log_2 \hat{\sigma} - \frac{1}{2}\log_2\left(\frac{n-1}{\chi^2_{1-\alpha/2}}\right)$

5.5.4 Discussion for a normal centered distribution

Under the normality hypothesis, the quantity $-\log_2 \frac{\sigma_X}{|\mu_X|}$ introduced by Stott Parker is pivotal, but needs to be refined. In our framework, Stott Parker's definition maps to $Z = \frac{X}{|\mu_X|} - 1$, which computes the relative error to the mean. In this case Stott Parker's formula computes the position of the bit until which the result has 68 % chance of being significant, and that contributes to the result precision with a probability of 70 %.

However, from this bit, for each desired probability level, there is a simple way to compute a quantity by which to move back to be sure that the result is significant. It is also easy to compute a quantity by which to move forward in order to guarantee that all bits contributing more than a fixed level are kept. Figure 5.9 demonstrates this on the Cramer's example.

We recall from section 5.3.4 that in this example

$$\hat{s}_{\text{mca}} = -\log_2 \left|\frac{\hat{\sigma}_X}{\hat{\mu}_X}\right| = -\log_2(\hat{\sigma}) \approx 28.48$$

First a lower bound, 28.45, on $-\log_2 \sigma$ is computed with the Chi-2 95 % confidence interval. With this confidence level, it is a lower bound of s_{mca} (as

introduced in definition 5.3.2). It is also a lower bound of s_{sto} with 68 % probability (as introduced in definition 5.5.1). To compute a lower bound on bits that are significant with probability 99 %, we simply subtract 1.37 from this number. By adding 4.32 to this number we get the number of bits that contribute or round towards the reference with a probability of 51 %. The remaining bits in the mantissa are random noise.

It is important to understand the difference between contributing and significant bits. To illustrate this difference, we show in figure 5.9 the number of significant bits with 50 % probability which we estimate at 29 bits (28.45 shifted by +0.57 bits). We deduce, since the probability of significant bits decreases monotonically, that bits in the range 30-33 are significant with a probability under 50 %; in other words they are likely to be non significant. Yet *taken individually*, these bits are contributing with probability over 51%. Therefore, bits in the range 30-33 still contain useful information about the computation and cannot be considered random noise. It is up to the practitioner to decide how many bits to keep depending on their use-case.

Taking this into account, we propose to give a result, by printing all contributing bits at the chosen probability and confidence levels and an annotation bounding the error term at the chosen probability and confidence levels. This would result, for $k = -\log_2 \sigma$, in the following: for an absolute error with $\lceil k+(e_y-1)+4.318108 \rceil$ bits with the annotation $\pm 2^{\lfloor k+(e_y-1)-1.365037 \rfloor}$ at 99 %; for a relative error with $\lceil k + 4.318108 \rceil$ and $\pm 2^{\lfloor k-1.365037 \rfloor} \times y$ at 99 % for a relative error. In this notation, only digits that are likely to round correctly the final result with a probability greater than 1 % are written; the error at probability 99 % is written. In decimal, this notation takes up to two additional digits ($4.318108 \times \log_{10} 2 \approx 1.29$ digits) that are probably wrong, but still have a chance to contribute to the result precision. As an example, using this notation to display the IEEE-754 result of Cramer's $X[0]$ yields, with 9 contributing digits and 8 significant digits:

$$1.999999996 \pm 1.4\text{e-}08 \text{ (at 99\% with confidence 95\%)}.$$

These 10 digits contain all the valuable information in the result, and are the only ones that it would make sense to save, for example in a checkpoint-restart scheme.

5.5.5 Accuracy in the general case

The hypothesis that the distribution Z is normal, or that it has expectation 0 is not always true. We propose statistical tools to study the significance of bits as well as their contribution to the result accuracy that do not rely on any assumption regarding the distribution of the results.

To address the problem in the general case we reframe it in the context of Bernoulli estimation, which is interesting because:

- it does not rely on any assumptions on the distribution of Z;
- it provides a strong confidence interval for determining the number of significant digits when using stochastic arithmetic methods;
- thanks to a more conservative bound, it allows to estimate *a priori* in all cases for a given probability and confidence a safe number of samples to draw from the Monte Carlo experiment.

5.5.6 Background on bernoulli estimation

In the next section, we restate the problem of estimating the number of significant bits as a series of estimations of Bernoulli parameters. We present here some basic results on such estimations.

Consider a sequence of independent identically distributed Bernoulli experiments with an unknown parameter p and outcomes (p_i). Each value of the parameter p gives a model of this experiment, and, among them, we will only keep an interval of model parameters under which the probability of the given observation is greater than α. The set of possible values for p will then be called a confidence interval of level $1-\alpha$ for p : if the actual value of p is not in the confidence interval computed from the outcome, it means that the observed outcome was an "accident" the probability of which is less than α.

A case of particular interest in our study is the one when all experiments succeed. Then, the probability of this outcome is p^n under the model that the Bernoulli parameter value is p. We then reject models (i.e., values of p) such that $p^n < \alpha \Leftrightarrow n\ln(p) < \ln(\alpha)$. Now, $\ln(p) \leqslant p-1$ and $\ln(p) \sim p-1$ is a first order approximation when p is close to 1. Thus, taking $p < 1 + \frac{\ln(\alpha)}{n}$ leads to a probability of the observation less than α, and one can reject these values of p. In particular, taking $1-\alpha = 95$ %, we keep values of p greater than $1-\frac{3}{n}$, and $\left[1-\frac{3}{n}, 1\right]$ is a 95 % confidence interval. This result is known in clinical trial's literature as the *Rule of Three* [56]. Vice versa, in an experiment with no negative outcome, one can conclude with confidence $1-\alpha$ that the probability of a positive outcome is greater than p after $\left\lceil \frac{\ln(\alpha)}{\ln(p)} \right\rceil$ positive trials.

The general case can be dealt with by using the Central Limit Theorem, which shows that for a number n of experiments large enough (with respect to $\widehat{p} = \frac{1}{n}\sum p_i$), $\sqrt{n}(\widehat{p}-p)/(\widehat{p}(1-\widehat{p}))$ is close to a Gaussian random variable with law $\mathcal{N}(0,1)$. This approximation is known to be unfit in many cases, and can be improved by considering $\tilde{p} = \frac{1}{n+4}(\sum p_i + 2)$ rather than $\hat{p}$ as shown by Brown et al. [19] (this paper also presents other estimators to build confidence intervals in this situation; in particular, it proposes a revised method when $\tilde{p}$ is close to 0 or 1, a situation in which the confidence interval below may be overly optimistic). Then, with F the cumulative distribution function of $\mathcal{N}(0,1)$,

$$\left[\tilde{p} - \sqrt{\tilde{p}(1-\tilde{p})/n}F^{-1}(1-\alpha/2), \tilde{p} + \sqrt{\tilde{p}(1-\tilde{p})/n}F^{-1}(1-\alpha/2)\right]$$

is a $1-\alpha$ confidence interval for p. If we focus on a lower bound on the parameter p, we can also use $\left[\tilde{p} - \sqrt{\tilde{p}(1-\tilde{p})/n}F^{-1}(1-\alpha), 1\right]$ as a confidence interval of level $1-\alpha$.

Thus, from n independent experiments, of which n_s have been a success, we can affirm with confidence 95 % that the probability of success is greater than $\frac{n_s+2}{n+4} - 1.65\sqrt{\frac{(n_s+2)(n-n_s+2)}{(n+4)^3}}$. We can note that when $n_s = n$, this confidence interval is valid, but much more conservative than the one obtained above, that can thus be preferred in this particular case.

5.5.7 Statistical formulation as Bernoulli trials

Now, for each of the four discussed settings, presented in section 5.5.1, we can form two series of Bernoulli trials based on collected data.

	reference x	reference Y
absolute precision	$Z = X - x$	$Z = X - Y$
relative precision	$Z = X/x - 1$	$Z = X/Y - 1$

When the reference is a constant x, we consider n samples X_i. We form N pieces of data by computing $Z_i = X_i - x$ or $Z_i = X_i/x - 1$ respectively.

When the reference is another random variable Y, we form N pieces of data by computing $Z_i = X_i - Y_i$ or $Z_i = X_i/Y_i - 1$. In the case where $X = Y$ and we study the distance between samples of a random process, this requires $2N$ samples from X.

From these N pieces of data, we form Bernoulli trials by counting the number of success of

$$S_i^k = \mathbb{1}_{|Z_i|<2^{-k}}$$

for studying k-th bit significance, and

$$C_i^k = \mathbb{1}_{\lfloor 2^k |Z_i| \rfloor \text{ is even}}$$

for studying k-th bit contribution, where $\mathbb{1}$ is the indicator function.

From these two Bernoulli samples, the estimation can be made as above to determine the probability that the k-th bit is significant and the probability that it contributes to the result, for any k. The result can then be plotted as two probability plots, one for significance, the other for the contribution. The significance plot is non-increasing by construction, should start at 1 if at least one bit can be trusted, and tends to 0. The contribution plot should tend to $\frac{1}{2}$ in most cases, since the last digits are pure noise and are not affected by the computation.

5.5.8 Evaluation

The main goal of the Bernoulli formulation is to deal with non normal distributions. In this section, we evaluate the Bernoulli estimate on Cramer's $X[0]$ samples which follow a normal distribution. This is to keep a consistent example across the whole paper and to compare the results with the Normal formulation estimates. Later, in section 7.1, we will apply the Bernoulli estimate to distributions produced by the industrial simulation code EuroPlexus, some of which are not normal.

Figure 5.10 plots the significance and the contribution per bit probabilities for $X[0]$ using the Bernoulli estimation. The estimation closely matches the empirical results. It is interesting to compare the Bernoulli estimates with 30 samples to the Normal estimates in figures 5.5 and 5.8. The Bernoulli estimates are less tight and more conservative. This is expected since they do not build upon the normality assumption of the distribution.

If we are only interested in the number of significant digits, we can consider the Bernoulli trial with no failed outcomes since it provides an easy formula giving the required number of samples. In this case, the number of needed samples is $n = \lceil -\frac{\ln \alpha}{\ln p} \rceil$. We then determine the maximal index k for which the first k bits of all n sampled results coincide with the reference:

$$\hat{s}_{\text{B}} = \max \left\{ k \in \{1, 2, \ldots, 53\} \text{ such that } \forall i \in \{1, 2, \ldots, n\}, S_i^k \text{ is true} \right\}. \quad (5.15)$$

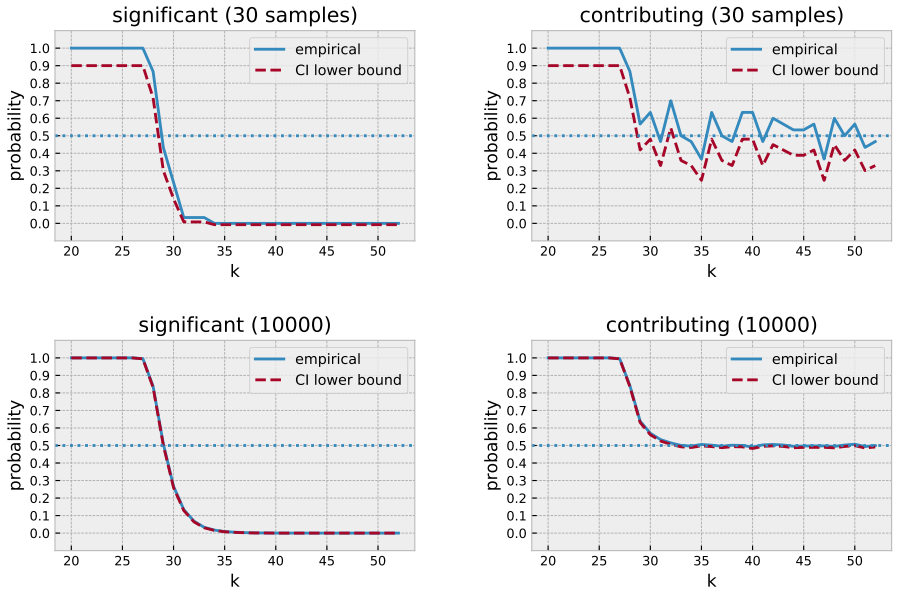


Figure 5.10: Significance and contribution per bit for variable $X[0]$ of the Cramer's system with 30 and 10000 samples.

We applied this method to the $X[0]$ sample from section 5.3.4. Assuming a $(1-\alpha) = 95\%$ confidence interval and a probability of $p = 99\%$ of getting s significant digits, we gather $n = 299$ samples. Among the collected samples, the 27^{th} digit is sometimes different compared with the reference solution, but the first 26 digits coincide for all samples. Therefore, we conclude with probability 99% and 95% confidence, that the first $\hat{s}_{\text{B}} = 26$ binary digits are significant.

5.6 Conclusion

In this chapter we presented Monte Carlo Arithmetic, a stochastic arithmetic method, which we use to estimate numerical accuracy in computer programs. We also proposed novel probabilistic measures of accuracy: the number of significant and contributing digits. We also give confidence intervals for these measures of accuracy in the normal and general cases. Thanks to the confidence intervals we can decide how many samples are needed when estimate the numerical error through MCA.

Unlike exact methods, MCA do not provide formal proofs of correctness. MCA provides accuracy assessments which are valid only for the specific set of input data that were used at run-time. Similarly to other run-time based analysis, it requires using numerous test cases to maximize the coverage of the analysis. On the other hand, MCA scales well to programs with greater code size and complexity and do not suffer from the intractability problems of exact methods. In the next chapter, we will introduce verificarlo, a tool for automatic error analysis in large simulation programs.

6 Verificarlo

Contents

This chapter includes contributions from the following publications and software projects:

- Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. "Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic." In: *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*. 2016, pp. 55–62. DOI: 10.1109/ARITH.2016.31. URL: http://dx.doi.org/10.1109/ARITH.2016.31
- Yohan Chatelain, Pablo de Oliveira Castro, Eric Petit, David Defour, Jordan Bieder, and Marc Torrent. "VeriTracer: Context-enriched tracer for floating-point arithmetic analysis." In: *25th IEEE Symposium on Computer Arithmetic, ARITH 2018, Amherst, MA, USA. June 25th-27th, 2018*. 2018
- Yohan Chatelain, Eric Petit, Pablo de Oliveira Castro, Ghislain Lartigue, and David Defour. "Automatic exploration of reduced floating-point representations in iterative methods." In: *Euro-Par 2019 Parallel Processing - 25th International Conference*. Lecture Notes in Computer Science. Springer, 2019
- E. Brun, D. Defour, P. De Oliveira Castro, M. Istoan, D. Mancusi, E. Petit, and A. Vaquet. "A Study of the Effects and Benefits of Custom-Precision Mathematical Libraries for HPC Codes." In: *IEEE*

Transactions on Emerging Topics in Computing 9.3 (2021), pp. 1467–1478. DOI: 10.1109/TETC.2021.3070422

- David Defour, Pablo de Oliveira Castro, Matei Istoan, and Eric Petit. "Custom-Precision Mathematical Library Explorations for Code Profiling and Optimization." In: *27th IEEE Symposium on Computer Arithmetic, ARITH 2020*. 2020, pp. 121–124

- Pablo de Oliveira Castro, Eric Petit, Yohan Chatelain, Alan Vaquet, Aurélien Delval, Killian Babilotte, Greg Kiar, Christophe Denis, Robert Lim, Matei Istoan, Nicolas Bouton, Marius Ghertescu, David Defour, and Olivier Jamond. *verificarlo/verificarlo: Verificarlo v0.7.0*. Version v0.7.0. Jan. 2022. DOI: 10.5281/zenodo.5833766. URL: https://doi.org/10.5281/zenodo.5833766

Verificarlo is an open-source tool, built upon the LLVM compiler [118], to analyze and optimize floating point computation in large programs. Verificarlo is available at http://www.github.com/verificarlo/verificarlo under an open source license.

Development on verificarlo started as a collaboration with Eric Petit, who at the time was working at UVSQ and now has moved to Intel, and Christophe Denis at ENS Cachan. Over the years, many people have contributed to the project. Yohan Chatelain, in particular, has extended significantly verificarlo during his Ph.D. thesis at UVSQ and continues to do so in his current post-doctoral work.

Verificarlo replaces at compilation time each floating point operation by a custom call. After compilation, the program can be run with various backends, such as Monte Carlo Arithmetic backend, or Variable Precision backend.

Figure 6.1 shows a bird's eye view of the verificarlo pipeline. It has specialized compiler passes that replace all FP operations by callbacks. The compiler passes also collect contextual information to locate numerical errors precisely in the code source and call stack.

Verificarlo includes seven backends which are extensively documented in the user manual. The four more important backends are:

- `mca and mca-int` backends which replace standard IEEE-754 computations by Monte Carlo Arithmetic.

- `vprec` backend which simulates the effect of using mixed-precision in a

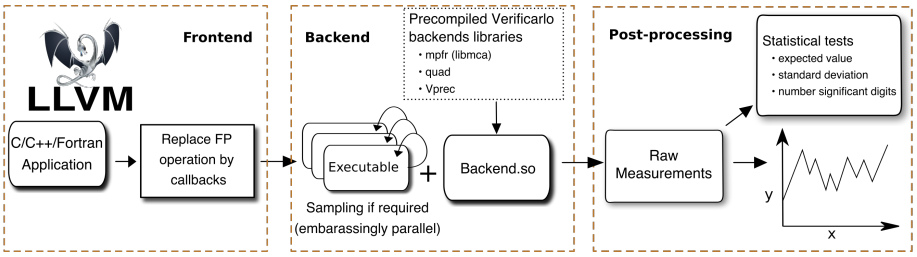


Figure 6.1: Bird's eye view of the verificarlo pipeline.

program.

- `cancellation` backend which can be used to detect catastrophic cancellations.

The latest version of verificarlo fully support OpenMP, Pthread and MPI parallel programs and backends have been designed to be reentrant and keep a coherent state between threads of execution. The state of the pseudo-random number generator uses thread-local storage variables which are local to each thread. The thread local seed is initialized when it is first accessed. This lazy initialization can handle dynamic thread spawning.

6.1 Compiler passes

Verificarlo relies on the LLVM compiler to instrument FP instructions. Like most compilers, LLVM architecture has three major components:

- The *Frontend* parses the source code, checks for errors, and produces an annotated Abstract Syntax Tree (AST). The AST produced is dependent on the particular source code language. It is translated to an independent Intermediate Representation (IR) language.
- the *Middle-end* optimizes and simplifies the code by applying multiple analysis and transformations passes on the IR. Depending on the optimization level, different sets of passes are selected.
- the *Backend* translates the IR to a specific instruction set assembly.

To transparently support multiple languages and architectures, verificarlo instruments the code through middle-end passes. Verificarlo includes two passes: *VFCInstrument* and *VFCFuncInstrument*. Both passes implement the standard LLVM `ModulePass` abstract class which operates on a whole compilation unit at the IR level.

VFCInstrument is responsible for replacing FP instructions by function calls to specific verificarlo hooks. This is quite straightforward but requires careful handling of vector operations and other corner cases such as comparison operations. Figure 6.2 shows a simple IR code extract before and after applying the *VFCInstrument* pass. The two FP subtractions on lines 3 and 5 are replaced by calls to the hook `float _floatsub(float, float)`. The original operands are passed as function arguments and the return of the hook call is used as the result.

VFCFuncInstrument instruments library and user defined functions. It will be presented in section 6.6.4

6.2 Advantages of operating at the optimized Intermediate Representation

Verificarlo instruments FP operations at the optimized Intermediate Representation level (IR). First, because the IR representation is independent of the

```
        ; Before applying VFCInstrument

        %30 = fsub float %28, %29
        %31 = load float, float* %7, align 4
        %32 = fsub float %30, %31
        store float %32, float* %6, align 4
        %33 = load float, float* %8, align 4
        store float %33, float* %5, align 4
        br label %34

        ; After applying VFCInstrument

        %30 = call float @_floatsub(float %28, float %29)
        %31 = load float, float* %7, align 4
        %32 = call float @_floatsub(float %30, float %31)
        store float %32, float* %6, align 4
        %33 = load float, float* %8, align 4
        store float %33, float* %5, align 4
        br label %34
```

Figure 6.2: A short IR excerpt from a Kahan summation algorithm before and after applying *VFCInstrument* pass.

```
float sum = f[0];
float c = 0.0, y, t;

for (int i=1;i<n;i++) {
    y = f[i] - c;
    t = sum + y;
    c = (t - sum) - y;
    sum = t;
}

return sum;
```

Figure 6.3: *Kahan compensated summation:* with `-O3 -ffast-math` flags the compiler simplifies and removes the computation of the compensation term `c`.

source language used, verificarlo can operate on any source language supported by the LLVM ecosystem that includes C and C++ through clang and Fortran through flang.

Second, the instrumentation pass is done after all the other front-end and middle-end optimization passes (which include all the floating point optimizations such as `-ffast-math` or `-freciprocal-math`). The interposition at compiler level takes into account the compiler optimization effect on the generated FP operation flow. On the other hand, Frechtling et al. [62] leverage source-to-source rewriting of floating point operations through the CIL tool for program transformation [137] to instrument a C program with MCA. Source rewriting approaches [62, 195] may hinder and miss compiler optimizations, since optimizations happen after FP interposition.

Third, instrumenting FP operations at the IR level allows to reduce the cost of this interposition by optimizing its integration with the original code.

In the following, we demonstrate the importance of capturing compiler effects on a standard use case: Kahan's compensated summation algorithm [86, p. 83] shown on figure 6.3. The C implementation is particularly sensible to

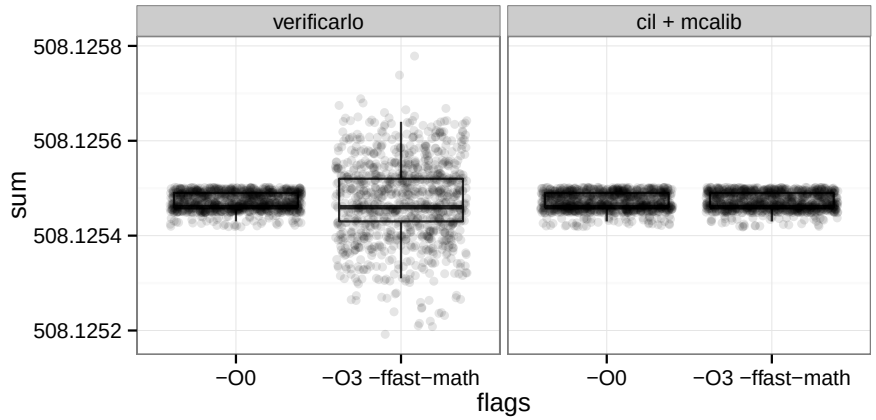


Figure 6.4: *One thousand MCA RR samples of Kahan summation:* CIL+MCALIB is unable to capture the compiler effect on Kahan's summation because it operates at the source level. On the other hand verificarlo operates after compiler optimizations and correctly shows that the `-O0` version is more precise than the `-O3 -ffast-math` version thanks to the compensation term `c`.

	-O0	-O3 -ffast-math
CADNA	7	7
CIL+MCALIB	7.3	7.3
verificarlo	7.3	**5.8**

Table 6.1: Number of significant digits estimated. Verificarlo is the only tool that detects that `-O3 -ffast-math` introduces a loss of accuracy.

compiler optimizations when floating point associativity rules are relaxed with `-ffast-math -O3`. The compiler uses simple common sub-expression elimination and rewrites line 9 as `sum = sum + f[i]` which is the naive non-compensated summation.

Using verificarlo and CIL+MCALIB [62] we measured 1000 sample executions of the Kahan summation code compiled with `-O3 -ffastmath` and `-O0`. The input array contains 1000 random single precision floats in the interval $[0, 1]$ and therefore has a condition number of 1. Only Random Rounding MCA errors were considered in this study.

Figure 6.4 compares the results between verificarlo and CIL+MCALIB. On one hand, CIL+MCALIB is unable to detect any difference between the two versions. Table 6.1 shows the number of significant digits predicted by CADNA, CIL+MCALIB and verificarlo for an array of 100000 floats. Again, CADNA and CIL+MCALIB are blind to compiler optimizations because they operate at source level. On the other hand, verificarlo correctly shows the loss of accuracy in the `-O3 -ffast-math` version.

6.3 Monte Carlo Arithmetic backend

As previously discussed in chapter 5, Monte Carlo Arithmetic is a powerful framework to understand the numerical stability of a function or program.

We recall that the result in Monte Carlo Arithmetic of $x = y \circ z$ with $\circ \in \{+, -, *, /\}$ and $y, z \in \mathbf{F}$ is computed as:

- $\text{mca}(y \circ z) = \text{round}(\text{inexact}(\text{inexact}(y) \circ \text{inexact}(z)))$ in full MCA mode,
- $\text{mca}(y \circ z) = \text{round}(\text{inexact}(y \circ z))$ in RR mode,

The inexact function models FP errors with ξ an uniformly distributed random variable in the range $\left(-\frac{1}{2}, \frac{1}{2}\right)$

- When x is representable, $\text{inexact}(x) = x$
- When x is inexact, $\text{inexact}(x) = x + 2^{e_x - t}\xi$

where t is the virtual precision and $e_x = \lfloor\!\lfloor \log_2(x) \rfloor\!\rfloor + 1$ is the order of magnitude of x.

In Stott Parker model's, all the operations performed in the right-hand side before the final rounding must be done in infinite precision. The final round operator (either IEEE-754 round-to-nearest or the corrected round operator in section 5.4.3), round the result to the machine precision p, ensuring that the MCA computation fits into the memory allocated for the original IEEE 754 result. Therefore, when instrumenting a program with Monte Carlo backend, verificarlo does not need to change the memory layout.

6.3.1 Implementing MCA with limited precision

When implementing MCA in software we cannot use infinite precision to compute the inexact terms. Let us consider the addition between two positive representables $x = y + z$ with $e_y \geq e_z$ and $e_x = e_y$. When summing, both values are normalized to the largest exponent e_y; so the mantissa of z is shifted to the right by $(e_x - e_z)$ bits as shown in figure 6.5.

$$inexact(x) = y + z + 2^{e_x - t}\xi = 2^{e_x}(m_y + 2^{e_z - e_x} m_z + 2^t \xi)$$

Because $t \in [0, p]$, the ξ mantissa left position is between 0 and p. Because $\xi \sim U(-1/2, 1/2)$, the first bit of the ξ mantissa is zero and is followed by an infinite number of random bits. We can separate the sum of the stochastic mantissa into two different terms: the bits of ξ that coincide with round-off bits and the remaining infinite random bits. The first term requires an addition with $2p + (e_x - e_z)$ bits. The second term would require an adder of infinite length, fortunately it can be handled with the adder's sticky bit [71, p. 19]. Because the remaining infinite random bits contain a one almost surely, we capture their effect in the final rounding by setting the sticky bit to one.

Nevertheless, this leaves us with a computation that must be performed with $2p + (e_x - e_z)$ bits. With binary64 numbers, $p = 53$ and normal exponents are in $[-1022, 1023]$, therefore we have a required precision up to 2151 bits which is much too costly.

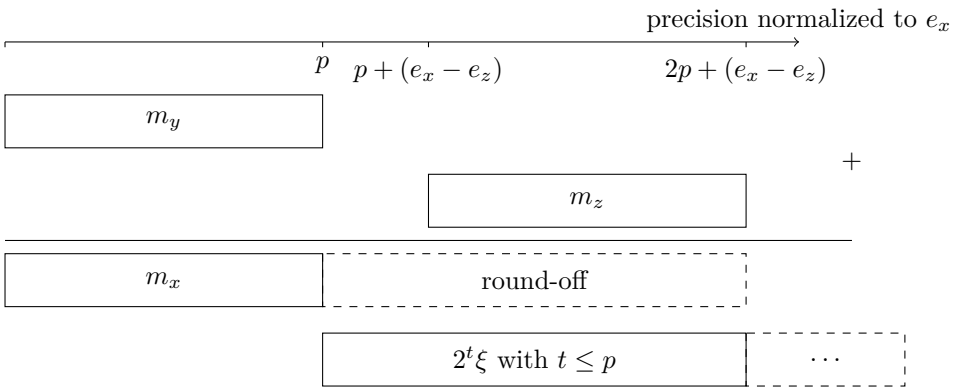


Figure 6.5: MCA computation $inexact(x) = inexact(y + z)$ with $e_x = e_y$.

To be efficient, our implementation will limit both the precision in which inexact terms are computed and the number of random bits in ξ. In verificarlo we choose to use an extended precision of $2p$. This follows Stott Parker recommendations [187, p. 38].

An extended precision of $2p$ bits requires drawing p random bits for ξ. Indeed, for the maximum virtual precision $t = p$, the stochastic bits are shifted just after the mantissa,

$$inexact(x) = x + 2^{e_x - p}\xi = 2^{e_x}(m_x + 2^p\xi) \quad \text{for } x > 0$$

Limiting the precision to $2p$ bits allows a much faster implementation of MCA, in particular for binary32 operands which can use the native binary64 type for extended computations. Truncating the precision introduces a relative error of at most 2^{-2p}, which is negligible compared to the machine precision 2^{-p}.

Similarly, when working with the full MCA mode, the inexact function is applied on the operands and the computation is performed with $2p$ bits of precision.

6.3.2 Quad and MPFR backends

The original MCA backend of verificarlo relied on mcalib [62]. Mcalib uses the GNU MPFR multi-precision floating-point library to perform the operations in extended precision. This simplifies the implementation since the extended precision operations are handled by a general library, but is not particularly efficient.

To optimize the original backend, we decided to use more efficient extended precision computations. When the original operands' type is binary32, we can use the standard binary64 IEEE-754 computations to compute the MCA result since it offers more than twice the precision of binary32. This is particularly efficient because binary64 computations are done in hardware.

When the original operands' type is binary64, we use GCC libquadmath. This library implements a quadruple precision format with a mantissa of 112 bits; on current architectures the quadruple operations are emulated in software. This faster MCA backend is called the *quad* backend.

Using binary64 and quadruple types for performing the computation handles nicely the case of denormalized values. Denormal operands, once converted to the extended precision, become normal.

6.3.3 MCA integer backend

The julia library `StochasticRounding.jl` [110] uses integer operations to perform the stochastic rounding. It is possible to extend their approach to compute the MCA inexact function using integers. We recall the original definition

$$\text{inexact}(x) = x + 2^{e_x - t}\xi$$

which we can write using the sign and mantissa of x as

$$\text{inexact}(x) = 2^{e_x - 1}((-1)^{s_x} m_x + 2^t \xi)$$

In the MCA integer backend we start by interpreting x as an integer i. The first α bits of i contain the sign and the exponent, then bits $\alpha + 1$ to $\alpha + p$ contain the pseudo-mantissa.

Then, we generate p random bits to form m_ξ the mantissa of ξ. The first bit acts as a sign bit in two-complement. We shift the mantissa to the right by $\alpha + t - 1$ bits, ensuring that the sign is extended during the shift. Next, we add the shifted random mantissa and i together,

$$i + (m_\xi >> (\alpha + t - 1))$$

Finally, we interpret back the result as a FP value and round it.

It is possible that the addition of i with the shifted random bits trigger a carry-out that increases or decreases the final exponent. Fortunately, this case is correctly handled on the integer computation because the IEEE-754 biased exponent representation ensures that an underflow or overflow from the mantissa carries on correctly to the exponent.

This particular case where an underflow changes the final exponent, corresponds exactly to the problematic case where x is close to a power of two demonstrated in section 5.4. When working with floating point numbers we had to be careful (cf. section 5.4.3) to use a consistent ϵ in the rounding operator. Interestingly, working with integers avoids the problem when $p = t$. Indeed, the stochastic noise in the mantissa is naturally reinterpreted when the exponent is decreased. So after moving across a power of two boundary, the stochastic noise is halved. This ensures when $t = p$ that the MCA integer backend avoids the rounding issues and matches the corrected MCA RR definition in section 5.4.3.

Let us illustrate this with the numerical example in section 5.4 with $x = 1.00001 \times 2^0$. We will only consider the pseudo-mantissas and ignore the sign and exponent bits in our notations. The pseudo-mantissa of x is 00001 since the initial 1 bit is implicit.

The minimal ξ mantissa in two-complement is 101, we arithmetically shift it to the right by $t-1 = p-1 = 2$ bits getting 11101. Now we can compute the sum as $00001 + 11101 = 11110$. The mantissa underflows in two-complement so the resulting exponent decreases by one. The final result is $\text{inexact}(x) = 1.1111 \times 2^{-1}$ which is above the midpoint 0.1111, so after the final rounding the result will be 1.0.

		verificarlo backends		
	original	IEEE	MCA quad	MCA integer
Kahan binary32	1.34s	2.36s (×1.7)	6.28s (×4.7)	7.76s (×5.8)
Kahan binary64	1.34s	2.34s (×1.7)	105s (×78)	64s (×48)
NAS CG A	0.80s	6.41s (×8)	173s (×216)	128s (×160)

Table 6.2: Execution time (and slowdown) for a Kahan sum of 100 millions elements and for the NAS CG A using different verificarlo backends.

6.3.4 Performance evaluation of the MCA backends

Simulating MCA on software is costly, this section compares the performance of the two previous MCA backends. Their overhead is measured on two benchmarks: the Kahan summation algorithm and the NAS CG conjugate gradient benchmark. Experiments were performed on a 6 core Coffee Lake i7-9850H at 2.60GHz with 15Gb of memory. The Kahan summation algorithm was run on an array with 100 million elements. NAS CG benchmark was run on dataset class A. The setup for both benchmarks can be found within the `tests/` directory shipped with verificarlo source-code.

Table 6.2 shows the execution time and overheads for the different backends. The IEEE backend in verificarlo, is a dummy backend where the FP hook mirror the standard IEEE-754 operation. It offers debugging possibilities and also provides a baseline to measure the instrumentation cost. On the Kahan benchmark, we see that just by diverting FP operations through called hooks we pay a 1.7 slowdown.

We observe that when summing binary32 elements with Kahan's algorithm, the overhead of the MCA backends translates into a slowdown factor between 4.7 and 5.8. Here the overhead is limited because the extended precision can be computed natively in hardware using a binary64 type.

Summing binary64 elements has a much higher cost because we have to emulate quadruple precision. The MCA quad backend has a ×78 slowdown, whereas the MCA integer backend has a ×48 slowdown. This shows that adding the ξ inexact term with integer types is significantly faster in the binary64 case.

For the NAS conjugate-gradient, which uses the binary64 type, we observe a ×8 slowdown for instrumentation cost, a ×215 slowdown for the MCA quad backend, and a ×160 slowdown for the MCA integer backend. Compared to the Kahan sum the overhead is higher since NAS CG is vectorized, and vector operations are serialized by the backends.

The large overhead difference in both MCA quad and MCA integer backends between binary32 and binary64 shows the high cost of emulating quadruple precision in software through GCC libquadmath. While emulating MCA for binary32 has a reasonable overhead, it is costly to use MCA on large binary64 programs. Fortunately, different MCA samples can be run in parallel with good scaling.

Nevertheless, a more efficient solution would be to take advantage of a hardware MCA implementation. Yeung et al. [203] have investigated MCA implementation at the hardware level through specialized FPGA coprocessors; but this prototype is not generally available to the practitioner. Stochastic round-

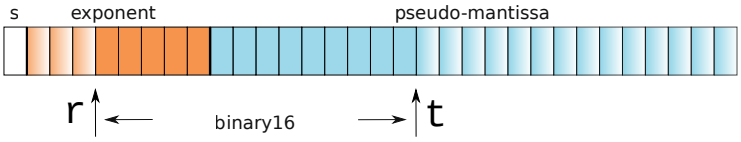


Figure 6.6: By setting $e = 5$ and $t = 10$, VPREC simulates a *binary*16 embedded inside a *binary*32. Opaque bits represent the new exponent range and precision available. The sign remains the same.

ing is more limited than MCA, since it only covers the RR mode with $t = p$. Nevertheless, it is used by machine-learning applications, which has fostered the development of specialized accelerators such as the Graphcore IPU and Intel's Loihi implementing stochastic rounding.

6.4 VPREC backend

The VPREC backend emulates FP formats that can fit into the IEEE-754 binary64 format. As illustrated in figure 6.6, VPREC allows to modify the bit length of the exponent $r \in [1, 11]$ and the pseudo-mantissa $t \in [0, 52]$.

At each instrumented floating-point operation, VPREC rounds the operands in (r, t), converts them to binary64, performs the operation in double precision and finally rounds the result in (r, t), and stores it in standard binary64 double representation. Converting the result back to double enables graceful degradation if some external libraries are not instrumented.

In VPREC, the custom FP representation is fully simulated in software. This allows to measure the effect of mixed-precision configurations or specialized FP hardware before doing the actual porting effort.

To correctly simulate a lower format different points must be addressed.

- Overflow and underflows must be correctly handled. If a computation falls outside the target virtual range, VPREC returns $\pm\infty$ for overflows and ± 0 for underflows.
- Rounding in the virtual precision t is achieved by adding one ulp at precision $t + 1$ followed by a truncation at t bits.
- Denormals are treated separately in the backend with a specific code path.

Unlike stochastic backends such as MCA, VPREC requires a single execution of the program. We observe a reasonable slowdown on full scale parallel applications ranging from $\times 2.6$ to $\times 16.8$ for very FP intensive codes.

We should note that VPREC performs the computation in binary64 which might trigger a first hardware rounding at 53 bits followed by a second rounding at precision t. Therefore VPREC might be subject to double-rounding issues. Figueroa [60] studies double rounding while emulating a lower precision format on a higher format. He shows that when $t \leq \frac{p-1}{2}$ double rounding is innocuous for the basic operations. Therefore, for $t \leq 26$ VPREC rounding is correct, and for $t > 26$ VPREC rounding is only faithful.

Many tools and strategies have been developed for exploring mixed precision in computer programs [168, 77, 115]. A review of these works and a comparison with VPREC is given in our paper [30].

6.5 Cancellation Backend

When subtracting two nearby FP values y and z, the most significant digits cancel. The result $x = y - z$ is renormalized, by shifting left the mantissa by the number of cancelled digits.

Not all cancellations should be considered numerical bugs. In particular, when y and z are two nearby *exact* values, subtracting y and z is an exact operation by the Sterbenz lemma.

On the other hand, when y and z are inexact computations which have accumulated rounding errors in the last digits; the renormalization will promote non-significant digits and increase the magnitude of the error. This effect is called *catastrophic cancellation*.

The cancellation backend is able to automatically detect cancellations. Cancellations can only happen when subtracting two nearby FP values. Therefore, the backend only instruments additions and subtractions. In the following, $y, z \in \mathcal{F}$ are the operands and $x \in \mathcal{F}$ the result,

$$x = y \circ z \text{ where } \circ \in \{+, -\}$$

The number of digits cancelled can be easily computed by comparing the exponents of the operands and the exponent of the result. This method was first proposed by Craft HPC [115] and is also used in Verrou [194]. The backend computes δ as

$$\delta = max(e_y, e_z) - e_x$$

A cancellation implies that δ is positive and in that case δ is the number of cancelled digits. The backend offers the possibility of reporting all cancellations for which δ is larger than a user-configured threshold. To simulate how the effect of cancellations propagates within complex programs, the cancellation backend can be combined with MCA Precision Bound mode.

6.6 Post-processing

Besides the already presented backends, verificarlo includes a set of post-processing tools to explore and analyze numerical bugs and optimization opportunities. These tools are extensively described in the verificarlo user manual. In the following, we will present the major ones.

6.6.1 Delta-Debug

Delta-Debug [204] is a general bug reduction method that allows to efficiently find a minimal set of conditions that trigger a bug. Here, we are going to consider the set of floating-point instructions in the program. Verificarlo Delta-Debug post-processing is built upon the stochastic delta-debug library developed by Bruno Lathuilière in the scope of Verrou [194] and Interflop projects.

In verificarlo we can use Delta-Debug for different objectives such as:

- finding a minimal set of instructions responsible for a numerical instability;
- finding a minimal set of instructions that cannot be run in lower precision.

Table 6.3 shows a simple Delta-Debug execution to find a reduced instruction set responsible for a numerical instability. By testing instructions sub-sets and their complement, Delta-Debug is able to find smaller failing sets step by step. It stops when it finds a failing set where it cannot remove any instruction. In this case, Delta-Debug finds a minimal failing set (ddmin) of size 1 (which is therefore also minimum). However, there is no guarantee of unicity.

Step	Instructions with MCA noise	Numerically Stable
1	1 2 3 4	stable
2	. . . . 5 6 7 8	unstable
3	. . . . 5 6 . .	stable
4	. . . . . 7 8	unstable
5	. . . . . 7 .	unstable
Result (ddmin)	. . . . . 7 .	

Table 6.3: Example of Delta-Debug bug minimization

By default, the Interflop Delta-Debug implementation iterates to find all the possible different ddmin sets. At the end, it produces the rddmin-cmp set which is the complement of the union of the ddmin sets. The rddmin-cmp set therefore includes the *stable* instructions and excludes the *unstable* instructions.

In verificarlo's tutorial, we give a complete example on how Delta-Debug can be used to fix an numerical bug in an iterative computation of the decimals of π. In the context of the TREX European Center of Excellence, François Coppens has used verificarlo's delta-debug on the Cornell-Holland Ab-initio Materials Package (CHAMP) application to detect opportunities for mixed-precision.

6.6.2 Verificarlo CI

Verificarlo CI is a continuous integration workflow for verificarlo. It tracks the numerically quality of an application over the course of its development on GitHub or GitLab. When launching verificarlo-CI for the first-time, it configures the CI system so each version of code pushed to the repository is checked through verificarlo.

The user can then add specific probes to the application regression test suites. For example, the call `vfc_probe_check` ensures that a test computed value reaches a given precision.

For each pushed code version, verificarlo-CI ensures that all the numerical assertions are valid. It then produces a detailed report that includes the measured numerical accuracy for each probed valued and a list of the failed assertions.

Verificarlo-CI has been developed by Aurelien Delval, during his internship at UVSQ funded by the TREX European Center of Excellence. It is used to continuously track the numerical accuracy of the Quantum Monte Carlo kernel library during its development.

6.6.3 Veritracer

VeriTracer automatically instruments an application and traces the accuracy of floating-point variables over time with one of the MCA backends. VeriTracer enriches the visual traces with contextual information such as the call site path in which a value was modified to understand how the floating-point errors propagate in complex codes.

VeriTracer was developed by Yohan Chatelain during his Ph.D. at University of Versailles. At the time, in a collaboration with Marc Torrent from CEA, we wanted to study the numerical stability within ABINIT [72]. ABINIT calculates, from the quantum equations of density functional theory (DFT), the optical, mechanical, vibrational, and others observable properties of materials.

Two important challenges in VeriTracer was to efficiently handle the vast amount of recorded data and to provide contextual information to correctly interpret the data. In this section we will give a simple example of VeriTracer outputs; more details about this work are found in Yohan Chatelain's thesis manuscript [29].

We consider the `simp_gen` function which computes an integral by Simpsons' rule over a generalized 1D-grid. This function is called within `Simp_gen` is called many times in ABINIT and appears in 31 different call-site paths (CSP).

VeriTracer used MCA RR with 53 bits of virtual precision to simulate IEEE round-off error in double precision. After post-processing, VeriTracer produces figure 6.7, which represents the number of significant digits for each call to `simp_gen`. Each point has a color that depends on its CSP. Many CSPs only correspond to a single call to `simp_gen` and are not easily identifiable on the figure.

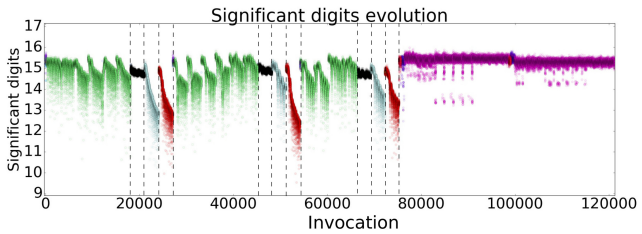
Among the four main CSPs, we observe several downward spikes that correspond to accuracy loss in the Simpsons' integral computation. Up to six digits of precision are lost in some of the calls.

Since `simp_gen` can be modeled as a large dot product computation, we tried to rewrite it using the compensated dot2 [58] operator. Dot2 is an error-free-transformation that compensates numerical errors through an error correcting term [144]. VeriTracer was run on the compensated `simp_gen` implementation. In 30 out of the 31 CSPs, the compensated algorithm fully fixed the precision loss. Interestingly, one of the CSPs (in red color) was not fixed by *dot2*. A dependency analysis of the code shows that `simp_gen`'s inputs in the failing CSP are themselves produced by upstream calls to `simp_gen`. The precision loss seems to be tied to the complex dependencies between the multiple calls and requires further study.

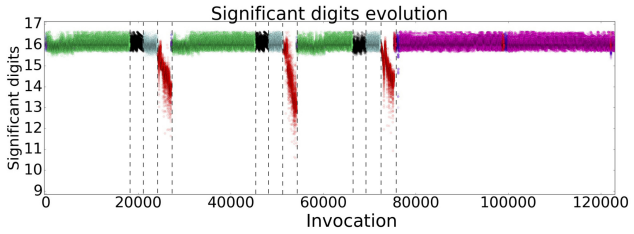
6.6.4 Variable precision in mathematical libraries

The default verificarlo pass only instruments the standard algebra and comparison operators. Nevertheless, some simulation codes make extended usage of calls to mathematical functions. For example the neutronic solver PATMOS [22] spends 70% of the execution time in calls to the libm mathematical library. To study such codes, the *VFCFuncInstrument* in verificarlo instruments library and user functions.

First, the pass maintains a lightweight stack-trace of the called functions; allowing us to retrieve the calling context. The stack-trace is updated at each



(a) Original



(b) Compensated

Figure 6.7: Profiles produced by VeriTracer of `simp_gen`. Several accuracy losses present in the original version (a) are improved by the compensated algorithm Dot2 (b). Each color maps one of the 31 distinct call-site paths. Dot2 improved 30 out of 31 call-site paths.

function prologue and epilogue. Stack unwinding libraries such as *libunwind* or the `backtrace()` call also provide the stack-trace. But these functions need to unwind the frame-pointer chain for each call. Verificarlo has a trace mode where the calling context is recorded for each FP call; in this case unwinding the stack is much too costly compared to updating the stack-trace twice per function.

Second, the pass captures and instruments the scalar arguments and return values of functions. This is useful to build a profile of the range and precision used for each argument and return value. Optionally, *VFCFuncInstrument* can apply the vprec backend on each argument or return value to reduce its precision. Using this feature, one can measure the effect of reducing the precision during function calls; such as calls to mathematical library functions.

Recent mathematical libraries, such as the Intel `MKL` Vector Math (`VM`) library, implement different precision presets for each call. The profile data collected with *VFCFuncInstrument* can be used to speculatively select the mathematical function implementation with the appropriate precision for a given scenario.

This pass was developed in the context of Matei Istoan's post-doctorate and Alan Vaquet's internship at the ECR lab (a joint lab between UVSQ, Intel, and CEA). FuncInstrument has allowed to study and optimize the Satellite Tracker Application SGP4 and the CEA's neutronic solver PATMOS. The details of this work are published in our papers [21, 46].

6.7 Conclusion

We presented verificarlo, an open-source project for debugging and optimizing numerical precision. Verificarlo instruments applications through an LLVM IR pass. Each FP operation transparently captured and redirected to one of the backends.

Monte Carlo Arithmetic backends introduce stochastic rounding noise in each operation, allowing through the statistical analysis proposed in chapter 5 to estimate the number of significant digits produced by a program.

VPREC backend simulates an execution in reduced precision. This is useful to measure the potential benefit of a mixed-precision version of a program. The backend allows us to decide which parts of the code could benefit from lower precision.

7 Numerical verification and optimization

Contents

This chapter includes contributions from the following publications:

- Devan Sohier, Pablo de Oliveira Castro, François Févotte, Bruno Lathuilière, Eric Petit, and Olivier Jamond. "Confidence Intervals for Stochastic Arithmetic." In: *ACM Transactions Mathematical Software* 47.2 (Apr. 2021). ISSN: 0098-3500. DOI: 10.1145/3432184. URL: https://doi.org/10.1145/3432184
- Gregory Kiar, Yohan Chatelain, Pablo de Oliveira Castro, Eric Petit, Ariel Rokem, Gaël Varoquaux, Bratislav Misic, Alan C. Evans, and Tristan Glatard. "Numerical uncertainty in analytical pipelines lead to impactful variability in brain networks." In: *PLOS ONE* 16.11 (Nov. 2021), pp. 1–16. DOI: 10.1371/journal.pone.0250755. URL: https://doi.org/10.1371/journal.pone.0250755
- Yohan Chatelain, Eric Petit, Pablo de Oliveira Castro, Ghislain Lartigue, and David Defour. "Automatic exploration of reduced floating-point representations in iterative methods." In: *Euro-Par 2019 Parallel Processing - 25th International Conference*. Lecture Notes in Computer Science. Springer, 2019
- El-Mehdi El-Arar, Sohier Devan, Pablo De Oliveira Castro, and Eric Petit. "Stochastic rounding variance and probabilistic bound: a new approach." In: *arXiv preprint arXiv:XXX* (2022)

In this chapter, we present how verificarlo has been used to numerically understand and optimize real applications.

Section 7.1 presents a study of the numerical stability in Europlexus, which resulted from a collaboration with Olivier Jamond at CEA. The aim was to ascertain if a buckling benchmark could be used as a numerical regression test when porting Europlexus to newer architectures.

Section 7.2 studies the numerical stability of Dipy, a Python toolchain for analyzing Magnetic Resonance brain images. This study was led by Greg Kiar, during his Ph.D. at the Montreal Neurological Institute and McGill University.

Section 7.3 uses the VPREC backend to optimize the Deflated Conjugate Gradient Solver (DPCG) used by YALES2. This study was led by Yohan Chatelain during his Ph.D. thesis at UVSQ. Using mixed-precision lowers the communication, computation, and energy cost of DPCG.

Finally, section 7.4 presents some perspectives and ongoing works around stochastic rounding.

7.1 Reproducibility analysis in the Europlexus simulation software

Europlexus is a fast transient dynamic simulation software co-developed by CEA, European Commission JRC, and other industrial and academic partners. The current source code has grown to about 1 million lines of Fortran 77 and Fortran 90. Europlexus has two main fields of application: simulation of severe accidents in nuclear reactors to check the soundness of the mechanical confinement barriers of the radioactive matters for the CEA; and simulation of explosions in public places to measure their impact on the surrounding citizens and structures for the JRC.

It handles several non-linearities, geometric or material, some of which lead to a loss of unicity of the evolution problem considered. This is, for example, the case for some configurations with frictional contact between structures or when the loadings cause fracture and fragmentation of the matter. Another obvious source of bifurcations of the dynamical system is the dynamic buckling.

Due to the non-associativity of FP arithmetic, the introduction of parallel processing in Europlexus raises a difficulty for the developer and the users: the solutions of a given simulation may differ when changing the number of processors used for the computation. We show here how verificarlo's significant digits estimate helps the developer to design relevant non-regression tests. To this end, we study a simple case that could serve as a non-regression test and symptomatic of a non-reproducibility related to FP arithmetic. It involves a vertical doubly clamped column to top and bottom plates. Vertical pressure is applied by lowering the top plate, which causes buckling of the column. The column is modeled as a set of discrete elements (here segments) connected at moving points called nodes.

The left plot in figure 7.1 shows the result after 300 simulation time-steps with the out-of-the-box Europlexus software using standard IEEE arithmetic. The sequential result is deterministic and does not change when run multiple times. We wish to study how the simulation is affected by small numerical errors.

We run the same simulations but this time using the verificarlo [47] compiler to introduce MCA randomized FP errors with a precision of $t = 50$. The cost to

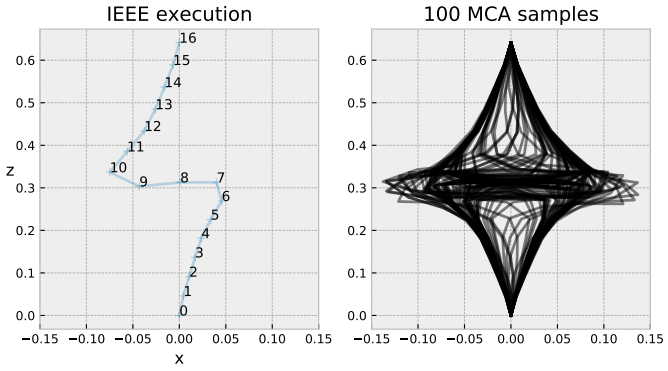


Figure 7.1: Europlexus buckling simulation of a doubly clamped column subject to a vertical pressure. The nodes of the column are labeled in the plot from 0 to 16. The left plot shows the deterministic and reproducible results produced by an IEEE-754 run of the simulation. In the right plot, a two-digits numerical error is simulated by collecting one hundred MCA samples with verificarlo ($t = 50$). The buckling direction is completely dominated by the small numerical error introduced.

instrument the whole Europlexus software and its accompanying mathematical libraries was low. In particular, no change to the source code was necessary thanks to the transparent approach of verificarlo.

The right plot in figure 7.1 shows the result of one hundred verificarlo executions. The direction of the buckling is chaotic and completely dominated by the FP errors introduced. This is not surprising as the buckling direction is physically unstable.

When parallelizing Europlexus or making changes to the code, it is important to check that there is no regression on standard benchmarks. Changing the order of the FP operations may introduce rounding errors. As we just saw, even minor numerical errors change the buckling direction making such a benchmark unsuitable for classical non-regression tests.

The column is modeled as a set of discrete elements connected by nodes. The distribution on the x-axis is normal, but whatever the node, there are no significant digits among the samples. The variation between samples is substantial on the x-axis; the x position clearly cannot be used as a regression test.

The distribution along the z axis is more interesting as it is non-normal for all the nodes. Figure 7.2 shows the quantile-quantile plot for node 1 (Shapiro-Wilk rejects normality with $W = 0.9$ and $p = 1.8e-06$). Because the distribution on the z axis is non-normal, we should apply the Bernoulli significant bits estimator. In this study, we measure the number of significant digits considering the relative error against the sample mean, so $Z = \frac{X}{\mu_X} - 1$.

To test the robustness of the proposed confidence interval, we computed the Bernoulli's estimate on the first 30 samples of the distribution. This corresponds to a probability of 90% with 95% confidence. We also computed the Normal estimate on the first 30 samples with the same probability and confidence.

Figure 7.3 compares the estimates to the empirical distribution observed on 100 samples. The Bernoulli estimate on 30 samples is precise and accurately

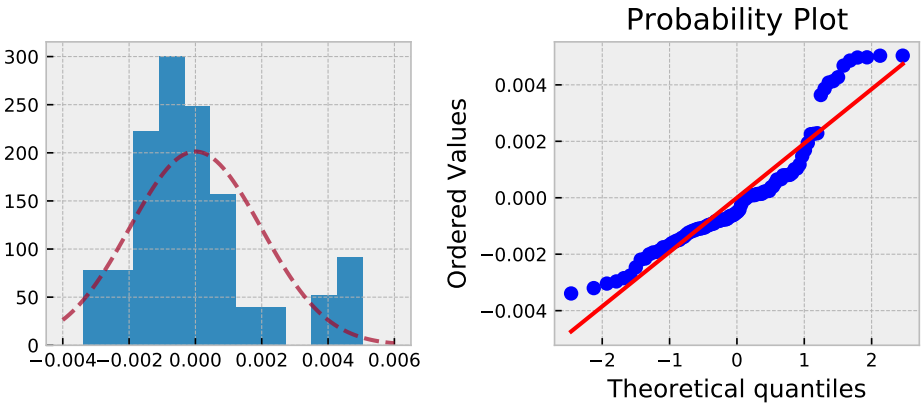


Figure 7.2: Non normality of buckling samples on z axis and node 1. Shapiro Wilk rejects the normality hypothesis.

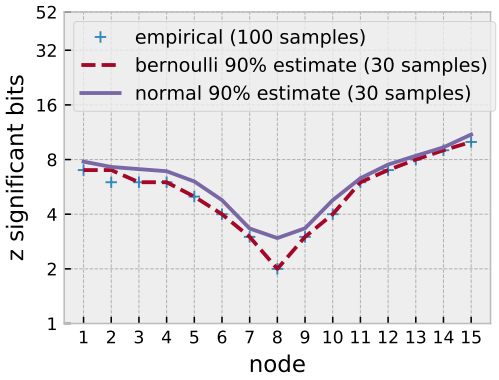


Figure 7.3: Significant bits on the z axis distribution. Bernoulli's estimation precisely captures the behavior (except for node 2). The normal formula overestimates the number of digits; this is expected since the distribution is strongly non-normal.

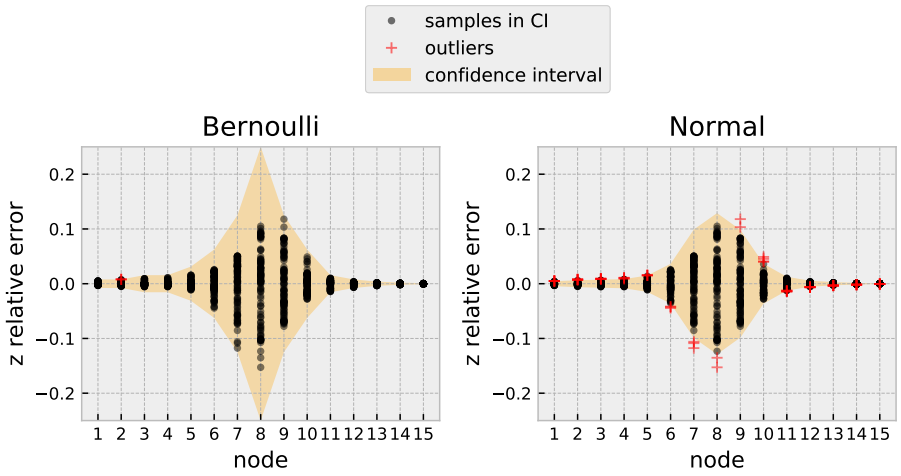


Figure 7.4: Relative error between the samples and the mean of the z-axis distribution. The shaded envelope corresponds to the computed confidence interval with 30 samples. Black dots are samples that fall inside the CI. Red crosses are outliers that fall outside the CI. In the Bernoulli case, only 3 samples out of 70 fall outside of the interval; which is compatible with the 90% probability threshold.

predicts the number of significant bits (except for node 2). The clamped node 16 has a fixed position, and therefore all its digits are significant. The other nodes have between 2 and 10 significant digits depending on their position.

Figure 7.4 shows the expected relative error on each node. We see that the Bernoulli estimate is robust and only mis-predicts the error on three samples of node 2. On the other hand, as expected, the Normal formula is not a good fit in this case due to the strong non-normality of the distribution: the normal estimate is too optimistic and fails to capture the variability of the distribution.

The previous experiments show that the x-axis has no significant digits and that the z-axis distribution has between 2 and 10 significant digits. For example, node 6 has 4 significant digits on the z-axis. Therefore, if the practitioner uses the z-axis position in this benchmark as a regression test, she should expect the first four digits of the mantissa to match in 90% of the runs. If the error is higher than that, then a numerical bug has probably been introduced in the code.

Another possibility for the practitioner is to adapt the benchmark slightly to make it more robust to numerical noise so it can be used in regression tests. For example, we can introduce a small perturbation in the numerical model by slightly moving node 2 along the x-axis. Then the buckling is expected to always occur in the same direction. Figure 7.5 shows what happens when node 2 is slightly displaced: the buckling becomes deterministic and robust to numerical noise: 51 bits are significant for the x-axis and z-axis samples with 90% probability; the two bits of precision lost correspond to the stochastic noise introduced. In this case, stochastic methods allow checking that the benchmark has become deterministic and assessing its resilience to noise.

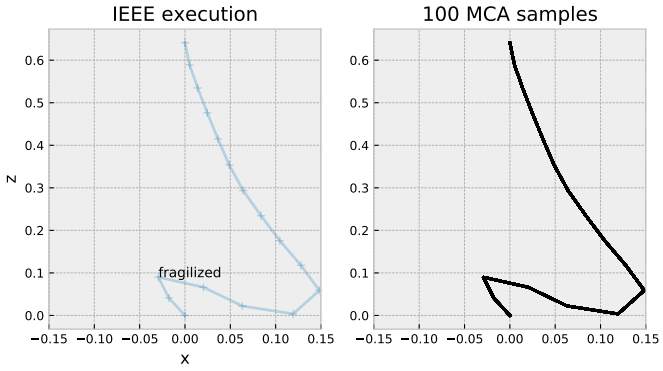


Figure 7.5: Europlexus buckling simulation with a fragilized node 2. By weakening the column, the physical process becomes reproducible in the presence of small numerical noise.

7.2 Evaluating brain-imaging numerical uncertainty

Connectomics is the study of the networks in the brain. Studying connections in the brain is important to understand *connectopathies* such as Alzheimer's Disease or Schizophrenia. In humans, connectomes are obtained by processing Magnetic Resonance Imaging (MRI) through complex software pipelines. G. Kiar et al [108] used verificarlo to evaluate the numerical uncertainty in the analytical software pipelines used to produce connectomes.

Their study used the Nathan Kline Institute Rockland Sample (NKI-RS) which contains high-fidelity imaging and phenotypic data from over 1000 individuals. First, MRI datasets are preprocessed using the standard FSL [97] library which corrects eddy-current movements, aligns the images, and subsamples the diffusion data.

Second, the preprocessed data is passed to Dipy [68] which generates the structural connectomes. Dipy offers two analysis pipelines. A deterministic pipeline estimates tensors at each voxel with a solid angle model and generates streamlines using the EuDX algorithm. A probabilistic pipeline fits a constrained spherical deconvolution model at each voxel and generates streamlines by iteratively sampling the fiber orientation distributions.

We collaborated with G. Kiar to compile Dipy with verificarlo in order to evaluate the pipelines' sensitivity to numerical noise. Dipy itself is coded in Python, nevertheless it depends on a complex software stack that uses Numpy, Cython, LAPACK, and BLAS. To facilitate the study and ensure future reproducibility, we developed Fuzzy, a docker image that contains a full Dipy stack compiled with verificarlo. In Fuzzy [106], all the FP operations in libraries and interpreters are routed to a verificarlo backend. G. Kiar applied MCA perturbations on the full software stack (dense perturbations) or only in the Dipy Python calls (Sparse perturbations). Here we only present dense perturbations results. For a much more detailed analysis, we refer the reader to the original paper [108].

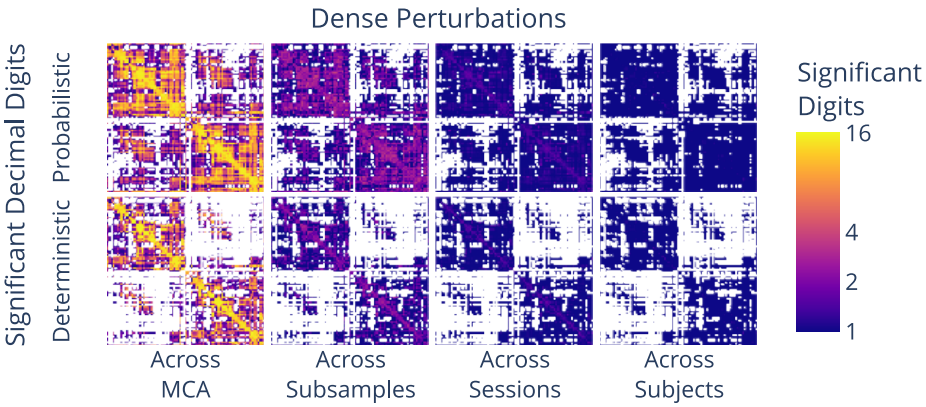


Figure 7.6: Significant digits in connectomes across MCA samples, subsambles, sessions, and subjects.

Figure 7.6 plots the number of significant digits of the produced connectomes on different sets of experiments. In all experiments 20 samples of MCA RR at precision 53 were produced.

- across MCA measures the significant digits of the connectomes between different MCA samples of the same input dataset.
- across subsamples measures the significant digits on different subsamples of the same imaging diffusion data. The subsamples come from the same subject and the same acquisition session.
- across sessions measures the significant digits between different acquisition sessions on the same subject.
- across subjects measures the significant digits between different individuals.

When exploring the variability across subsamples, sessions, and subjects, the number of significant digits progressively drops.

The precision of the connectome for a fixed input perturbed with MCA RR can be high, with some edges reaching 16 significant decimal digits. Nevertheless, the precision varies and for some edges drops to only 2 significant digits. We observe that the macro-scale network structure is stable but specific edge weights are affected by MCA.

Connectomes are often used as input to machine learning models used to predict brain-phenotype relationships. Kiar et al. used MCA perturbed connectomes to evaluate the robustness of machine learning models to numerical errors in the analytical processing pipeline of MRI. They used Principal Component Analysis followed by a Logistic Regression Classifier to predict the Body Mass Index (BMI) of a subject from their connectome.

Figure 7.7 reports the accuracy, F1 score, and explained variance for the BMI classification. The distribution of the accuracy and F1 score for the perturbed models is centered around the reference scores. Nevertheless, the standard deviation is high, and a few perturbed outliers drop below random classifier

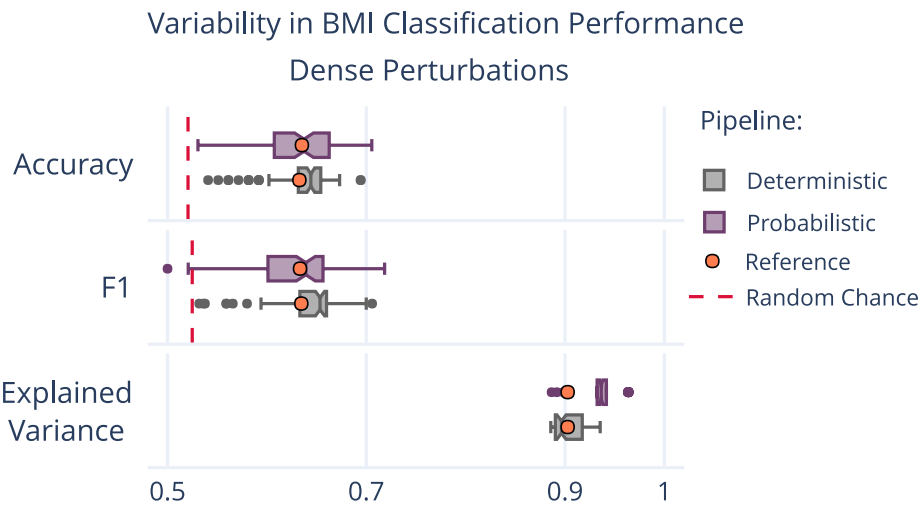


Figure 7.7: Variability in BMI classification on an MCA RR perturbed connectome.

performance. The numerical uncertainty should be accounted for when interpreting the results of brain-phenotype models. As previous works show, careful statistical analysis is needed to interpret neuro-imaging studies[93].

Kiar et al. use verificarlo to study the numerical variability in brain imaging studies. MCA distributions help them estimate numerical errors in connectomes and bound the precision one can expect from further models. They have also used MCA as a data-augmentation technique [107].

7.3 Mixed-Precision optimization in YALES2

YALES2 is a parallel CFD library that aims at solving the unsteady Navier-Stokes equations in the low-Mach number approximation for multiphase and reactive flows. A projection method enforces the mass-conservation constraint on the flow thanks to the resolution of a Poisson equation at each time step. The solver in YALES2 relies on the Deflated Preconditioned Conjugate Gradient (DPCG) [141, 126]. In this algorithm, a coarse grid is built from the fine mesh by merging a fixed number of elements together in super-cells. The general principle of deflation is the following. The coarse grid is used to converge the low-frequency eigenmodes of the solution, which represent the long-range interactions of the Poisson equation. This requires much less work than performing a classical CG on the fine mesh, which is only used to obtain the remaining high frequencies in a small number of iterations.

Numerically, the deflated operator is solved in *iterdef* iterations using a usual PCG method such that the convergence criteria *convcrit* is met. The solution is then expanded and injected into the main PCG loop on the finer grid. This whole process is repeated until the *maxnorm* of the fine grid residual is below a threshold. In both CG solvers (on coarse and fine grids), the system is preconditioned by the inverse of the diagonal. In all experiments, we constrained the total number of iterations performed by the algorithm to be below 1% of

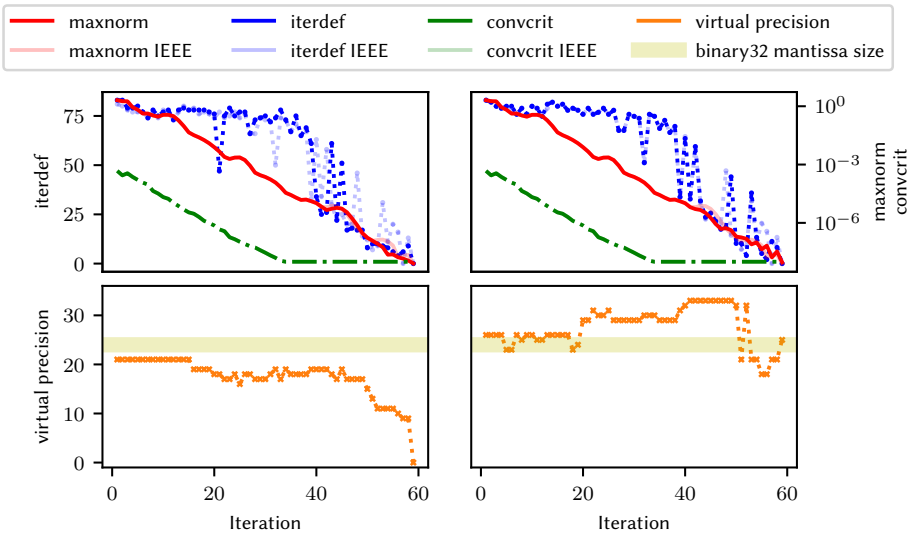


Figure 7.8: Adaptive precision searching on YALES2's DPCG with the deflated part (*left*) and the entire code (*right*). On both plots, we can see that our reduced precision solution follows the reference IEEE convergence profile.

additional iterations compared to the original.

The representative use case we focus on is the PRECCINSTA burner [117]. It is a well-known lab-scale burner used to validate combustion CFD solvers. We use 3 different mesh sizes of 1.75 million, 40 million and 870 million tetrahedral elements. For all configurations, the super-cell size in the coarse grid was set to 500 elements. We set the max norm of residual convergence criteria to 10^{-8}.

To reduce the search space, we consider two sets of functions. The first set, named *deflated*, is the set of functions used on the coarse grid to solve the deflated operator. The second set, named *all*, contained all the functions used to solve the fine grid operator.

7.3.1 Adaptive precision algorithm experiment on DPCG

We applied VPREC backend on the 1.75M mesh case to explore valid variable precision implementation. We statically set the exponent range to 8-bits in VPREC exploration.

Figure 7.8 shows the result of the exploration. In both graphs, the x-axis represents the number of iterations on the fine grid. In the top plot, the right y-axis represents, on the same scale, the norm *maxnorm* of the residual between two successive iterations and the convergence criterion *convcrit* of the deflated operator. The left y-axis represents $iterdef$, the number of iterations on the deflated operator. In the bottom plot, the y-axis represents the virtual precision used to compute the given iteration on the x-axis.

Figure 7.8 *(left)* shows that less than 23 bits of precision are required for the deflated operator on the 1.75 million elements mesh, with an average precision of 18 bits. Therefore, the deflated operator can be computed with `binary32`,

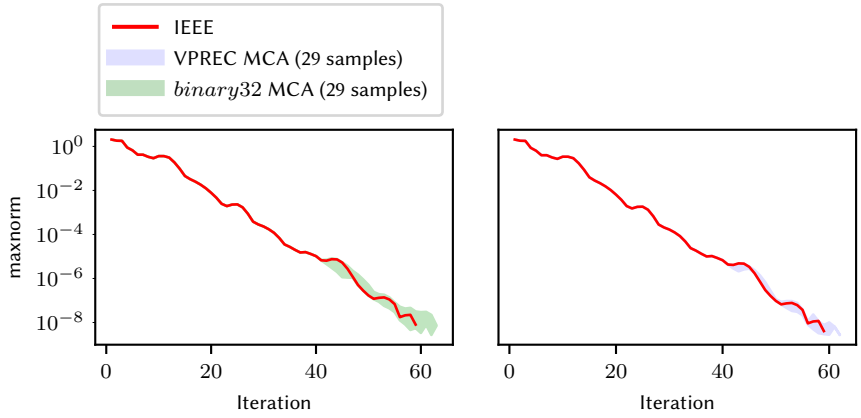


Figure 7.9: Resiliency of VPREC and `binary32` configurations. In red is the IEEE maxnorm convergence for reference. The blue envelope shows the 29 MCA samples for the previously found VPREC configuration. Green envelop shows the 29 MCA samples for the `binary32` configuration. All samples converge, showing the resiliency of both configurations.

resulting in a mixed-precision implementation detailed in section 7.3.3.

Figure 7.8 *(right)* shows the results of the proposed explorations on both coarse and fine grids at the same time. We notice that, contrary to the deflated experiment, the required precision increases over iterations. This is expected because the solver needs more and more precision to converge as it refines the solution.

7.3.2 Validating resiliency to round-off errors

We use MCA as the second step of our VPREC analysis to find a configuration resistant to round-off errors. We model this process as a Bernoulli trial. We run 29 MCA samples to simulate the effect of rounding errors. If any sample fails to converge, we conclude that the solution is not robust to round-off errors. On the other hand, if all samples converge, we conclude that the probability of convergence in the presence of round-off errors is 90% with a 0.95 confidence level, thanks to the confidence intervals from section 5.5.5.

Figure 7.9 shows that the VPREC solution found in the previous section is robust and converges for all the samples. Since the solution is very close to the `binary32` precision, our objective is to achieve a robust `binary32` configuration. The binary32 constant-precision configuration represented by the light red envelop in figure 7.9 converges in 57 to 63 iterations in the presence of round-off errors for all samples. This demonstrates that it is possible to safely rewrite the coarse grid operator of DPCG in `binary32`.

7.3.3 Evaluating mixed-precision version

The deflated operator of DPCG can be computed within the `binary32` format for most iterations, as shown in previous sections. To validate the results,

we compiled a mixed-version of YALES2 where the deflated operator can be executed either in `binary32` or `binary64` format.

We evaluated the mixed-precision version on the three different grids of PRECCINSTA and 10^{-9} convergence criteria. We limit the exploration algorithm to double and single precision since we are not running on variable precision hardware.

We use CRIANN cluster constituted of 366 bi-socket Intel Xeon E5-2680 nodes and Intel Omnipath interconnect. We gather statistics using IPM interface for Intel MPI.

Convergence

As predicted by our methodology, the computation converges, and all versions satisfy all accuracy constraints on the results. However, we noticed that larger experiments require extra initial double-precision iterations on the deflated grid. For example, two and four extra double-precision iterations are necessary for the 40M and 870M mesh. This is coherent with the observations of Cools et al. [37] about the importance of being precise in the first iteration of a CG recurrence. In these larger cases, it is necessary to switch the deflated grid from single to double precision when the deflated convergence criteria approaches the threshold of single-precision $\sim 10^{-8}$.

This effect did not appear in the smaller case with 500 elements per group. One explanation requiring further verification is that the granularity difference between the two grid levels is larger on the small mesh, and therefore the errors on the coarse grid iteration are less impacting on the fine grid iterations [126].

Communication and execution time

We measure a 28% to 67% reduction in the communication volume. Since DPCG is mostly bounded by communication latency, the performance gain is limited when the number of processors grows for a given size falling from 28% speedup to -2% slowdown on strong-scaling experiments. However, on the configuration that is used in production for this code, the expected speedup for daily usage will be in the 10% range.

Energy reduction

To evaluate the energy consumed, we used the Running Average Power Limit (RAPL) interface that is present in Intel processors. RAPL estimates the power consumption of different hardware domains such as the cores, the package, and the memory. Unlike a Watt-meter, RAPL does not directly measure power. Instead, it combines multiple hardware performance counters into a micro-architecture power consumption model.

In our experiments, we aggregated the power of two domains: the package domain, which contains the CPU and cache memories, and the memory domain. We instrumented the code to measure the energy in the deflated CG iterations for a single process. Figure 7.10 compares the energy consumed per iteration for the binary32 and binary64 versions. We measure a 16.6% energy gain for a single process.

This measure does not account for the energy reduction due to reduced communications; we cannot measure this directly but the simple model used

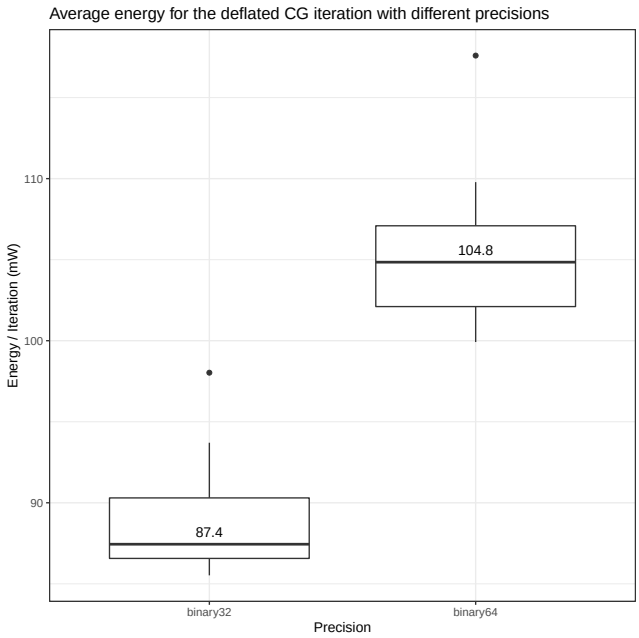


Figure 7.10: Energy per iteration of the deflated CG for a single Yales2 process (measured on an Intel Core i7-4770).

in [6] estimates that the energy gains are proportional to the communication reduction in a communication-bound iterative solver.

Using VPREC backend we explored the precision requirements over time, choosing an optimal precision for each application phase. Our methodology goes beyond mixed-precision approaches by exploring precision configurations at bit level. Tailoring the precision of Yales2's DPCG achieved consequent savings in performance and energy.

7.4 Perspectives on Stochastic Rounding

In theorem 5.4.2 on page 81 we showed that MCA RR with $t = p$ is equivalent to Stochastic Rounding (SR), first proposed in the 1950s by von Neumann and Goldstine [139]. While verificarlo indirectly uses this rounding mode to estimate the accuracy of a program, SR is also used as a replacement for the IEEE-754 default rounding mode in some computation domains. In 2021, El-Mehdi El-Arar started working on his Ph.D. at UVSQ, co-advised by Devan Sohier and myself. We are investigating the probabilistic properties of SR and in particular, its forward error bounds in different algorithms.

Previously, it has been demonstrated that in multiple domains such as neural networks, ODEs, PDEs, and Quantum mechanics [39], SR provides better results compared to the IEEE-754 default rounding mode. M. P. Connolly and Mary [125] show that SR successfully prevents the phenomenon of stagnation that takes place in various applications such as neural networks, ODES and PDEs. Recent works by Ipsen, Zhou, Higham, and Mary have computed SR

probabilistic error bounds for basic linear algebra kernels. For example, the inner product SR probabilistic bound of the forward error is proportional [95] to $\sqrt{n}u$ instead of nu for the default rounding to nearest mode.

El-Arar et al. [7] proposes a new approach to characterize SR errors based on the computation of the variance. We pinpoint common error patterns in a set of numerical algorithms and propose a lemma that bounds their variance. For each probability and through Bienaymé–Chebyshev inequality, this bound leads to better probabilistic error bound in several situations. Our method has the advantage of providing a tighter probabilistic bound for all algorithms fitting our model. This section will succinctly illustrate the advantage of using SR over nearest round when computing the inner product. For a detailed exposition of this work and proof of the error bounds please refer to [7].

Consider the inner product $y = a^\top b$, where $a, b \in \mathbb{R}^n$. We evaluate the product from left to right, i.e, $s_i = s_{i-1} + a_i b_i$, starting with $s_1 = a_1 b_1$, and finishing with $y = s_n = a_1 b_1 + ... + a_n b_n$. Let us note $\widehat{y}$ a computed result either with nearest round or SR. The condition number using the 1-norm of the inner product is

$$\mathcal{K}_1 = \frac{\sum_{i=1}^{n} |a_i b_i|}{|\sum_{i=1}^{n} a_i b_i|}.$$

The following bounds on the forward error with SR, where $u = \beta^{1-p}$ and $\gamma_n(u) = (1+u)^n - 1$, have been proved:

- Deterministic bound [86] for nearest round,

$$\frac{|\widehat{y} - y|}{|y|} \leq \mathcal{K}_1 \gamma_n(u/2), \tag{Det-IP}$$

- Probabilistic bound based on the Azuma-Hoeffding inequality for SR proved by Ipsen and Zhou [95],

$$\frac{|\widehat{y} - y|}{|y|} \leq \mathcal{K}_1 \sqrt{u\gamma_{2n}(u)} \sqrt{\ln \frac{2}{\lambda}} \text{ with probability at least } 1 - \lambda, \tag{AH-IP}$$

- Probabilistic bound based on the Bienaymé-Chebyshev inequality for SR (ours [7]),

$$\frac{|\widehat{y} - y|}{|y|} \leq \mathcal{K}_1 \sqrt{\gamma_n(u^2)} \sqrt{\frac{1}{\lambda}} \text{ with probability at least } 1 - \lambda. \tag{BC-IP}$$

The deterministic bound, Det-IP, applies to round to nearest and is proportional to $nu/2$. The probabilistic bounds apply to SR computations and are tighter because they grow proportionally to $\sqrt{n}u$.

We show in [7] that BC-IP is the tightest bound when n increases because $\sqrt{\gamma_n(u^2)} \leq \sqrt{u\gamma_{2n}(u)}$ for all $n \geq 1$.

To showcase the advantage of using BC method for large n, we present a numerical application of the inner product for vectors with positive FP values chosen uniformly at random between 0 and 1.

For smaller n, AH and BC bounds are comparable with a slight advantage for (AH-IP). However, for a large n, the AH bound grows exponentially faster than the BC bound. Asymptotically, the BC bound is therefore much tighter.

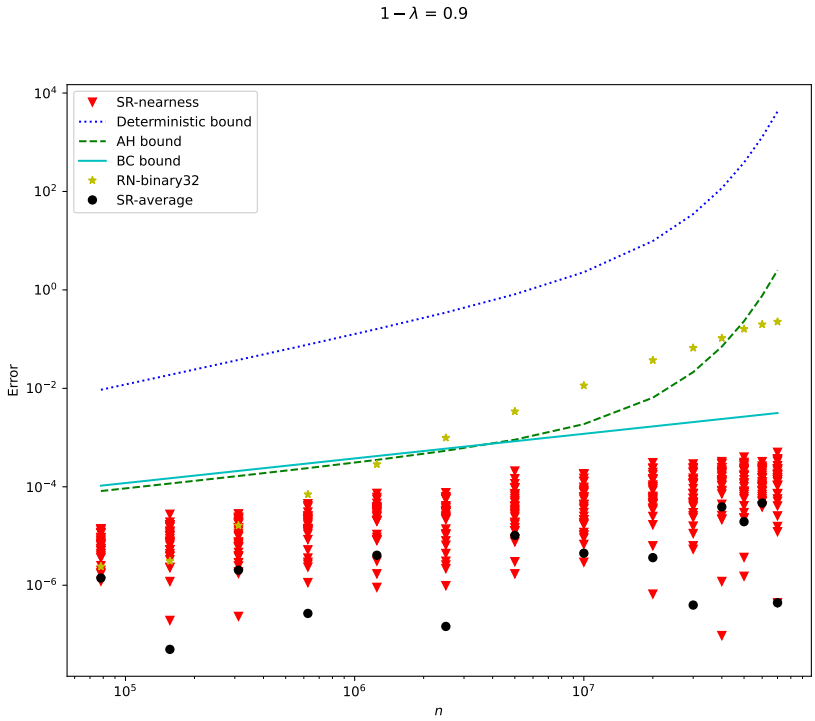


Figure 7.11: Probabilistic bounds with probability $1 - \lambda = 0.9$ vs deterministic bound of the computed forward errors of the inner product with $u = 2^{-23}$.

Interestingly, when n increases, a single instance of SR in binary32 precision is more accurate than IEEE-754 binary32 round to nearest. Therefore, for large vectors, using stochastic rounding instead of the default round to nearest improves the computation accuracy of the inner product.

Implications for MCA RR

In section 5.3.2 we showed that MCA RR was designed to capture the accumulation of rounding errors with an IEEE-754 computation. Indeed, in most of the programs we have studied, IEEE-754 computations are within the range of the distribution of MCA RR values.

However, for some algorithms, MCA RR is more accurate than round to nearest, for example, by avoiding stagnation during sums. In particular, the previous experiment shows that MCA RR with $t = p$ (corresponding to SR) deviates from IEEE-754 round to nearest behavior for large n.

When using MCA RR to estimate numerical accuracies, we should account for the cases where MCA RR diverges from round to nearest. Fortunately, they are easy to detect by computing the deviation between the SR samples and the IEEE-754 value.

Section 5.5.1 discusses the importance of choosing an adequate reference to compute the significant digits. Taking the IEEE-754 round to nearest computation as reference directly flags cases where IEEE-754 and MCA RR deviate. When taking as reference the average of SR values or the expected mathematical result, we need additionally to monitor the deviation against IEEE-754.

If a deviation between MCA RR and IEEE-754 is detected, it probably signals an accuracy problem and warrants investigation. For example, Stott Parker shows that for summations [187, p. 46] MCA RR is unbiased. Therefore, a deviation between MCA RR and IEEE-754 indicates that IEEE-754 has diverged from the expected mathematical result. In such a case, monitoring the deviation detects the numerical bug, but depending on the chosen reference, the number of significant digits computed will not necessarily be representative.

To conclude, the fact that MCA RR sometimes computes more accurately than IEEE-754 round to nearest does not invalidate its usage for error detection. However, the deviation between IEEE-754 and MCA RR should be monitored as part of this analysis.

8 Conclusion: HPC energy consumption

Contents

Faced with an unprecedented climate crisis, reducing our environmental impact is paramount. HPC's direct effects on the environment are twofold.

First, producing HPC hardware involves complex industrial processes and depends on costly resources such as rare-earth elements. Large areas are mined to extract rare-earth elements with major ecological and health impacts. Mining releases toxic and radioactive materials that poison the land and the people around the extraction sites. Moreover, rare-earth mines operate in developing countries where the labor cost is low and environmental regulations are less strict, leading to human rights abuses and child labor to satisfy the demands of the high-tech industry. Disposing of obsolete HPC hardware is also problematic, and waste is often exported to developing countries. For example, the Agbogbloshie district in Ghana has become one of the largest dumps where electronic waste is dismantled under unsafe conditions with harmful health [119] and environmental effects.

Second, HPC requires energy to fabricate the hardware and perform the computations, which translates into greenhouse gas emissions, particularly carbon dioxide emissions. Reducing these emissions is critical to mitigate global warming.

However, HPC also has indirect effects on the environmental crisis. The technological advances it brings can cause *rebound effects* where the efficiency gained increases demand for computation. Paradoxically, the increased demand can result in a net increase in carbon emissions and new hardware manufacture, worsening the ecological impact.

My research has focused on optimizations at the compiler, operating system, and software levels. Therefore, in this survey, I look at the environmental impact through the prism of power consumption and the associated carbon emissions. This choice is guided by my research field and does not imply that the other topics, such as rare-earth extraction, are less important.

One could think that optimizations reducing the computation cost of a code also improve the power consumption. Instead, the gained efficiency often amortizes an increase in the computation size or the simulation complexity. In that case, the larger problem size offsets the gained efficiency, and the net effect is that the power consumption stays the same or even increases.

The climate crisis is shifting the focus to reducing energy consumption and limiting the environmental impact of computations. Improving the computation efficiency might not be enough and could even be counterproductive because of *rebound effects*. I think more sobriety in our computation demands is required; in particular, instead of pushing for larger and more complex models, we should aim for the smallest model satisfying the required accuracy.

8.1 Energetic sobriety

We are in the midst of an ecological crisis, documented extensively by the scientific community [130]. One major aspect of this crisis is global warming, which calls for a drastic reduction of our carbon footprint. The *Stratégie Nationale Bas Carbone* is the French national roadmap for reaching carbon-neutrality in 2050; its last revision in 2020 operates under the hypothesis that France will reduce its energy consumption by 40% either through increased efficiency or sobriety. In this context, improving energy efficiency is paramount when optimizing HPC programs and architectures.

The TOP500 and Green500 lists have ranked the computing and power efficiency of the top supercomputers. In 2013, the most efficient supercomputer in these lists achieved 3208 MFlop/s/W; in 2022, the most efficient supercomputer achieved 39.4 GFlop/s/W. In a decade, the energy efficiency has been multiplied by 12. Efficiency improvements result from improvements in chip manufacturing technology, the use of specialized accelerators, and careful optimization of applications.

However, this efficiency improvement comes with a comparable increase in computation capacity. The total computing capacity of the TOP500 list has gone from 228.6 PFlop/s in 2013 to 3 EFlop/s in 2022, representing a 13 times increase in computing capacity.

When considering the net carbon impact of HPC, the efficiency savings are lost due to the increased computation demand. One might wonder whether HPC is subject to rebound effects (discussed in section 8.5), where the efficiency gained in a technology fosters its widespread adoption, nullifying the net energy savings.

Another option to reduce HPC's power budget is to compute less. Often, the trend is to reach for the more complex and fine-grained models available, which come with an increased computation cost. But we should not aim for the most precise and accurate computation when a simpler or less precise model would do the job. As demonstrated in section 7.3 with floating-point numbers, reducing the precision of the model can save computation and energy.

8.2 The global carbon impact of computation

Most studies that evaluate the carbon impact of computation do not specifically focus on HPC centers but consider the broader ecosystem of data centers which, besides high-performance computations, also runs data-oriented internet services.

Data centers worldwide use an estimated 200TWh [99] each year, representing 1% of the global energy demand and 0.3% of the global carbon emissions. That figure is under-estimated because it does not account for the embodied emissions [64] of data centers: the carbon emitted to fabricate the servers and their surrounding infrastructure.

When personal digital devices, mobile phone networks, and televisions are also included, the whole information and communication technology (ICT) carbon footprint is estimated to be 1.8%-2.8% of the global carbon emissions [63]. Embodied emissions are important in the ICT sector: they represent 23% of the total carbon impact. In France, the part of embodied emissions is even higher because the carbon intensity of French electricity is low. Therefore, besides reducing computations, it is essential to prolong the lifetime of ICT appliances.

A 1.8%-2.8% total carbon footprint for ICT might appear unimportant. In particular, when compared to the carbon emitted by the industry (29.4%), transportation (16.2%), or residential sectors (10.9%) [81]. Nevertheless, it should be put into perspective since the demand for HPC, particularly AI, is growing quickly; for example, it is estimated that DNN computations have increased by a 300 000 factor from 2012 to 2018 [176].

While this discussion focuses on carbon impact to keep matters simple, as mentioned in the introduction, ICT raises other important sustainability issues such as rare-earth mining and electronic waste.

8.3 Low-carbon electricity is not a silver bullet

To mitigate the carbon impact, some computation-centers buy renewable energy on the grid [154] or are built close to renewable energy sources such as geothermic plants. Others perform their computations when the demand on the grid is low, reducing their operation cost since they buy the electricity at a premium price and optimize the usage of the grid scheduling computation when there is a surplus of electricity.

Despite these strategies, Freitag et al. [63] conclude renewable energies are not a silver bullet due to their limited availability with current technology. They note that renewable energy is a scarce resource and that any energy taken by ICT will not be available for other uses.

The carbon intensity of electricity in France is low, in 2021, the annual average was 36 gCO2/kWh, as reported by Réseau de Transport d'Électricité (RTE). Indeed, 92% of french electricity comes from low-carbon production methods: 69% comes from nuclear power plants, 12% from hydroelectric power stations, 7% from wind turbines, and 3% from solar panels. Nevertheless, only 25% of the French final energy consumption comes from electricity. Fossil fuel's annual energy consumption was 1005TWh in 2021, which represents 61% of the final energy mix.

Due to the low carbon intensity of electricity, some argue that reducing computing energy consumption in France will not significantly lower the national carbon footprint and advocate focusing on decarbonating other sectors such as transport or heating, which depend mainly on fossil fuels. Yet, moving to electric heaters and cars will increase the required electricity budget drastically. Satisfying such an increase in electricity demand is impossible with the current electricity production infrastructure and scaling the renewable and nuclear production is a complex challenge [167] with multiple technological uncertainties. Given the forecasted growth of the ICT sector, reducing ICT energy consumption appears necessary to free renewable electricity for decarbonating heating, transport, or other essential industries.

Many countries still depend massively on fossil fuels for their energy production. For example, in 2021, US carbon intensity was 379 gCO2/kWh, ten times more than France. The world average carbon intensity in 2021 is 442 gCO2/kWh. The low carbon intensity in France is the exception and not the rule.

8.4 HPC efficiency

8.4.1 Dennard's scaling: 1970-2009

From 1946 to 2009, the power efficiency of processors doubled every 1.57 years [112]. This efficiency gain was achieved mainly through improvements in the lithographic and chemical manufacturing process of semiconductors, which reduced the gate length of transistors in each generation of processors.

Current logic circuits use Complementary Metal Oxide Semiconductor (CMOS) transistors. The power consumption of a CMOS gate is modeled by two terms representing the dynamic and static power, respectively,

$$P = \underbrace{1/2.C.V^2.f}_{P_{\text{dynamic}}} + \underbrace{V.I_{\text{leak}}}_{P_{\text{static}}}$$

where C is the equivalent charge capacity of the gate, V the voltage applied to the gate, f the clock frequency of the gate double-transitions (rising and falling), and I_{leak} the intensity of the leak currents.

In 1975, Dennard observed that for each new transistor generation, the transistor dimensions were reduced by 30% through manufacturing improvements.

This has the following implications:

- The voltage and capacitance also diminish by 30% since they vary linearly with the transistor size.
- The propagation time diminishes by 30% since the distance is reduced and the frequency grows by 42% since it varies inversely to the propagation time.
- The surface area is reduced by 50% $\approx 0.7 \times 0.7$.

Given these changes, P_{dynamic} should be roughly halved in each new transistor generation,

$$\Delta P_{\text{dynamic}} = \Delta C.\Delta V^2.\Delta f = 0.7 \times 0.7^2 \times \frac{1}{0.7} \approx 0.5$$

Assuming that the static power is small, the power dissipation per surface unit remains constant across CPU generations. Indeed, the surface is halved for each generation, but so is the power dissipation.

In practice, CPU manufacturers have used the surface gain to double the number of transistors in each generation, so the overall power consumption has stayed constant. Nevertheless, since the frequency also increases by 42%, the computing efficiency in FLOPS/W has improved across generations.

These observations, called Dennard's scaling, were empirically verified from 1970 to 2009. However, the rate of scaling has started slowing since 2009. The main reason is that with the continuing miniaturization of transistors, leak currents increase, which in turn increases the static power limiting the miniaturization of transistors and efficiency scaling [17].

8.4.2 Multi-Processing and accelerators: 2009-2022

Because frequency scaling has reached a limit, chip manufacturers turn to other strategies to improve FLOP/s. We have seen two main trends in this last decade. First, manufacturers have started increasing the number of processing elements per socket. For applications that expose enough fine-grained independent tasks, massive parallelism improves the performance despite the frequency limit. Second, there is increased usage of specialized processors such as GPU (Graphical Processing Units), TPU (Tensor Processing Units), FPGA (Field Programmable Gate Arrays). These processors target massively parallel applications where tasks are mostly synchronous and execute the same operations. They excel in specific domains such as dense linear algebra computations, machine learning, crypto-currencies mining, and others. Specialized processors tailored to a particular application domain can achieve impressive performance and often better power efficiency.

In this decade, power efficiency in computing elements has continued to improve due to technological advances and reductions in idle power. Efforts have also been made to optimize the other components in an HPC system, such as memory, storage, network interfaces, or power converters. The top two machines in the Green500 list for November 2021 illustrate the previous trends. The first supercomputer, MN-3, achieves 39.38 GFlops/W with 1 664 MN cores, specialized chips for matrix arithmetic. The second supercomputer, SSC-21, achieves 33.96 GFlops/W with 16 704 AMD EPYC 7543 cores. In recent years, AMD has significantly improved the power consumption [175, 188] of general-purpose CPUs.

As discussed in section 8.1, when comparing the most efficient supercomputers between 2013 and 2021, we see a 12× improvement in power efficiency. This trend is also true for internet data centers, Masanet et al. [129] shows that, in the last decade, server efficiency has improved owing to more efficient CPUs, storage-drive density and efficiency gains, better server utilization thanks to virtualization, and better data-center power usage effectiveness.

8.4.3 Software optimizations

Current processors offer multiple idle modes. For example, Intel processors have different P-States and C-States. In P-States, the CPU is active but operates in a power-saving mode. C-States are sleep modes, which turn off parts of the

system, deep C-States offer substantial energy saving during idle time, but there is a transition delay from P-States to C-States to wake up the processor.

P-States are based on DVFS (Dynamic Voltage Frequency Scaling) [124]: voltage and frequency are reduced to decrease the static and dynamic power, as modeled in equation 8.4.1. P-States are either changed by the operating system or runtime governors [202] or are directly managed by the hardware itself, depending on the workload. For some HPC applications, DVFS can achieve up to 16.5% [186] energy savings without significantly reducing their performance. The key idea is to reduce the frequency and voltage during the less computation-intensive phases or selectively reduce them for the uncore components [5].

On the application side, many factors affect the energy efficiency such as the choice of the algorithm and the data structures. The language, optimization, and compiler also impact the energy used. In general, compiled languages tend to be more energy-efficient [156] than interpreted languages. Yet, many factors are at play, and one should be careful when comparing languages since the results depend on the developers' implementation and expertise.

Often, optimizations that improve the performance also improve energy saving. For example, the DPCG mixed-precision optimization proposed in section 7.3 improves both the performance and the energy savings. Since the energy consumed is the product of the power and the computation time, reducing the computation time shrinks the energy product. In some corner cases, optimizations for energy and execution time are different, but on most platforms, the optimizations improving execution time also improve energy [151]. For this reason, the LLVM Compiler does not offer a specific optimization level targeting energy.

8.5 Rebound effects

In 1865, the economist William S. Jevons observed that Watt improvements of the steam engines' efficiency was accompanied by an increase in coal consumption. Watt's engine efficiency fostered its adoption by a wide range of industries. Jevons paradox, also called the rebound effect, states that an increase in efficiency in resource use will generate an increase in resource consumption rather than a decrease.

Gossart [75] reviews the literature on rebound effects of ICT. He distinguishes different levels of rebound effects.

- Direct rebound effects increase the spread of ICT technologies.
- Indirect rebound effects happen when increasing ICT efficiency reduces the cost of goods or services produced with ICT. This indirect cost reduction increases the consumption of other resources.
- Economy-wide rebound effects structurally change production and consumption patterns, potentially increasing production and associated carbon emissions in other fields.

As previously seen in section 8.4, from 1970 to 2009, the power efficiency of processors doubled every 1.57. Nevertheless, Hilty, Lohmann, and Huang [87] show that in the same period, the computation power for personal computers

doubled every 1.5 years. Moreover, the number of installed computers doubled every three years from 1980 to 2008. The increase in computation demand offset efficiency gains. The continuous improvements in chip manufacturing cost and frequency lead to quick obsolescence of old slower models and an explosion in demand. This is an example of a direct rebound effect in CPUs.

For HPC, the 12 × efficiency improvement achieved between 2013 and 2022 is shadowed by a 13 × computation power increase. Figure 8.1 confirms this trend by looking at the evolution of the first 100 systems in the TOP500 supercomputer list in the last decade. The geometric growth of the energy efficiency is offset by a geometric growth of the peak computation power. This results in a moderate growth in power consumption.

The same is true for data-centers, for which increases in demand are balanced by efficiency gains, producing a net power increase of 6% from 2010 to 2018 [129]. All in all, despite large efficiency gains, the net effect is an increase in carbon emissions.

In the last decade, the growth of the total power consumption has been moderate. But with the slowing down of CMOS scaling, there is a risk that the efficiency improvements in processors will reach a limit [63]. In that scenario, increases in computation volume would directly translate into higher energy consumption and carbon emissions.

Few studies directly quantify the indirect and economy-wide rebound effects of ICT. Yet, in its last report [181], the ICPP working group III recognizes rebound effects in digitalization as a risk towards carbon emissions increase since they "have the potential to steeply increase energy efficiency in all end-use sectors through material input savings and increased coordination. [...] economic growth resulting from higher energy and labour productivities can increase energy demand and associated GHG [Green House Gas] emissions. Importantly, digitalization can also benefit carbon-intensive technologies".

8.6 Computation sobriety: when less is more

In computer simulations, efficiency gains are often leveraged to increase model complexity or size. Much like in Jevons' paradox, efficiency gains can increase computation and data storage capacity demands.

For example, Neural Networks have received many optimizations in the last decade. Codes have been optimized to run on GPU and, later, on dedicated architectures such as Google Tensor Processing Unit. Algorithms and data representations have been optimized [24]. For example, networks exploit smaller floating-point formats, such as bfloat16, to reduce bandwidth, computation time, and storage size.

Despite the optimizations, the training cost of neural networks has spiked [191]. Figure 8.2 shows that from 2012 onwards, the training cost for AI systems doubles every 3.4 months. The training cost from 2012 to 2018 has grown by a $\times 300\,000$ factor.

Schwartz et al. [176] studies different generations of image recognition neural networks. In figure 8.3, Schwartz compares the accuracy, the training cost, and the number of parameters. The accuracy, expressed as a percentage, is the success rate in an object recognition task. The training cost is measured as the

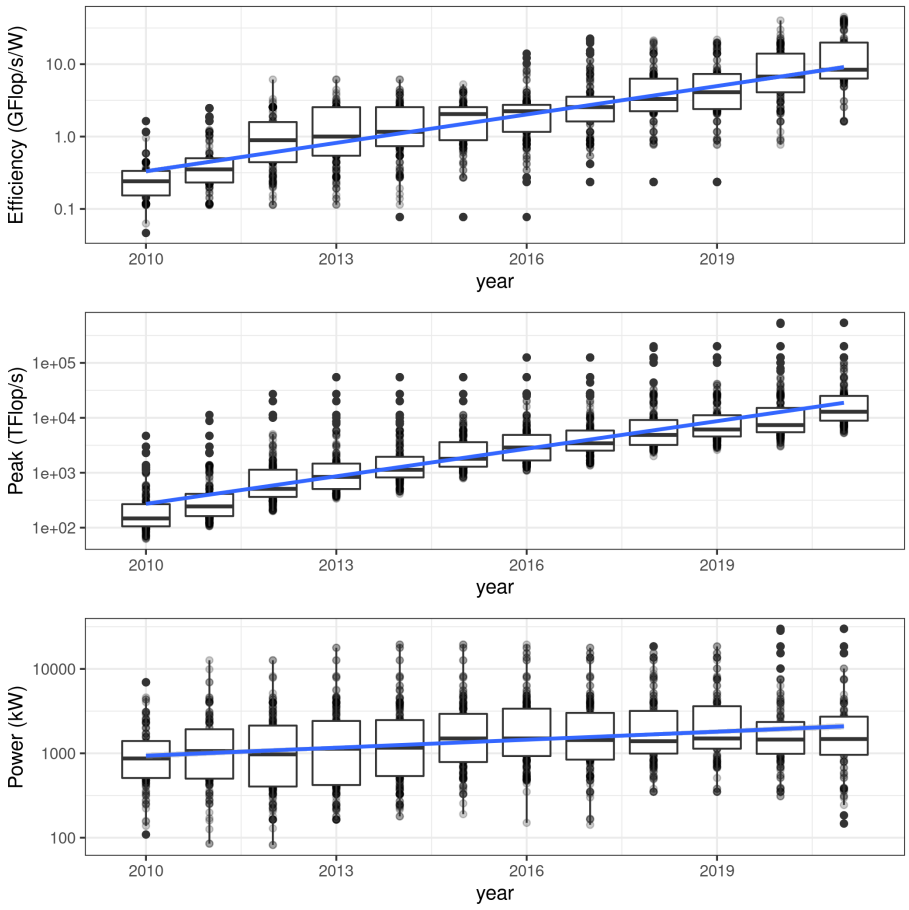


Figure 8.1: Evolution over time of the first 100 systems in the TOP500 list. Peak is the theoretical peak computing performance. Each point is an HPC system. Box plots span from the first to the third quartile. The blue line is a linear regression on the individual HPC systems' metrics. Because the vertical axis is logarithmic, Peak and Efficiency follow roughly a geometric growth.

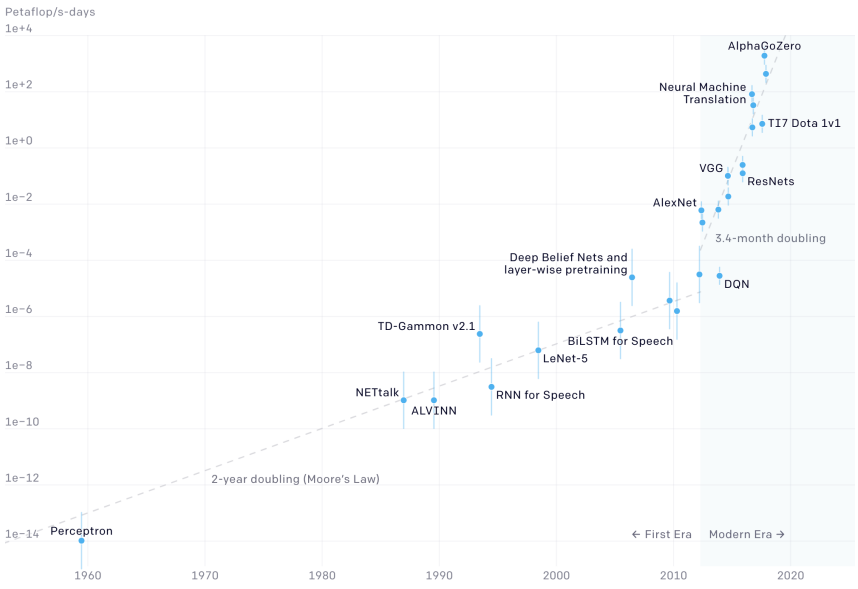


Figure 8.2: Computation power required to train AI systems, data and figure by OpenAI [4]. From 2012, AI computation demands have become steeper doubling every 3.4 months.

number of floating-point operations used during training. The number of model parameters is also reported.

Schwartz et al. [176] shows that there are diminishing returns on accuracy. The accuracy gains are marginal for a linear increase in computation power and model complexity.

Schwartz shows that in the last decade, linear accuracy gains have required an exponential increase in model complexity and computation cost. Such as trend appears unsustainable. It appears necessary to weigh higher accuracy's benefits against the increased computation cost. Since the needed accuracy depends on the model finality, it is difficult to draft general guidelines. Nevertheless, Schwartz et al. [176] advocates for systematically reporting the model efficiency, such as the total FLOP or energy used during training, alongside the accuracy. Considering an efficiency metric makes it possible to select the smallest model satisfying the target accuracy.

8.6.1 Case study in healthcare

To illustrate the previous discussion with a concrete case study, I turn to the work of D'Acremont [41] and her colleagues at the Swiss Tropical and Public Health Institute. They have developed e-POCT, an algorithm for the management of febrile illnesses in resource-limited countries [104]. D'Acremont discusses the diminishing returns of AI models when applied to healthcare.

e-POCT uses Classification and Regression Trees to help diagnose Tanzanian

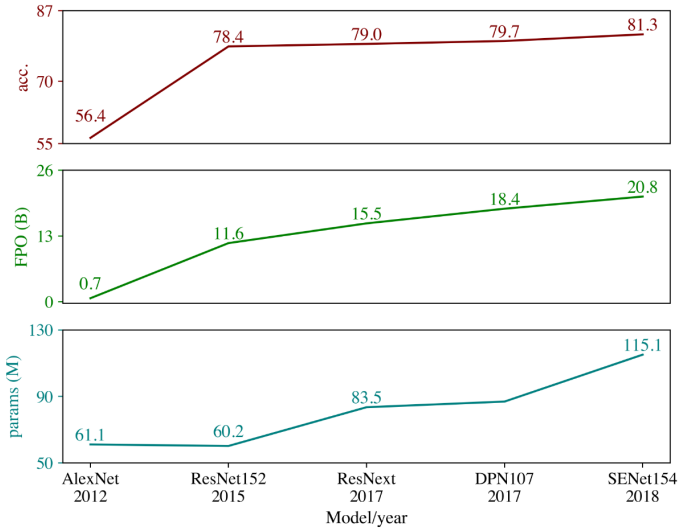


Figure 8.3: Accuracy (acc.) vs. computation (FPO) for AI neural networks identifying objects in the ImageNet dataset. Figure from Schwartz et al. [176]. acc. is the top-1 accuracy percentage. FPO is the total number of floating point operations required for training the network. params is the number of model parameters.

children with febrile illnesses and recommends adequate treatment. e-POCT is embedded on an Android tablet which gathers data from different sources: objective measures from an oxymeter, hemoglobinometer and glucometer; Point of Care (POC) tests for Malaria or VIH; clinical signs observed by the treating clinician. The algorithm follows the classification tree step-by-step, suggesting tests to perform and, when reaching a leaf establishing a diagnosis, recommending treatment, or referring the patient for higher-level care. In a randomized trial in Tanzania with 1 586 children, e-POCT improved clinical outcomes while reducing antibiotic prescription from 30% to 11%.

The World Health Organization has produced the Integrated Management of Childhood Illness (IMCI) is a strategy to help treat children's illnesses in resource-limited countries. Interestingly, IMCI contains paper-based classification trees similar to the ones produced by e-POCT. Yet unlike the IMCI classification tree, e-POCT algorithm is tailored and trained using real data, which reflects the statistical prevalence of diseases in Tanzania [173]. This training allows fitting the CART model to the target population.

After an initial successful field trial for e-POCT, the development continues in the DYNAMIC [53] Study, which would deploy a more sophisticated AI systems to continuously collect patient data and adapt the algorithm. Such a solution incurs higher carbon emissions because it requires more computing power and increases the amount of data collected and transmitted.

There seems to be a law of diminishing returns at play. On the low end of the technology spectrum, we have paper-based classification algorithms like the ones proposed in IMCI. In the middle, we have the tablet-based e-POCT algorithm, where the algorithm is static and updated manually. At the high

end of the spectrum, we have sophisticated AI algorithms that require dedicated HPC servers and continuous data collection.

Developing countries are strongly affected by climate change. Among other problems, droughts, floods, or air pollution caused by the environmental crisis negatively affect the health of children in Africa. Fabricating the tablets used for e-POCT currently involves child labor in rare-earth mines where children are exposed to pollutants. D'Acremont [41] wonders if the increased accuracy of diagnosis with the more sophisticated solutions would justify the increased carbon emissions.

A first step to guide the technical choice would require precisely quantifying the environmental impact of the different solutions by estimating the power consumption and performing a life-cycle assessment of the hardware involved. This impact should be weighed against the potential improvements in healthcare that each solution brings and how well the local clinicians and patients will accept the technology. Choosing an appropriate model involves complex social and environmental factors and requires considering all ethical factors.

8.6.2 Closing thoughts

D'Acremont study perfectly illustrates the dilemma of choosing the right complexity for a computation model and shows that the most accurate and advanced technological solution is not necessarily the better one when all factors are considered.

The field of HPC offers many choices and avenues for optimization. First, we choose the physical phenomena to model and the finesse of the discretization. Then, we select a target machine that will execute the program. Finally, we write an implementation algorithm and apply different optimizations at the level of the programming language, the compiler, or the operating system. Usually, this process is not sequential and requires back-and-forths to find a good fit between the model, the implementation, and the architecture.

Despite significant optimizations targeting the model, the software, and the hardware; the demand for computation keeps rising. Indeed, the efficiency gained is harnessed to increase the model complexity. Curbing HPC power consumption is mostly not a technical issue but a methodological one. Like D'Acremont, we need to consider how many resources we dedicate to a given computation. It is not possible to answer this question in general. The allocated resources must be weighed against the expected requirements and outcomes for the computation, which are specific and contextual questions.

Yet remaining general, we can question the tendency to always go for the more complex or accurate model. Going for the most accurate model is a shortcut that saves us from thinking about the fit between the model and our research question. Yet, I believe that asking this question before running the computation would help us reduce the size of models and increase the quality of our research. For example, in the field of image recognition, there is a tendency nowadays to go for DNNs regardless of the problem because of their flexibility. In many cases [142], thinking about the structure of the problem beforehand allows using classical computer-vision methods that are as efficient and less costly.

Simulation has changed the way we do science, creating a new epistemological tool that lies between experiment and theory, which has brought incredi-

ble scientific advances in many fields. However, faced with a shrinking energy budget, we should carefully consider when simulation is needed and not use it blindly. After all, the most environmentally friendly code is the one that is not run.

Bibliography

[1] Chadi Akel et al. "Is Source-code Isolation Viable for Performance Characterization?" In: *Parallel Processing Workshops (ICPPW), 2013 42nd International Conference on*. IEEE. 2013.

[2] A.R. Alameldeen and D.A. Wood. "Variability in architectural simulations of multi-threaded workloads." In: *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. 2003, pp. 7–18. DOI: 10.1109/HPCA.2003.1183520.

[3] A. Alexandrescu. *The D Programming Language*. Pearson Education, 2010. ISBN: 9780132654401.

[4] Dario Amodei et al. *AI and Compute*. OpenAI. May 16, 2018. URL: https://openai.com/blog/ai-and-compute/ (visited on 05/30/2022).

[5] Etienne André et al. "DUF: Dynamic Uncore Frequency scaling to reduce power consumption." In: *Concurrency and Computation: Practice and Experience* 34.3 (2022), e6580.

[6] Hartwig Anzt et al. "Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers." In: *Concurrency and Computation: Practice and Experience* (2017), e4460.

[7] El-Mehdi El-Arar et al. "Stochastic rounding variance and probabilistic bound: a new approach." In: *arXiv preprint arXiv:XXX* (2022).

[8] Ehsan K Ardestani and Jose Renau. "ESESC: A fast multicore simulator using time-based sampling." In: *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE. 2013, pp. 448–459.

[9] ARM. *ARM Cortex-A57 MPCore Processor Technical Reference Manual revision r1p1*. 2016.

[10] David H Bailey et al. "The NAS parallel benchmarks summary and preliminary results." In: *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*. IEEE. 1991, pp. 158–165.

[11] Edip Baysal. "Reverse time migration." In: *Geophysics* 48.11 (Nov. 1983), p. 1514. ISSN: 1070485X. DOI: 10.1190/1.1441434.

[12] Emily M. Bender et al. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. FAccT '21. Virtual Event, Canada: Association for Computing Machinery, 2021, pp. 610–623. ISBN: 9781450383097. DOI: 10.1145/3442188.3445922. URL: https://doi.org/10.1145/3442188.3445922.

[13] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. "A dynamic program analysis to find floating-point accuracy problems." In: *ACM SIGPLAN Notices*. Vol. 47. 6. ACM. 2012, pp. 453–462.

[14] Christian Bienia, Sanjeev Kumar, and Kai Li. "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors." In: *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE. 2008, pp. 47–56.

[15] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. "Combining Coq and Gappa for certifying floating-point programs." In: *International Conference on Intelligent Computer Mathematics*. Springer. 2009, pp. 59–74.

[16] Sylvie Boldo and Guillaume Melquiond. "Flocq: A unified library for proving floating-point algorithms in Coq." In: *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*. IEEE. 2011, pp. 243–252.

[17] Shekhar Borkar and Andrew A Chien. "The future of microprocessors." In: *Communications of the ACM* 54.5 (2011), pp. 67–77.

[18] L. Breiman et al. "CART: Classification and regression trees." In: *Wadsworth: Belmont, CA* (1983).

[19] Lawrence D Brown, T Tony Cai, and Anirban DasGupta. "Interval estimation for a binomial proportion." In: *Statistical science* (2001), pp. 101–117.

[20] Tom B Brown et al. "Language models are few-shot learners." In: *arXiv preprint arXiv:2005.14165* (2020).

[21] E. Brun et al. "A Study of the Effects and Benefits of Custom-Precision Mathematical Libraries for HPC Codes." In: *IEEE Transactions on Emerging Topics in Computing* 9.3 (2021), pp. 1467–1478. DOI: 10.1109/TETC.2021.3070422.

[22] Emeric Brun, Stéphane Chauveau, and Fausto Malvagi. "Patmos: A prototype Monte Carlo transport code to test high performance architectures." In: *Proceedings of International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering, Jeju, Korea*. 2017.

[23] Ariel N Burton and Paul HJ Kelly. "Performance prediction of paging workloads using lightweight tracing." In: *Future Generation Computer Systems* 22.7 (2006), pp. 784–793.

[24] Maurizio Capra et al. "Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead." In: *IEEE Access* 8 (2020), pp. 225134–225180.

[25] CAPS. *Codelet Finder*. 2013. URL: http://www.caps-entreprise.com/.

[26] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. "Sampled Simulation of Multi-Threaded Applications." In: *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 2–12.

[27] Trevor E Carlson et al. “BarrierPoint: Sampled Simulation of Multi-Threaded Applications.” In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014.

[28] John Cavazos et al. “Automatic performance model construction for the fast software exploration of new hardware designs.” In: *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM. 2006, pp. 24–34.

[29] Yohan Chatelain. “Outils de débogage et d’optimisation des calculs flottants dans le contexte HPC.” Theses. Université Paris-Saclay, Dec. 2019. URL: https://tel.archives-ouvertes.fr/tel-02614237.

[30] Yohan Chatelain et al. “Automatic exploration of reduced floating-point representations in iterative methods.” In: *Euro-Par 2019 Parallel Processing - 25th International Conference*. Lecture Notes in Computer Science. Springer, 2019.

[31] Yohan Chatelain et al. “VeriTracer: Context-enriched tracer for floating-point arithmetic analysis.” In: *25th IEEE Symposium on Computer Arithmetic, ARITH 2018, Amherst, MA, USA. June 25th-27th, 2018*. 2018.

[32] Françoise Chatelin. “On the general reliability of the CESTAC method.” In: *C. R. Acad.Sci. Paris* 1 (1988), pp. 851–854.

[33] Jean-Marie Chesneaux and Jean Vignes. “On the robustness of the CESTAC method.” In: *C. R. Acad.Sci. Paris* 1 (1988), pp. 855–860.

[34] Daniel Citron. “MisSPECulation: partial and misleading use of SPEC CPU2000 in computer architecture conferences.” In: *ACM SIGARCH Computer Architecture News*. Vol. 31. 2. ACM. 2003, pp. 52–61.

[35] D.A. Cohn. “Neural network exploration using optimal experiment design.” In: *Neural Networks* 9.6 (1996), pp. 1071–1083.

[36] Thomas M Conte, Mary Ann Hirsch, and Kishore N Menezes. “Reducing state loss for effective trace sampling of superscalar processors.” In: *Computer Design: VLSI in Computers and Processors, 1996. ICCD’96. Proceedings., 1996 IEEE International Conference on*. IEEE. 1996, pp. 468–477.

[37] Siegfried Cools et al. *Analysis of rounding error accumulation in Conjugate Gradients to improve the maximal attainable accuracy of pipelined CG*. Research Report RR-8849. Inria Bordeaux Sud-Ouest, Jan. 2016. URL: https://hal.inria.fr/hal-01262716.

[38] Keith D Cooper, Philip J Schielke, and Devika Subramanian. “Optimizing for reduced code space using genetic algorithms.” In: *ACM SIGPLAN Notices*. Vol. 34. 7. ACM. 1999, pp. 1–9.

[39] Matteo Croci et al. “Stochastic rounding: implementation, error analysis and applications.” In: *Royal Society Open Science* 9.3 (2022), p. 211631.

[40] K. Crombecq et al. “A Novel Hybrid Sequential Design Strategy for Global Surrogate Modeling of Computer Experiments.” In: *SIAM Journal on Scientific Computing* 33 (2011), p. 1948.

[41] Valérie D'Acremont. *Santé, Technologies, Environnement : Quels compromis éthiques ?* Université de Lausanne. Nov. 27, 2021. URL: https://youtu.be/oKcy_cY0QOw.

[42] S. Dasgupta and D. Hsu. "Hierarchical sampling for active learning." In: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, pp. 208–215.

[43] K. Datta et al. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures." In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press. 2008, pp. 1–12.

[44] Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. "Assisted verification of elementary functions using Gappa." In: *Proceedings of the 2006 ACM symposium on Applied computing*. ACM. 2006, pp. 1318–1322.

[45] Pablo De Oliveira Castro et al. "Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.1 (2015), p. 6.

[46] David Defour et al. "Custom-Precision Mathematical Library Explorations for Code Profiling and Optimization." In: *27th IEEE Symposium on Computer Arithmetic, ARITH 2020*. 2020, pp. 121–124.

[47] Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. "Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic." In: *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*. 2016, pp. 55–62. DOI: 10.1109/ARITH.2016.31. URL: http://dx.doi.org/10.1109/ARITH.2016.31.

[48] Olivier Dessombz et al. "Analysis of mechanical systems using interval computations applied to finite element methods." In: *Journal of Sound and Vibration* 239.5 (2001), pp. 949–968.

[49] U.M. Diwekar and J.R. Kalagnanam. "Efficient sampling technique for optimization under uncertainty." In: *AIChE Journal* 43.2 (1997), pp. 440–447.

[50] Lamia Djoudi et al. "Maqao: Modular assembler quality analyzer and optimizer for itanium 2." In: *The 4th Workshop on EPIC architectures and compiler technology, San Jose*. 2005.

[51] Jason Duell. "The design and implementation of Berkeley lab's linux checkpoint/restart." In: *Lawrence Berkeley National Laboratory* (2005).

[52] H. Dursun et al. "A multilevel parallelization framework for high-order stencil computations." In: *Euro-Par 2009 Parallel Processing* (2009), pp. 642–653.

[53] Dynamic Study. *Project web page*. 2022. URL: https://dynamic-study.com/.

[54] Lieven Eeckhout, John Sampson, and Brad Calder. "Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation." In: *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE. 2005, pp. 2–12.

[55] B. Efron and R. Tibshirani. "Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy." In: *Statistical science* 1.1 (1986), pp. 54–75.

[56] Ernst Eypasch et al. "Probability of adverse events that have not yet occurred: a statistical reminder." In: *BMJ* 311.7005 (1995), pp. 619–620. ISSN: 0959-8138. DOI: 10.1136/bmj.311.7005.619. eprint: http://www.bmj.com/content/311/7005/619. URL: http://www.bmj.com/content/311/7005/619.

[57] Asma Farjallah. "Preparing depth imaging applications for Exascale challenges and impacts." Theses. Université de Versailles-Saint Quentin en Yvelines, Dec. 2014. URL: https://tel.archives-ouvertes.fr/tel-01165085.

[58] Francois Févotte and Bruno Lathuilière. *LibEFT: a library implementing Error-Free transformations*. 2017. URL: https://www.github.com/ffevotte/libeft.

[59] François Févotte and Bruno Lathuilière. "VERROU: a CESTAC evaluation without recompilation." In: *International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN)*. Uppsala, Sweden, Sept. 2016.

[60] Samuel A. Figueroa. "When is Double Rounding Innocuous?" In: *SIGNUM Newsl.* 30.3 (1995), pp. 21–26. ISSN: 0163-5778. DOI: 10.1145/221332.221334. URL: https://doi.org/10.1145/221332.221334.

[61] George E Forsythe. "Reprint of a note on rounding-off errors." In: *SIAM review* 1.1 (1959), p. 66.

[62] Michael Frechtling and Philip H.W. Leong. "MCALIB: Measuring sensitivity to rounding error with Monte Carlo programming." In: *ACM Transactions on Programming Languages and Systems* 37.2 (2015), p. 5.

[63] Charlotte Freitag et al. *The climate impact of ICT: A review of estimates, trends and regulations*. 2021. DOI: 10.48550/ARXIV.2102.02622. URL: https://arxiv.org/abs/2102.02622.

[64] Charlotte Freitag et al. "The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations." In: *Patterns* 2.9 (2021), p. 100340. ISSN: 2666-3899. DOI: https://doi.org/10.1016/j.patter.2021.100340. URL: https://www.sciencedirect.com/science/article/pii/S2666389921001884.

[65] J.H. Friedman. "Greedy function approximation: a gradient boosting machine.(English summary)." In: *Ann. Statist* 29.5 (2001), pp. 1189–1232.

[66] Grigori Fursin et al. "Milepost gcc: Machine learning enabled self-tuning compiler." In: *International Journal of Parallel Programming* 39.3 (2011), pp. 296–327.

[67] Xiaofeng Gao et al. "Reducing overheads for acquiring dynamic memory traces." In: *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE. 2005, pp. 46–55.

[68] Eleftherios Garyfallidis et al. "Dipy, a library for the analysis of diffusion MRI data." In: *Frontiers in neuroinformatics* 8 (2014), p. 8.

[69] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference.” In: *arXiv preprint arXiv:2103.13630* (2021).

[70] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. “The Zonotope Abstract Domain Taylor1+.” In: *International Conference on Computer Aided Verification (CAV)*. 2009. DOI: 10.1007/978-3-642-02658-4_47.

[71] David Goldberg. “What every computer scientist should know about floating-point arithmetic.” In: *ACM computing surveys (CSUR)* 23.1 (1991), pp. 5–48.

[72] Xavier Gonze et al. “ABINIT: First-principles approach to material and nanosystem properties.” In: *Computer Physics Communications* 180.12 (2009), pp. 2582–2615.

[73] *Google Performance Tools v2.2.1*. Version v2.2.1. URL: http://code.google.com/p/gperftools.

[74] D. Gorissen et al. “A surrogate modeling and adaptive sampling toolbox for computer based design.” In: *The Journal of Machine Learning Research* 11 (2010), pp. 2051–2055.

[75] Cédric Gossart. “Rebound effects and ICT: a review of the literature.” In: *ICT innovations for sustainability* (2015), pp. 435–448.

[76] Eric Goubault and Sylvie Putot. “Static analysis of numerical algorithms.” In: *International Static Analysis Symposium*. Springer. 2006, pp. 18–34.

[77] Stef Graillat et al. “PROMISE: floating-point precision tuning with stochastic arithmetic.” In: *Proceedings of the 17th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN)*. 2016, pp. 98–99.

[78] R.B. Gramacy. “tgp: An R package for Bayesian nonstationary, semiparametric nonlinear regression and design by treed gaussian process models.” In: *Journal of Statistical Software* 19.9 (2007), p. 6.

[79] R.B. Gramacy and H.K.H. Lee. “Adaptive design and analysis of supercomputer experiments.” In: *Technometrics* 51.2 (2009), pp. 130–145.

[80] Christopher Haine et al. “Exploring and Evaluating Array Layout Restructuration for SIMDization.” In: *Proceedings of the 27th international conference on Languages and Compilers for Parallel Computing*. LCPC’14. 2014.

[81] Max Roser Hannah Ritchie and Pablo Rosado. “CO2 and Greenhouse Gas Emissions.” In: *Our World in Data* (2020). URL: https://ourworldindata.org/co2-and-other-greenhouse-gas-emissions.

[82] Eldon Hansen. “Interval arithmetic in matrix computations, Part I.” In: *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis* 2.2 (1965), pp. 308–320.

[83] John W Haskins Jr and Kevin Skadron. “Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation.” In: *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*. IEEE. 2003, pp. 195–203.

[84] Malte Henkel. “Sur la solution de Sundman du problème des trois corps.” In: *Philosophia Scientiae* 5.2 (2001), pp. 161–184.

[85] Timothy Hickey, Qun Ju, and Maarten H Van Emden. “Interval arithmetic: From principles to implementation.” In: *Journal of the ACM (JACM)* 48.5 (2001), pp. 1038–1068.

[86] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. Siam, 2002.

[87] Lorenz Hilty, Wolfgang Lohmann, and Elaine M Huang. “Sustainability and ICT-an overview of the field.” In: *Notizie di POLITEIA* 27.104 (2011), pp. 13–28.

[88] Sunpyo Hong and Hyesoon Kim. “An integrated GPU power and performance model.” In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 280–289.

[89] 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*. ACM. 2008, pp. 165–174.

[90] Kenneth Hoste and Lieven Eeckhout. “Comparing benchmarks using key microarchitecture-independent characteristics.” In: *Workload Characterization, 2006 IEEE International Symposium on*. IEEE. 2006, pp. 83–92.

[91] Kenneth Hoste and Lieven Eeckhout. “Microarchitecture-independent workload characterization.” In: *Micro, IEEE* 27.3 (2007), pp. 63–72.

[92] Kenneth Hoste et al. “Performance prediction based on inherent program similarity.” In: *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM. 2006, pp. 114–122.

[93] Jean-Michel Hupé. “Statistical inferences under the Null hypothesis: common mistakes and pitfalls in neuroimaging studies.” In: *Frontiers in neuroscience* 9 (2015), p. 18.

[94] Walter L. Hürsch and Cristina Videira Lopes. *Separation of Concerns*. Tech. rep. College of Computer Science, Northeastern University, 1995.

[95] Ilse CF Ipsen and Hua Zhou. “Probabilistic error analysis for inner products.” In: *SIAM journal on matrix analysis and applications* 41.4 (2020), pp. 1726–1741.

[96] Tomoaki Ishiyama, Kohji Yoshikawa, and Ataru Tanikawa. “High Performance Gravitational N-body Simulations on Supercomputer Fugaku.” In: *International Conference on High Performance Computing in Asia-Pacific Region*. 2022, pp. 10–17.

[97] Mark Jenkinson et al. “Fsl.” In: *Neuroimage* 62.2 (2012), pp. 782–790.

[98] M.E. Johnson, L.M. Moore, and D. Ylvisaker. “Minimax and maximin distance designs.” In: *Journal of statistical planning and inference* 26.2 (1990), pp. 131–148.

[99] Nicola Jones. “How to stop data centres from gobbling up the world’s electricity.” In: *Nature* 561.7722 (2018), pp. 163–167.

[100] William Kahan. "Numerical Linear Algebra." In: *Canadian Mathematical Bulletin*. 1966, pp. 756–801.

[101] William Kahan. "The improbability of probabilistic error analyses for numerical computations." In: *UCB Statistics Colloquium, Evans Hall edition*. 1996, p. 20.

[102] Yuriy Kashnikov et al. "Evaluating architecture and compiler design through static loop analysis." In: *High Performance Computing and Simulation (HPCS), 2013 International Conference on*. IEEE. 2013, pp. 535–544.

[103] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. Vol. 344. John Wiley & Sons, 2009.

[104] Kristina Keitel et al. "A novel electronic algorithm using host biomarker point-of-care tests for the management of febrile illnesses in Tanzanian children (e-POCT): A randomized, controlled non-inferiority trial." In: *PLOS Medicine* 14.10 (Oct. 2017). Publisher: Public Library of Science, pp. 1–29. DOI: 10.1371/journal.pmed.1002411. URL: https://doi.org/10.1371/journal.pmed.1002411.

[105] Richard E. Kessler, Mark D Hill, and David A Wood. "A comparison of trace-sampling techniques for multi-megabyte caches." In: *Computers, IEEE Transactions on* 43.6 (1994), pp. 664–675.

[106] Greg Kiar et al. *verificarlo/fuzzy: [v0.6.0] 2022/01/07*. Version v0.6.0. Jan. 2022. DOI: 10.5281/zenodo.5838384. URL: https://doi.org/10.5281/zenodo.5838384.

[107] Gregory Kiar et al. "Data augmentation through Monte Carlo arithmetic leads to more generalizable classification in connectomics." In: *arXiv preprint arXiv:2109.09649* (2021).

[108] Gregory Kiar et al. "Numerical uncertainty in analytical pipelines lead to impactful variability in brain networks." In: *PLOS ONE* 16.11 (Nov. 2021), pp. 1–16. DOI: 10.1371/journal.pone.0250755. URL: https://doi.org/10.1371/journal.pone.0250755.

[109] Toru Kisuki et al. "Iterative compilation in program optimization." In: *Proc. CPC'10 (Compilers for Parallel Computers)*. 2000, pp. 35–44.

[110] Milan Kloewer. *StochasticRounding.jl project*. 2020. URL: https://github.com/milankl/StochasticRounding.jl (visited on 12/13/2021).

[111] Donald E Knuth. "An empirical study of Fortran programs." In: *Software: Practice and Experience* 1.2 (1971), pp. 105–133.

[112] Jonathan Koomey et al. "Implications of historical trends in the electrical efficiency of computing." In: *IEEE Annals of the History of Computing* 33.3 (2010), pp. 46–54.

[113] Scott Robert Ladd. *Acovea: Analysis of compiler options via evolutionary algorithm*. 2007.

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

[115] Michael O Lam et al. "Automatically adapting programs for mixed-precision floating-point computation." In: *Proc. of the 27th International conference on supercomputing*. ACM. 2013, pp. 369–378.

[116] Jean-Luc Lamotte, Jean-Marie Chesneaux, and Fabienne Jézéquel. "CADNA_C: A version of CADNA for use with C or C++ programs." In: *Computer Physics Communications* 181.11 (2010), pp. 1925–1926.

[117] G Lartigue, U Meier, and C Bérat. "Experimental and numerical investigation of self-excited combustion oscillations in a scaled gas turbine combustor." In: *Applied thermal engineering* 24.11-12 (2004), pp. 1583–1592.

[118] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[119] Tamba S. Lebbie et al. "E-Waste in Africa: A Serious Threat to the Health of Children." In: *International Journal of Environmental Research and Public Health* 18.16 (Aug. 11, 2021), p. 8488. ISSN: 1661-7827. DOI: 10.3390/ijerph18168488.

[120] Yoon-Ju Lee and Mary Hall. "A code isolator: Isolating code fragments from large programs." In: *Languages and Compilers for High Performance Computing*. Springer, 2005, pp. 164–178.

[121] B. Li, L. Peng, and B. Ramadass. "Accurate and efficient processor performance prediction via regression tree based modeling." In: *Journal of Systems Architecture* 55.10-12 (2009), pp. 457–467.

[122] Wenbin Li. "Numerical accuracy analysis in simulations on hybrid high-performance computing systems." PhD thesis. University of Stuttgart, 2013. ISBN: 978-3-8440-1990-2. URL: http://d-nb.info/1035203235.

[123] Chunhua Liao et al. "Effective source-to-source outlining to support whole program empirical optimization." In: *Languages and Compilers for Parallel Computing*. Springer, 2010, pp. 308–322.

[124] Yongpeng Liu and Hong Zhu. "A survey of the research on power management techniques for high-performance systems." In: *Software: Practice and Experience* 40.11 (2010), pp. 943–964.

[125] N. J. Higham M. P. Connolly and Théo Mary. "Stochastic rounding and its probabilistic backward error analysis." In: 43, No. 1, pp. A566–A585 (2021). DOI: https://doi.org/10.1137/20M1334796.

[126] Mathias Malandain, Nicolas Maheu, and Vincent Moureau. "Optimization of the deflated Conjugate Gradient algorithm for the solving of elliptic equations on massively parallel machines." In: *Journal of Computational Physics* 238 (2013), pp. 32–47. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2012.11.046. URL: http://www.sciencedirect.com/science/article/pii/S0021999112007280.

[127] Gabriel Marin and John Mellor-Crummey. "Cross-architecture performance predictions for scientific applications using parameterized models." In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 32. 1. ACM. 2004, pp. 2–13.

[128] Matthieu Martel. "An Overview of Semantics for the Validation of Numerical Programs." In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. 2005. ISBN: 978-3-540-30579-8.

[129] Eric Masanet et al. "Recalibrating global data center energy-use estimates." In: *Science* 367.6481 (2020), pp. 984–986.

[130] Valérie Masson-Delmotte et al. *Climate change 2021: the physical science basis. Contribution of working group I to the sixth assessment report of the intergovernmental panel on climate change*. 2021.

[131] Abdelhafid Mazouz, Denis Barthou, et al. "Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures." In: *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE. 2011, pp. 273–279.

[132] *MB3 D6.5 Initial report on automatic region of interest extraction*. Tech. rep. Mont-Blanc project, 2016. URL: https://ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5b571f675&appId=PPGMS.

[133] AT McKay. "Distribution of the Coefficient of Variation and the Extended t Distribution." In: *Journal of the Royal Statistical Society* 95.4 (1932), pp. 695–698.

[134] Dmitry Mikushin et al. *KernelGen–the design and implementation of a next generation compiler platform for accelerating numerical models on GPUs*. Tech. rep. Technical Report 2013/02, University of Lugano, July 2013. http://www. old. inf. usi. ch/file/pub/75/tech_report2013. pdf, 2013.

[135] Ramon E Moore. *Methods and applications of interval analysis*. Vol. 2. Siam, 1979.

[136] Ramon E Moore, R Baker Kearfott, and Michael J Cloud. *Introduction to interval analysis*. Vol. 110. Siam, 2009.

[137] George C Necula et al. "CIL: Intermediate language and tools for analysis and transformation of C programs." In: *Compiler Construction*. Springer. 2002, pp. 213–228.

[138] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation." In: *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*. San Diego, California, USA, 2007.

[139] John von Neumann and Herman H. Goldstine. "Numerical inverting of matrices of high order." In: *Bulletin of the American Mathematical Society* 53 (1947), pp. 1021–1099.

[140] Anthony Nguyen et al. "3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs." In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–13. ISBN: 978-1-4244-7559-9. DOI: http://dx.doi.org/10.1109/SC.2010.2.

[141] Roy A Nicolaides. "Deflation of conjugate gradients with applications to boundary value problems." In: *SIAM Journal on Numerical Analysis* 24.2 (1987), pp. 355–365.

[142] Niall O'Mahony et al. "Deep learning vs. traditional computer vision." In: *Science and information conference*. Springer. 2019, pp. 128–144.

[143] Georg Ofenbeck et al. "Applying the roofline model." In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014, pp. 76–85. DOI: 10.1109/ISPASS.2014.6844463.

[144] Takeshi Ogita, Siegfried M Rump, and Shin'ichi Oishi. "Accurate sum and dot product." In: *SIAM Journal on Scientific Computing* 26.6 (2005), pp. 1955–1988.

[145] Pablo de Oliveira Castro. *Montblanc3 ARM64 codelet repository*. 2018. URL: https://benchmark-subsetting.github.io/cere/montblanc3-arm64-codelets/ (visited on 10/27/2021).

[146] Pablo de Oliveira Castro et al. "Adaptive Sampling for Performance Characterization of Application Kernels." In: *Concurrency and Computation: Practice and Experience* (2013). ISSN: 1532-0634. DOI: 10.1002/cpe.3097.

[147] Pablo de Oliveira Castro et al. "ASK: Adaptive Sampling Kit for Performance Characterization." In: *Euro-Par 2012 Parallel Processing - 18th International Conference*. Ed. by Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis. Vol. 7484. Lecture Notes in Computer Science. Springer, 2012, pp. 89–101. ISBN: 978-3-642-32819-0.

[148] Pablo de Oliveira Castro et al. *benchmark-subsetting/cere: CERE v0.3.1 release*. Version v0.3.1. Nov. 2018. DOI: 10.5281/zenodo.5910793. URL: https://doi.org/10.5281/zenodo.5910793.

[149] Pablo de Oliveira Castro et al. "Fine-grained Benchmark Subsetting for System Selection." In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM. 2014, p. 132.

[150] Pablo de Oliveira Castro et al. *verificarlo/verificarlo: Verificarlo v0.7.0*. Version v0.7.0. Jan. 2022. DOI: 10.5281/zenodo.5833766. URL: https://doi.org/10.5281/zenodo.5833766.

[151] James Pallister, Simon J Hollis, and Jeremy Bennett. "Identifying compiler options to minimize energy consumption for embedded platforms." In: *The Computer Journal* 58.1 (2015), pp. 95–109.

[152] Zhelong Pan and Rudolf Eigenmann. "Fast, automatic, procedure-level performance tuning." In: *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM. 2006, pp. 173–181.

[153] David Patterson et al. "Carbon Emissions and Large Neural Network Training." In: *arXiv:2104.10350 [cs]* (Apr. 23, 2021). arXiv: 2104.10350. URL: http://arxiv.org/abs/2104.10350 (visited on 06/07/2021).

[154] David Patterson et al. "Carbon Emissions and Large Neural Network Training." In: *arXiv:2104.10350 [cs]* (Apr. 23, 2021). arXiv: 2104.10350. URL: http://arxiv.org/abs/2104.10350 (visited on 06/07/2021).

[155] Roger D Peng. "Reproducible research in computational science." In: *Science* 334.6060 (2011), pp. 1226–1227.

[156] Rui Pereira et al. "Energy efficiency across programming languages: how do energy, time, and memory relate?" In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. Vancouver BC Canada: ACM, Oct. 23, 2017, pp. 256–267. ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136031. URL: https://dl.acm.org/doi/10.1145/3136014.3136031.

[157] Erez Perelman et al. "Detecting phases in parallel applications on shared memory architectures." In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE. 2006, 10–pp.

[158] Eric Petit, Guillaume Papaure, and François Bodin. "Astex: a hot path based thread extractor for distributed memory system on a chip." In: *Proceedings of Compilers for Parallel Computers workshop (CPC2006)*. 2006.

[159] Eric Petit et al. "Computing-Kernels Performance Prediction Using DataFlow Analysis and Microbenchmarking." In: *International Workshop on Compilers for Parallel Computers*. 2012.

[160] Aashish Phansalkar, Ajay Joshi, and Lizy K John. "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite." In: *ACM SIGARCH Computer Architecture News*. Vol. 35. 2. ACM. 2007, pp. 412–423.

[161] Mihail Popov et al. "PCERE: Fine-grained parallel benchmark decomposition for scalability prediction." In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE. 2015, pp. 1151–1160.

[162] Mihail Popov et al. "Piecewise holistic autotuning of parallel programs with CERE." In: *Concurrency and Computation: Practice and Experience* (2017), e4190. ISSN: 1532-0634. DOI: 10.1002/cpe.4190. URL: http://dx.doi.org/10.1002/cpe.4190.

[163] William H. Press et al. *Numerical recipes: The art of scientific computing*. Cambridge university press, 1986.

[164] Nathalie Revol and Fabrice Rouillier. "Motivations for an arbitrary precision interval arithmetic and the MPFI library." In: *Reliable computing* 11.4 (2005), pp. 275–290.

[165] G. Ridgeway. "Generalized Boosted Models: A guide to the gbm package." In: *Update* 1 (2007), p. 1.

[166] Gabriel Rivera and Chau-Wen Tseng. "Tiling optimizations for 3D scientific computations." In: *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Supercomputing '00. Washington, DC, USA: IEEE Computer Society, 2000. ISBN: 0-7803-9802-5.

[167] RTE. *Futurs Énergétiques 2050 - Principaux résultats*. Tech. rep. Oct. 2021.

[168] Cindy Rubio-González et al. "Precimonious: Tuning assistant for floating-point precision." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM. 2013, p. 27.

[169] Siegfried M Rump. "INTLAB—interval laboratory." In: *Developments in reliable computing*. Springer, 1999, pp. 77–104.

[170] Siegfried M Rump. "Verification methods: Rigorous results using floating-point arithmetic." In: *Acta Numerica* 19 (2010), pp. 287–449.

[171] Alex Sanchez-Stern et al. "Finding root causes of floating point error." In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018, pp. 256–269.

[172] D Sands. "Reimplementing llvm-gcc as a gcc plugin." In: *Third Annual LLVM Developers' Meeting*. 2009.

[173] Olga De Santis et al. "Predictive value of clinical and laboratory features for the main febrile diseases in children living in Tanzania: A prospective observational study." In: *PLOS ONE* 12.5 (May 2, 2017), e0173314. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0173314. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0173314 (visited on 05/10/2022).

[174] Gilbert Saporta. *Probabilités, analyse de données et statistiques (3eme édition)*. Editions Technip, 2011.

[175] Robert Schöne et al. "Energy Efficiency Aspects of the AMD Zen 2 Architecture." In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2021, pp. 562–571.

[176] Roy Schwartz et al. "Green AI." In: *arXiv:1907.10597 [cs, stat]* (Aug. 13, 2019). arXiv: 1907.10597. URL: http://arxiv.org/abs/1907.10597 (visited on 02/03/2021).

[177] B. Settles. "Active Learning Literature Survey." In: *Science* 10.3 (1995), pp. 237–304.

[178] Timothy Sherwood, Erez Perelman, and Brad Calder. "Basic block distribution analysis to find periodic behavior and simulation points in applications." In: *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*. IEEE. 2001, pp. 3–14.

[179] Timothy Sherwood et al. "Automatically characterizing large scale program behavior." In: *ACM SIGARCH Computer Architecture News*. Vol. 30. 5. ACM. 2002, pp. 45–57.

[180] P. Shirley et al. "Discrepancy as a quality measure for sample distributions." In: *Proceedings of Eurographics¿ 91*. Eurographics Society. 1991, pp. 183–94.

[181] P.R. Shukla et al. *Mitigation of Climate Change. Contribution of Working Group III to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change*. 2022.

[182] T.W. Simpson, D.K.J. Lin, and W. Chen. "Sampling strategies for computer experiments: design and analysis." In: *International Journal of Reliability and Applications* 2.3 (2001), pp. 209–240.

[183] Devan Sohier et al. “Confidence Intervals for Stochastic Arithmetic.” In: *ACM Transactions Mathematical Software* 47.2 (Apr. 2021). ISSN: 0098-3500. DOI: 10.1145/3432184. URL: https://doi.org/10.1145/3432184.

[184] Alexey Solovyev et al. “Rigorous estimation of floating-point round-off errors with symbolic taylor expansions.” In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41.1 (2018), pp. 1–39.

[185] M. Stein. “Large sample properties of simulations using Latin hypercube sampling.” In: *Technometrics* (1987), pp. 143–151.

[186] Mathieu Stoffel and Abdelhafid Mazouz. “Improving Power Efficiency Through Fine-Grain Performance Monitoring in HPC Clusters.” In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2018, pp. 552–561.

[187] Douglas Stott Parker. *Monte Carlo Arithmetic: exploiting randomness in floating-point arithmetic*. Tech. rep. CSD-970002. UCLA Computer Science Dept., 1997.

[188] Lisa T Su, Samuel Naffziger, and Mark Papermaster. “Multi-chip technologies to unleash computing performance gains over the next decade.” In: *2017 IEEE International Electron Devices Meeting (IEDM)*. IEEE. 2017, pp. 1–1.

[189] Jan Treibig, Georg Hager, and Gerhard Wellein. “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments.” In: *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE. 2010, pp. 207–216.

[190] Jan Treibig, Gerhard Wellein, and Georg Hager. “Efficient multicore-aware parallelization strategies for iterative stencil computations.” In: *J. Comput. Science* 2.2 (2011), pp. 130–137.

[191] Denis Trystram, Romain Couillet, and Thierry Ménissier. *Apprentissage profond et consommation énergétique : la partie immergée de l’IA-ceberg*. The Conversation. Dec. 8, 2021. URL: http://theconversation.com/apprentissage-profond-et-consommation-energetique-la-partie-immergee-de-lia-ceberg-172341 (visited on 05/30/2022).

[192] Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. “A co-phase matrix to guide simultaneous multithreading simulation.” In: *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE. 2004, pp. 45–56.

[193] Hans Vandierendonck and Koen De Bosschere. “Many benchmarks stress the same bottlenecks.” In: *Workshop on Computer Architecture Evaluation Using Commercial Workloads*. 2004.

[194] Verrou. *Project repository*. 2018. URL: https://github.com/edf-hpc/verrou.

[195] Jean Vignes. “Discrete Stochastic Arithmetic for Validating Results of Numerical Software.” In: *Numerical Algorithms* 37.1-4 (2004), pp. 377–390.

[196] Jean Vignes and Michel La Porte. “Error Analysis in Computing.” In: *Proceedings of IFP 1974*. IFP. 1974, pp. 610–614.

[197] Jean Virieux, Henri Calandra, and R.É. Plessix. "A review of the spectral, pseudo-spectral, finite-difference and finite-element modelling techniques for geophysical imaging." In: *Geophysical Prospecting* 59.5 (2011), pp. 794–813. DOI: 10.1111/j.1365-2478.2011.00967.x.

[198] Joe H. Ward. "Hierarchical Grouping to Optimize an Objective Function." In: *Journal of the American Statistical Association* 58.301 (1963), pp. 236–244. DOI: 10.1080/01621459.1963.10500845. eprint: http://www.tandfonline.com/doi/pdf/10.1080/01621459.1963.10500845. URL: http://www.tandfonline.com/doi/abs/10.1080/01621459.1963.10500845.

[199] Thomas F Wenisch et al. "SimFlex: statistical sampling of computer system simulation." In: *IEEE Micro* 26.4 (2006), pp. 18–31.

[200] Darrell Whitley. "A genetic algorithm tutorial." In: *Statistics and computing* 4.2 (1994), pp. 65–85.

[201] Egon Willighagen. *GNU R package 'genalg'*. 2013. URL: http://cran.r-project.org/web/packages/genalg/.

[202] Rafael J. Wysocki. *CPU Performance Scaling — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/v5.18/admin-guide/pm/cpufreq.html (visited on 05/23/2022).

[203] Jackson H.C. Yeung, Evangeline F.Y. Young, and Philip H.W. Leong. "A monte-carlo floating-point unit for self-validating arithmetic." In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2011, pp. 199–208.

[204] Andreas Zeller. *Why Programs Fail*. Second. Boston: Morgan Kaufmann, 2009. ISBN: 978-0-12-374515-6.