STEFANO CHERUBIN

COMPILER-ASSISTED DYNAMIC PRECISION TUNING

[June 28, 2019 at 16:11 – XXXI cycle]

COMPILER-ASSISTED DYNAMIC PRECISION TUNING

PHD CANDIDATE: STEFANO CHERUBIN
SUPERVISOR: PROF. GIOVANNI AGOSTA

PhD thesis in Information Technology

Dipartimento di Elettronica, Informazione e Bioingegneria

Politecnico di Milano

June 2019 – XXXI cycle

Dedicated to the loving memory of my lost soul.
1989 – 2019

ABSTRACT

Given the current technology, approximating real numbers with finite-precision is unavoidable. Determining which finite-precision representation to exploit for each variable in the program is a difficult task. To face this problem, several precision mix solutions have been proposed so far in the state-of-the-art. However, the best precision mix configuration may vary at runtime along with input data.

In this thesis we aim at suggesting two effective approaches to solve the precision tuning problem. The first approach follows the static precision tuning paradigm, i.e. it generates a single mixed precision version from the original code, which is designed to be used in place of the original version. We later allow the possibility of changing the input conditions that may affect the best precision mix configuration. To solve this problem we propose a novel approach and a new toolchain that automitizes a large portion of this process. We present each component of the toolchain, and we provide guidelines to use them properly. We refer to this second approach as *dynamic precision tuning*.

We evaluate the static and the dynamic precision tuning solutions on a set of high performance computing and approximate computing benchmarks. We show how the dynamic precision tuning toolchain can be used – under certain conditions – also for static precision tuning. Our second toolchain is capable of achieving good results in terms of performance gain while maintaining acceptable precision loss threshold. In the future we aim at further improving this toolchain to extend its applicability to other use cases. Additionally, we highlight which improvements on the current toolchain may provide greater benefits on the quality of the output.

SOMMARIO

La precisione finita è una approssimazione dei numeri reali inevitabile con la attuale tecnologia. Determinare quale rappresentazione a precisione finita sia meglio sfruttare per ciascuna variabile nel programma è un compito difficile. Per risolvere questo problema, diverse soluzioni per determinare un adeguato mix di precisione sono state proposte finora nello stato dell'arte. Tuttavia, la miglior configurazione di mix di precisione può variare durante l'esecuzione assieme ai dati in ingresso.

In questa tesi proponiamo due approcci efficaci per risolvere il problema di determinare miglior il mix di precisione. Il primo approccio riflette il paradigma di determinazione statica della precisione, ossia genera dal codice originale un singolo mix di precisione, il quale dovrà poi essere usato al posto della versione originale. Successivamente ammettiamo la possibilità di avere cambiamenti delle condizioni di ingresso che possano inficiare la miglior configurazione di mix di precisione. Per risolver questo problema proponiamo un approccio innovativo e una nuova toolchain per automatizzare larga parte di questo processo. Noi introduciamo ciascun componente della toolchain, e forniamo linee guida per usarla in modo appropriato. Chiamiamo questo secondo approccio *dynamic precision tuning*.

Valutiamo le soluzioni di static e dynamic precision tuning su un insieme di benchmark per high performance computing e calcolo approssimato. Mostriamo come la toolchain per dynamic precision tuning può essere usata, sotto certe condizioni, anche per static precision tuning. La nostra seconda toolchain è capace di raggiungere buoni risultati in termini di miglioramento delle performance alla stessa soglia di perdita accettabile di precisione. In futuro auspichiamo di migliorare ulteriormente questa toolchain per estenderne la applicabilità ad altri casi d'uso. Inoltre, evidenziamo quali miglioramenti sulla attuale toolchain possono garantire i maggiori benefici sulla qualità del risultato.

RELEVANT PUBLICATIONS

We present an handout summary of the contribution of this PhD thesis over the state-of-the-art. We summarize the works we authored that are later mentioned in this thesis.

- **Tools for reduced precision computation: a survey**
  S. Cherubin, G. Agosta
  Currently under review.
  In this work we survey the state-of-the-art of precision tuning techniques and tools.

- **Implications of Reduced Precision in HPC: Performance, Energy and Error**
  S. Cherubin, G. Agosta, I. Lasri, E. Rohou, O. Sentieys
  International Conference on Parallel Computing (ParCo).
  Bologna, Italy. September 2017 [22]
  In this work we re-purpose a source-to-source compiler to perform static precision tuning. In particular, we evaluate the effect of using fixed point data types on HPC-like computer architectures.
  This work is the result of a collaboration with INRIA Rennes.

- **Embedded Operating System Optimization through Floating to Fixed Point Compiler Transformation**
  D. Cattaneo, A. Di Bello, S. Cherubin, F. Terraneo, G. Agosta
  21st Euromicro Conference on Digital System Design (DSD).
  Prague, Czech Republic. August 2018. [17]
  In this work we experiment a compiler-level approach to floating point to fixed point conversion. In particular, we compare the results achieved by using a dedicated compiler plugin for the conversion, with respect to those achieved by performing such conversion manually.

- **libVersioningCompiler: An easy-to-use library for dynamic generation and invocation of multiple code versions**
  S. Cherubin, G. Agosta
  SoftwareX, Volume 7
  January – June 2018 [21]
  In this work we introduce a C++ library to ease the exploitation of the dynamic compilation paradigm. In particular, we test this library on the continous optimization use case, and on the optimization of a legacy code base.

- **Continuous Program Optimization via Advanced Dynamic Compilation Techniques**
  M. Festa, N. Gervasoni, S. Cherubin, G. Agosta
  Proceedings of the 10th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 8th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms.
  Valencia, Spain. January 2019. [47]
  In this work we extend the dynamic compilation library to implement the Just-In-Time compilation paradigm. We evaluate the benefits of performing Just-In-Time compilation with respect to standard dynamic compilation approach, and with respect to the use of a state-of-the-art C++ interpreter.

- **TAFFO: Tuning Assistant for Floating point to Fixed point Optimization**
  S. Cherubin, D. Cattaneo, A. Di Bello, M. Chiari, G. Agosta
  IEEE Embedded Systems Letters.
  April 2019. [23]
  In this work we propose a framework to guide the user in the task of static precision tuning. It is based on the conversion from floating to fixed point, which is performed on the intermediate representation of the program within the compiler. This framework is packaged as a set of compiler plugins.

- **Dynamic Precision Autotuning with TAFFO**
  S. Cherubin, D. Cattaneo, A. Di Bello, M. Chiari, G. Agosta
  Currently under review.
  In this work we introduce the concept of dynamic precision tuning as a particular case of continous program optimization. We describe a methodology based on the TAFFO framework, and on the dynamic compilation library libVersioningCompiler.

In an ideal world,
programming languages and compilers are boring.
They do what users expects.
They exhibit the same behavior with and without optimizations,
at all optimizations levels, on all hardware.
"Boring", however, is surprisingly difficult to achieve [...].

— Nötzli, Andres and Brown, Fraser [112]

ACKNOWLEDGMENTS

First of all, we thank our advisor prof. Agosta. Not only he represented a wise guide during our doctoral studies, he was – and he currently is – a good friend.

Creating a productive and stimulating environment is far from something that can be defined as an easy task, we thus want to thank the whole research group we spent our life in during the last years. Uncountable times we spent nights over days fighting together for the same cause, or the same deadlines.

Last and least of this list we want to thanks our institution. Odd and peculiar, with all its services and inefficiencies, clever flexibility and meaningless restrictions, it allowed us to improve ourselves from a professional and from a personal point of view.

There is one more category of people we did not explicitly thank. Underlining their importance in our life is not a thing we can do in a few lines of this section, we prefer to omit the list of people who represented more than something to us because they already know how much we care about them and they know we prefer to manifest our gratitude in person.

That's all for the acknowledgments, we hope the reader will enjoy the content of this thesis. Thanks to anyone who will find useful our work and will continue pushing our research forward. In the end, this is the reason why we are doing this.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Since the early days of automatic computing machines, the problem of representing infinite real numbers in a limited memory location represents one of the most important scientific challenges. Turing defines the concept of *computable number* in 1936 [155]. Long before the advent of modern computers, Church and Turing demonstrated that only certain classes – and not the whole set – of real numbers can be computed by a machine [28]. Thus, given the current technology, approximating real numbers with finite-precision is unavoidable. This approximation – as any approximation – introduces errors in the computation.

The effect of this error can be mitigated by increasing the precision of the computation. The precision level represents a trade-off between error and memory space. However, the error could be avoided only by using an infinite memory, which is not possible.

The problem of computing using finite precision operand is not limited to the precision of the result, which has to be finite as well. Indeed, the error introduced on the input eventually propagates and magnifies throughout the automatic computation. Nowadays the automatic computing systems scaled up to a point where keeping track of the error propagation is unfeasible without dedicated tools.

The research branch of computer science that deals with the trade-off between error and other performance metrics is named *Approximate Computing* [162]. It proposes several techniques whose focus range from the hardware level to the software level of the automatic computing system.

In our work we focus on one specific approximate computing technique: precision tuning. With precision tuning we aim at leveraging the trade off between the error introduced in the computation and the performance of the application by changing the representations of the real numbers in the application. This technique leads towards two different goals: error minimization, and performance maximization. We present two examples to highlight the importance of each of these goals.

On large scale automatic computing systems, the numerical sensitivity magnifies. As a consequence, during recent years, an increasing share of applications which embraced parallelism to scale up in performance, started to focus also on the numerical precision problem [141]. Such applications must guarantee that an increased level of parallelization does not increase the error in the computation. We can use precision tuning to find which is the most efficient number represen-

tation that can satisfy a given precision requirement. In this way it is possible to use large precision data types only in a restricted portion of the application.

Another point of view to look at the error-performance trade-off originates from the request for performance improvements on error-tolerant applications. Large classes of applications – such as media streaming and processing applications – can trade off unnecessary precision to improve performance metrics. The precision tuning approach uncovers optimization opportunities by exploiting lower precision data types with respect to the original implementation in different portions of the application.

These two goals are actually two sides of the same coin. The user specifies the requirement, and the precision tuning provides a mix of data types that satisfies the requirement while minimizing a given cost function. This cost function represents the degradation of the performance metric we want to improve by trading off some precision during the computation. Traditionally, users aim at improving the throughput of their applications. Thus, the most used cost function is the execution time. However, other systems may have different requirements. Indeed, in the internet of things domain the device battery lifetime is often a critical aspect that developers have to take into account. In such systems it is more common to use the system energy consumption as cost function. Hardware developers, instead, push to minimize the area occupied by the computational unit on their circuits. All the methodological considerations on precision tuning do not depend on the cost function we aim at minimizing.

Precision tuning, as many other approximate computing techniques, is a delicate task that may change the semantics of the program. Thus, developers traditionally apply it manually to always keep the control over the approximation level. This approach does not scale well with the complexity of the program, and neither it does with respect to the size of the program. Automatic tools can ease the process. However, due to the implications of this approximation – which are largely application dependent – no holistic tool can effectively apply precision tuning without any human supervision or input. According to the nature and to the form of the information that the user can provide to the tool, different tools and frameworks have been proposed to automatize one or more stages of the precision tuning process.

PROBLEM DEFINITION Precision tuning manipulates the original application by replacing the original representation of real numbers with a different one. This process typically moves the needle of the trade off towards performance improvement or towards quality improvement. There exists corner cases when a change in the representation of real numbers causes both a performance improvement and a quality improvement. The opposite case, i.e. changes that cause a

performance and quality degradation, is more common. The problem of precision tuning entails the problem of defining metrics and methodologies to measure the quality of the application output, and to measure the application performance. As the name may suggest, tuning is about finding the best level for a given knob. The definition of which knobs – real numbers in the program – need to be tuned and which levels – representation methods or data types – are allowed for each knob are challenges that each precision tuning tool has to address.

SCOPE OF THE THESIS In our work we consider two different approaches to precision tuning: the static and the dynamic ones. On one hand, **static precision tuning** entails the ahead of time analysis and verification of the program to obtain a single mixed precision version. On the other hand, **dynamic precision tuning** involves the continuous adaptation of the mixed precision version at runtime, thus generating multiple mixed precision versions. These two approaches share a large common ground of challenges that have to be addressed. We analyze such challenges in this thesis along with the solutions that have been proposed in the literature so far. We discuss strong and weak points for each of the proposed solutions. In this thesis we aim at providing an overview of the differences between the well-known static precision tuning approach and the dynamic precision tuning one, which represents the main contribution of this thesis. In particular, we compare such approaches in terms of methodological requirements and proposed toolchains.

ORGANIZATION OF THE THESIS Precision tuning is a complex process which requires a set of code analysis and code transformation tools. We introduce a selection of theoretical background concepts in Chapter 2. Several proposals have been presented in the state-of-the-art to improve the automation of such process. We present a survey of the state-of-the-art in Chapter 3. We subsequently propose effective solutions to perform precision tuning in Chapter 4. We distinguish between two toolchain structures to perform of precision tuning: static (Section 4.1) and dynamic (Section 4.2) ones. Most of the work in the state-of-the-art focuses on static precision tuning. We present a dynamic precision tuning solution by combining different software components we developed. Therefore, we first introduce each software component, and we later discuss their composition to implement dynamic precision tuning. We show in Chapter 5 the experimental evaluation and validation of the tools we developed, and those of the precision tuning solutions we introduced. Finally, in Chapter 6 we draw some conclusions.

2

THEORETICAL BACKGROUND

The content of this Chapter has been submitted for publication in the journal ACM Computing Surveys *and it is currently under revision.*

Numerical representation methods were designed to have a meaningful subset of existing numbers mapped in a finite memory location. These methods introduce a trade-off between the required number of bits and the cardinality of the subset of numbers which can be exactly represented. The remaining numbers can either be ignored or approximated. Regardless of the representation method, in fact, only 2^n distinct numbers can be exactly represented, where n is the number of bits in the operands.

Given a fixed n available in the machine, integer numbers are usually represented exactly in the range $[0; 2^n - 1]$ or $[-2^{n-1}; 2^{n-1} - 1]$ whereas the rest of the numbers are not considered. When the result of a computation that uses such representation exceeds those ranges, the representation of the result is not defined. This is the case of the overflow and underflow problem.

Due to the nature of real numbers, every representation method is susceptible not only of overflow and of underflow, but also of round-off errors. In the case of real numbers, given $r_1, r_2 \in \mathrm{R}$ such that $r_1 < r_2$, there exists infinite other numbers r' such that $r_1 < r' < r_2$. Since it is impossible to represent infinite different numbers using a finite memory, it is common practice in number representation methods to approximate a number r' that does not have an exact representation with the representation of another real number. This alternative representation is defined by the representation method and it is chosen to minimize a cost function, which usually depends on a combination of the approximation error and of the implementation cost.

A representation method can be seen as a function $F(r)$ that maps a real value r to its finite-precision representation. This function is defined for all the values $r \in \mathrm{R}$ within the range bounds of the representation method, and it has a co-domain cardinality at most equal to 2^n.

In general purpose computing, the most popular way to represent real numbers is via the floating point IEEE-754 standard [71]. With this standard, a number is described by a sign, a mantissa, and an exponent. The IEEE-754 standard provides partitions of the available bits among mantissa and exponent – depending on the total number of available bits – thus implementing a trade-off between precision and range of the representation.

Whilst IEEE-754 is extremely effective and popular, its implementation is much more complex than that of integer arithmetic operations.

Indeed, there are at least two dimensions across which it is possible to select the best trade-off among range, precision, and performance. First, every representation, IEEE-754 included, is available in multiple sizes. Smaller sized representations have limited range and precision, but may, under appropriate implementations, provide opportunities for increased throughput – e.g., by allowing vector operations on small size numbers instead of scalar, large size number operations – or for reduced energy consumption – by selecting smaller, and therefore less energy-hungry, hardware implementations. Second, by using less complex representations – such as fixed point – it is possible to leverage less complex hardware units, possibly leading to reductions in energy consumption.

These trade-offs have been traditionally exploited in embedded systems, where the emphasis is on low-energy devices – e.g., for long-running, battery-powered sensors. As a consequence, researchers proposed design methodologies where algorithms are initially designed with high-level tools such as Matlab, using precise floating point representations, and they are later manually re-written using fixed point representations [93]. A large amount of domain knowledge is required in this case, since the small integers' limitations in range and precision used in such systems force the developer to finely tune the representation, possibly readjusting it in different phases of the algorithm.

More recently, customized precision [137] has emerged as a promising approach to improve power/performance trade-offs. Customized precision originates from the fact that many applications can tolerate some loss in quality during computation, as in the case of media processing (audio, video and image), data mining, machine learning, etc. Error-tolerating applications are increasingly common in the emerging field of real-time HPC [163].

Given the increased interest in customized precision techniques, and given the error-prone and time-consuming nature of the adaptation and tuning processes, several techniques, and associated tools, have been proposed to help developers select the most appropriate data representation (See, e.g. *FRIDGE* [74], *Precimonious* [133], *CRAFT* [85]). This way they are now able to automatically or semi-automatically adapt an original code, usually developed using a high-precision data type – e.g., IEEE-754 double precision floating point – to the selected lower-precision type. Such tools may target different hardware solutions, ranging from embedded microcontrollers to reconfigurable hardware to high-performance accelerators.

2.1 DATA TYPES

To overcome the inaccuracy problem, which is intrinsic to finite precision computation, several proposals have been made. The most

common one consists in multiple-word data types [51], which allow an arbitrary number of machine words to be used for the storage of a single value. This technique is named *Multiple Precision* or *Arbitrary Precision* [8].

Arbitrary precision implementations such as *ARPREC* [8], *MPFR* [51], *MPFUN2015* [7] come with explicit control over the desired accuracy. This poses no upper bound to the accuracy employed in the computation. Therefore, there is no numerical reference to measure the error introduced by using a reduced precision data type. On the contrary, symbolic analysis [144] can derive error functions without a numerical reference, but they are difficult to be interpreted by programmers. Hence, it is common practice to agree on a data type to be the reference for benchmarking. By *reduced precision computation* we refer to the use of data types that have less precision than the reference data type, which is usually provided by the algorithm designer. The set of data types considered in reduced precision computation is heterogeneous and diverse from one tool to another. Below we shortly introduce theoretical background and naming conventions regarding data types.

2.1.1 *Fixed point Data Types*

A fixed point number is a tuple $< sign, integer, fractional >$ that represents a real number defined in Equation 2.1.

$$(-1)^{sign} * integer.fractional \tag{2.1}$$

A fixed number of digits is assigned to *sign*, *integer*, and *fractional* within the data type format. As integer data types can be signed or unsigned, in fixed point numbers the *sign* field can be omitted or can be part of the representation, such as two's complement. This is the case of unsigned fixed point numbers, which represent the absolute value of the real number defined in Equation 2.1.

The majority of hardware implementations treat fixed point numbers as integer numbers with a logical – not physical – partitioning between integer and fractional part. Such bit partitioning is defined at compile time. Figure 2.1 illustrates two examples of bit partitioning.

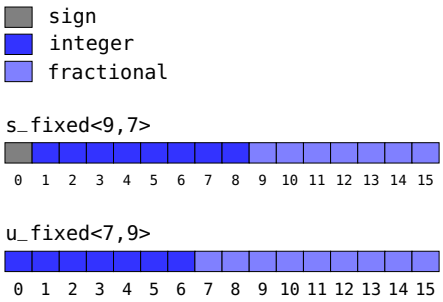


Figure 2.1: Bit partitioning for signed and unsigned fixed point data types.

An evolution of the fixed point representation as presented in Equation 2.1 is the dynamic fixed point representation. Dynamic fixed point representations have a fixed number of digits for the whole number – like the plain fixed point representation – however, they allow to move the point in order to implement a trade-off range/precision.

As the representation of fixed point data types resembles the representation of integer numbers, it is common to have architectures reuse the implementation of the integer arithmetic unit also for fixed point operations. In these cases, there is no hardware constraint that forces a clear separation between *integer* and *fractional*. Thus, the support for dynamic fixed point data types comes at no cost. From now onward we consider only dynamic fixed point representations.

The type cast among fixed point data types aims at aligning the position of the point in the representation via a signed shift operation. This move is also known as *scaling*. The most common arithmetic operations – e.g. addition, subtraction, multiplication, and division – are implemented by using the corresponding operations for a standard integer data type. Indeed, only in the case of multiplication and division additional scaling is required after the operation.

One of the most basic tools in support of programmers working with fixed point representations is the automatic insertion of scaling operations. Usually this feature is implemented via the definition of a C++ class to represent the fixed point data type. Hence, proper C++ operators can be defined with a scaling operation for that class at every operation and type cast. Nowadays the automatic scaling, which was initially described in [75], is a key feature in all of the relevant tools in the literature. Although tools may differ in their implementation, the perception of data type abstraction is guaranteed for the users. Thus, the programmer can consider fixed point representations implemented over integers representations as data types for real numbers disregarding implementation details and architectural differences.

2.1.2 *Floating point Data Types*

A floating point number is a tuple $< sign, mantissa, exponent >$ that represents a real number defined in Equation 2.2.

$$(-1)^{sign} * mantissa * b^{exponent} \tag{2.2}$$

The base b of the representation is an implicit constant of the number system. In our work we focus on floating point representations where the b is equal to 2, and the *mantissa* (or *significand*), the *sign* and the *exponent* are represented in the binary system. A fixed number of digits is assigned to *sign*, *mantissa*, and *exponent* within the data type format.

Figure 2.2 illustrates the bit partitioning for the most used floating point data types. The initial IEEE standard for floating-point arith-

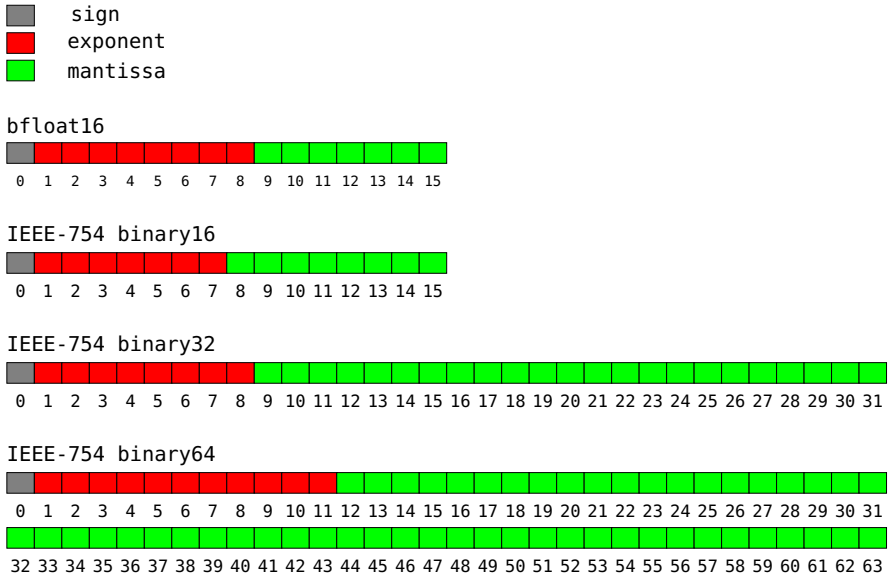


Figure 2.2: Bit partitioning for the `bfloat16`, `binary16`, `binary32`, and `binary64` floating point data types.

metic [70] defines the format for `single` and `double` floating point numbers using 32 and 64 bit respectively. The last revision of the IEEE-754 standard [71] introduces a 128 bit format named `binary128` as **basic format** – also known as floating point `quadruple` precision – while it renames the previous format as `binary32` and `binary64`. As hardware implementations of floating point units working on the basis of the `binary128` standard are costly in terms of area and complexity, they have been implemented only on very specific hardware platforms – e.g. [87, 91]. Therefore, the 128 bit arithmetic operations are often emulated via 64 bit arithmetic units and they do not represent a fair baseline from the energy and performance point of view.

Other data types which provide an improved accuracy over the `binary64` data type have been proposed. The most relevant one is certainly the `double extended` floating point precision, which is a data format resembling the IEEE-754 `binary64` data type. It features the same number of exponent bits and a larger number of mantissa bits for a total of 80 bit width. Hardware units implementing this format are also known as *x87* architectures [72]. It was discussed in [105] that the precision of computation declared as `double extended` actually depends on the compiler register allocation. Indeed, even when compilers use native 80 bit registers, they force a downcast of the values to spill from 80 bit registers into 64 bit memory locations when all the 80 bit registers are saturated. The evolution of computer architectures towards the exploitation of the SIMD parallel paradigm did not favor the exploitation of this data type, as its width is not an integer power of 2. The use of the *x87* unit is deprecated in the more common *x86_64* architecture.

Although the revision of the IEEE-754 standard defines only five basic floating point formats – `binary32`, `binary64`, `binary128`, `decimal64`, and `decimal128` –, it provides guidelines and definitions for other floating point binary interchange formats. Indeed, it suggests also a `binary16` interchange format – also known as `half` precision. This format, which was originally intended for data storage only, is nowadays investigated for arithmetic computation [121, 127]. Since GPU hardware manufacturers began to provide native support for `binary16` floating point arithmetic computations, tools to explore the benefits of such data format started to appear. We classify such tools separately from those working on IEEE-754 basic data types.

Another notable exception from the IEEE-754 standard is the recently-introduced `bfloat16` floating point data type [6]. This is heavily inspired by the `binary32` format. Indeed, it is derived by truncation: it uses only 7 mantissa bits instead of the 23 mantissa bits of the IEEE-754 `binary32`. This representation was originally designed to be used in deep-learning dedicated hardware. It allows fast type casting from and to the original `binary32` format.

2.2 ERRORS

Since we rely on finite-precision representations of numbers, we can incur in several types of error. We briefly discuss the most relevant ones.

ARITHMETIC OVERFLOW The arithmetic overflow is an error related with integer numbers. In particular, it occurs whenever an instruction is attempting to save to a memory location an integer value that exceeds the range of values that can be represented with the available number of digits in the given memory location. The result of the arithmetic operation that generates an overflow error is implementation-dependent. In the most common case, only the least significant bits of the number are stored in the destination memory location. This is the case of a *wrapping overflow*. The most notable alternative is the *saturating overflow*, which entails the saturation of the exceeding value to the boundary value of the range of values that can be stored in the destination memory location.

REPRESENTATION MISMATCH The origin data type can allow symbols – other than real numbers – to be represented in its number system without a counterpart in the reduced precision data type. In this situation, whenever we replace the origin data type with the reduced precision one, we talk about representation mismatch. This is the case of $\pm\infty$, and *NaN* that have a representation in the IEEE-754 floating point standard data types. There is instead no equivalent in most of the fixed point representations.

ROUND-OFF The round off error derives from the truncation we are forced to apply to represent a real number using finite precision. The round off error depends on the data type we use. Given two elements $F(r_1), F(r_2)$ of a representation of a subset R on n bits, such that there is no other $F(r')$ with $F(r_1) < F(r') < F(r_2)$ in the representation, the distance $|F(r_2) - F(r_1)|$ gives a measure of the approximation with which numbers in the range $[r_1, r_2]$ can be represented. From this idea, the function $ulp(x)$ – *Unit in the Last Place* – can be defined for every point representation method and it can be used as an error measure. In the case of floating point numbers, the function $ulp(x)$ is defined as the distance between the closest straddling floating point numbers a and b such that $a \leq x \leq b$ and $a \neq b$, assuming that the exponent range is not upper bounded [68]. We can generalize the ulp definition by considering the Goldberg's definition [59] of the function $ulp(x)$ for a representation method with radix β, precision p extended to reals as in Equation 2.3 [109].

$$\text{If } |x| \in [\beta^e, \beta^{e+1}), \text{ then } ulp(x) = \beta^{\max(e, e_{min})-p+1} \tag{2.3}$$

We call machine epsilon $\epsilon_M = ulp(1)$ of a given data type the smallest distance between 1 and its successor that can be represented with the given data type. Fixed point representations have a fixed round off error $\epsilon_{r,fixed} = \epsilon_{M,fixed}$ which is equal to their own machine epsilon. On the contrary, floating point representations have a variable round off error $ulp_{float}(x)$ which depends also on the value x itself, and not only on the data type representation.

The trade-off introduced by reduced precision computation can be applied to a large variety of use cases. In all these cases, the quality degradation of the output must stay within given bounds and it is not an easy task to provide such guarantees. We consider as quality degradation introduced by the reduced precision computation the difference $\epsilon = |output_{orig} - output_{mix}|$ between the output of the original version of the program $output_{orig}$ and the output of the reduced precision version of the program $output_{mix}$. Although there are approaches, such as the code rewriting implemented by *Xfp* [36], that aim at minimizing the representation error on the output without changing the data type, the most challenging aspects of precision tuning involve the possibility to exploit data types which are different from the original version of the program. Indeed, the problem of finding safe bounds for quality degradation entails the analysis of the propagation of the initial round off errors across all the intermediate values of the program. Each intermediate value of the program may have a different data type, which has its own machine epsilon and round off error formulation.

To this end, it is possible to compare the output of one or more executions of the reduced precision version of the code against the output of the original version. Although this profiling approach gives

a precise quantification of the error, it strictly depends on the input test set.

To ensure limits on the error bounds, it is possible to perform a static data flow analysis on the source code. Such analysis, which provides guarantees on the error bounds for every possible input, have also limits. In particular, it is impossible to obtain numerical error bounds on iterative and recursive algorithms whose bounds depend on the input data.

2.3 THE PROCESS OF REDUCED PRECISION COMPUTATION

In this section we outline the main stages that tools addressing reduced precision computation need to tackle in order to provide an effective support. Reduced precision computation is not simply about changing the data type in the source code with the *Find-and-Replace* button of any text editor. It is a more complex technique that presents several challenges, from the architectural to the algorithmic ones.

The process of precision tuning consists in a finite number of steps, each one representing a problem that has been addressed in the state-of-the-art:

1. Identify potential precision/performance trade-off opportunities in the given application.
2. Understand the boundaries of this trade-off.
3. Perform the code patching to replace the original code region with an equivalent one that exploits a reduced precision data type.
4. Verify the quality degradation.
5. Evaluate the overhead due to the introduction of type casting instructions.

SCOPE The noise tolerance that allows us to apply approximate computing techniques is a property of the individual algorithm, and it is not uniformly distributed among the whole program. Different regions of the program typically implement distinct algorithms. The Pareto principle applied to computer science states that only a restricted portion of the application is performance critical. Therefore, the identification of potential trade-offs should focus on those critical code regions, whilst the rest of the application code can be ignored.

ANALYSIS Once the code regions of interests have been identified, the program has to be analyzed to characterize its error sensitiveness with respect to changes in the data type. This analysis usually entails the computation or the estimation of the dynamic range of each

variable in the program. From this piece of information it is possible to allocate a data type to each variable such that the performance of the code is improved and the introduced error remains within a given threshold.

CODE MANIPULATION The code patching is the precision tuning step in which the actual reduced precision version is generated. This aspect is mostly considered an implementation detail and several research tools do not invest great effort in it. However, the code manipulation approach determines the portability of the tool to real-world scenarios. This challenge can be addressed with different solutions, which are better suited for different use cases. Indeed, any tool for precision reduction needs to be aware of both the source language and the target platform of the compilation process it is part of. On the source language side, this requirement is mostly due to the need to cope with best practices in different domains, such as embedded systems or high performance computing. On the target side, different platforms implement the arithmetic unit using different architectures. Thus, each of them provides different trade-off opportunities. As a result, the tool needs to be well aware of the properties of each platform, or able to deduce them in some way.

ERROR BOUNDS The verification of the error introduced by the use of a reduced precision data type comes unavoidably after the allocation of the data types for the code regions of interest. The goal of this step is to validate the precision mix and to accept it or to reject it with a given threshold, which is usually associated to an output value. This threshold can be expressed in terms of ulp, absolute numeric value, or relative value. In case of rejection it is possible to – partially or fully – change the precision mix in the regions of interest and evaluate it again.

TYPE CASTING OVERHEAD The quest for reduced precision computation often entails the search for the smaller data type to use for each region of the program. However, the change of the data type in a code region requires to convert the data from the original precision version to the new data type before the reduced-precision code region. The same process is needed for restoring the original precision after that code region. These conversions insert a set of type casting operations, which result in additional overhead for the reduced precision version of the application. In the case of large data to be processed, this overhead is not negligible. The performance improvements due to the precision/performance trade-off have to overtake the cost of data conversion. The use of a very heterogeneous set of data types within the same region of code increases the overhead. For this reason, a uniform data type selection reduces the impact of this overhead.

The estimation of the type casting overhead during the data type allocation is still an open challenge that has been poorly addressed in the literature.

3

STATE-OF-THE-ART

The content of this Section has been submitted for publication in the journal ACM Computing Surveys *and it is currently under revision.*

Researchers with different scientific backgrounds and with different objectives contributed to the literature on reduced precision computation. Even though they proposed tools and approaches to solve some of the problems described in Section 2.3, their contribution was not always focused on the reduced precision computation itself. Researchers often use this technique as a mean to reach a more complex goal. In this chapter we briefly describe the most relevant works from the reduced precision perspective.

3.1 SCOPE OF THE TOOL

It is important to identify which portions of the program can be approximated and which ones should keep using the original precision level. This problem can be addressed with different approaches.

The most trivial approach consists in ignoring the concept of code regions. The tool analyzes the whole application code and provides a suggested data type allocation for each value in the program. This approach allows the tool to find the global optimum solution. It is the case of formal verification tools – such as *GAPPA* [38] – and holistic frameworks – such as the one implemented by *PetaBricks* [4]. Besides them, many other tools adopt the whole program analysis approach. Examples are *Autoscaler for C* [79], the approach proposed by Menard [99], *GAPPA* [38], *PROMISE* [62, 63], *SHVAL* [84], the binary mode of *CRAFT* [85], and the method proposed by Rojek [129]. However, the whole program analysis approach is likely to lead to long processing time due to the exponential amount of possible precision solutions. Another weak point of this approach is the difficulty that static analysis techniques face when they have to process a large code base. In fact, the static conservative approach is likely to diverge or to provide over-approximations of the results, which are too large to be actually useful on large programs.

On the contrary, the annotation-based approach asks the end-user to manually specify which regions of the code have to be considered for precision tuning and which ones should not. These annotations can be implemented in several ways. *Precimonious* [133] uses external XML description files to declare which variables in the source code should be explored and which data types have to be investigated. Other tools define a custom annotation language that mixes with the original programming language. In particular, *ID.Fix* [22] relies on as scoped `pragma` declarations in the source file, *FRIDGE* [74], *CRAFT* [85] (using

variable mode), *FloPoCo* [44], and *FlexFloat* [149] implement custom C++-like data types. In particular, these tools act only on the variables declared with the tool-specific data types. *Rosa* [35] and *Daisy* [32] use a custom contract-based programming model. The annotation-based approach forces the tool to investigate only the critical region(s) of the program and thus it allows to save processing time, which can be spent on more accurate analysis. On one hand, the effort of the user in manually annotating the source code is paid back by improved scalability with respect to the size of the code base. On the other hand, this approach does not fit use cases where the size of the critical section is particularly large. Additionally, a fine-grained annotation process requires the end-user to have a deep domain knowledge of the application.

A higher level approach to the definition of the scope of the tool consist in the selection of the computational-intensive kernel function. This approach is typically adopted by tools that operate on hardware-heterogeneity-aware programming languages, such as OpenCL. The tool mentioned in [111] relies on an already existing domain specific language to describe which is the OpenCL kernel that has to be analyzed. Although the main focus of the framework mentioned in [3] are OpenCL kernels too, they process the whole computational kernel and do not expose any hook to the end-user.

Another approach which requires user intervention is the expression-based approach. Tools of such kind are designed to work on individual expression. The end-user is supposed to manually extract the expressions they are interested in from the program, have them processed by the tool, and take care of combining the results. These are typically analysis tools that do not apply any code manipulation. This is the case of *Xfp* [36], which is a tool for floating point to fixed point conversion. Other examples are *FPTuner* [25] – which is an analysis tool that provides precision allocation for an input floating point expression – and *PRECiSA* [154], which is an error estimation tool.

An ultimately automatic approach consists in performing a profile run of the application to identify the hot code regions, which may especially benefit by the performance/precision trade-off. *HiFPTuner* [65] combines a static data flow analysis with a dynamic profiling on the source code. It builds a hierarchical representation of the data-dependencies in the program. In this graph the initial static analysis creates the hierarchical structure whereas the dynamic profiling highlights the hottest dependencies.

A similar approach is the so-called Dynamic Precision Scaling [164], which defines the concept of dynamic region. These regions are delimited according to runtime conditions of the program and may not be mapped on a specific region of source code.

3.2 PROGRAM ANALYSIS

Precision tuning is supposed to provide the most performing data type for each value. The use of a reduced precision data type can introduce several errors: round off error, cancellation error, overflow errors, and representation mismatch. The precision loss in the precision/performance trade off determines the round off error. There are cases when the cancellation error can be considered acceptable and other where the precision tuning tool should either track the dynamic precision requirements in the preliminary analysis, or verify the absence of cancellation error via a subsequent precision mix validation. The representation mismatch error does not happen when the precision options are limited to the IEEE-754 data types. When the precision tuning considers also other floating point and fixed point data types, the representation mismatch error may be neglected due to its erratic and infrequent nature. On the opposite, in these cases overflow errors must be avoided. Thus, the analysis always have to provide safe bounds for the dynamic range of each value.

A considerable share of precision tuning tools just apply a trial-and-error paradigm to precision tuning, such as *CRAFT* [85], *Precimonious* [133], *PROMISE* [62], and others [3, 111, 129]. They lower the precision of the values in the program and observe the error on the output of a testing run. These approaches select a precision mix to evaluate without any knowledge on the dynamic range of values. Among these tools, there are some [111] that perform a full-factorial exploration of the possible precision mix. Other tools – e.g. *CRAFT* [85], *Precimonious* [133], and *PROMISE* [62] – implement search algorithms to explore the space of possible precision mix more efficiently. As long as the list of data types that are candidates to be allocated to each value is relatively short (e.g. only IEEE-754 data types) and the space to be explored is sufficiently smooth, greedy algorithms can be used to reach a good-enough suboptimal solution. Thus, the preliminary analysis can prove to be slower than a trial-and-error approach. However, this approach does not scale with the number of possible number representation that can be used. Furthermore, discontinuities in the program can trick a greedy algorithm into a local optimum, which may be considerably distant from the global optimum.

The goal of the preliminary analysis of the program is to provide a fine-grained description of the precision requirements in the code regions of interest. This description allows the precision tuning tool to significantly reduce the space of possible data types admitted for each intermediate value. The end-user may want to provide as little information as possible to the precision tuning tool. Thus, the precision tuning tool usually receives as input either a set of annotation over the input values either an input batch which is representative of the typical workload of the application. Respectively, the precision tuning tool can

statically propagate the annotation metadata to all the intermediate values in the program, or it can instrument the application to profile the dynamic range of the program variables. The literature describes a large variety of approaches to program analysis. We can classify them into *static analysis* and *dynamic analysis*.

3.2.1 *Static Approaches*

Static analyses extract additional knowledge from the program source code without testing it with input data. They can propagate partial information from a portion of the code, such as input data, to the rest of the program, or they can characterize the program by adding new metadata which can be used to improve the precision/performance tradeoff.

Xfp [36] uncovers expressions equivalent to the input ones whose fixed point implementation have a lower error in terms of ulp. It exploits genetic programming to generate alternative expressions according to a known set of rewriting rules. *Xfp* reaches a sub-optimal solution without performing an exhaustive exploration. *Rosa* [34, 35] is a source-to-source compiler that provides a precision mix for a given program on real values. It considers fixed point data types (8, 16, 32 bit) as well as floating point data types (`binary32`, `binary64`, `binary128`, and a floating point extended format with 256 bit width). *Rosa* introduces a contract-based programming paradigm based on the Scala functional programming language. For each expression, *Rosa* asks the programmer to provide a pre-condition statement that describes the range of values expected as input. By running a static analysis on the intermediate values, *Rosa* provides safe approximations of range bounds for nonlinear real-value expressions. In particular, *Rosa* initially computes safe ranges using interval arithmetic [106]. Then, it applies a binary search using the Z3 [39] Satisfiability Modulo Theories (SMT) solver to check if the range could be tightened. The approach implemented in *Rosa* [35] has been extended in *Daisy* [32] by integrating the rewriting capabilities of *Xfp* [36]. Additionally, *Daisy* improves with respect to *Rosa* by using a different SMT solver – dReal [56] instead of Z3.

Salsa [30] asks the end-user to annotate the source code with the range of expected initial values. Then, it statically analyzes the intermediate representation of the program to propagate these metadata to all intermediate values. More precisely, it performs an inter-procedural value range analysis based on interval arithmetic [106].

A combined approach between interval arithmetic and affine arithmetic [145] is adopted also in hardware/software codesign environments, such as *Minibit* [114]. However, both interval arithmetic and affine arithmetic can lead to over-conservative ranges. Indeed, *Minibit+* allows the user to select whether to run a static analysis or a dynamic

profiling-based analysis. Static approches have been proposed in the literature to tighten these conservative output ranges. They mainly suggest to apply iterative refinement methods to the result of interval arithmetic and affine arithmetic. In particular, Kinsman and Nicolici [76] refine the ranges using a binary search until an SMT-solver is not able to demonstrate the validity of a stricter interval. Pang et al. [119] refine the ranges using *Arithmetic Transform* [118]. The resulting analysis is faster with respect to the SMT-based approach.

3.2.2 *Dynamic Approaches*

Autoscaler For C [79] performs an explorative run over the original floating point code to obtain an estimation of the dynamic range for each variable. This range estimation analysis is built upon the SUIF [161] compiler by instrumenting its intermediate representation. The goal is to find the smaller data width required by the algorithm for hardware implementation. While in the case of area optimization there are no limitations on the number of bits that are supposed to be used, in the case of performance optimization for general purpose computing the number of bits is usually a multiple of 8 – which is the number of bits in a byte.

Other dynamic analysis works aim at improving already existing precision tuning frameworks. The approach suggested by *Menard et al.* [99] provides an improved methodology to determine the bit width of the fixed point data type – in addition to an optimized code generation – for the *FRIDGE* [74] framework. In their work they describe a methodology to generate multiple data flow graphs from the target application using the SUIF [161] compiler framework. For each block of such data flow graphs, their approach involves the separate profiling of the input values. After the profiling, a static data flow analysis propagates the information about the dynamic range within the rest of the block.

Blame Analysis [134] is a dynamic technique which aims at reducing the space of variables involved in a precision mix exploration. This analysis extends the *Precimonious* [133] tool and – as for *Precimonious* – it acts on the LLVM [86] intermediate representation. *Blame Analysis* instruments the LLVM bitcode to create a shadow execution environment which tracks the evolution of floating point values at runtime. The result of the *Blame Analysis* does not contain the dynamic range of values, it only highlights the variables whose precision minimization had no impact on the output. Thus, it provides a list of variables which can be safely ignored for the purpose of searching the best precision mix.

There are tools and analyses in the literature that aims at detecting overflow errors at runtime. They modify the original program by wrapping instructions that may lead to overflow errors with dynamic

a check. This approach has been explored at source-level [42], at binary-level [14], and at compiler-level [128]. However, these works only detect overflow errors and do not focus on precision tuning.

Other relevant analysis have been proposed in the literature. Although they do not focus on the range of values, they provide information on the sensitivity of the output with respect to variable approximations. *ASAC* [130] automatically inserts an annotation to define an input data as *approximable* or *non-approximable*. Its approach is based on the observation of the perturbation of the output corresponding to a perturbation of the input variables. Whilst *ASAC* has been designed for generic approximate computing techniques, a similar tool – named *ADAPT* [101] – specifically targets floating point precision tuning. *ADAPT* is based on algorithmic differentiation (AD) [110]. According to its approach, the programmer selects which variables in the source code should be analysed. The tool performs a profile run using AD APIs from CoDiPack [135] (or Tapenade [69], depending on which is the source language) to estimate the sensitivity of the program with respect to each of the selected floating point variables. *ADAPT* produces an error model that can be used to estimate the effect of a precision lowering on a given variable. This sensitivity analysis approach assumes that the application to be tuned is always differentiable, and that the algorithmic derivative of the variable can be used as a proxy for the sensitivity of the program. Such assumptions prevent the code that can be tuned to contain even simple discontinuities, such as conditional statements.

3.3 CODE MANIPULATION

The code manipulation is the core part of a scalable reduced precision tool. It allows to automatically apply the precision mix to the target code regions of interest. However, it is often considered only as an implementation detail or as an engineering problem. For this reason several state-of-the-art tools do not implement code conversion features at all. This lack prevents real-world scenarios where applications have a large code base to adopt such tools.

3.3.1 *The Generality Problem*

Any other tool that aims at providing a reduced precision version of the input code needs to be aware of both the source language and the target platform of the compilation process it is part of.

On the source language side, this requirement is mostly due to the need to cope with best practices in different scenarios. For example, embedded systems programmers typically write their code in C, with C++ being an emerging solution. However, many algorithms are initially designed in Matlab or similar high-level languages, and tools

supporting precision reduction could be inserted in the Matlab-to-C conversion step instead of in the C-to-assembly step. In High Performance Computing, the scenario is not too different, with C++ and FORTRAN being heavily used, but other application domains might require different languages.

On the target platform side, different platforms provide different trade-offs. As a result, the tool needs to be keenly aware of the properties of each platform, or it needs to be able to deduce them in some way. On the one hand, the use of smaller data types requires the use of less complex – and thus more energy efficient – hardware platforms. On the other hand, it allows complex architectures to exploit more aggressively the SIMD parallelism through data vectorization. Which one is the most beneficial aspect depends on the target platform. In the embedded system domain it is common to have architectures with limited or no support for floating point computation, whereas in high performance computing the goal is to maximize the data parallelism.

As a result, in embedded systems the goal is typically to avoid the use of software emulation of large data types, especially when targeting microprocessors not endowed with a floating point unit, typically of ultra-low power platforms. In such platforms, the conversion is almost always beneficial, if reasonable accuracy can be obtained. In higher-end embedded systems, where floating point units are available, a conversion to fixed point may not be desirable. Still, even in these cases, limiting the data size to `binary16` or `binary32` floating point may be useful.

In high performance computing, the main goal is to reduce power consumption and to increase parallelism. The `binary16` representation finds its main application here, whereas the main impact of moving the computation from floating point to fixed point is given by the ability to exploit higher degrees of Single-Instruction Multiple-Data (SIMD) parallelism via vector instructions.

Consequently, tools tend to focus on one type of target platform. Source-to-source tools in particular need to have information about the target platform, whereas tools implemented as part of a compiler framework can leverage the target information already available as part of the back-end.

3.3.2 *A Technological Taxonomy*

When it comes to start writing new software and configure a programming toolchain, developers rarely focus on tools whose purpose is to enable or optimize reduced precision computation. Therefore, such tools have been specialized to target already existing programming toolchains. The technological approach followed by each of those tools reflects the needs and the integration requirements of the programming environment they were originally developed for. We classified

the technological approaches present in literature in five different categories:

1. Tools that accept as input a program written in a valid human-readable programming language and that produce as output another version of the same program written in a valid human-readable programming language. In our discussion we consider only tools that emit the same programming language they accept as input. These tools are also known as **source-to-source compilers**.

2. Tools based on **Binary modification** or instrumentation. Such tools operate on machine code which can directly be executed by the hardware.

3. **Compiler-level** analyses and transformations which are implemented as compiler extension or customizations. Such tools are executed as intermediate stages of the process that compiles program source code into machine-executable code.

4. **Custom programming environments** which involve complex ad-hoc toolchains that require a dedicated programming or code generation paradigm.

5. **Other** kinds of tools and utilities with specific purposes.

3.3.2.1 *Source-to-Source Compilers*

The source-to-source compilers are best-suited for those use cases where there is the need for an automatic code replacement with human supervision. A source-to-source approach is recommended whenever optimizations other than precision tuning are scheduled to be applied after the precision tuning.

Autoscaler For C [79] is a source-to-source compiler that complies with the ANSI C programming language. It converts every variable to fixed point by using a data size which guarantees the absence of overflow. The conversion applies within their compiler, which is built upon the Stanford University Intermediate Format (SUIF) compiler system [161]. Although the output of *Autoscaler For C* is ANSI C, which is portable to different architectures, it specifically targets digital signal processors without hardware floating point units.

A compiler-aided exploration of the effects of reduced precision computation involving native `binary16` code is described in [111]. The core of their toolchain is a source-to-source compiler, which is based on an aspect-oriented domain specific language. It creates multiple versions of the OpenCL source code with different precision mix through *LARA DSL* aspects [124]. This precision mix tuning tool explicitly targets OpenCL kernels for GPU architectures with hardware support for `binary16`-based vector data types.

Salsa [30] is a source-to-source compiler written in Ocaml whose purpose is to improve the accuracy of a floating point program without via source-level code transformations. It features both intra-procedural and inter-procedural code rewriting optimizations.

Xfp [36] is a tool for floating point to fixed point conversion which selects the fixed point implementation that minimizes the error with respect to the floating point implementation. As an exhaustive rewriting of the expression with different evaluation orders is unfeasible, *Xfp* exploits genetic programming to reach a sub-optimal implementation.

Rosa [34, 35] is a source-to-source compiler that provides a precision mix for a given program on real values. *Rosa* operates on a subset of the Scala programming language. It also introduces a contract-based specification language to allow the programmer to describe proper preconditions and precision requirements for each function. It parses such specifications and then allocates a data type for each `Real` value – placeholder for any value that can be either floating or fixed point – in the function. *Rosa* internally exploits the Z3 SMT solver [39] to process the precision constraints derived from the program accuracy specifications. The source code of *Rosa* is available at [31]. *Real2Float* [94] exploits a similar approach. In particular, they rely on global semidefinite programming optimization instead of SMT solvers. It relies on the semidefinite programming solver *SDPA*. The source code of *Real2Float* is available at [95].

Daisy [32] is a source-to-source compiler derived from *Rosa* [35]. It supports Scala and C programming languages. *Daisy* implements also code rewriting optimization from [36] and [117]. It explores code alternatives that may improve accuracy. The evolutionary algorithm that *Daisy* uses to evaluate the code alternatives is provided by the *Xfp* tool [36]. The source code of *Daisy* is available at [37].

PROMISE [62, 63] is a tool that provides a subset of the program variables which can be converted from `binary64` to `binary32`. It is based on the delta debugging search algorithm [167] – which reduces the search space of the possible variables to be converted. *PROMISE* is written in Python and it relies on the CADNA software [73] to implement the Discrete Stochastic Arithmetic verification in C, C++, and Fortran program source code. The output of *PROMISE* is a program which implements the best precision mix found by the tool.

3.3.2.2 *Binary Modification*

This kind of tools does not need to access the source code of the application as they work directly on the machine-executable code. Binary patching tools are particularly useful whenever the source code of the application cannot be accessed or modified. However, just as these tools are unbound from the program source language, they have a very strict focus on a specific binary format. Therefore, binary instrumentation tools are designed to work only on a given computer

architecture, and the porting to other architectures may require a significant effort.

To detect the effect of digit cancellation in floating point code, [83] provide a description of a binary instrumentation tool which reports digit cancellation events at runtime. This tool is based on the Dyninst [15] and on the Intel Pin [92] binary instrumentation libraries.

The same authors later presented in [85] a whole framework – named *CRAFT* – which aims at minimizing the instructions that exploit the `binary64` format by replacing them with equivalents based on the `binary32` format. *CRAFT* is based on machine code instrumentation, and it performs binary code patching for Intel X86_64 architectures. The whole *CRAFT* framework is mostly written in C++. It initially instruments the target application executable file and then it provides a set of mixed precision configurations. Later, it evaluates the mixed precision versions on a given input test set to find the most promising one in terms of error and performance. It is based on the Dyninst [15] binary instrumentation library. Although the original implementation of the *CRAFT* framework – called *binary mode* – relies on binary instrumentation, authors recently added an experimental *variable mode* which allows the end user to restrict the scope of the tool to variables defined in the source code. This variable mode of *CRAFT* relies on the Typeforge tool [136] from the *Rose* compiler framework [125] to perform source-to-source translation on C/C++ applications. The *CRAFT* framework is available online as free software [80].

SHVAL is an open-source [81] library to perform the Shadow Value analysis described in [84]. It simulates the execution of a floating point program that exploits the `binary64` as it was executed using a different data type. The described implementation supports the emulation of the `binary32` and the `binary128` data types. It also supports `binary64` emulation for verification purposes. In particular, *SHVAL* instruments the binary code of a floating point program to measure the inaccuracy introduced by the floating point rounding. It inserts a *shadow* execution flow which runs coupled with the original program without interfering with its values. This shadow execution is used as a reference to compare the result of the original program. Thus, it exploits higher accuracy with respect to the original program. It allows several precision levels to be tested with this approach, even though they are not natively supported by the target architecture thanks to the GNU MPFR arbitrary precision library [51], which can emulate larger data types, such as the `binary128`. *SHVAL* is based on the the Intel Pin binary instrumentation library [92]. The *Shadow Value Analysis*, which originally targets only x86_64 architectures [81, 84], has been extended in [96, 97] to demonstrate its portability on a Raspberry Pi 3.

3.3.2.3 *Compiler-level Transformations*

They operate within the standard compilation flow of the target application. Therefore, it is relatively easy to integrate them in the development process of the target application. They work on the intermediate representation of the program which is built internally by the compiler infrastructure they rely on. The user is typically not asked to revise the output of the precision tuning tool, as it is not in a human-friendly form.

Precimonious [133] is a tool based on the customization of the LLVM [86] compiler framework which aims at suggesting the most efficient precision mix within a given error threshold. *Precimonious* accepts as input a C program and produces a description of the suggested precision mix as output. It exploits LLVM-IR bitcode files to test various versions of the code. Thus, such bitcode files can be saved for later reuse. It supports the standard C data types `float`, `double`, and `long double`, which respectively implement the `binary32`, `binary64`, and the 80-bit `double extended` precision data type. *Precimonious* is being used as a reference framework for further related works. In particular, *Blame Analysis* [131, 134], and *HiFPTuner* [65] are preliminary analysis that aims at reducing the search space of *Precimonious*. The source code of *Precimonious* is available online at [132].

A framework for automated accuracy reduction is described in [3]. The goal of this work is to reduce the memory impact of floating point values by using smaller data types. The presented work is based on the LLVM [86] compiler framework. This framework supports three classes of real number representations: IEEE-754 formats – more specifically `binary32`, and `binary16` –, mantissa-truncated data types – which are obtained by truncating mantissa bits from the basic IEEE-754 data types, such as `bfloat16` –, and IEEE-754-style data types – which are data types with variable bit width but constant ratio between the number of mantissa bits and that of exponent bits. It extends LLVM with the custom defined data types and transparently converts the floating point values. As the target hardware is not guaranteed to support the custom defined data types, the proposed approach entails wrapping every memory access instruction to unpack and to pack the data from and to such data types. Although this approach is target independent, it is particularly relevant for architectures where the cache and the memory size are critical. In the case of HPC accelerators – e.g. GPU – a reduced memory footprint may allow to run a higher number of parallel jobs.

Verificarlo [41] is an analysis tool designed to estimate the round off errors. It instruments the LLVM-IR of the program to substitute the IEEE-754 floating point arithmetic operations with equivalent instructions based on Monte Carlo Arithmetic. *Verificarlo* is available online at [166].

3.3.2.4 *Custom Environments*

Whenever the technical implementation of the precision tuning utilities require a complex chaining of multiple tools, the application development and precision tuning processes are likely to be customized to reflect the environment requirements.

FRIDGE [74] is a comprehensive hardware/software codesign environment that provides analysis and code conversion tools. The whole environment is composed of a source-to-compiler, a simulation environment, and an additional software component to determine the fixed point parameters. It simulates the effect of the fixed point operations on the hardware description via its own simulation system – called *HYBRIS*. *FRIDGE* accepts as input programs written in ANSI-C with floating point variables and it produces as output an equivalent code which exploits fixed point arithmetic. For simulation and evaluation purposes it features a VHDL and an assembly back-end. *FRIDGE* is designed to support digital signal processors (DSPs) without hardware floating point units.

More recently the focus of hardware/software codesign environments moved from DSPs to FPGAs. *Minibit+* [114] provides bit-width optimization analysis and precision mix features for FPGA-specific applications. It improves over the previous *Minibit* [88] tool by allowing the synthetization of a floating point hardware unit in addition to the fixed point one. This additional option allows the framwork to preserve accuracy in cases of extremely costly fixed point implementation alternatives. The whole framework is built upon the *BitSize* [55] code analysis framework.

ADAPT [101] is an analysis tool for floating point precision tuning. It is based on algorithmic differentiation [110]. *ADAPT* is shipped as a C++ library that has to be compiled along with the application to be tuned. Their approach is composed by two steps. First, the programmer selects which variables in the source code should be subject of analysis. The tool performs a profile run to estimate the sensitivity of the program with respect to each of the selected floating point variables. Later, it iteratively reduces the precision of the program variables by one precision step – e.g. from `binary64` to `binary32` – starting from the variable which is estimated to have the least impact on the degradation of the output. *ADAPT* stops when the estimated error reaches the tolerance threshold. The source code of *ADAPT* is available online under the GPLv3 license [100].

PRECiSA [108, 154] is an error estimation environment. It consists in an abstract interpretation framework that defines an analysis which is parametric over a set of input-predicated conditions. *PRECiSA* is based on the definitions of the basic floating point (hardware and software) standard [12, 102] via the SRI's Protoype Verification System [115]. It supports the basic floating point data types according to the definitions available in the SRI Prototype Verification System. The source code is

available online at [152]. It can also be used as an online service via web page [153].

FPTuner [25] is an analysis tool based on Symbolic Taylor Expansion which is able to provide a precision mix allocation of values that keep the error within a given bound. It supports `binary128`, `binary64`, and `binary32` data types. *FPTuner* is based on the *Gelpia* [2] global optimizer, which provides bounds to the expressions processed by *FPTuner*. *Gelpia* is capable to leverage SIMD parallelism via the *GAOL* [60] library for interval arithmetic. Finally, a solver library is required to obtain the best precision mix allocation. To this end *FPTuner* relies on *Gurobi* [66] and on the *FPTaylor* [144] error estimation tool. *FPTuner* targets mainly the scientific computing domain. However, it is limited to conditional-free expressions. *FPTaylor* is open source [143] and it can also be used as a standalone tool. Authors open sourced their tool at [24].

PetaBricks [4] is a programming language that exposes the concepts of *accuracy metric* and *accuracy guarantee* to the programmer. The implementation of a library function can be defined multiple times with different precision or accuracy specifications. The library user can specify which level of accuracy guarantee to use in his code. There are potentially infinite different implementations at different accuracy levels: they might differ for data types as well as for the algorithm implementation. *PetaBricks* has its own compiler and setup environment. The latter features an autotuner framework that explores the accuracy of the several implementations during a training phase and selects the most suitable for runtime. The *PetaBricks* environment implements a closed-loop system where an autotuner profiles the application at deploy time. The application runs in tight coupling with the autotuner, which observes the program output and decides at runtime which code version the program should use. The *PetaBricks* approach enables the automatic multithreading parallelization of the code via compiler transformations. The effectiveness of this approach is demonstrated on a multi-core platform.

An emerging approach is the so-called *Transprecision* technique which consists in tuning the accuracy of the computational kernel at runtime. This approach allows the precision level to be tightly coupled with the current state of the application, and with the input data. The work presented in [89] introduces *Transprecision* for iterative refinement algorithms. It employs `binary32`, `binary64`, and `binary128` data types. This work guarantees the convergence of the the iterative algorithm, as every source of inaccuracy introduced by the described approach is proven not to impact on the convergence of the refinement. It targets HPC-like computing cores featuring x86_64 architectures.

A proof of concept for *DPS* – acronym for Dynamic Precision Scaling, which is meant to be the porting to reduced precision of the more well-known concept of Dynamic Voltage and Frequency Scaling

(DVFS) – is described in [164]. The purpose of DPS is to run the program on reduced-precision floating point functional units whenever the data can tolerate the degradation, and to dynamically switch to the original floating point data types when there is the need to preserve the accuracy. The proof-of-concept implementation features an offline profiler, a runtime monitor, and an accuracy controller software component. The *DPS* system continuously monitors the quality of the output at runtime. The accuracy controller component runs a regulator which adjusts the precision level according to the feedback from the monitor. The described implementation of the offline profile and the accuracy controller are based on the approximate computing framework iACT [103]. The runtime monitor is emulated via binary instrumentation of load/store instruction in the binary to export performance counters. The reduced-precision floating point functional units should be obtained by reducing the number of mantissa bits from the `binary64` and `binary32` floating point data types.

3.3.2.5 *Other Notable Tools and Methods*

The literature related to precision tuning presents a large variety of works. However, not all of them can be classified in any of the aforementioned categories. In particular, we highlight utilities and tools with a very specific purpose, which are particularly interesting for precision tuning.

FlexFloat [149] is an emulation *Transprecision* framework for variable width floating point data types. It supports any configuration of exponent bit width and mantissa bit width for floating point data types smaller than 32 bit. It can be used to evaluate the functional effects of custom-defined floating point data types. The goal of this work is to implement ultra low-power architectures featuring transprecision floating point hardware units. Two different implementations are available online: a full featured one [148] and a lightweight version with only precision analysis [49].

FloPoCo [44] is a framework written in C++ that generates VHDL code to design custom arithmetic data path of floating point cores. It provides to the hardware designer C++ classes to represent custom floating point units. It is designed to provide a high level abstraction of the design of floating point units with custom bit width of mantissa and exponent. *FloPoCo* generates a synthetizable hardware description according to the parameters specified via C++ code. The source code of *FloPoCo* is available at [43].

A machine-learning based method for the dynamic selection of the precision level for GPU computation is presented in [129]. It implements a modified version of the random forest algorithm to decide whether a variable type should be `binary32` or `binary64` floating point. The proposed approach prune the less promising branches of the exploration tree to reduce the number of code versions to test. The work

presented in [129] strictly fits the single use case presented in the paper, which is a GPU application that implements a 3D stencil iterative algorithm.

3.4 VERIFICATION

Once the precision tuning tool processed the program and generated a reduced precision version of it, the tool should provide guarantees on the maximum degradation of the output quality. The metric that is used to describe the output quality is typically application-dependent. The most common use case involves the end-user specifying a threshold of maximum acceptable degradation. Such threshold may be zero when the user aims at maximizing performance without any quality degradation. Just as the preliminary program analysis, the verification methodologies can be either **static** or **dynamic**, depending on whether they need to execute the mixed precision version to be verified or not.

The literature on software verification is rich of research works on this field, as it captures interest from the software engineering research area [57]. Thus, the problem of estimating a safe upper bound of the error introduced in the computation by the use of certain data types has widely been addressed by dedicated tools. Along with the verification procedures adopted by precision tuning tools, in this Section we also survey relevant verification-specific works which can be related to reduced precision computation.

3.4.1 *Static Approaches*

Static approaches compute a worst-case scenario of the error projection on the output without the need to exhaustively test every input case. The conservative approach of the static analyses may lead to extremely large error bounds, which eventually can degenerate in no information or useless partial information. Whenever static analyses do not diverge, they can provide a formal proof of the output, which is required by the use cases whose errors must not reach the given threshold – such as safety-critical systems.

Salsa [30] employs on a set of rewriting rules to minimize the error in a given program. In this case – given a correct set of rewriting rules – the final version of the program is proven correct by construction. Although the rewriting engine of *Xfp* [36] exploits a similar methodology, the final evaluation of the mixed precision version is based on a profile run.

In the case of the contract-based programming paradigm – such as in the *Rosa* [34, 35] and *Daisy* [32] tools – the precision allocation follows the user-defined constraints. The solution of such constraints is typically provided by an automatic SMT solver. Therefore, also in the case of these tools, the error introduced in the final mixed precision

version stays within the given threshold by construction. A similar approach, implemented in *Real2Float* [94], uses semidefinite programming optimizations instead of SMT-based solutions. *Real2Float* is able to provide error bounds with proof of correctness for Ocaml programs.

FPTaylor [144] is an error-estimation tool which relies on symbolic Taylor expansion. It approximates the floating point expression with its first order Taylor expansion, and it uses the second order term as an upper bound for the error. Is is also used by the *FPTuner* [25] as key component of the verification process for the mixed precision versions. This approach works properly on floating point expressions that are smooth enough to be polynomially approximated. In the case of discontinuities, the approximation may not fit the real error.

A different symbolic analysis is implemented in *PRECiSA* [108, 154]. This tool implements a symbolic static analysis for computing provably-sound over-approximation of floating point round-off errors. It takes as input an expression defined over real values, and predicates on the input values. The output of the *PRECiSA* analysis [108, 154] is a set of tuples $< eb, cond >$ where *eb* is the estimated error bound and *cond* is the set of validity conditions for *eb*. In addition to the error bounds, *PRECiSA* provides also the verification lemmas to prove the correctness of such bounds.

GAPPA [38] is a tool based on interval arithmetic and forward error analysis which is able to provide formally verified error bounds. The *GAPPA* input language allows expressions on real values with floating point and fixed point data types. The former are built-in data types whereas the latter need to have their parameters manually specified – e.g. data width and rounding mode. The source code of *GAPPA* has been published online at [98].

A similar approach based on affine arithmetic [145] has been implemented within *Fluctuat* [61]. This analysis tool provides error bounds for algorithms implemented via IEEE-754 floating point data types, and hints about their numerical accuracy.

Although its focus is not on precision tuning, the *Astrée* [29] analysis tool can prove effective also in this domain. *Astrée* is a static analyzer which is able to ensure the absence of several categories of runtime errors. Among such categories, all possible floating point rounding errors are considered.

3.4.2 *Dynamic Approaches*

Dynamic approaches compare the result of the reduced precision version to be tested with a more accurate version of the program. It is common practice to generate an executable version of the reduced precision version and run it on a representative input set. This is the case of tools like *Autoscaler for C* [79], *Precimonious* [133], *HiFPTuner* [65], and *CRAFT* [85].

Although the approach followed by *SHAVAL* [84] is not significantly different from the one previously mentioned, it should be analysed in more details. It is in fact able to instrument the executable code of the application to be analyzed. Then, *SHAVAL* traces the evolution of the variables at runtime. In particular, it inserts an independent data flow in the program to replicate the original operations, having this *shadow execution* based on a data type which is more precise with respect to the original one. The goal of *SHAVAL* is not related to verification of mixed precision version. Rather its goal is to empirically measure the error dependent on the data type used in the program to represent the real values using a larger-precision data type as reference.

Researchers also proposed probabilistic methodologies to perform reliable profiling of the mixed precision version to compute the error bounds with a certain degree of confidence. *PROMISE* [63] estimates the error bounds using Discrete Stochastic Arithmetic [159]. It executes each arithmetic operation 3 times using a random rounding mode – 0.5 probability of rounding up, 0.5 probability of rounding down. Another floating point rounding error estimation tool based on probabilistic analysis is *Verificarlo* [41]. It replaces the arithmetic instructions based on the IEEE-754 floating point standard with equivalents based on Monte Carlo Arithmetic [122].

Other tools do not produce a fully-working executable version, but only perform emulation or simulations of the mixed precision version instead. This approach is typical of design and prototyping tools, such as *FRIDGE* [74], and *FlexFloat* [149]. In the context of hardware simulation, the *FRIDGE* [74] verification system performs a simulation of the fixed point operations. Its own simulation system – called *HYBRIS* – allows the framework to collect the precision profiles of the mixed precision hardware designs with a bit-level accuracy. *FlexFloat* [149] emulates the functional behavior of custom transprecision data types on generic by means of other floating point standards, such as IEEE-754 data types. Both *FRIDGE* and *FlexFloat* require the end-user to provide an input set to test the configuration on.

More in general, the verification and validation of a mixed precision configuration can be also applied at runtime, in a closed-loop control-system way. Precision autotuning frameworks, such as the one proposed for *Dynamic Precision Scaling* [164] and for the *PetaBricks* [4] runtime monitor system, follow this approach. These works measure the quality of the output while the system is in production and adjust the precision when the runtime conditions change.

3.5 TYPE CASTING OVERHEAD

The strict implementation of the data width minimization may lead to a very heterogeneous precision mix. This is particularly the case of fixed point representations. Every type mismatch in the data flow of

the program requires performing a type cast operation before continuing the execution of the program. The overhead introduced by type cast operations may overtake the benefits of using the smaller data types. It follows that the minimization of the data type width does not guarantee performance improvements. A dynamic performance profiling can instead solve this problem. Indeed, whenever the precision tuning system validates the benefits of a data type variation by executing the code, the measurement includes also the type casting overhead. Examples of this approach are implemented in *FRIDGE* [74], *Precimonious* [133], *CRAFT* [85], and *PetaBricks* [4]. Although performance profiling solves this problem, a representative profiling run on each code version may not always be possible. This limitation is possibly due to the time required to perform it, which can exceed the time budget that a programmer is willing to allocate to precision tuning.

To overcome this problem, it is possible to estimate the overhead by measuring the number of type cast instructions introduced in the code, or by using other similar heuristics. This is a promising approach that has been explored only by few tools in the literature. In particular, *Autoscaler for C* [79] and the approach suggested by [99] iteratively optimize the fixed point code by reordering instructions to collapse the shift operations whenever possible. *FPTuner* exposes to the end-user a threshold on the number of type cast that the tool is allowed to insert in the code. However, this parameter is hard to hand-tune for the end-user. *Daisy* [32] uses the number of type cast operations in a cost function that estimates the profitability of the precision lowering. *HiFPTuner* [65] adopts a hierarchical-based approach to minimize the number of type cast operations. It builds a data-dependency tree and it tries to assign the same data type to all values in the same cut of the tree.

3.6 A COMPARATIVE ANALYSIS

In this section we summarize the most relevant contributions to the state-of-the-art. We compare tools and approaches over a fixed set of *functional capabilities* and *portability characteristics*. Functional capabilities represent the core of the research innovation represented by tools and approaches in the state-of-the-art. Let us define portability characteristics as the implementation details and the features of the tool that practically enable it to fit to a given use case. The effectiveness of each work depends on the combination of advanced functional capabilities and wise implementation choices.

3.6.1 *Functional Capabilities*

For each work, Table 3.1 describes the following capabilities:

SCOPE of the tool or analysis

ANALYSIS to discover properties of the algorithm

OVERHEAD handling to mitigate the effect of type cast operations at runtime

VALIDATION approach to calculate error bounds

GUARANTEES provided by the validation

In Table 3.1 we distinguish tools whose scope is the whole program (P), from the ones that allow to restrict the scope to a user-defined portion of the whole program (U) or to a single computational kernel (K). The capability to limit the scope of the analysis (or of the transformation) can turn out to be fundamental in the case of, for instance, very large and complex programs. The focus of the reduced precision computation should be only on the most computationally intensive kernel whereas precision tuning typically brings little or no benefit to the input/output routines, and other marginal code. This way it becomes possible to avoid the additional time and space complexity of the problem which is generated by the marginal portion of code of the program. However, there exists use cases – e.g. the full repurpose of an application to support an architecture with different arithmetic units – where it may be desirable to act on the whole program without having to specify explicit bounds.

A preliminary analysis stage can help precision tuning tools to expand the knowledge on the code by uncovering properties or by propagating partial knowledge about some values in the program to the rest of the code being analyzed. Depending on which type of information the precision tuning tool is able to exploit, and depending on which type of information is available before the processing, a preliminary analysis can be performed statically (S) – by only processing

Table 3.1: Tool Capabilities Synopsis

Tool / Approach Name	Scope	Prelim. Anal.	Ovh Handl.	Valid. Method	Guarantees
ADAPT [101]	U	D	none	P	none
Angerd et al. [3]	K	–	–	–	none
ASAC [130]	U	D	–	–	none
ASTRÉE [29]	P	–	–	S	F
Autoscaler for C [79]	P	none	R	P	none
CRAFT [85] (binary mode)	P	none	P	P	none
CRAFT [85] (variable mode)	U	none	P	P	none
Daisy [32]	U	S	C	S	F
DPS [164]	U	D	none	A	none
FlexFloat [149]	U	–	–	P	none
FloPoCo [44]	U	–	–	–	none
Fluctuat [61]	P	–	–	S	F
FPTaylor [144]	P	–	–	S	F
FPTuner [25]	P	D	L	S	F
FRIDGE [74]	U	none	P	P	none
GAPPA [38]	P	–	–	S	F
HiFPTuner [65]	P	M	H	P	none
Menard et al. [99]	P	D	R	P	none
Minibit+ [114]	P	S, D	C	P	F
Nobre et al. [111]	K	none	P	P	none
PetaBricks [4]	P	none	P	A	none
Precimonious [133]	U	none	P	P	none
PRECiSA [108]	P	–	–	S	F
PROMISE [62]	P	none	none	S	prob.
Real2Float [94]	P	–	–	S	F
Rojek [129]	P	none	none	P	none
Rosa [35]	U	S	none	S	F
Salsa [30]	P	S	none	S	F
SHVAL [84]	P	–	P	P	none
Verificarlo [41]	U	–	–	D	prob.
Xfp [36]	P	S	none	P	none

the code and the user input – or dynamically (D) – by running the original version of the program. Singular cases reported in Table 3.1 are *HiFPTuner* [65], which implements a mixed approach (M) between static and dynamic analysis, and *Minibit+* [114], which allows the end-user to select whether to use a static or dynamic approach. This preliminary analysis step is particularly important for tools that deal with fixed point representations, as the position of the point can be different for each intermediate variable. In Table 3.1 we denote with *none* precision tuning tools that do not perform any preliminary analysis and that directly start processing the code without any attempt to extract additional pieces of information. We distinguish tools where a preliminary analysis is not applicable – because their purpose is not to generate a mixed precision version of the code – and we denote them with a dash symbol.

The use of smaller data types does not necessarily imply a reduction in the execution time. Although the reduced precision data type can prove to be more efficient to compute, there are side effects of the precision reduction that impact on the overall code performance. The most important side effect is the introduction of the type cast operations. Such operations convert variables from a larger to a smaller data type, and vice-versa. A precision mix that minimizes the data size of the variables does not guarantee that the required type cast operations have a limited impact. Indeed, it may happen that the overhead given by the type cast instructions overcomes the benefits of the reduced precision data type. Among the other side effects of changing the data type it is worth mentioning the increased heterogeneity of the data types within the computational kernel, which may lead to fewer vectorization opportunities for architectures that support SIMD instructions. In Table 3.1 we can observe that usually the overhead introduced by the change in the data type is not considered by the precision tuning tools. The most common approach to evaluate the effect of such overhead is to profile (P) the mixed precision version. Static approaches aim at the reduction of shift operations in the code (R), limiting the number of type cast operations with a threshold (L), or using a cost function (C) to weight their impact in the mixed precision version. *HiFPTuner* [65] exploits a hierarchical based approach (H) to minimize *a priori* the number of type cast instructions in the mixed precision code. As for the preliminary analysis, a dash symbol means that this capability is not amenable to consideration.

Even though the quality of the output can be lowered to improve the time-to-solution or the energy-to-solution of the application, this quality degradation has to be bound within acceptable limits to preserve a meaningful computation. Different applications have different requirements on the quality of the output. E.g. media streaming applications require the output to be limited on the average case, whilst equation solvers require that the error at each step of the computation stays

within a given threshold. Methodology to verify the error bounds can be static (S) or dynamic (D). Some tools for reduced precision computation provide a formal proof (F) – or a probabilistic test (prob.) to verify – that the error is less then a given value in the worst case. Other tools perform an explorative profile run (P) of the reduced precision versions using a significant input test set that verifies whether the result is acceptable. Another approach consists in a continue monitoring of the output quality by sampling it, and by dynamically adjusting the precision level at runtime (A).

3.6.2 *Portability Characteristics*

The evaluation of generality and portability capabilities of each tool or approach is not easy. We rely on the known implementations of the previously mentioned works to structure two tables that can help the reader understand which tool can suit which use case. In particular, Table 3.2 reports the following implementation details for each work:

INPUT LANGUAGE of the tool.

OUTPUT LANGUAGE of the tool.

DATA TYPES supported or considered by the tool.

Although an approach can be extended or generalized, what matters for its usability are the features that its implementation supports. The most common approach to precision tuning is a proper replacement of the data type in variable declaration at source code level. It follows that the tool's input and output languages limit its implementations. Even though tools and approaches may work at binary level to be source-language independent, they are bound to a given binary format, which usually depends on the architecture type. An intermediate approach consists in operating the precision tuning within the compiler. In this case the limits to the usability concern the compiler's capabilities and not the tool's ones. Some other tools work by abstracting the description of the program via a custom defined description language. Those tools require the user to manual port the program from the source programming language to the input language accepted by the tool. Similarly, the user has to convert the output of the tool from a precision mix description to a program implementation.

Most of the tools that automatically provide a precision mix only focus on the most popular data types, which are usually IEEE-754-compliant floating point data types. In Table 3.2 we denote with *fixed* the capability to deal with fixed point representations, while we generally denote with *IEEE754* the capability to support the `binary32`, `binary64`, and `binary128` data types from the IEEE-754 standard.

1 *SHVAL* supports IEEE754, MPFR, Unums, and Posits by default. It also allows the user to extend this framework to support custom data types

Table 3.2: Tool Implementation Synopsis

Tool / Approach Name	Input Language	Output Language	Considered Data Types
ADAPT [101]	C/C++/Fortran	description	IEEE754
Angerd et al. [3]	LLVM-IR	LLVM-IR	`binary32`, custom
ASAC [130]	LLVM-IR	LLVM-IR	`binary64`, `binary32`
ASTRÉE [29]	C	description	IEEE754
Autoscaler for C [79]	ANSI-C	C++	fixed
CRAFT [85] (binary mode)	x86 bin	x86 bin	`binary64`, `binary32`
CRAFT [85] (variable mode)	C/C++	C/C++	`binary64`, `binary32`
Daisy [32]	Scala/C	Scala/C	IEEE754, fixed
DPS [164]	–	–	custom
FlexFloat [149]	C++	C++	custom
FloPoCo [44]	C++	VHDL	user-defined
Fluctuat [61]	C, ADA	description	IEEE754
FPTaylor [144]	expression	description	IEEE754
FPTuner [25]	expression	expression	IEEE754
FRIDGE [74]	ANSI-C	C++	fixed
GAPPA [38]	Gappa	Col/HOL	IEEE754, fixed
HiFPTuner [65]	C	description	IEEE754
Menard et al. [99]	C, ARMOR	C	fixed
Minibit+ [114]	ASC/C/C++	description	fixed, `binary32`
Nobre et al. [111]	OpenCL	OpenCL	IEEE754, `binary16`
PetaBricks [4]	PetaBricks	C++	–
Precimonious [133]	LLVM-IR	description	IEEE754
PRECiSA [108]	PVS	PVS proof	IEEE754
PROMISE [62]	C/C++	C/C++	`binary64`, `binary32`
Real2Float [94]	Ocaml	description	IEEE754, custom
Rojek [129]	CUDA	CUDA	`binary64`, `binary32`
Rosa [35]	Scala	Scala	IEEE754, fixed
Salsa [30]	C	C	IEEE754
SHVAL [84]	x86 bin	x86 bin	several[1]
Verificarlo [41]	LLVM-IR	LLVM-IR	IEEE754
Xfp [36]	Matlab	Matlab	fixed

As the input language can be a limit to the tool usability, which platform a given tool targets is enforced by the technological structure of the tool itself. Most of the source-to-source compiler tools can be considered platform independent, whereas the binary instrumentation and the hardware/software codesign tools are strictly bound to a given architecture type.

Other solutions specifically focus on particular use cases or architectures. The hardware/software codesign tools for Digital Signal Processors (DSP) focus on the ANSI-C programming language. They accept C as input and they all produce C or C-like source code. The verification tools, such as *PRECiSA* [154] and *GAPPA* [38], produce proof lemmas that can be verified as output. The tools proposed by Nobre et al. [111] and by Rojek [129] operate on GPU kernels by using OpenCL/CUDA programming languages whereas Angerd et al. [3] propose a GPU-oriented approach while working within the intermediate representation of the compiler. *FlexFloat* [149] is designed to emulate code for Ultra Low Power (ULP) architectures via C++ code encapsulation. The original version of *CRAFT* [82] is source-programming-language independent as it operates at binary level on Intel x86 architectures. Although in its newer variable mode the tool's scope can be controlled by the end user, it loses the source-language portability.

In Table 3.3 we classify tools and approaches based on the the platform they support. In the case of tools that run on general purpose computers but that are specifically designed to support the development of a particular architecture, we report the target architecture. In particular, we consider the following additional characteristics for each tool:

TARGET PLATFORM the tool runs on, or it is designed for.

FRAMEWORK the tool is built upon.

LICENSE the tool is released under.

The literature on tools for reduced precision computation dates back to the 1990s and some of those tools are still relevant in the current state-of-the-art. Among them there are tools that have been maintained and kept updated by private companies and/or open source communities. Other tools and approaches that did not received much attention are based on frameworks and/or technologies which may have become obsolete during the years. The proper setup to make such tools interact with a modern toolchain may require additional effort to satisfy software dependencies. For each tool, we report in Table 3.3 the framework their implementation is based on, and which license their source code has been released under (if any). No license is indicated for works that onlny describe methodologies.

Even though there are tools in the state-of-the-art that solve most of the challenges presented in Section 2.3, there is no tool that can

Table 3.3: Tool Release Synopsis

Tool / Approach Name	Target Platform	Base Framework	Licensing
ADAPT [101]	analysis	CoDiPack, Tapenade	GNU GPL v3
Angerd et al. [3]	GPU	LLVM 3.5	proprietary
ASAC [130]	analysis	LLVM	proprietary
ASTRÉE [29]	generic	–	proprietary
Autoscaler for C [79]	DSP	SUIF	proprietary
CRAFT [85] (binary mode)	x86	Dyninst	GNU LGPL v3
CRAFT [85] (variable mode)	generic	Rose compiler	GNU LGPL v3
Daisy [32]	generic	dReal, Z3	BSD-2 Clause
DPS [164]	simulation	iACT	–
FlexFloat [149]	ULP	–	FSF Apache v2
FloPoCo [44]	FPGA	–	FSF AGPL
Fluctuat [61]	analysis	–	proprietary
FPTaylor [144]	analysis	–	MIT
FPTuner [25]	x86	Gurobi 6.5	MIT
FRIDGE [74]	DSP	HYBRIS	proprietary
GAPPA [38]	analysis	–	CeCILL
HiFPTuner [65]	generic	LLVM	proprietary
Menard et al. [99]	DSP	SUIF, CALIFE	–
Minibit+ [114]	FPGA	BitSize	proprietary
Nobre et al. [111]	GPU	–	proprietary
PetaBricks [4]	generic	PetaBricks	MIT
Precimonious [133]	generic	LLVM 3.0	BSD-3 Clause
PRECiSA [108]	analysis	SRI's PVS	NASA
PROMISE [62]	generic	CADNA for C/C++	GNU LGPL v3
Real2Float [94]	generic	NLCertify, SDPA	CeCILL
Rojek [129]	GPU	–	proprietary
Rosa [35]	generic	Z3	BSD-2 Clause
Salsa [30]	generic	–	proprietary
SHVAL [84]	x86	Intel Pin	GNU LGPL v2.1
Verificarlo [41]	generic	LLVM	GNU GPL v3
Xfp [36]	generic	–	proprietary

properly solve all of them. In particular, the problem of avoiding a priori the overhead of type cast to consume the benefits of the mixed precision versions is poorly explored. The implementation of a tool that can be applied to several use cases is also an open issue as nowadays no de facto standard is available in the state-of-the-art.

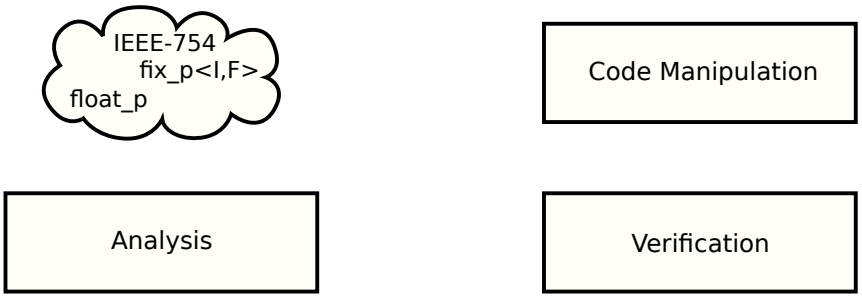
4

EFFECTIVE PRECISION TUNING SOLUTIONS

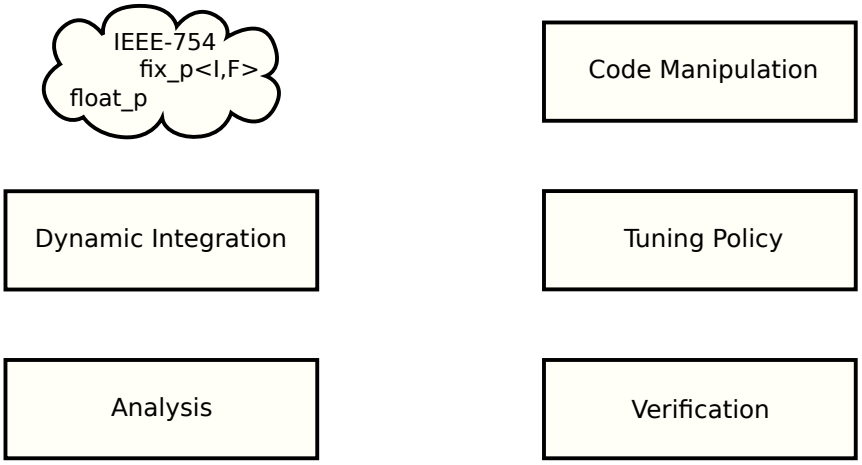
As we previously discussed in Chapter 3, the problem of precision tuning has been addressed from diverse and heterogeneous points of view. We have already discussed the differences between the various tools and approaches in the state-of-the-art in terms of functional and portability characteristics. We now focus on the components that are common among the previously mentioned proposals. In this chapter we discuss in detail the structure of two distinct solutions adopted to perform precision tuning. Such solutions can be used to apply precision tuning in different points of the software design flow depending on the goal we want to pursue.

Precision tuning can be performed either statically or dynamically. Static precision tuning produces a single mixed precision version that runs for every possible input data. Dynamic precision tuning adapts the mixed precision version to varying runtime conditions such as resource availability and input data. Despite many similarities between those two processes, they present different challenges and thus they require a slightly different toolchain structure. Figure 4.1 summarizes the components required by static and by dynamic precision tuning.

Static mixed precision tuning is performed once for every application and it is considered part of the system design. Indeed, such precision tuning tools can be part of wider hardware/software co-design environments. The base component of every mixed precision tool is a set of data type representations. The IEEE-754 standard offers a selection of widely supported floating point representations. However, fixed point representations are usually implemented either as signed or as unsigned integers representations. Once a floating point or fixed point representation has been consolidated, there is the need to consistently change the data type in the application. This code conversion can be performed on the source code of the application, on the binary machine code, or at an intermediate level within the compiler. Different variables in the code may require to be converted to different data types. The analysis of the application that has to be tuned usually involves a profiling phase to empirically measure the range of possible values that each variable can assume at runtime. This profiling phase needs to run a sufficiently large input set to cover all the relevant branches in the application. The complexity of this task scales up exponentially with the number of variables that have to be tracked, and with the number of control-flow statements in the application. Thus, this analysis requires a large amount of time for real-world applications.



(a) Static precision tuning components.



(b) Dynamic precision tuning components.

Figure 4.1: Software components required to effectively perform static (4.1a) and dynamic (4.1b) precision tuning.

Dynamic mixed precision tuning is a recurring task that gets invoked multiple times while the application is running. In this case, the precision tuning is considered as a possible code transformation that performs continuous program optimization. Thus, the re-configuration overhead must be minimized. While the selection of the set of supported floating and fixed point number representations is not different from the static precision tuning, in this case the code conversion tool either supports the dynamic generation of a mixed precision version, or it should rely on ahead-of-time compilation techniques. Thus, Figure 4.1b introduces a *Dynamic Integration* component that is not present in the static case. Another difference with respect to the static precision tuning lies in the *Tuning Policy*: this software component decides which mixed precision version should be used at runtime and when to dynamically generate a new mixed precision version. The profiling phase of the analysis of the application is performed with a reduced input test set, or it is replaced by heuristics or static analysis

to further reduce the overhead. Similarly to the analysis, the verification phase is also considered as part of the overhead resulting from the generation of a new mixed precision version. Therefore, different verification techniques are recommended with respect to the static precision tuning.

In the rest of this chapter we are going to present two solutions to perform precision tuning. The first one is designed for static precision tuning, whilst the second one performs dynamic precision tuning.

4.1 STATIC PRECISION TUNING

The content of this Section has been presented in International Conference on Parallel Computing (ParCo). Bologna, Italy. Sep 2017. [22]

Static precision tuning does not present any runtime overhead, as it provides a mixed precision code version at design time.

In this section we propose a floating to fixed point conversion solution that can be classified as a technology enabler according to the taxonomy presented in Chapter 3. Fixed-point representations are typically used in hardware design, where the width can be arbitrarily chosen for each value, on a per-bit basis. Since the widths of the integer and fractional parts are fixed and pre-computed, they must be carefully chosen to limit the precision loss. For a given computation, this task is accomplished by assessing the dynamic range (minimum and maximum) of its input values, and by propagating these ranges through all intermediate values – in a data-flow manner – to the results. The analysis phase does not have strict time bounds. Therefore, to profile the dynamic range for each variable, it is recommended to employ representative input data sets and to run the application several times. Based on all ranges, an appropriate fixed-point representation that minimizes the added noise is selected. When using general-purpose processors, on the contrary, the actual bit-widths are constrained by the underlying hardware, typically the width of registers. In practice, such containers are 16-bit, 32-bit or 64-bit wide. Still, the cost of floating-point arithmetic, even in optimized hardware implementations, is high enough to make it worth investigating the benefits of fixed-point operation even in the context of high performance computing. Thus, we use a parametric fixed point representation as target data type.

To allow the programmer to retain a good control during the design phase we propose to exploit a source-to-source compiler as a code manipulation component. This kind of tool provides a description of the mixed precision version which is ideally written in a language familiar to the programmer. Although there are exceptions that insert language extensions to enable features not supported by the original language, those extensions usually aim at preserving the code readability.

The verification procedure is highly application dependent. Whenever it is not possible to define an output quality metric, we rely on relative error metrics by using the original full-precision version of the application as reference.

Figure 4.2 summarizes the component description for this proposed solution.

4.1.1 *A source-to-source solution*

Our proposed solution takes advantage of the programmers' application domain knowledge of the nature of the processed values. In particular, we rely on source code annotations written by the program-

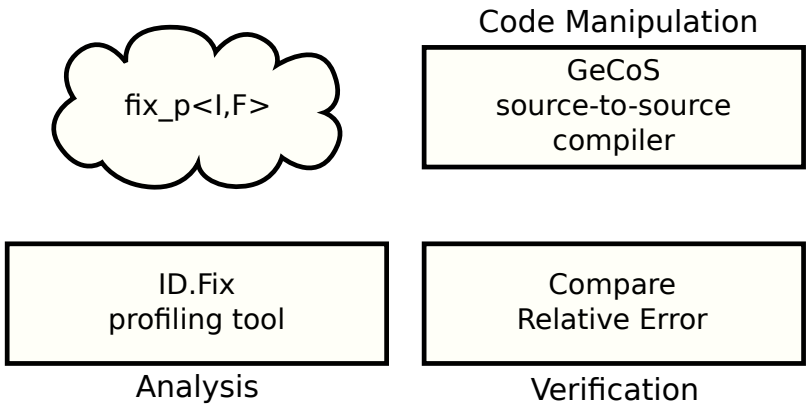


Figure 4.2: Software components for static precision tuning.

Listing 4.1: ID.Fix Annotation Example

```
input :

#pragma VARIABLE_TRACKING variable
for (int i=0, i<10, i++) {
  variable=i;
}

output :

variable_min = 0
variable_max = 9
```

mer (we consider as input a valid C/C++ source file) to know which variables should be converted to fixed-point. The input annotations for a simple example are shown in Listing 4.1.

Then, we perform a *value range propagation analysis* to propagate the value range information from annotated variables along data-dependence chains, thus inferring the value range for each variable involved in the computation. The output of this analysis is a fully annotated C/C++ source code having the dynamic range of each variable annotated in their declaration. To perform the value range propagation analysis, we re-purposed the GeCoS[1] framework [40, 50, 120]. GeCoS was originally designed as an hardware/software codesign environment. In particular, the ID.Fix [142] plugin for GeCoS is the component that tracks the dynamic range of the annotated variables via an automatic instrumentation of the code. We modified the data type allocation to enforce the use of a data width multiple of the word size for the specific architecture.

From the value ranges, it is then possible to compute the number of bits needed for the representation of the integer part of the fixed-point

1 http://gecos.gforge.inria.fr

representation. The width of the fractional part is then obtained as the difference between the architectural constraint on the total bit size and the size of the integer part.

The GeCoS source-to-source compiler takes then care of replacing the annotated floating-point variables with their fixed-point equivalent. It also adds to the original source code the utility functions to perform data type conversions from floating-point to fixed-point and vice versa.

The output of this stage is a new version of the kernel source code exploiting fixed-point computation instead of floating-point computation. The fixed-point code can then be compiled as a standard C++ source file using any compiler compliant with the C++11 standard. We developed a C++ library [20] that defines a template type `FixedPoint<integer_bits,fractional_bits>` with operators properly defined to make its use convenient.

Listing 4.2: Before GeCoS Source-To-Source

```
#define SIZE1 10
#define SIZE2 10

#pragma VARIABLE_TRACKING m tmp foo
double m[SIZE1][SIZE2];

double tmp;

double foo;

foo = 0;
for (size_t i = 0; i < SIZE1; ++i) {
  for (size_t j = 0; j < SIZE2; ++j) {
    if (m[i][j] > m[j][i]) {
      foo = foo + m[i][j] * m[j][i];
      tmp = m[i][j];
      m[i][j] = m[j][i];
      m[j][i] = tmp;
    }
  }
}
```

In the case of our running examples, the output of the source-to-source compilation process for our running example is shown in Listing 4.3 for the input of Listing 4.2.

Listing 4.3: After GeCoS Source-To-Source

```
#define SIZE1 10
#define SIZE2 10

double m[SIZE1][SIZE2];
FixedPoint<3,29> m_fixp[SIZE1][SIZE2];

double tmp;
FixedPoint<3,29> tmp_fixp;

double foo;
FixedPoint<8,24> foo_fixp;

convert2DtoFixP<double,SIZE1,SIZE2>(m, m_fixp);

foo_fixp = 0;
for (size_t i = 0; i < SIZE1; ++i) {
  for (size_t j = 0; j < SIZE2; ++j) {
    if (m_fixp[i][j] > m_fixp[j][i]) {
      FixedPoint<8,24> _s2s_tmp_foo_0;
      _s2s_tmp_foo_0 = m_fixp[i][j] * m_fixp[i][j];
      foo_fixp = foo_fixp + _s2s_tmp_foo_0;
      tmp_fixp = m_fixp[i][j];
      m_fixp[i][j] = m_fixp[j][i];
      m_fixp[j][i] = tmp_fixp;
    }
  }
}

convertScalarToFloat<double>(foo_fixp, foo);
convert2DToFloat<double,SIZE1,SIZE2>(m_fixp, m);
```

4.2 DYNAMIC PRECISION TUNING

Dynamic precision tuning is a form of continuous program optimization that entails the reconfiguration at runtime of the main application itself.

The reconfiguration happens every time the runtime conditions significantly vary. Tuning the threshold on the runtime conditions triggering the reconfiguration is a task that depends on the reconfiguration time and on the benefits that the mixed precision version brings. We obviously want to maximize the benefits of the mixed precision version. However, dynamic precision tuning adds reconfiguration time in the equation, and this overhead needs to be minimized. Indeed, we prefer to exploit a suboptimal mixed precision version whenever its setup is significantly faster with respect to the optimal mixed precision version.

The need for a quick reconfiguration time changes the structure of the precision tuning toolchain originally employed for static precision tuning. In particular, the profiling phase should be replaced by a static analysis, the source-to-source compilation stage should be replaced by a compiler-level transformation, and the validation can be performed by a static estimation of the error bounds rather than by a reference-based error verification.

In the dynamic precision tuning structure we also introduce a dynamic integration component, which has no corresponding equivalent in the static precision tuning structure. This component is the joint point between the precision tuning process and the continuous program optimization paradigm. The main application should be able to reconfigure itself multiple times via the dynamic integration system.

Given this abstract description of the required components, we propose a dynamic precision tuning solution, whose structure is summarized in Figure 4.3. Our solution is based on the dynamic compilation library LIBVERSIONINGCOMPILER and on the compiler-level Tuning Assistant for Floating point to Fixed point Optimization TAFFO.

4.2.1 TAFFO

The content of this Subsection has been accepted for publication in the journal Embedded Systems Letters.

Floating to fixed point conversion is a key task in the field of embedded application design. Since it is generally performed manually, it can incur in delays and potentially in additional errors. No existing open source tool is mature enough for industry adoption, and most of them have little hope of achieving production status, as they are designed as primary research tools, thus employing compiler frameworks that, differently from LLVM and GCC, do not benefit from industry-grade maintenance. Moreover, developers' needs and habits evolve over time. In recent years the popularity of ANSI C as main development language is decreasing in favor of more expressive programming lan-

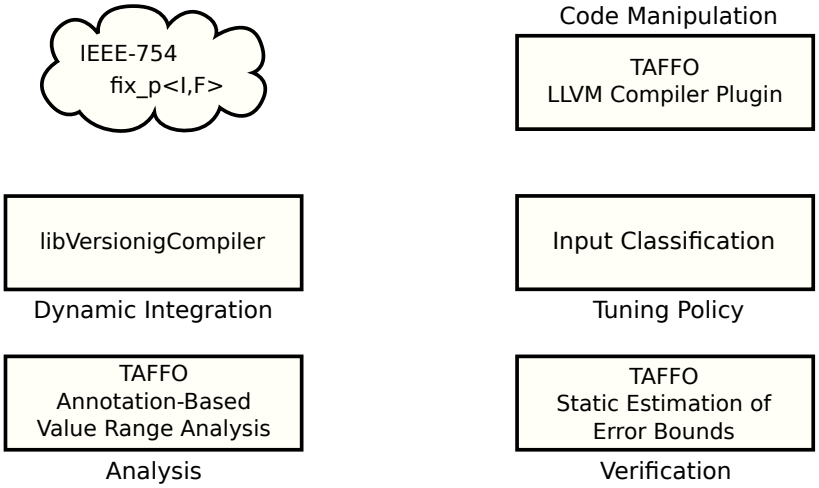


Figure 4.3: Software components for dynamic precision tuning.

guages [58]. The quest for tools that can support an increasingly large number of languages is still an open challenge. To bridge these gaps, we introduce *Tuning Assistant for Floating point to Fixed Point Optimization* – TAFFO. This is a toolset that automatically converts computation from floating point to fixed point, and that tunes the precision of the resulting fixed point code according to the application goals. TAFFO leverages programmer hints to understand the characteristics of the input data, and performs the conversion to the appropriate data types. The tools are robust enough to support automated conversion for complex C++ benchmarks without rewriting the computational kernels into less expressive languages, such as ANSI C.

TAFFO is a flexible and lightweight framework. Flexibility comes from the capability to address an arbitrary large number of source languages. Although an approach based on static analysis provides less strict bounds on the runtime values, the achieved result is proved to be safer with respect to profile-based approaches, which depend on input representatives. Moreover, the combination of profile-based approaches with whole-program analysis proved to be extremely time-consuming [134].

The implementation of TAFFO does not require any modification of the standard compiler toolchain. As shown in Figure 4.4, TAFFO introduces new compiler passes without modifying neither the existing compiler passes nor the compiler front-end – as most competitors do [32, 35, 74, 79].

The complexity of finding the best precision mix for a given program grows exponentially with the number of values to be tuned and with the possible precision levels. An exhaustive exploration is feasible only with a small number of values, and with a reduced selection of precision levels. Although the space of possible solutions is wide, a large portion of it is composed of precision mix permutations of

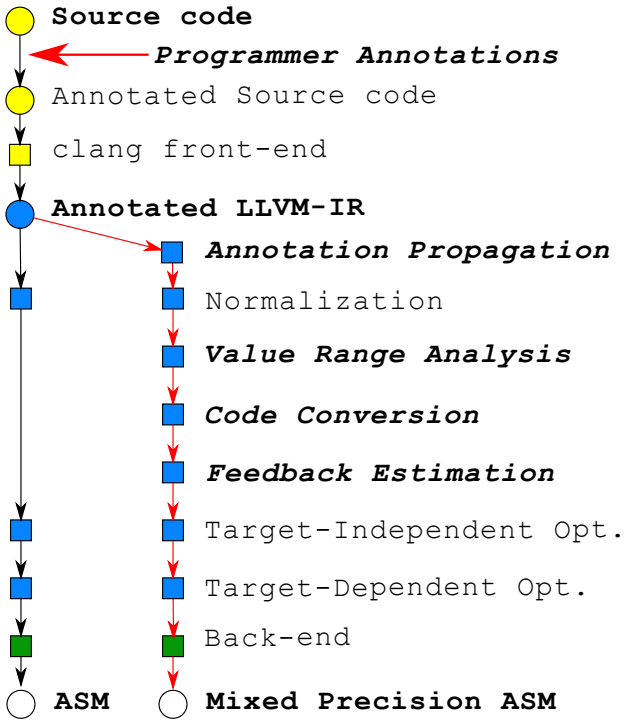


Figure 4.4: Outline of the compilation pipeline using the CLANG compiler front-end with and without TAFFO. We highlight with red arrows the TAFFO pipeline stages. Yellow elements refer to source code and the compiler front-end. Blue elements refer to passes of the optimizer. Finally, the green element represents the compiler back-end.

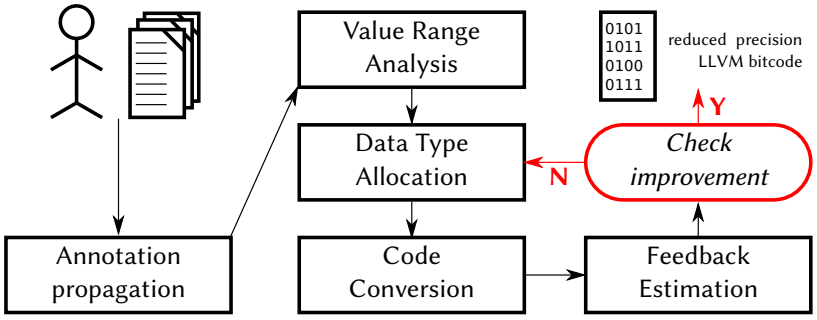


Figure 4.5: Component schema of the TAFFO framework, highlighting the control loop driven by the performance/error prediction carried out in the Feedback Estimation component.

marginal code. These code versions can be safely removed from the exploration. To this end, we rely on the programmer's knowledge of the application. We ask the programmer to restrict the scope of TAFFO's analysis and transformation via annotations on the source code. The annotation-based approach is a common practice in compiler construction to forward pieces of information towards further compiler stages. We use compiler metadata to keep information about the range of possible values for each variable. TAFFO requires the user to specify the range only on the input values. Our framework makes sure these values gets propagated to all the intermediate values. We then decide the allocation of data types and – in case of fixed point – the position of the point for each value. This process takes place in the intermediate representation of the compiler. Thus, we allow each intermediate value to be represented with a potentially different data format.

After the code conversion, we evaluate the converted code to check if it actually represents an improvement with respect to the baseline. This process – called *Feedback Estimation* – entails both a functional and a performance evaluation. The error bounds are computed via a data flow analysis whilst the performance is estimated via a previously computed platform-dependent performance model.

The structure of TAFFO is summarized in Figure 4.5. A detailed description of each component of TAFFO follows.

ANNOTATIONS Annotations specify which portion(s) of the code should be analyzed for precision tuning. The insertion of `annotate` attributes is natively supported by the CLANG compiler. Thus, there is no need to extend the compiler front-end for any language extension. This approach holds also with other front-ends for the LLVM compiler infrastructure.

Listing 4.4: Example of annotated C code where the programmer is asking to transform the variables `a` and `b` to a fixed point, and is providing the value range for the `c` variable.

```
float a __attribute((annotate("taffo 20 12")));
float b __attribute(
    (annotate("taffo 7 25 signed 0.4 0.9 1e-8")));
float c __attribute((annotate("range 0.05 1 0")));
```

As shown in Listing 4.4, the programmer may specify additional parameters to TAFFO. Indeed, the annotation on line 3 adds the value range, and the value uncertainty. In addition to the fixed point format, it is also possible to add the value range and the value uncertainty, as the annotation on line 3. If the programmer defines only the range, TAFFO can derive the most appropriate fixed point format on its own. It is possible to convert the computations whose result is used by the annotated variable by using `"force_taffo"` instead of `"taffo"`.

VALUE RANGE ANALYSIS After the propagation of the annotations, we run a data flow analysis based on interval arithmetic [107] to propagate the value ranges to all the intermediate values defined in the LLVM-IR. We recall that LLVM-IR is a low-level representation in which every compound expression is split into their minimal terms. Thus, we are able to analyze and propagate the precision requirements for each term of the compound expressions.

We denote with the symbol $\bot$ an undefined range, and with the symbol $\top$ the limit we allow to be represented in the range of a variable. In our case study, we limit them to be the smallest/largest value that can be represented by a 32 bit integer variable, whereas in the general case $\top$ is defined based on the available data type in the target architecture. The data flow analysis is initialized by the range as defined in the annotations. In particular, we define the set of ranges available at initialization time as the *InSet* of the root node of the Control Flow Graph, $InSet_0$. All other $InSet$s are initialized to undefined values:

$$InSet_I = \{r_i = \bot | v_i \in V\}$$

where V is the set of all variables.

To define the data flow equations for our analysis, we need to introduce the definition of an operator $\sqcup$ to combine two sets of ranges $A = \{r_i^A | v_i \in V\}$ and $B = \{r_i^B | v_i \in V\}$:

$$A \sqcup B = \{< \hat{\min}(l_i^A, l_i^B), \hat{\max}(u_i^A, u_i^B) > | r_i^A \in A \wedge r_i^B \in B\}$$

where

$$\hat{\max}(x,y) = \begin{cases} x & \text{if } y = \bot \\ y & \text{if } x = \bot \\ \max(x,y) & \textit{otherwise} \end{cases}$$

and $\hat{\min}$ is defined in the same way.

We can now introduce the following equations, characterizing a forward data flow analysis:

$$InSet_I = \bigsqcup_{J \in pred(I)} OutSet_J \tag{4.1}$$

$$OutSet_I = InSet_I \setminus kill(I) \cup gen(I) \tag{4.2}$$

where the *gen* and *kill* sets are defined as follows:

$$gen(I) = \begin{cases} gen_{\texttt{fmul}}(I) & \text{if } I \text{ is } \texttt{fmul} \\ gen_{\texttt{fadd}}(I) & \text{if } I \text{ is } \texttt{fadd} \end{cases} \tag{4.3}$$

$$kill(I) = \{r_i | i \in def(I)\} \tag{4.4}$$

where the gensets $gen_{\texttt{fadd}}(I)$ and $gen_{\texttt{fmul}}(I)$ are defined as follows:

$$gen_{\texttt{fadd}}(I) = \{r_i =< l_{\texttt{fadd}}, u_{\texttt{fadd}} > | i \in def(I)\} \tag{4.5}$$

$$gen_{\texttt{fmul}}(I) = \{r_i =< l_{\texttt{fmul}}, u_{\texttt{fmul}} > | i \in def(I)\} \tag{4.6}$$

where

$$l_{\texttt{fadd}} = \sum_{j \in use(I)} l_j$$

$$u_{\texttt{fadd}} = \sum_{j \in use(I)} u_j$$

$$l_{\texttt{fmul}} = \min_{j,k \in use(I) \wedge j \neq k} (l_j * l_k, u_j * l_k, u_k * l_j, u_j * u_k)$$

$$u_{\texttt{fmul}} = \max_{j,k \in use(I) \wedge j \neq k} (l_j * l_k, u_j * l_k, u_k * l_j, u_j * u_k)$$

Essentially, the transfer equation replaces the range of the defined variable(s) with a combination of the ranges of the variables used as inputs for the operation that generates the new value, according to the operation class. In the above equation, we only report the definition of $gen(I)$ when I is `fadd` or `fmul`, which is sufficient for our case study, but the definition can be trivially extended to other floating point operations. It is worth noting that the definition of the l_I and u_I terms of Equation 4.5 and Equation 4.6 – and for each generic floating point instruction I – must be slightly modified to include the values of $\top$ and $\bot$, thus the operations saturate to $\top$ and accept $\bot$ as a neutral element.

We provide a sketch of the proof of convergence for the data flow analysis.

Lemma 1. *The data flow analysis defined by equations 4.1 and 4.2 converges.*

Proof. The proof is derived from the following properties. *Monotonicity*: the operations on ranges never reduce them. *Bounded cardinality*: the ranges are bounded by the definition of $\top$, which acts as an absorbing value for all operations. *Halt*: the analysis halts when a fixed point is reached, which at most happens when the all ranges are set to $\top$. □

In the general workflow of our solution, the data flow analysis we have just defined is used to perform a LLVM-IR-level evaluation of the code to be converted. The n_{bits} values found through this analysis are then used to annotate every variable – if it was not already annotated.

These concepts can be extended to consider also procedure invocations. In particular, our implementation performs inter-functional analysis of the program across all the functions defined in the translation unit being analyzed. To retain a fine-grained control over the precision requirements we handle multiple call sites using the same function by cloning the function being invoked.

Intra-functional control flow operations – such as loops – are handled by relying on compile-time knowledge whenever it is possible – i.e. loop bounds known at compile time – or by asking the end-user to provide a safe bound to the number of iterations.

DATA TYPE ALLOCATION AND CODE CONVERSION Given the previously computed annotations, we can derive the data width required by each intermediate value. Let $r_i =< l, u >$ be the range of the variable v_i with lower bound l and upper bound u. Equation 4.7 relates the range of a variable with the minimum bit width required.

$$n_{bits} = \begin{cases} \lceil \log_2 \max(abs(l), abs(u)) \rceil & l \geq 0 \\ \lceil \log_2 \max(abs(l), abs(u)) \rceil + 1 & l < 0 \end{cases} \tag{4.7}$$

Our solution transforms the LLVM-IR as if a type change was performed in the original source code. Integer and fractional parts are logically partitioned so as to prevent a priori any overflow problem. The transformation generates code to perform the computation with fixed point arithmetic alongside the already existing floating point code. We allocate a separate memory location for the fixed point values. The code conversion process supports the inter-procedural transformation of memory operation on scalar, array, and pointers values via `load`, `store`, and `getelementptr` instructions. By their nature, the conversion of constants – both literals and in-memory constants – do not require any memory duplication.

Function calls are handled via duplication of the function in the LLVM-IR. We apply the same code conversion procedure to the cloned function as if its parameters were annotated by the programmer.

When the code conversion pass meets an instruction with an unknown conversion – as in the case of calls to an external function – it restores the original data type and it leaves that instruction unchanged. This *fallback* behavior guarantees a strong preservation of the program semantic.

In the absence of fallbacks, all the uses of the floating point values should have been replaced by their fixed point equivalent. Finally, we schedule a Dead Code Elimination (DCE) optimization pass from the LLVM compiler infrastructure, which safely removes all the floating point instructions – including the `allocas`.

Listing 4.5 shows snippets of the source code from the *gramschmidt* benchmark from the PolyBench/C benchmark suite, its translation to LLVM-IR, and the corresponding fixed point equivalent LLVM-IR after the code conversion.

FEEDBACK ESTIMATION We evaluate the mixed precision LLVM-IR bitcode with two metrics. First, we run a static error propagation analysis to project the truncation error we introduced with the fixed point computation on the output. The propagation is performed by representing the absolute error associated to each LLVM-IR instruction by means of *affine forms* [48], combined with the intervals resulting from the value range analysis (c.f. [35]). This approach allows us to keep track of each single error source, exploiting error cancellation when possible.

An *affine form* is the representation of a variable x as

$$x = x_0 + \sum_{i=1}^{n} x_i \epsilon_i,$$

where x_0 is the *central* value, and each ϵ_i is a *noise symbol* of magnitude x_i. Each noise symbol is a symbolic variable representing a single error source. Affine forms are combined with the intervals resulting from the value range analysis, as detailed in [35]. In this setting, to each instruction x we associate a range r_x and an error e_x, represented as a zero-centered affine form. Sum and subtraction are performed term-wise on the magnitudes of corresponding noise symbols. This allows us to exploit the possible cancellation of errors due to the same noise symbol. The error of a `mul` instruction with operands x and y is computed as

$$\begin{aligned} x \times y &= (r_x + e_x)(r_y + e_y) \\ &= r_x \times r_y + r_x \times e_y + r_y \times e_x + e_x \times e_y. \end{aligned}$$

Division is treated similarly, that is a multiplication by the inverse of the divisor.

Non-linear operations such as mathematical functions from the C standard library are treated with linear approximations, as suggested in the state-of-the-art [33, 48]. Whenever it is possible we exploit

C source code

```
[...]
POLYBENCH_2D_ARRAY_DECL(A,DATA_TYPE,M,N,m,n);
POLYBENCH_2D_ARRAY_DECL(R,DATA_TYPE,N,N,n,n);
POLYBENCH_2D_ARRAY_DECL(Q,DATA_TYPE,M,N,m,n);
[...]
nrm = SCALAR_VAL(0.0);
[...]


nrm += A[i][k] * A[i][k];


[...]




R[k][k] = SQRT_FUN(nrm);




[...]
R[k][j] += Q[i][k] * A[i][j];
[...]
```

LLVM-IR before conversion

```
[...]
%A.addr.i41 = alloca [240 x float]*, align 8
%R.addr.i42 = alloca [240 x float]*, align 8
%Q.addr.i43 = alloca [240 x float]*, align 8
[...]
store float 0.000000e+00, float* %nrm.i, align 4
[...]


%mul.i56 = fmul float %36, %40


[...]
%43 = load float, float* %nrm.i, align 4


%call.i59 = call float @sqrtf(float %43) #4


store float %call.i59, float* %arrayidx17.i, align 4
[...]
%add60.i = fadd float %80, %mul55.i
[...]
```

LLVM-IR after conversion

```
[...]
%A.addr.i41.fixp = alloca [240 x i32]*, align 8
%R.addr.i42.fixp = alloca [240 x i32]*, align 8
%Q.addr.i43.fixp = alloca [240 x i32]*, align 8
[...]
store i32 0, i32* %nrm.i.fixp, align 4
[...]
%59 = sext i32 %52 to i64
%60 = sext i32 %58 to i64
%61 = mul i64 %59, %60
%62 = ashr i64 %61, 10
%63 = trunc i64 %62 to i32
[...]
%67 = load i32, i32* %nrm.i.fixp, align 4
%68 = sitofp i32 %67 to float
%69 = fdiv float %68, 1.024000e+03
%call.i59 = call float @sqrtf(float %69) #4
%70 = fmul float 1.024000e+03, %call.i59
%call.i59.fallback = fptosi float %70 to i32
store i32 %call.i59.fallback, i32* %75, align 4
[...]
%134 = add i32 %133, %127
[...]
```

Listing 4.5: Comparison between the source code of the *gramschmidt* benchmark of PolyBench/C 4.2, of its LLVM-IR code before the conversion to fixed point, and its LLVM IR code after the conversion to fixed point. The same line numbers correspond to the same point in the code in all listings. In the first snippet, the pass has converted the storage space to fixed point. In the second snippet, the pass has converted an immediate constant to fixed point at compile time. In the third snippet, the `fmul` instruction is replaced by a more complex fixed point equivalent. In the fourth snippet, the call to `sqrtf` – which does not have a fixed point equivalent – has been handled by the fallback algorithm. In the last snippet, the `fadd` instruction was replaced by an integer `add`.

the LLVM facilities to unroll loops, in order to analyze the error they introduce. The unrolling is performed on a copy of the loop, which is discarded after the analysis.

The second evaluation is based on a platform-dependent performance model. The goal of this step is to estimate the impact of the type cast overhead, and the performance gain due to the precision lowering. To this end, we train a performance model using code statistics before and after the code conversion. We rely on machine-learning tools from `Scikit-learn` [123] to analyze the impact of the introduction and removal of different instructions. We identify 26 classes of LLVM-IR instructions that can be used as features in statistical learning. For each class, the relevant feature is the change in instruction frequency from the floating point version of the code to its mixed precision version. As the target response, we consider the ratio T_{fix}/T_{flt} between the execution times of the fixed point conversion and that of the original code. We use the results of the conversion of a set of small computational kernels as the training set for a range of ensemble classification and regression methods (random forests, extremely randomized trees, bagging meta-estimators on decision trees and k-neighbors classification and regression, AdaBoost and gradient trees). We consider the most stable approach among these candidates to build the final performance estimation model.

4.2.2 *Precision Tuning Policies*

In a dynamic environment, the application processes several batches of input data and it can decide whether to keep the same kernel version running or to change it. For simplicity we assume a sequential application. This argument, however, can also be ported in the parallel case by implementing the decision process on each processing element. The precision tuning policy is the algorithm that controls the trade off between code reuse and exploration of new versions.

Depending on the assumptions that hold for the code generation and verification, the problem that the policy has to solve can be largely different. In our dissertation we distinguish three cases:

AHEAD OF TIME GENERATED AND VERIFIED The mixed precision code versions are generated before the execution of the application. The verification of their accuracy is performed offline.

RUNTIME-GENERATED NON-VERIFIED The mixed precision code versions can be generated during the execution of the application. The verification of the accuracy has to be performed by running the mixed precision versions multiple times with the workload of the application.

RUNTIME-GENERATED VERIFIED The mixed precision code versions can be generated during the execution of the application. The

verification of the accuracy can either be performed statically or by profiling it with a fixed number of input samples.

In the first case the precision tuning policy has to characterize and to classify each input batch and to decide which statically generated mixed precision versions need to be exploited to process the given input batch. Thus, the algorithm should implement an input partitioning and a suitability ranking for each input class. The input partitioning is intrinsically an application-dependent problem whereas ranking code versions according to a linear combination of known metrics is a problem which has been already solved in the state-of-the-art by autotuning tools such as mARGOt [54].

In the second case the precision tuning policy does not know in advance which version is most suitable for each input class. This can be discovered by selecting and executing one or more times the version itself. Hence, the suitability ranking has to be generated online. This is an optimization problem where the reward associated to a version for a given input class depends on the version's suitability for that input. The policy should explore the mixed precision versions to eventually achieve a stable suitability ranking for each input class. To this end, the algorithm has to minimize the *regret* function that represents the difference between the reward associated to the optimal version and the sum of rewards associated to the versions that have been selected. This problem is also known in the literature as the *multi-armed bandit problem* [158].

In the third case the precision tuning policy can decide to generate a new version on-the-fly. The suitability of such version can be discovered within a constant time by its generation. This is the case of TAFFO-generated versions. In our proposed solution we define a policy which provides an input partitioning based on the workload equivalence function for each target platform. In this situation, it is possible to define a priori which parameters should be used to generate the optimal version for the given input class.

WORKLOAD EQUIVALENCE The workload equivalence function is the component that properly partitions the input space into classes which can be considered equivalent from a continuous optimization point of view. In particular, all the input batches that belong to the same input class must:

1. share the same hardware resource **requirements**;
2. share the same input and output **format**;
3. share the same **verification** metrics and methods;
4. have a defined ratio between their problem **sizes**.

The first two constraints guarantee that two batches from the same input class can be processed using the same platform-optimized kernel

version. The latter ones enforce that the execution statistics for those input batches can be fairly compared in terms of output quality and performance per input size.

Additional desirable similarities between elements of the same partition sets given by the equivalence function are:

5. kernel **code coverage**;
6. kernel **memory footprint**;
7. **accuracy** requirements.

These properties minimize the distance between elements of the same partition measured in terms of the quality of the output and of the performance metrics. Indeed, a good workload equivalence function classifies in the same class input batches that are indistinguishable from the application profile point of view.

The workload equivalence function is required to partition the input into classes which are large enough to allow the optimization time to be paid back by the performance improvements.

A SIMPLE EXAMPLE An illustrative example is offered by the continuous program optimization on the well-known sorting algorithm *counting sort*, whose implementation is shown in Listing 4.6.

Listing 4.6: C++ implementation of the counting sort algorithm

```
void sort(std::vector<int32_t> &array, const int32_t
    min, const int32_t max) {
  const size_t my_range_size = max - min;
  std::vector<size_t> my_counter(my_range_size);
  std::fill(my_counter.begin(), my_counter.end(), 0);

  for( size_t i = 0; i < array.size(); i++ ) {
    my_counter[array[i] - min]++;
  }

  auto it = array.begin();
  for( int32_t i = min; i < max; i++ ) {
    const size_t increment = my_counter[i - min];
    std::fill_n(it, increment, i);
    std::advance(it, increment);
  }
  return;
}
```

The presented implementation is not platform dependent. The input and the output are defined only by the signature of the function. There is no other input from – neither output to – external sources, such as the file system or external memory locations. The quality of the output is given by a comparison function between the elements to be

sorted, which is defined statically for every input. The problem size is defined by the number of elements in the data structure that need to be ordered, which in the counting sort example is a vector. Thus, the ratio between problem sizes is a function defined for every non-zero input size.

The above mentioned requirements hold for every input of the target function. Thus, every workload equivalence function satisfies them. For the counting sort example we focus only on the desirable design features of the equivalence function.

Let us analyze the behaviour of this algorithm. First, the counting sort algorithm linearly iterates over the input batch. Later, it iterates over the range of possible input values. As the the input batch does not affect the code coverage, we are going to ignore this aspect in the design of the workload equivalence function. The memory footprint of the algorithm strictly depends on the `min` and on the `max` input parameters. Those values are good candidates to be considered as possible input partitioning criteria. The counting sort implementation presented in Listing 4.6 is properly defined for integer values and it can not deal with any real value data type. Accuracy requirements can be analyzed by analyzing which is the smallest integer data type that can be used for processing the input batch. However, no floating point nor fixed point precision tuning technique can be applied here.

In the case of counting sort algorithm we propose a workload characterization and partitioning based only on the `min` and on the `max` values. The function should equally classify all the diverse input batches that have the same `min` and `max` values.

4.2.3 *libVersioningCompiler*

The content of this Subsection has been published in the journal SoftwareX. *[21]*

Designing and implementing High Performance Computing (HPC) applications is a difficult and complex task that requires mastery of several specialized languages and performance-tuning tools; however, this prerequisite is incompatible with the current trend that opens HPC infrastructures to a wider range of users [78, 168]. The current model that sees the HPC center staff directly supporting the development of applications will become unsustainable in the long term. Thus, the availability of effective APIs and programming languages is crucial to provide migration paths towards novel heterogeneous HPC platforms as well as to guarantee the developers' ability to work effectively on these platforms.

Profile-guided code transformations at compile-time usually provide a good optimization level in a general-purpose scenario. On the contrary, in HPC scenarios where large data sets are employed, a proper profiling may be unfeasible. In these cases, which are becoming more and more common [126], dynamic approaches can prove more effective. The practice of improving the application code at runtime

through dynamic recompilation is known as *continuous program optimization* [46, 77, 113]. Although it has been studied for more than a decade, very few people adopt it in practice since it is difficult to perform manually, and, when performed automatically, it can compromise software maintainability. At the same time, autotuning is used both to tune software parameters and to search the space of compiler optimizations for optimal solutions [11]. Autotuning frameworks can select one of a set of different versions of the same computational kernel to best fit the HPC system runtime conditions, such as system resource partitioning, as long as such versions are generated at compile time. Few of these frameworks are actually able to perform continuous optimization, and those that support it do so only through specific versions of a dynamic compiler [9, 18] or through cloud-based platforms [27].

libVersioningCompiler (abbreviated libVC) can be used to perform continuous program optimization using simple C++ APIs. libVC allows different versions of the executable code of computational kernels to be transparently generated on the fly. Continuous program optimization with libVC can be performed by dynamically enabling or disabling code transformations, and changing compile-time parameters according to the decisions of other software tools such as a generic application autotuner.

DESCRIPTION OF THE SOFTWARE The goal of libVC is to allow C/C++ compute kernels to be dynamically compiled multiple times while the program is running, so that different specialized versions of the code can be generated and invoked. This capability is especially useful when the optimal parametrization of the compiler depends on the program workload. In these cases, the ability to switch at runtime between different versions of the same code can provide significant benefits, as shown in [19, 150].

Indeed, in general-purpose code it is preferable to profile the application to statically generate the most efficient versions ahead of time. However, in HPC code the execution times are usually so long that a profiling run may not be an attractive choice. On the contrary, libVC enables the exploration and tuning of the parameter space of the compiler at runtime, while the program is performing useful work.

libVC considers as valid compute kernels any C-like procedure or function that can be compiled to object code. There is just one constraint that should be enforced on the compute kernel: it must respect C linkage rules. This means that no name mangling should be applied to the compute kernel itself. Within our model, the `Compiler` is the tool used to compile the compute kernel, and the `Version` is the configuration passed to the compilation task. We assume to work with deterministic `Compilers`. In this scenario, a `Version` produces at

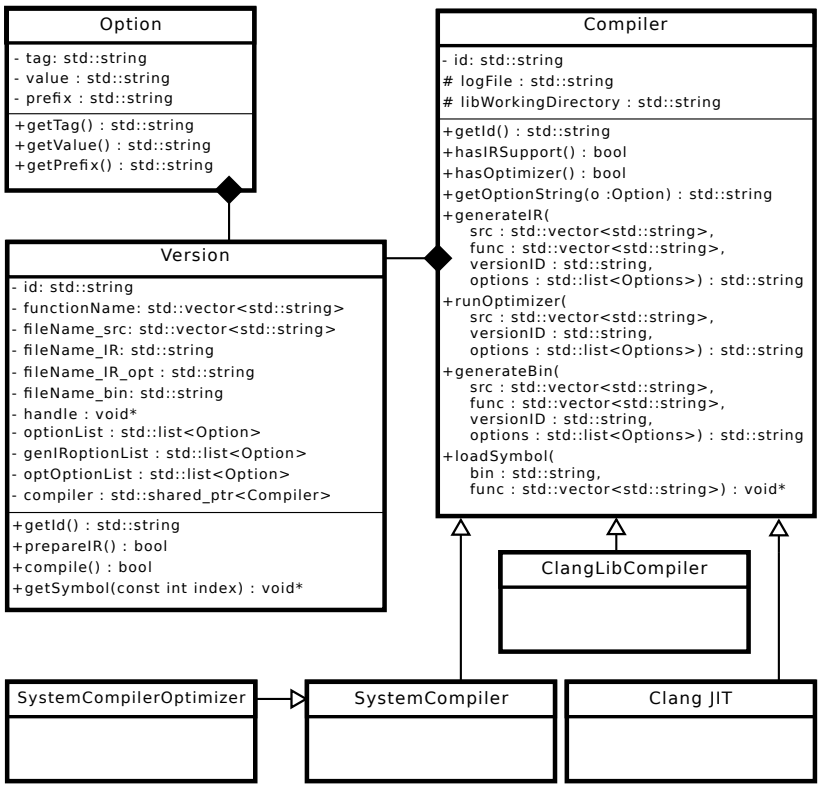


Figure 4.6: Simplified UML class diagram of LIBVC

most one executable code. No executable code is generated when the specified configuration is invalid.

The LIBVC source code is available under the LGPLv3 license. It is compliant with the C++11 standard and it comes with configuration files to ease the setup by using the `CMake` build system.

The minimum required `CMake` version is 3.0.2. The build system automatically checks the presence of the optional dependencies `LLVM` and `libClang`, whose version must be greater than `6.0.0`. Whenever these dependencies are not satisfied, some features are automatically disabled during the library installation. Please refer to the official code repository[2] for a detailed and exhaustive list of dependencies.

Figure 4.6 shows a simplified UML class diagram of this software. It is possible to identify three main classes in the source code. The simplest class, which is called `Option`, represents each of the flag and parameters that are passed to LIBVC in order to compile a version of a computing kernel.

The `Compiler` abstract class defines the interface that allows the host application to interact with `Compiler` implementations. LIBVC provides up to four possible implementations for the `Compiler` abstract

2 https://github.com/skeru/libVersioningCompiler

class: `SystemCompiler`, which relies on system calls to external compilers that are already installed in the host system; `SystemCompilerOptimizer`, which is an extension of a `SystemCompiler` that also supports external optimization tools (such as the LLVM optimizer `opt`); `ClangLibCompiler`, which exploits the compiler-as-a-library paradigm through the `Clang` APIs[3]; and `ClangJIT`, which implements the Just-In-Time compiler paradigm via `Clang`. The main difference between the compiler-as-a-library paradigm and the just-in-time compiler on – which are respectfully implemented by the `ClangLibCompiler` and the `ClangJIT` classes – consists in the interaction between the compiler and the host application. With the former approach the host application invokes the compiler as any other dynamic library. Whereas, with the latter approach the host application removes the overhead of performing a call to an external library API and embodies the compiler capabilities in its own executable code. Please note that `ClangLibCompiler` and `ClangJIT` are installed only if the optional `LLVM` and `libClang` dependencies are satisfied.

The last important class is the `Version` class, which represents a set of compute kernels defined in a specific list of source files, with a given compiler configuration. A `Version` object is compiled with the chosen `Compiler` using an ordered list of `Options`. It contains a unique identifier, references to `Compiler` and `Options` used to compile it, and references to the files that are generated by the `Compiler` while compiling the `Version`. The configuration of a `Version` object is immutable throughout the lifetime of that object. The `Version` class also provides APIs to control the stages of the compilation process: it is possible to create a `Version` object and postpone the execution of the selected `Compiler` to a later stage.

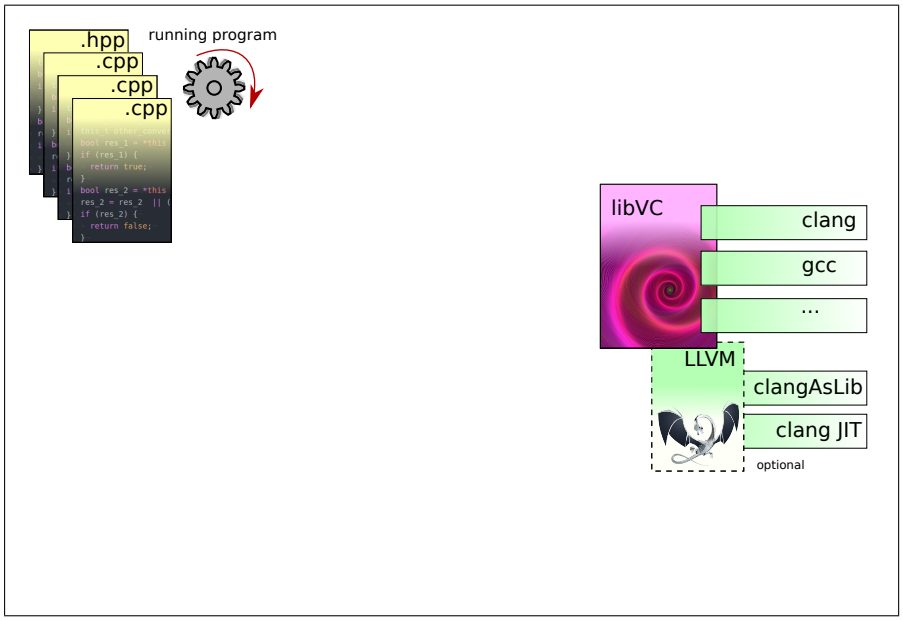
libVC provides an easy-to-use interface that can be employed to perform the dynamic compilation of the kernel, and to load compiled `Versions` as C-like function pointers. libVC itself does not provide any automatic selection of which `Version` should be executed. The decision of which `Version` is the most suitable for a given task is left to policies defined by the programmer or other autotuning frameworks such as mARGOt [54] or cTuning [53]. Figure 4.7 illustrates the configuration and the exploitation paradigm of a `Version` object step by step.

An introductory video about libVC *is available online at* `https://youtu.be/1p8IajxOgoY`.

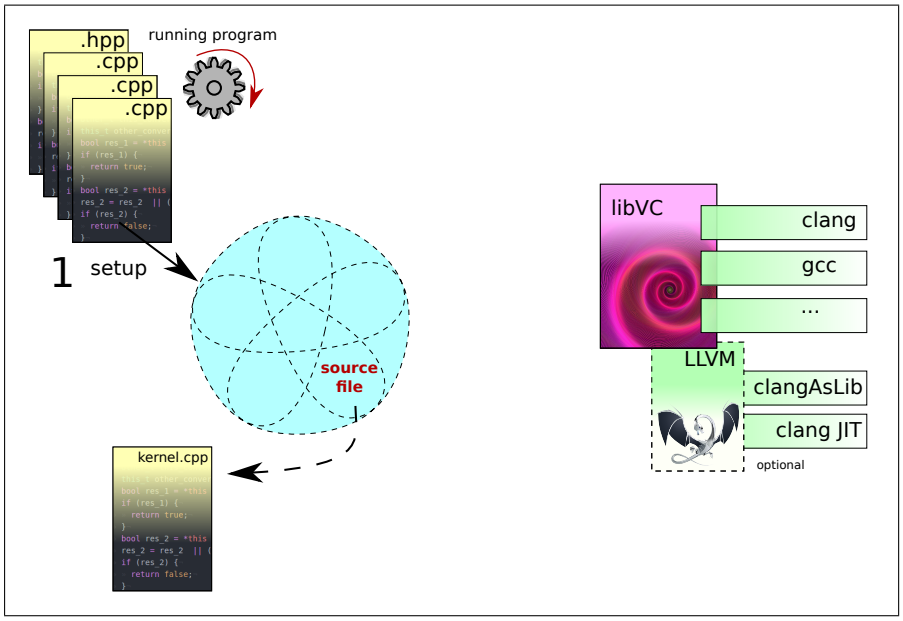
libVC comes in two different flavors: with detailed low-level APIs and with simple high-level APIs. The latter is optimized for the most common use cases, they exploit the default system compiler and do not support any external optimization tool, whereas low-level APIs allow a more fine grained setup and support split-compilation techniques [26]; hence, the resulting source code is slightly more verbose.

The typical usage of libVC involves different stages. The first task must be the declaration and initialization of the `Version`-independent tools, such as `Compilers` and `Version` builders, which are helper ob-

3 `http://clang.llvm.org/docs/Tooling.html`

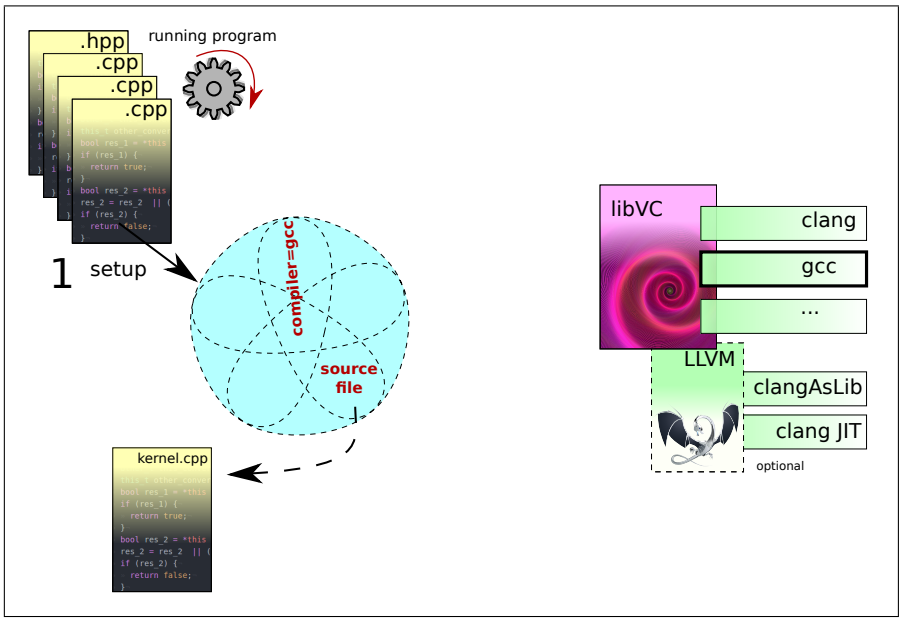


(a) We start from a C++ host application, which is linked against LIBVC.

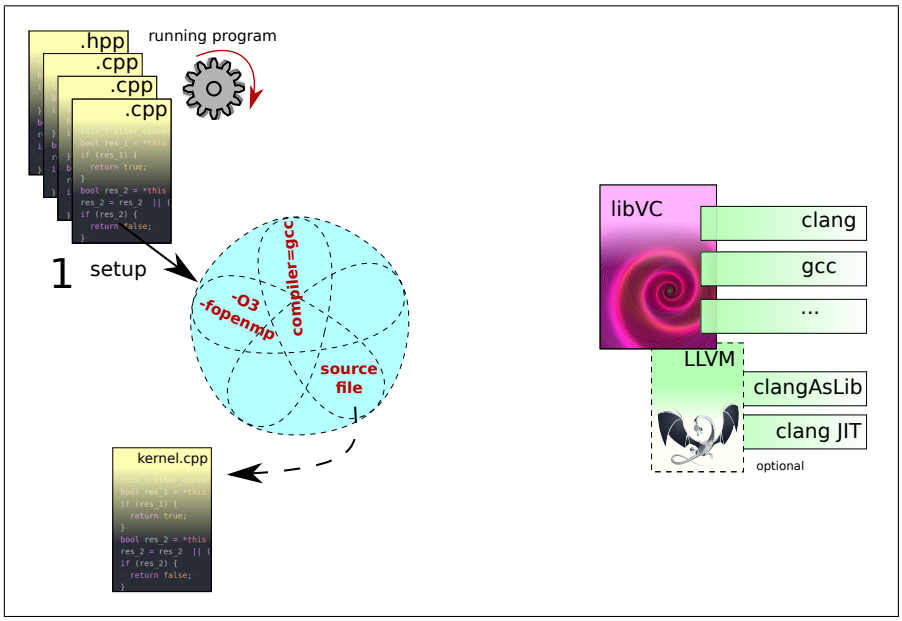


(b) During the setup of a `Version` object, the host application specifies the source file, ...

Figure 4.7: LIBVC configuration and usage steps

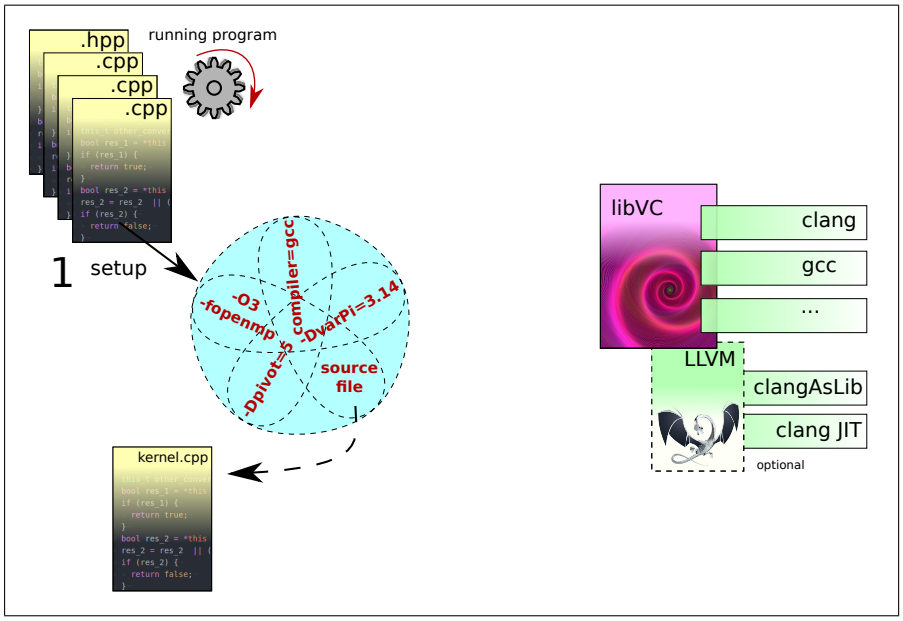


(c) ... the desired compiler implementation, ...

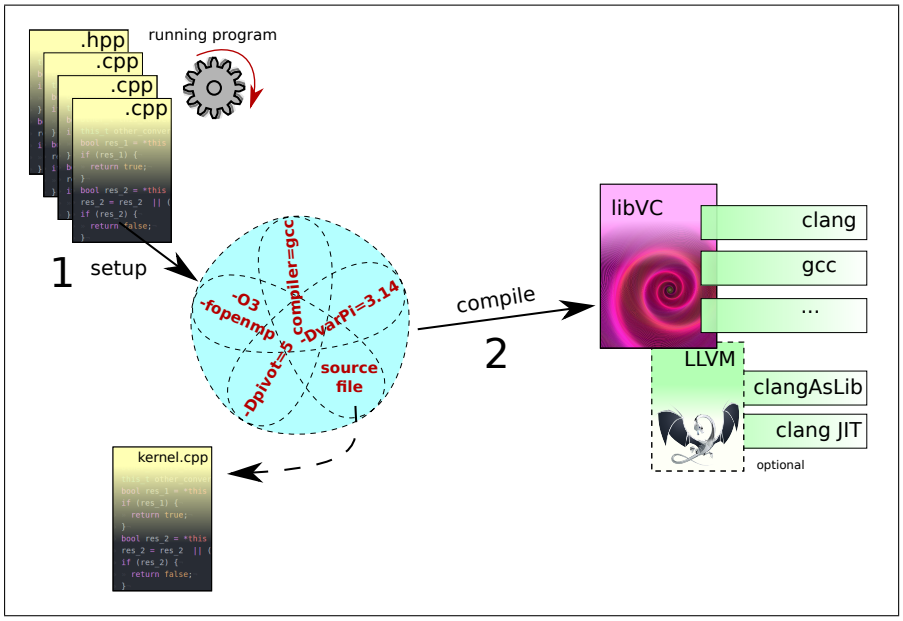


(d) ... the desired compilation options, ...

Figure 4.7: LIBVC configuration and usage steps

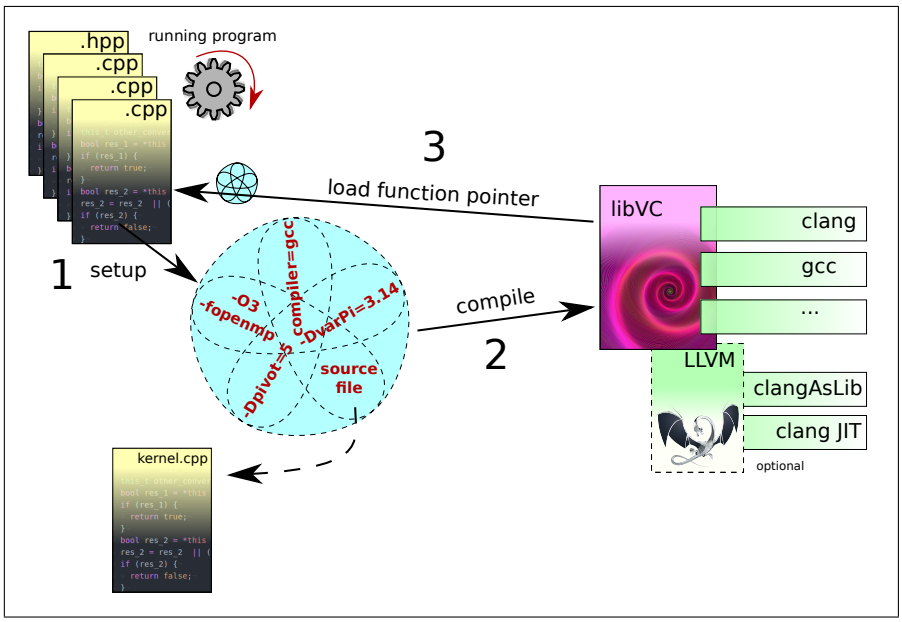


(e) ... such as constants defined at compile time.

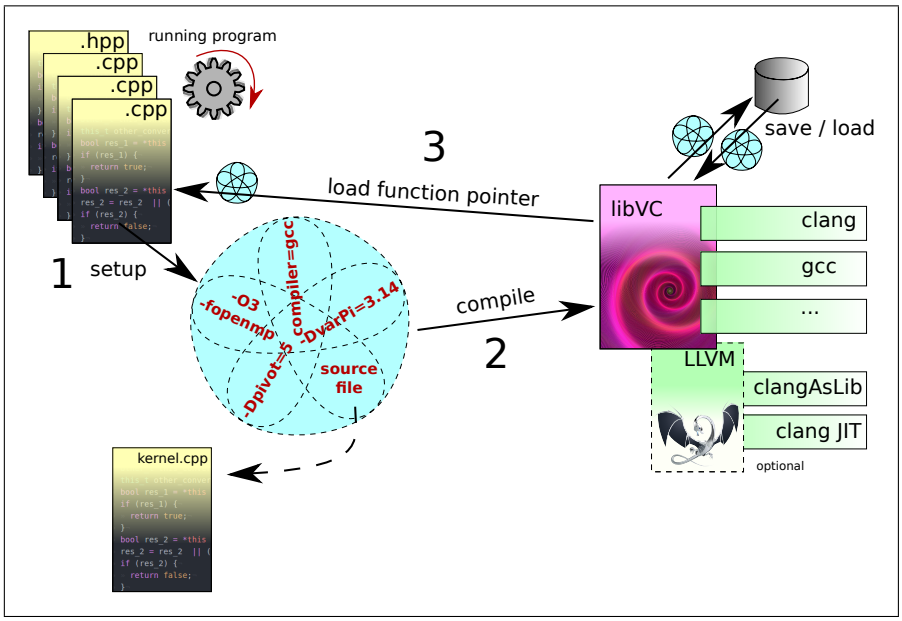


(f) Once the `Version` object has been finalized, the host application explicitly triggers the compilation step via the LIBVC APIs.

Figure 4.7: LIBVC configuration and usage steps



(g) LIBVC provides a function pointer to the host application to invoke the dynamically generated executable code.



(h) The host application can also decide whether and when to offload the already generated `Version` object to the mass memory.

Figure 4.7: LIBVC configuration and usage steps

jects designed to properly setup a `Version` configuration. Low-level APIs allow the programmer to customize one or more `Compiler` implementations. High-level APIs expose a special function to transparently perform this task; it is required to be invoked just once in the whole process lifetime. After that, it is possible to proceed to the `Version` configuration. The programmer can, by using low-level APIs, dynamically forge and arrange `Options`, set the chosen `Compilers`, manipulate file and kernel names to identify the code to be compiled. The `Version` builder is the component which allows this low-level setup. Once the `Version` builder has its fields filled up, it can be finalized to generate a `Version` object. High-level APIs receive all these parameters and produce a `Version` object in a single function call. High-level APIs limit the configuration to one `Version` at a time while low-level APIs allow the parallel configuration of multiple `Versions`. Once a `Version` object is finalized, it has to be compiled. The compilation task is activated by the programmer through a dedicated API. It may trigger more than one sub-task when it involves split-compilation techniques. In the absence of compilation errors, and regardless of which APIs are being used, at the end of this stage libVC generates a binary shared object. From this same shared object libVC loads function pointer symbols, which points to the kernels.

The target kernels may include other files or refer to external symbols. libVC will act just as a compiler invocation and will try to resolve external symbols according to the given compiler and linker options.

libVC defers the resolution of the compilation parameters to runtime. The only piece of information that is needed at design-time is the prototype of the kernel, which has to be used for a proper function pointer cast. Additionally, libVC provides hooks to enable tracking and versioning of the compiled versions.

ILLUSTRATIVE EXAMPLES libVC can be exploited to apply a wide range of optimizations through the dynamic compilation.

In this section we show and discuss a generic use case of continuous program optimization performed through libVC. Listing 4.7 illustrates the dynamic adaptation of a counting sort algorithm to the data workload. In particular, the counting sort implementation is specialized through recompilation using libVC every time the `min` and `max` value of range of the data to be sorted change. When the `min` and `max` values of the range of the data are known at compile-time it is possible to perform array allocation and loop optimizations more efficiently.

Listing 4.7: Benchmark of a statically linked kernel performing counting sort against a dynamically compiled version of the same kernel using libVC high-level APIs

```
1 // libVersioningCompiler High-Level API header file
  #include "versioningCompiler/Utils.hpp"
3
  // define kernel signature
```

```
typedef void (*kernel_t)(std::vector<int32_t> &array);

vc::version_ptr_t getDynamicVersion(int32_t min, int32_t max) {
  // version configuration using libVC - start
  const std::string kernel_dir = PATH_TO_KERNEL;
  const std::string kernel_file = kernel_dir + "kernel.cpp";
  const std::string functionName = "vc_sort";
  const vc::opt_list_t opt_list = {
    vc::make_option("-O3"),
    vc::make_option("-std=c++11"),
    vc::make_option("-I"+kernel_dir),
    vc::make_option("-D_MIN_VALUE_RANGE="+std::to_string(min)),
    vc::make_option("-D_MAX_VALUE_RANGE="+std::to_string(max)),
  };
  vc::version_ptr_t version = vc::createVersion(kernel_file,
      functionName, opt_list);
  // version configuration using libVC - end

  // version compilation - start
  kernel_t f = (kernel_t) vc::compileAndGetSymbol(version);
  if (f) {
    return version;
  }
  // version compilation - end
  return nullptr;
}

int main(int argc, char const *argv[]) {
  const std::vector<std::pair<int, int> > data_range = {
    std::make_pair<int,int>(0,256),
    std::make_pair<int,int>(0,512),
    std::make_pair<int,int>(0,1024),
  };
  const size_t data_size = 1000000000;

  // initialize libVersioningCompiler
  vc::vc_utils_init();

  for (const auto range : data_range) {
    TimeMonitor time_monitor_ref;
    TimeMonitor time_monitor_dyn;
    TimeMonitor time_monitor_ovh;

    // running reference version - statically linked
    for (size_t i = 0; i < iterations; i++) {
      // produce workload to process
      auto wl = WorkloadProducer<int32_t>::get_WL_with_bounds(range.
          first, range.second);
      const auto meta = wl.getMetadata();
      time_monitor_ref.start();
      sort(wl.data, meta.minVal, meta.maxVal); // call reference
      time_monitor_ref.stop();
    }

    // measuring overhead of preparing a new version - start
    time_monitor_ovh.start();
    auto v = getDynamicVersion(range.first, range.second);
    kernel_t my_sort = (kernel_t) v->getSymbol();
    time_monitor_ovh.stop();
    // measuring overhead of preparing a new version - end

    // running dynamic version - dynamically compiled
    for (size_t i = 0; i < iterations; i++) {
      // produce workload to process
      auto wl = WorkloadProducer<int32_t>::get_WL_with_bounds(range.
          first, range.second);
```

```
      time_monitor_dyn.start();
69    my_sort(wl.data); // just a call to a function pointer
      time_monitor_dyn.stop();
71  }

73  // consider average time-to-solution
    std::cout << range.second << " " << time_monitor_ref.getAvg())
75  << " " << time_monitor_dyn.getAvg())
    << " " << time_monitor_ovh.getAvg()) << std::endl;
77 }
  return 0;
79 }
```

Listing 4.7 reveals the several stages of the compilation flow of libVC. In the `main` function, an initialization is needed before using libVC. This is done in line 40 using a simple API invocation. From line 8 to line 20 we see how to configure a new `Version` for dynamic compilation. The following lines (22 - 27) perform the actual dynamic compilation. It is possible to notice in line 69 the call to the dynamically compiled kernel, which is very similar to the call to a statically linked kernel (line 53).

As proof of concept, we tested the benefits of continuous program optimization implemented with libVC by comparing the time-to-solution of the statically linked kernel against a dynamically compiled version of the same kernel, as shown in listing 4.7. We compiled both the statically linked and the dynamically compiled kernels using the same compiler and the same optimization level. A full project using code from listing 4.7 is available on GitHub[4]. We run this example to sort an array of 1 billion 32-bits integers. The platform used to execute the experiment is a supercomputer NUMA node that features two Intel Xeon E5-2630 V3 CPUs (@2.4 GHz) with 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration.

Table 4.1 shows that dynamically compiled kernels always perform better with respect to the reference statically linked implementation. We define as range size the difference between `max` and `min` values of the range of the data to be sorted. We observe an important speedup when the range size is smaller than 8192 possible values. In those cases the main part of the speedup comes from a more efficient memory allocation of the array in the dynamically compiled kernels. We also notice that the overhead of dynamically compiling a new `Version` is not related to the range size. This overhead can be absorbed within 3 iterations when the range size is small, and within less than one thousand iterations in the worst case.

It is also possible to use libVC to dynamically compile and run several functions or the same function with different options. A more complex example of usage of libVC which exploits these features can be found on GitHub[5] where we dynamically compile and run the full PolyBench/C [165] benchmark suite within the same C++ program.

4 https://github.com/skeru/countingsort_libVC
5 https://github.com/skeru/polybench_libVC

Table 4.1: Experimental results of Time-To-Solution (TTS) averaged over 100 executions on a Ubuntu x86_64 system. Kernels were compiled using `gcc` 5.4.0 with optimization level `-O3`.

Range size [elements]	TTS reference [ms]	TTS LIBVC [ms]	speedup [%]	overhead [ms]	payback [iterations]
256	2831.33	2368.12	19.56	1355.99	3
512	2822.84	2352.27	20.00	1345.25	3
1024	2820.67	2347.28	20.17	1356.86	3
2048	2831.92	2351.99	20.41	1361.37	3
4096	2914.13	2440.47	19.41	1353.05	3
8192	3967.59	3966.21	0.03	1354.12	982
16384	5168.64	5163.51	0.10	1370.82	268
32768	6459.75	6430.77	0.45	1358.26	47

4.2.4 *Combining the Continuous Optimization Toolchain*

Given the previously described components, we can build upon them a toolchain to properly perform continuous optimization using dynamic precision tuning. We start from a generic application and we describe step by step the procedure to achieve our goal.

First, we need to clearly identify which are the bounds of the most computationally-intensive application kernel. The application kernel is a function or a set of functions that process data without focusing on I/O operation, error recovery, or corner case handling. Whenever the application does not clearly highlight a set of functions that matches these requirements, it should be refactored to allow the application kernel to be isolated from the rest of the source code of the application. This set of functions will be the target of our optimization.

After the kernel identification, we need to make sure it processes a finite amount of data per iteration. Whenever the application is designed to process a variable amount of data, we recommend to group the input in batches of similar input data. We consider for similarity within the batch similar input properties, common data, spatial and temporal data locality. We thus suggest to rewrite the application to resemble the code pattern shown in Listing 4.8.

Once the shape of the kernel has been defined according to the previous steps, we can focus on its content. We start to characterize the kernel and to design the workload equivalence function. We annotate the source code of the kernel with the information required by TAFFO coherently with the precision requirements of the kernel for each input class. In case of different requirements, for each input class we

Listing 4.8: Iteration over application kernel for each input batch.

```
void kernel(const float* batch) {
  const float pivot = batch[0];
  // ...
}

int main(int argc, char*argv[]) {
  // ...
  while(hasInput()) {
    float* batch = fetchData();
    kernel(batch);
  }
  // ...
}
```

Listing 4.9: Example of parameterized TAFFO annotation of the kernel.

```
// kernel.cpp
#if defined(MIN) AND defined(MAX)
#define PIVOT_RANGE "range("MIN","MAX")"
#else
#define PIVOT_RANGE
#endif

extern "C"
void kernel(const float* batch) {
  float pivot __attribute(annotate(PIVOT_RANGE));
  pivot = batch[0];
  // ...
}
```

suggest to parameterize the annotations, as in the example shown in Listing 4.9.

It is worth of noticing that the refactoring and adjustments applied to the application should have not modified nor the functional nor the extra-functional behavior of the application. Indeed, the TAFFO annotations are simply ignored by the compiler when the TAFFO plugins are not scheduled for execution on the application code.

The first invasive edit on the original application is the introduction of the dynamic compilation of the kernel. To this end we exploit the APIs provided by libVersioningCompiler. In particular, we setup a `Version` object to dynamically compile all the source files required by the kernel of the application – not the whole application – using clang as a compiler and `opt` as a code optimizer.

Later we introduce the TAFFO optimization stages. More specifically, we select a split-compilation approach where the generation of

Listing 4.10: Example of libVC setup to perform dynamic compilation using TAFFO toolchain.

```
// main.cpp
const std::string functionName = "kernel";
const std::string fileName = "../kernel.cpp";
typedef void (*signature_t)(const float*);

vc::compiler_ptr_t taffo = vc::make_compiler<vc::
    TAFFOCompiler>();

vc::version_ptr_t prepareVersion(const float* batch) {
  vc::Version::Builder b;
  b._functionName.push_back(functionName);
  b.addSourceFile(fileName);
  b.addDefine("MIN", batch[0]);
  b.addDefine("MAX", batch[0]);
  b._compiler(taffo);
  vc::version_ptr_t v = b.build();
  return v;
}

int main(int argc, char*argv[]) {

  // ...
  while(hasInput()) {
    float* batch = fetchData();
    vc::version_ptr_t v = prepareVersion(batch);
    v->compile();
    signature_t kernel = (signature_t)v->getSymbol();
    kernel(batch);
  }
  // ...
}
```

the intermediate representation is performed with a minimal set of code optimizations (`-O1`), TAFFO operates on the LLVM-IR, and finally CLANG compiles to shared object the final version using the higher optimization level available (`-O3`). In the example in Listing 4.10 the explicit invocation of the TAFFO optimization stages is encapsulated within the `vc::TAFFOCompiler` compiler implementation, which we designed to simplify user experience.

We replicate the setup of the `Version` object for each input class defined by the workload equivalence function. Ideally every different `Version` should have different precision requirements. With respect to the example in Listing 4.10 we suggest to add memoization techniques to avoid the re-compilation of the kernel multiple times for the same input class. Such techniques can be either applied manually or via automatic tools [139, 147].

Finally, in addition to the precision tuning optimization, we can add to the code generation stage other compile-time options to enable code transformations that are known to be effective for each specific input class. Examples of such code transformation are function specialization via constant propagation or control-flow simplification [21], and approximate computing techniques such as loop perforation or memoization [140]. The effect of such code transformations heavily depends on the input data and on the application kernel itself. A deeper discussion on this topic can be found in dedicated surveys [104, 162]. Therefore, in our work we do not investigate – nor we aim at suggesting – the profitability conditions for any code transformation other than precision tuning.

5 CASE STUDIES

In this chapter we present a series of experimental campaigns that validated the individual tools, and the methodologies presented in Chapter 3. These results have either been published or submitted for publication in peer-reviewed workshops, conferences and journals.

Section 5.1 discusses the evaluation of static precision tuning using a source-to-source translation solution on a set of microkernel benchmarks.

Section 5.2 introduces the exploitation of TAFFO to perform static precision tuning on a limited code base such as the scheduling procedures of a real-time operating system.

Section 5.3 evaluates the TAFFO toolchain for static precision tuning using a well-known approximate computing benchmark suite, which contains applications with pre-defined metrics to measure the quality of the result.

Section 5.4 discuss a few use cases where libVersioningCompiler proved to be effective to perform continuous program optimization. A deeper comparative analysis focusing on the Just-in-Time compilation paradigm in libVC is provided in Appendix a.1.

Section 5.5 provides a deep evaluation of the whole dynamic compilation process using three approximate computing benchmarks.

5.1 IMPLICATIONS OF REDUCED PRECISION COMPUTING IN HPC

The content of this Section has been presented in International Conference on Parallel Computing (ParCo). Bologna, Italy. Sep 2017. [22]

In this section we present the effects of the source-to-source static precision tuning solution described in Section 4.1 in a case study of simple linear-algebra kernels run in an High Performance Computing environment.

As the source-to-source code transformation to enable fixed point representations introduces instruction pattern which are not usually not considered by modern `x86_64` compiler backends, compiler optimization presets – such as the optimization level – may fail to improve the code performance and may instead worsen them.

5.1.1 *Issues with Vectorization*

The `GCC 5.4.0` and `Clang 4.0.0` compilers are not designed to efficiently vectorize kernels for the `x86_64` architecture when the fixed-point conversion is applied. In particular, sign extension and shift operations that are introduced when performing fixed-point multiplications are not handled automatically by the vectorizers.

Listing 5.1: Floating-point SAXPY kernel written in C

```
void saxpy (const float a, float x[], const float y[]) {
  for (int i = 0; i < LEN; ++i) {
    y[i] = a * x[i] + y[i];
  }
  return;
}
```

This problem can be clealy explained using the `saxpy` kernel as an example. Listing 5.1 shows the C version of such kernel. Listing 5.2 shows the fixed point version of the `saxpy` kernel, compiled with `GCC 5.4.0` to assembly code. It is possible to see that several unpacking instructions are generated to perform the shift operation, which the compiler generates as an independent instruction.

However, it is possible to use the `pmulhw` from the MMX vector extension to replace the 16-bit shift, as shown in Listing 5.4. A similar solution can be applied in the case of 32-bit operands: it is possible to replace the 32-bit shift by expressing the multiplication as a sequence of `pmuldq` and `pshufd` instructions.

Since the implementation of the vectorizer is beyond the scope of this work, we did not apply this set of optimizations for the experimental evaluation.

Listing 5.2: Fixed-point SAXPY kernel, compiled with baseline GCC

```
.L34:
  movdqa    (%rdx), %xmm2
  movdqa    %xmm3, %xmm1
  addq      $16, %rax
  addq      $16, %rdx
  pmullw    %xmm2, %xmm1
  movdqa    %xmm1, %xmm0
  pmulhw    %xmm3, %xmm2
  punpckhwd %xmm2, %xmm1
  punpcklwd %xmm2, %xmm0
  psrad     $16, %xmm1
  psrad     $16, %xmm0
  movdqa    %xmm0, %xmm2
  punpcklwd %xmm1, %xmm0
  punpckhwd %xmm1, %xmm2
  movdqa    %xmm0, %xmm1
  punpcklwd %xmm2, %xmm0
  punpckhwd %xmm2, %xmm1
  punpcklwd %xmm1, %xmm0
  paddw     -16(%rax), %xmm0
  movaps    %xmm0, -16(%rax)
  cmpq      %rax, %rcx
  jne       .L34
```

Listing 5.3: Fixed-point SAXPY kernel with unsigned multiplication, compiled with baseline GCC

```
.L34:
  movdqa    (%rdx), %xmm0
  addq      $16, %rax
  addq      $16, %rdx
  pmullw    %xmm1, %xmm0
  paddw     -16(%rax), %xmm0
  movaps    %xmm0, -16(%rax)
  cmpq      %rcx, %rax
  jne       .L34
```

Listing 5.4: Fixed-point SAXPY kernel after post-processing, integer multiplication restored

```
.L34:
  movdqa    (%rdx), %xmm0
  addq      $16, %rax
  addq      $16, %rdx
  pmulhw    %xmm1, %xmm0
  paddw     -16(%rax), %xmm0
  movaps    %xmm0, -16(%rax)
  cmpq      %rcx, %rax
  jne       .L34
```

5.1.2 *Experimental Evaluation*

HARDWARE SETUP The platform used to run the experiments is a NUMA node with two Intel Xeon E5-2630 V3 CPUs (@2.4 – 3.2 GHz Turbo) for a total of 16 cores, with hyper threading enabled and 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration. The selected hardware is therefore representative of modern supercomputer nodes. The operating system is Ubuntu 16.04 with version 4.4.0 of the Linux kernel. We rely on the performance power settings with Turbo Boost activated to effectively drive all of the CPU cores up to 3.2 GHz from the base clock of 2.4 GHz. The compiler in use is GCC 5.4.0.

We collected for each kernel two performance indicators (Time-To-Solution and Energy-To-Solution), as well as the error with respect to the reference version and the instruction mix. Performance measurements are averaged over 100 executions for each same kernel. Time-To-Solution is measured using the `clock()` API from the standard C++ `sys/time.h`. Energy-To-Solution is measured using the Intel RAPL (*Running Average Power Limit*), a set of hardware counters pro-

viding energy and power information. These counters are updated automatically by the hardware. Linux provides an interface to read the counter values. Intel defines a hierarchy of power *domains*, where the top-level domain is the *package*. In our experiment we consider Energy-To-Solution the $\sum_i^{all} Energy_{package,i}$. Note that RAPL does not map energy to processes therefore, $Energy_{package,i}$ represents the energy consumed by the package i as a whole. We used a controlled and unloaded machine for our experiments to guarantee that the energy consumption is due to the benchmarks we run.

We measured the error on the output of each kernel with respect to the highest data precision, which is floating-point quadruple-precision (128 bits). To measure the instruction mix we rely on Intel Software Development Emulator (SDE), a Pin tool [92] that produces instruction traces and classifies them into categories.

BENCHMARKS PolyBench/C is a collection of benchmarks consisting of regular kernels written in C language. We evaluated our approach over a subset of the linear algebra family of PolyBench/C [165]. The subset is chosen based on the ability of the compiler to vectorize the code, which directly impacts the speedups that can be achieved with fixed-point arithmetic. The benchmarks that can be fully vectorized are: `floyd-warshall`, `atax`, `jacobi-1d`, `jacobi-2d`.

`floyd-warshall` Shortest path in a weighted graph

`atax` Matrix Transpose and Vector Multiplication

`jacobi-1d` 1-D Jacobi stencil computation

`jacobi-2d` 2-D Jacobi stencil computation

PolyBench defines five presets of input data sizes for each benchmark: mini, small, medium, large, extralarge. In these experiments, measurements in terms of error, energy, and time are obtained by exploiting the medium data-set size.

The set of benchmarks is characterized by the instruction mix reported in Figure 5.1. The four benchmarks show a variety of floating-point instruction mixes, ranging from `floyd-warshall`, which has almost only additions, to `atax` which has a more balanced mix of additions and multiplications. The `jacobi-1d` and `jacobi-2d` kernels fall in the middle.

ANALYSIS OF RESULTS With the above described setup, we collected for each benchmark measures for the following metrics: time-to-solution, energy-to-solution, and error. We report the time-to-solution and energy-to-solution normalized with respect to the execution of the quadruple-precision floating-point version of each benchmark, which provides the greatest accuracy, but also the slowest time-to-solution

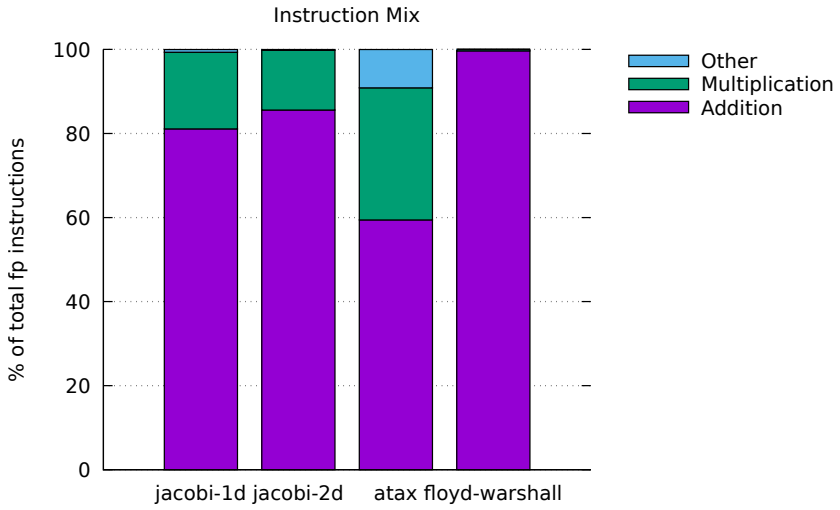


Figure 5.1: Instruction mix for the selected PolyBench benchmarks.

and the largest energy-to-solution. For what concerns the error, we report the *relative solution error* (or relative forward error), defined as follows:

$$\eta = \frac{\|A_{approx} - A\|_F}{\|A\|_F}$$

where $\|A\|_F$ is the Frobenius matrix norm:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{i,j}^2}$$

As it can be seen from Figure 5.2, different benchmarks achieve maximum performance with different approximation solutions.

In particular, for `atax`, we obtain the best performance/error trade-off using 32-bit fixed-point arithmetic, as with 16-bit fixed-point arithmetic it is impossible to find a good compromise between the need to preserve precision for small numbers and the need to provide a sufficiently large number of integer part bits to avoid overflows. On the other hand, 32-bit fixed-point arithmetic provide a major speed-up at only a limited cost in precision.

For `floyd-warshall`, 16-bit fixed-point arithmetic are sufficient for a reasonably good precision, and provide a good boost in performance. The algorithm does not include multiplications, so impacts on both metrics are more limited than in other cases.

For `jacobi-1d` and `jacobi-2d`, 16-bit fixed-point arithmetic is inefficient due to limited opportunities to vectorize. Indeed, the 16-bit fixed-point becomes slower than 32-bit fixed-point arithmetic, as similar operations are used, but more conversion overhead is incurred. 32-bit fixed-point arithmetic, on the other hand, provide a reasonable boost to performance while incurring in reasonable error.

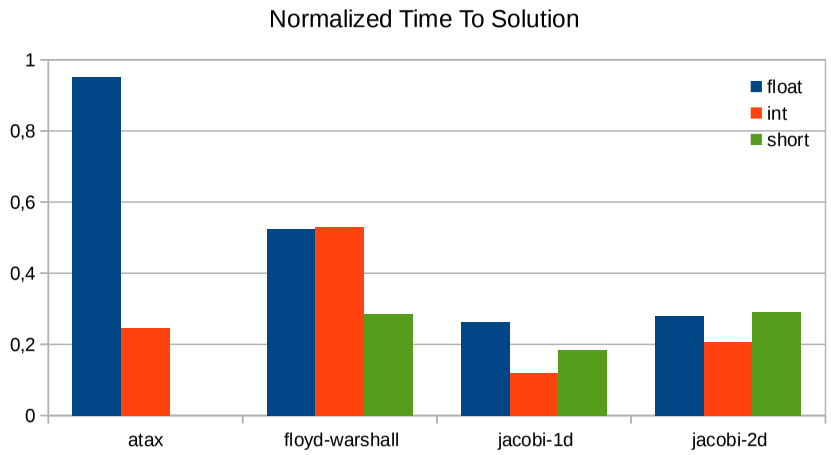


Figure 5.2: Normalized time-to-solution for each benchmark and data type, normalized to the same benchmark with double precision floating-point arithmetic.

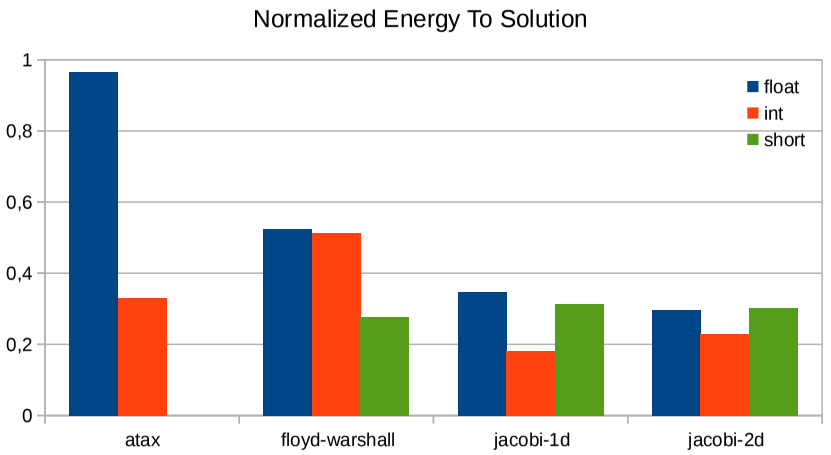


Figure 5.3: Normalized energy-to-solution for each benchmark and data type, normalized to the same benchmark with double precision floating-point arithmetic.

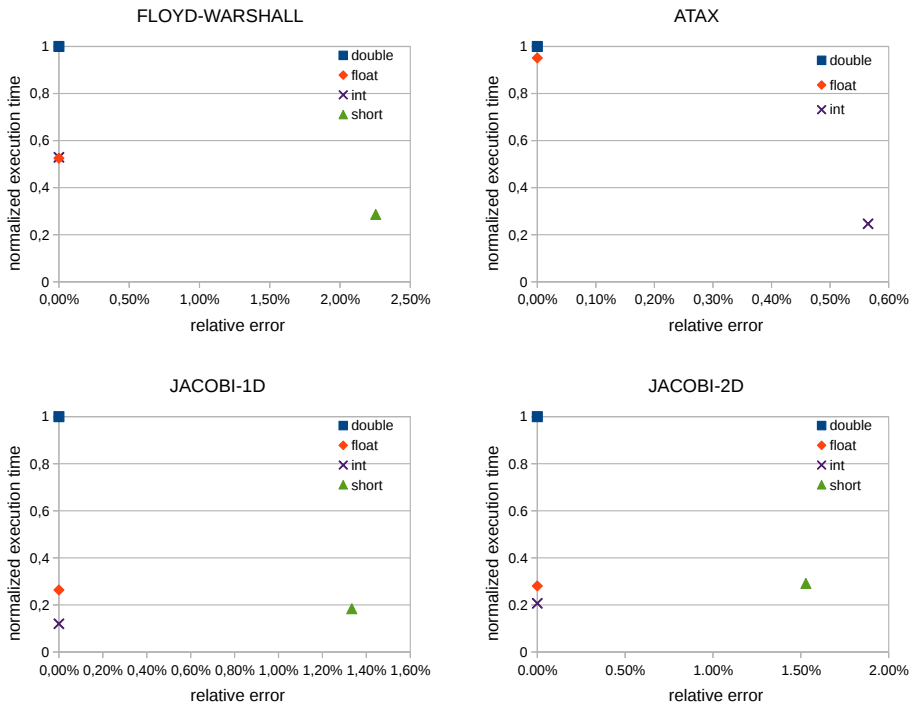


Figure 5.4: Relative solution error vs time-to-solution

It is important to note that the approximation also needs to be taken into account, as with fixed-point, and sometimes also with low-precision floating-point, there is typically a price to pay. The graphs in Figure 5.4 plot the relative solution error against the time-to-solution for each benchmark to highlight Pareto-optimal solutions. The graph shows that, while for `jacobi-1d` and `jacobi-2d` there is a single Pareto-optimal solution (32-bit fixed-point), for `atax` and `floyd-warshall` there are two (single-precision floating-point and either 32-bit or 16-bit fixed-point). Therefore, a solution can be selected based on the target relative solution error.

Figure 5.3 reports the normalized energy-to-solution for each benchmark and data type, normalized to the double precision version. In general, energy to solution is strongly related to time to solution. In most cases, the energy saving is more limited than the performance improvement, though. The only exception is `floyd-warshall`, which differs from the other benchmarks for a distinctly lower use of multiplications. This difference in the instruction mix is reflected in the energy to solution, which is also lower than in other benchmarks when compared to time to solution.

5.2 EMBEDDED OPERATING SYSTEM OPTIMIZATION USING TAFFO

The content of this Section has been presented in Euromicro DSD 2018 *[17].*

The structure of TAFFO is well suited to perform dynamic precision tuning. Before proceeding to evaluate it in this task, we demonstrate its effectiveness in the static case. In this section we discuss the floating to fixed point conversion of the implementation of the scheduler of the MIOSIX[1] embedded real-time operating system. We compare the TAFFO approach with the original floating point implementation, a hand-tuned fixed point one, and a solution based on a C++ library for fixed-point arithmetic.

5.2.1 *About* MIOSIX

MIOSIX is a real time OS targeting embedded system. It has been ported to embedded systems hardware platforms ranging from wireless sensor network nodes to development and evaluation boards. It runs also on wearable devices such as smartwatches, and control boards for experimental sounding rockets of the Skyward program[2]. The kernel is used as a platform for academic research, such as to design clock synchronization solutions [151] and schedulers.

MIOSIX design aims at satisfying three main requirements: hardware variety, software compatibility, and rapid prototyping of new kernel modules. It supports hardware ranging from performance-oriented systems to resource-constrained and real-time systems. MIOSIX provides support for C and C++ standard libraries, and for POSIX threads. Moreover, it is based on a small C++ kernel which provides a proper separation of concerns among its component. The resulting operating system structure is simple enough to allow researchers to quickly implement prototypes of new features and test them on a working system.

In this work we focus on MIOSIX operating system applied to embedded and real-time applications. From an architectural point of view the kernel of MIOSIX is built upon the hardware board support package (BSP). Whilst there are kernel modules that have to interact with the BSP, the scheduler exploits only the APIs provided by the kernel. Three different *pluggable* implementations of the scheduler are shipped with MIOSIX. This design decision allows embedded system developers and researchers to fairly compare different scheduling algorithms by switching among the available implementations at compile-time. For performance reasons, scheduler switching is done using C++ compile-time polymorphism. More schedulers can be implemented and plugged in MIOSIX however, in this work we only consider the available implementations.

1 https://miosix.org
2 https://www.skywarder.eu

Focusing on the scheduler, the kernel supports different scheduling algorithms, one of which is based on control theory [93] and uses floating point operations. This scheduler is the subject of our optimization. The goal of the control-based scheduler is to guarantee on average to each task the CPU share the programmer assigned to it. Scheduling is performed in rounds, at the end of which a control-theoretical regulator is run to compute the next execution time for each task, using the time previously used by tasks to provide feedback.

For each task, a floating point variable `alfa` represents its CPU share set point. Listing 5.5 shows the most important portions of the code of the regulator. Two different operations involve computations on `alfa`. The `IRQrecalculateAlfa()` function is called whenever the CPU distribution requirements change. It partitions the scheduling round among threads. It gives a higher share to those with higher priority whilst it keeps the invariant $\sum_{t \in T} \texttt{alfa}_t = 1$ where T is the set of tasks to be scheduled. The function `IRQrunRegulator()` is instead called by the context switch interrupt service routine, and implements the control theoretical regulator.

To be able to guarantee that no overflows will occur in the fixed point computation, we perform a range analysis of the algorithm. Figure 5.5 shows the analysis result, restricted to the relevant part of the MIOSIX scheduler. Certain variables such as the measured CPU time burst of each tasks have well defined ranges, while for the maximum number of tasks and priorities we set a limit to 64 for the fixed point conversion.

Based on these assumptions it is possible to compute the initial range of values for each of the involved variables.

5.2.2 *Experimental Evaluation*

We compare TAFFO against three other alternatives: a fixed point C++ header library, a manual porting of the algorithms from floating point to integer arithmetic, and the original floating point implementation. In the rest of this section we refer to TAFFO solution as the LLVM *pass* version.

We call *reference* version the implementation that exploits floating point variables and arithmetic. Note that for architectures without hardware floating point support, MIOSIX relies on the default GCC software floating point emulation support.

We also generate a manually ported and optimized version of the original algorithm, using only integer arithmetic. We refer to this optimized version as the *manual* version.

Finally, we also convert the floating point code into fixed point equivalent code by exploiting a template-based C++ library that provides an implementation of both signed and unsigned fixed point data types. This is an open-source library [3] which is designed for ap-

3 source code available at https://github.com/skeru/fixedpoint

Listing 5.5: Relevant portions from the source code of the MIOSIX kernel which implements a I+PI regulator to assign burst times to threads in a preemptive multitasking environment. Notice the usage of floating point variables. This code has been sightly edited for clarity and brevity.

```
void ControlScheduler::IRQrecalculateAlfa()
{
  Thread *it;
  unsigned int sumPriority=0;
  for (it=threadList; it; it=it->schedData.next)
    sumPriority +=
      it->schedData.priority.get() + 1;

  if(sumPriority==0)
    return;

  float base = 1.0 / sumPriority;
  for (it=threadList; it; it=it->schedData.next)
    it->schedData.alfa = base *
      ((float)(it->schedData.priority.get() + 1));

  reinitRegulator=true;
}
/* ... */
static void IRQrunRegulator(bool allThrdsSat)
{
  int eTr=SP_Tr-Tr;

  int bc = bco + (int)(krr*eTr-krr*zrr*eTro);
  if (!allThrdsSat || bco > bc)
    bco=bc;
  bco=min<int>(max(bco, -Tr), bMax*threadListSize);

  float nextRoundTime = Tr + bco;
  eTro=eTr;
  Tr=0;
  Thread *it;
  for (it=threadList; it; it=it->schedData.next) {
    //Recalculate per thread set point
    it->schedData.SP_Tp=
      it->schedData.alfa * nextRoundTime;
    //Run each thread internal regulator
    int eTp =
      it->schedData.SP_Tp - it->schedData.Tp;
    int b =
      it->schedData.bo + eTp;
    //saturation
    it->schedData.bo =
      min(max(b,bMin*multFactor), bMax*multFactor);
    // burst allocated for the task =
    // curInRound->schedData.bo / multFactor
  }
}
/* ... */
static int SP_Tr; // Set point of round time
static int Tr;    // Round time
static int bco;   // Old burst correction
static int eTro;  // Old round time error
/* ... */
class ControlSchedulerData
{
    /*...*/
    ControlSchedulerPriority priority;
    int bo;       // burst time
    float alfa;   // Sum of all alfa=1
    int SP_Tp;    // Processing time set point
    int Tp;       // Real processing time
    Thread *next; // Next thread in list
};
```

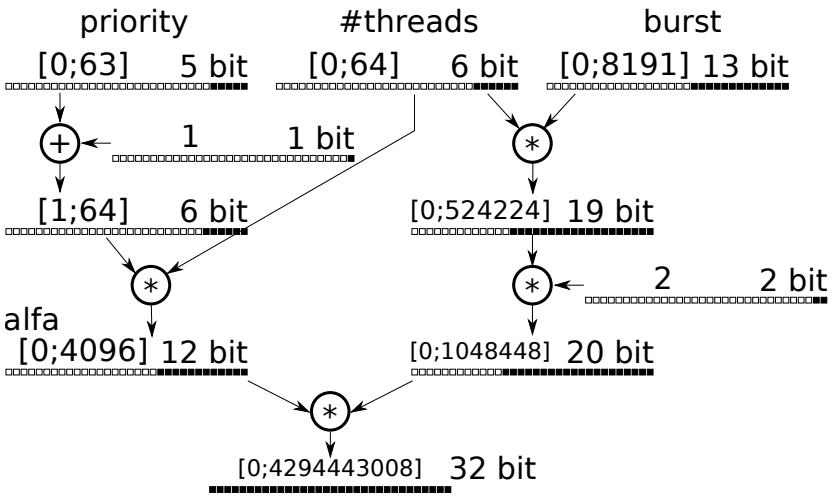


Figure 5.5: Data flow analysis of the floating point values used in a fragment of the Miosix scheduler. For each node we report the range and the minimum data width required.

proximate computing purposes in HPC [22] as part of the ANTAREX project [138, 139]. We refer to this version as the *C++ lib* version.

All the aforementioned conversion approaches require the software developer some effort to be applied. We quantify such effort in terms of newly inserted or modified lines of code (LOCs). The *manual* version requires a complete rework of the scheduler component. It represents the most costly solution in terms of LOCs to be modified. The *C++ lib* solution requires the insertion of 10 LOCs, and the modification of 6 LOCs. The LLVM *pass* solution requires the insertion of attributes near the floating point variables to characterize their initial value range. This procedure costs 9 LOCs and represents the lowest-effort solution. It is worth noting that the LLVM *pass* solution satisfies the *elision* property, i.e. when the conversion pass is not explicitly enabled the source code represents valid C++ code using floating point arithmetic.

HARDWARE SETUP We selected two representative off-the-shelf development boards among those supported by Miosix, one without floating point hardware support, and one with single precision floating point hardware support:

F207 An STM3220G-EVAL board featuring a 120MHz ARM Cortex M3 microcontroller without hardware floating point support. This board has 1 MByte of on-chip flash memory from which code is executed, and 2 MByte of off-chip SRAM used for the kernel and application data.

F469 An STM32F469I-DISCO board featuring a 168MHz ARM Cortex M4 microcontroller which has hardware support for single precision floating point. This board has 2 MByte of on-chip flash memory from which code is executed, and 16 MByte of off-chip SDRAM used for the kernel and application data.

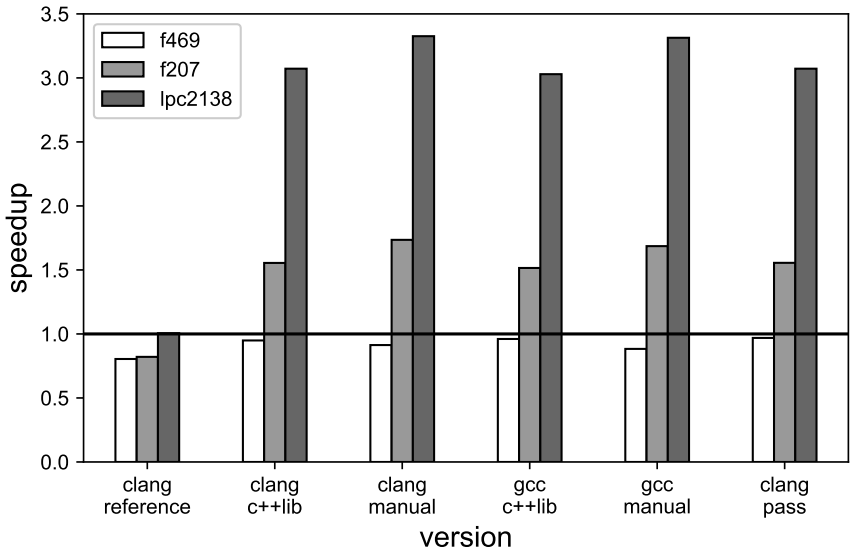
LPC2138 A development board featuring a 59MHz ARM7TDMI microcontroller without hardware floating point support, using the ARM 32 bit instruction set rather than the mixed 16/32 bit instruction encoding Thumb2 instruction set of the other two boards. The board has 512 KB of on-chip flash memory from which code is executed, and 32 KB of on-chip SRAM.

SOFTWARE SETUP For each board we run two series of experiments. We run all of the above mentioned versions of the scheduler with the Hartstone uniprocessor benchmark suite [160] and with benchmarks from the MiBench suite [67]. We rely on the official compiler of the MIOSIX toolchain, which is GCC 4.7.3 with some minor patches to the standard library, and on its default compiler optimization set enabled by the `-O3` optimization level. Our solution compiles only the scheduler component via CLANG 4.0.0 with the same optimization level, and it integrates the compiled scheduler within the original MIOSIX compiler toolchain. To measure the time spent in the scheduling functions that have been affected by the transformation, we added a device driver that interfaces with a hardware timer/counter. The counter was clocked at the maximum frequency possible, corresponding to a timestamping resolution of two CPU clock cycles. To measure the execution time of the whole benchmark, we instead relied on standard C++ library APIs.

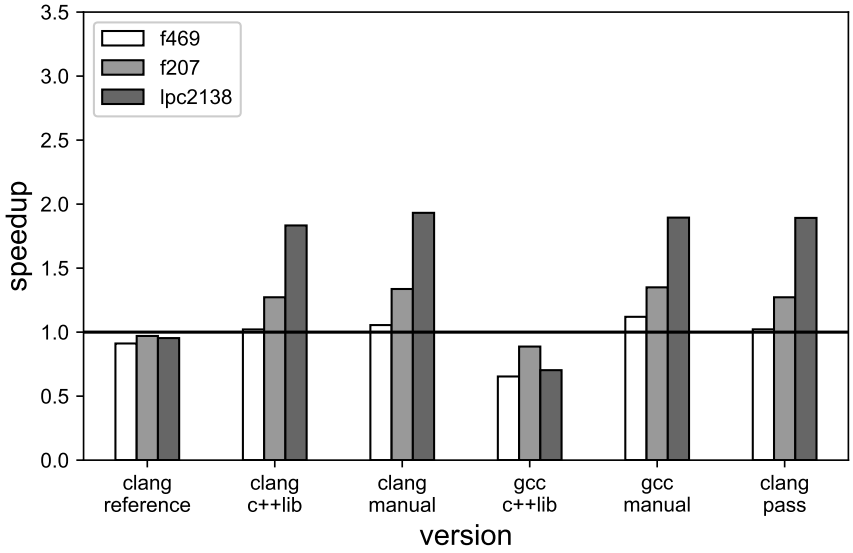
5.2.3 *Result Analysis*

We recall that TAFFO relies on the LLVM compiler framework and it uses CLANG as compiler frontend for the C++ language. However, MIOSIX is ordinarily compiled using a slightly customized GCC toolkit. Thus, in Figure 5.7 we use the *reference* version compiled with GCC as baseline to measure the speedup of the other versions. On the other hand, as GCC and CLANG are different compilers as for design and implementation, we use both of them to compile the other versions to evaluate the performance differences due solely to the compiler.

We find that both CLANG and GCC produce code with negligible performance differences, except in two cases. First, the *reference* version is generally slower when compiled with CLANG than with GCC due to the differences between the architectural model used by LLVM and GCC. Second, the `IRQrecalculateAlfa` function in the *C++ lib* version is slower when compiled with GCC with respect to the same code when it is compiled using CLANG, because we observe different machine-

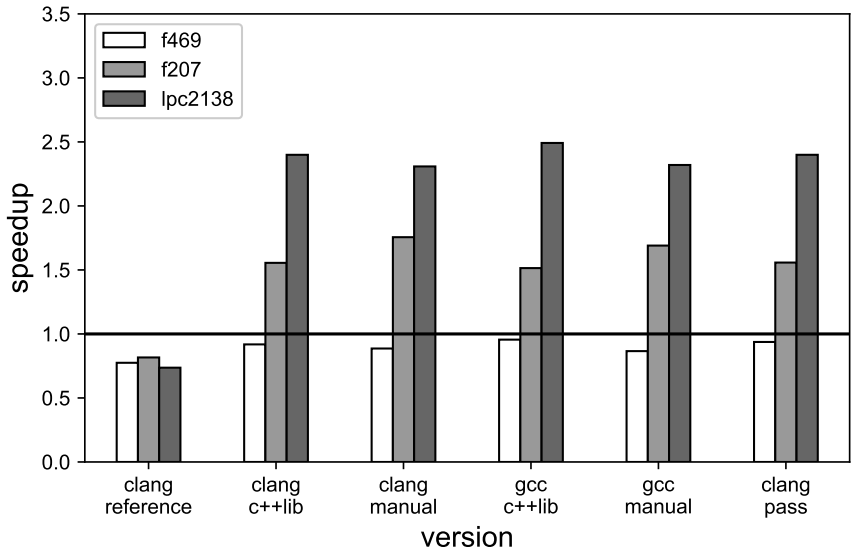


(a) Speedup of `IRQrunRegulator` during the MiBench benchmark.

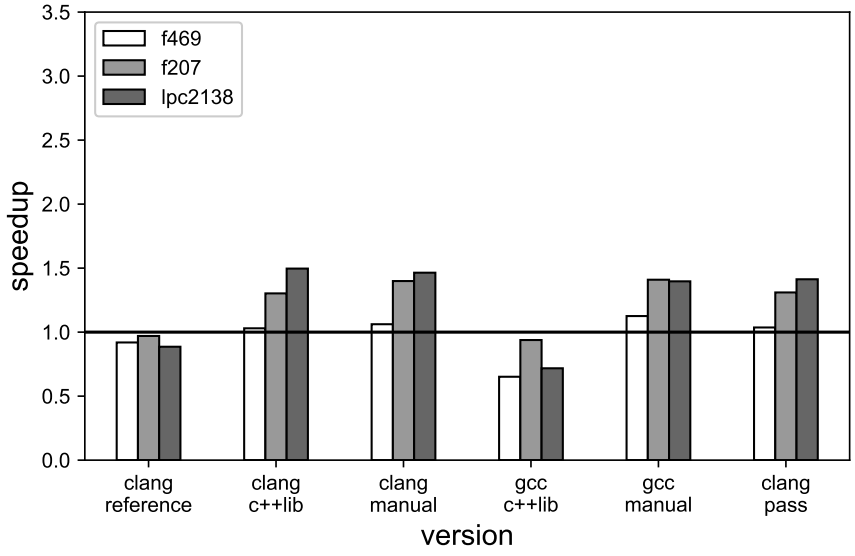


(b) Speedup of `IRQrecalculateAlfa` during the MiBench benchmark.

Figure 5.6: Average speedup of the fixed point versions of the `IRQrunRegulator` and `IRQrecalculateAlfa` methods compared to their *reference* version compiled with GCC, measured during the execution of the MiBench benchmark. The speedup has been computed as the average execution time of one call to each non-reference version, divided by the average execution time of one call to the *reference* version.



(a) Speedup of `IRQrunRegulator` during the Hartstone benchmark.



(b) Speedup of `IRQrecalculateAlfa` during the Hartstone benchmark.

Figure 5.7: Average speedup of the fixed point versions of the `IRQrunRegulator` and `IRQrecalculateAlfa` methods compared to their *reference* version compiled with GCC, measured during the execution of the Hartstone benchmark. The speedup has been computed as the average execution time of one call to each non-reference version, divided by the average execution time of one call to the *reference* version.

independent optimizations. In particular, the *C++ lib* version creates a multiplication with two 64 bit integers operands. Those operands are 32 bit values which have been sign-extended to avoid precision loss. CLANG optimizes this pattern of multiplication to a 32 bit by 32 bit operation with a 64 bit result, whilst GCC does not. Indeed, GCC produces a 64 bit by 32 bit multiplication with a 64 bit result. Thus, the use of CLANG over GCC gives a slight advantage in very specific conditions while it becomes a disadvantage in other conditions. For this reason we show data both for CLANG and GCC.

When it comes to using the execution time as a quality indicator, the most important method to consider is `IRQrunRegulator`, as it runs every scheduling round.The `IRQrecalculateAlfa` method is run only once every workload change. Thus, it has a minor impact on real-world applications.

In Figure 5.6a and in Figure 5.7a we show the speedups measured on `IRQrunRegulator`. Values refer to the average time spent while executing `IRQrunRegulator`. We run each version 10 times and we report the speedup measured on the median value. We observe the speedup achieved by the fixed point representation is consistent across the two sets of benchmarks. We get a slowdown on the `f469` board as that board has support for hardware floating point, and thus we are effectively measuring the overhead intrinsic to fixed point computations. On the `f207` board, we measure a consistent speedup of roughly 1.5 to 1.8 times for all fixed point versions. Among the fixed point implementations, the *C++ lib* version and the *pass* version are equivalent, and they place only sightly below the *manual* version. Finally, the `lpc2138` board is the one which benefits of the highest speedup, roughly up to 3.3 times. This is due to the ARM7TDMI CPU architecture, which features a more limited pipelining with respect to the Cortex architecture.

In Figure 5.6b and in Figure 5.7b we show the speedups measured on `IRQrecalculateAlfa`. Similarly to the previous case, the results are consistent across the two set of benchmarks. On the `f207` and `lpc2138` boards the fixed point versions achieve speedups of roughly 1.2 times (for the `f207` board) and roughly 1.7 times (for the `lpc2138` board). We also achieve small speedups for the `f469` board, up to 1.1 times. The *C++ lib* version compiled with GCC is an outlier because of a missed optimization, as previously discussed. Overall, the *manual* version always sightly exceeds both the *C++ lib* version and the *pass* version, due to inter-procedural optimization performed by hand.

The assembly code generated by the *C++ lib* version compiled with CLANG and the code generated by the LLVM *pass* version are identical for all the boards, except for very small optimizations. Performance-wise, the difference between these two versions is always less than the timer resolution.

Finally, we evaluate the fixed point based solutions over the functional properties of the scheduler being discussed, which must not be invalidated. To this end we compare the quality metrics reported by the Hartstone benchmark, which represent the number of iterations before at least one thread misses a deadline. These indicators are consistent with the *reference* version for every fixed point version.

5.3 STATIC PRECISION TUNING USING TAFFO

The content of this Section has been published in the journal Embedded Systems Letters *[23].*

In this section we evaluate TAFFO on a set of approximate computing benchmark applications.

5.3.1 *Experimental Setup*

We evaluate TAFFO on two different types of hardware: an HPC-like computer architecture and an embedded systems' development board:

AMD a server NUMA node featuring four Six-Core AMD Opteron 8435 CPUs (@2.6 GHz, AMD K10 microarchitecture), with 128 GB of DDR2 memory (@800 MHz);

F207 An STM3220G-EVAL board (**f207**) featuring a 120MHz ARM Cortex M3 microcontroller without hardware floating point support. This board has 1 MByte of on-chip flash memory from which code is executed, and 2 MByte of off-chip SRAM used for the kernel and application data.

We rely on version 6.0 of the LLVM compiler framework, and on `clang` as compiler front-end – aligned with the same version. We collected time measures on the *AMD* node via the `clock_gettime` API of Ubuntu 16.04 LTS operating system, whilst on the *f207* node we exploited the `times` API of the Miosix [90] real-time operating system.

5.3.2 *Benchmarks*

To assess the effectiveness of TAFFO we exploit the set of CPU applications from the AxBench [163] benchmark suite, which is composed of representative error-tolerant applications (a key feature, since we need to objectively assess the precision impact of our transformation). The benchmark suite provides metrics to measure the quality of the result for each application. In particular, *Blackscholes*, *FFT*, and *Inversek2j* use the average relative error (ARE); *Jmeint* uses miss rate (MR); *K-means* and *Sobel* use the root mean square error (RMSE) of the image. We do not consider the *JPEG Encoding* benchmark application as it does not feature any floating-point intensive computational kernel.

All the experiments on the *AMD* node use the largest data set available in the AxBench benchmark suite. Due to hardware memory constraints, on the *f207* node we used the largest data set from AxBench that could fit the memory. In particular, input images for *K-means* and *Sobel* have been scaled down to 256×256 resolution, *Blackscholes* runs on the 10*K* data set, *FFT* uses the 65536 data set, *Inversek2j* uses the 100*K* data set, and *Jmeint* uses the 10*K* data set.

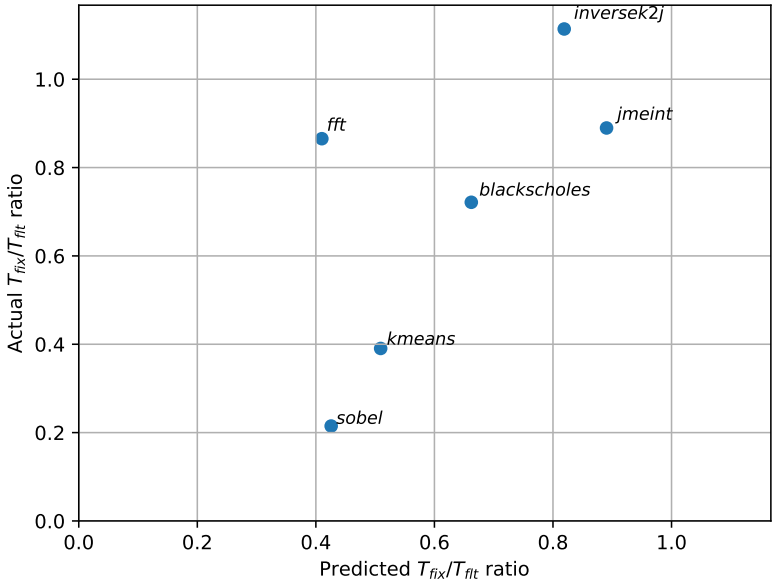


Figure 5.8: Comparison between measured and estimated T_{fix}/T_{flt} ratio using a Bagging ensemble method to boost the accuracy of a K Neighbor regressor.

5.3.3 *Model Construction*

We selected the PolyBench/C benchmark suite [165] as training set for the performance model of the feedback estimator. For each kernel of PolyBench/C we prepare and run three different versions: *vanilla*, *optimized*, and *mixed*. The *vanilla* version exploits only floating point computations using the `binary32` data format from the IEEE-754 standard [71]. The *optimized* version is obtained by converting to fixed point computation the whole kernel via TAFFO framework. Finally, the *mixed* version is obtained by randomly converting a section of the kernel to exploit fixed point computation, and explicitly forcing the rest of the kernel to use floating point computation as in the *vanilla* version. We consider only *mixed* versions that feature at least one type cast in the kernel. For each version we consider a set of features comprising the instruction count per class of instructions, relative to the original float point version. The response metric selected for regressor estimation is the ratio T_{fix}/T_{float} between the execution times of the converted and original version of the code. Based on the outcome of the training and of the test, we select a Bagging ensemble [13] using k-neighbors estimators as the base estimators, which proved more stable than the other candidates.

We validate the selected Bagging ensemble of k-neighbors estimators, trained on the PolyBench/C benchmark suite, on the AxBench benchmarks. Figure 5.8 shows the comparison between prediction

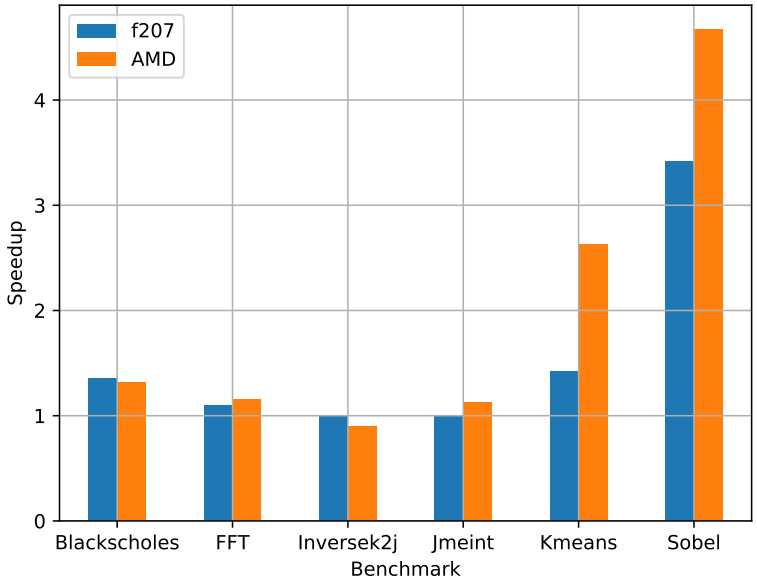


Figure 5.9: Measured speedup (T_{flt}/T_{fix}) of the mixed precision versions over the reference floating point implementation.

and actual values. Although for some benchmarks the prediction is inaccurate (*FFT*, *Inversek2j*), only in one case it leads to an incorrect classification. The regression on the execution time ratio provides a useful tool to understand the effectiveness of the prediction, and may in the future be used to better tune the conversion. We expect the accuracy of the predictor to improve by using real-world applications instead of small kernel benchmarks for model training.

5.3.4 *Result Discussion*

Figure 5.9 shows the measured speedup achieved by the mixed precision versions created by TAFFO with respect to the corresponding floating point reference implementations. The only application that does not benefit from the mixed precision approach is *Inversek2j*, whereas all the other benchmarks show speedups ranging from 12.5% to 366.8% on the HPC *AMD* node. Although the speedup is not as important as in the *AMD* platform, the trend is confirmed also on the embedded system *f207* node – with the exception of *Jmeint*, which has only 0.02% speedup.

Table 5.1 shows the impact of the error introduced by the precision reduction on the output, for all applications. *K-means* and *Sobel* applications do not have a single-value output. We did not show the absolute error for those applications, as it is locally pointless, and it has to be compared on the whole output. In all the other cases, when we compare the absolute error computed on the output against the

Table 5.1: Quality of the result for the mixed precision versions according to the AxBench metrics

Benchmark	$Error_{feedback}$	$Error_{abs}$	$Error_{rel}$	metric
Blackscholes	0.005579455	0.00000006	0.4502%	ARE
FFT	0.079661725	0.02281871	1.2478%	ARE
Inversek2j	0.0005	0.0000485	0.0051%	ARE
Jmeint	0.09673511	0.01654037	0.0118%	MR
K-means	-	-	2.8583%	RMSE
Sobel	-	-	0.0316%	RMSE

error provided by the static feedback estimation, we can observe that the feedback error prediction is always conservative.

5.4 DYNAMIC COMPILATION

The content of this Section has been published in the journal SoftwareX. *[21]*

libVC is a software tool that supports the generation and execution of multiple versions of C++ kernels. This means that libVC allows a wider range of users to adopt continuous optimization practices by generating workload-dependent specializations of one or more kernels. Accordingly, libVC enables the development of autotuning techniques, as well as the comparison of different autotuning algorithms within a neutral platform with any desired compiler. By providing the option to select multiple compilers, libVC can be easily adopted by industrial users, such as supercomputing centers, as they are often constrained to vendor-specific compilers.

libVC is used within the European project ANTAREX [137, 138], which aims at expressing the capability of applications to self-adapt to runtime conditions (we call this practice *autotuning*) through a Domain Specific Language (DSL) and at providing runtime management and autotuning support for applications that target green and heterogeneous HPC systems up to Exascale. The application functionality is expressed through C/C++ code (possibly including legacy code), whereas the non-functional aspects of the application, including parallelization, mapping, and adaptivity strategies are expressed through the DSL developed in the project. The application autotuning is delayed to the runtime phase, where the *software knobs* (application parameters, code transformations and code variants) are configured according to the runtime information that is retrieved from the execution environment. libVC serves to dynamically provide code transformations and code variants in the ANTAREX tool flow. The ANTAREX consortium includes two major European supercomputing centers, as well as industrial users in the automotive and bioinformatics application domains.

5.4.1 *Geometrical Docking Miniapp*

To assess the impact of the proposed tools on a real-world application we employ a miniapp developed within the ANTAREX project [138] to emulate the workload of the geometric approach to molecular docking. This class of application is useful in the in-silico drug-discovery process, which is an emerging application of HPC, and consists in finding the best fitting ligand molecule with a pocket in the target molecule [10]. This process is performed by approximating the chemical interactions with the proximity between atoms.

We processed a database of 113161 ligand molecule-pocket pairs. The platform used to execute the experiment is a supercomputer NUMA node that features two Intel Xeon E5-2630 V3 CPUs (@2.4 GHz) with 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration. The evaluation of every ligand molecule-pocket

pair is independent with respect to the other pairs. Therefore, we implemented an MPI-based version of the same miniapp. The input dataset is partitioned among the slave processes.

The initial code base was developed by another team at Politecnico di Milano. We integrated the code which is executed by each slave process with LIBVC, as for the serial version. It took one hour of work to integrate the miniapp source code with the LIBVC. The integration required to add or modify a total of 60 lines of code over an original code size of 1300 lines of code, which is less than 5% of the code size.

The baseline miniapp took 4354.95 seconds before the integration. After the integration the miniapp took 1783.93 seconds – including the overhead for dynamic compilation – for a speedup of 2.44× with respect to the baseline. The speedup is achieved by exploiting code specialization on geometrical functions.

Although the overhead of performing dynamic compilation on every parallel process slows down the running time, the speedup we obtained in the serial version of the miniapp is confirmed also in the parallel case. We run the MPI-based miniapp using 4, 8, 16, and 32 parallel processes. We obtained a speedup of 2.39×, 2.24×, 1.99×, and 1.63× respectively.

5.4.2 *OpenModelica Compiler*

To assess the impact of the proposed tools on a legacy code we employ the C code which is automatically generated by a state-of-the-art compiler for Modelica. Modelica is a widely-used object-oriented language for modeling and simulation of complex systems. OpenModelica [52] is an open source compiler for the Modelica language. It translates Modelica code into C code, which is later compiled with `clang` and linked against an external equation solver library.

As test case, we simulated a transmission line model [16] of 1000 elements. We modified the C and Makefile code automatically generated by the OpenModelica compiler to integrate the simulation C source code with LIBVC and properly compile it. It took two hours of work to integrate the automatically generated code with the LIBVC. The integration required to add or modify a total of 65 lines of C code and 5 lines of Makefile code over an original code size of 633390 lines of code, which is less than 0.015% of the code size.

The baseline code took 374.25 seconds before the integration. After the integration the simulation took 295.00 seconds – including the overhead for dynamic compilation – for a speedup of 1.27× with respect to the baseline. The speedup is achieved by recompiling the C code which implements the model description by using a deeper optimization level (`-O3`) with respect to the default one (`-O0`). In this case, the compilation time that it is spent on optimizations is widely paid back by a faster execution time

5.5 DYNAMIC PRECISION TUNING USING TAFFO

The content of this Section has not been submitted for publication yet. We thank Daniele Cattaneo and Michele Chiari who helped with the technical development and experimental campaign.

In this section we evaluate TAFFO in the dynamic precision tuning toolchain on a set of approximate computing benchmark applications.

5.5.1 *Benchmarks*

To assess the effectiveness of TAFFO we exploit a subset of CPU applications from the AxBench [163] benchmark suite, which is composed of representative error-tolerant applications. All the experiments use the largest data set available in the AxBench benchmark suite. The benchmark suite provides metrics to measure the quality of the result for each application. A detailed description of the considered benchmarks follows.

K-MEANS This benchmark implements the *K-means* machine learning algorithm to perform clustering on image files. We consider as input batch a single image file and we partition the input batches according to the number of pixels of the image. We thus generate a new version of the mixed precision code for each different image resolution. This benchmark uses the root mean square error (RMSE) of the output image as error metric and we set 1% as acceptable error threshold.

SOBEL This benchmark implements the *Sobel* edge detection filter on image files. As for the previous case, we consider as input batch a single image file and we partition the input batches according to the number of pixels of the image. We thus generate a new version of the mixed precision code for each different image resolution. This benchmark uses the root mean square error (RMSE) of the output image as error metric and we set 1% as acceptable error threshold.

BLACK-SCHOLES This benchmark implements a mathematical model of a financial market. It prices financial options according to their parameters, namely the *call option price* C, the *current stock price* S, the *strike price* K, the *risk-free interest rate* r, the *time to maturity* t, and *volatility* σ. We consider as input batch a single option to be priced and we partition the input batches according to the values of their parameters and relations between them. In particular, we distinguish options with *low volatility* ($\sigma < 0.5$) and *high volatility* ($\sigma \geq 0.5$). A further classification we apply is based on the ratio ρ between *strike price* and *current stock price*. We distinguish between *low ratio* ($\rho < 0.95$), *average ratio* ($0.95 \leq \rho < 1.10$), and *high ratio* (≥ 1.10). This benchmark uses the average relative error (ARE) as error metric and we set 1% as acceptable error threshold.

5.5.2 *Experimental Setup*

We evaluate TAFFO on two HPC-like hardware architectures:

AMD a server NUMA node featuring four Six-Core AMD Opteron 8435 CPUs (@2.6 GHz, AMD K10 microarchitecture), with 128 GB of DDR2 memory (@800 MHz);

INTEL-X a server NUMA node that features two Intel Xeon E5-2630 V3 CPUs (@1.2 GHz) in powersave mode with Intel Turbo Boost disabled, with 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration;

INTEL-I a server node featuring one Intel i3-8350K CPU (@4 GHz), with 32 GB of DDR4 memory (@2.4 GHz).

We rely on version 6.0 of the LLVM compiler framework, and on `clang` as compiler front-end – aligned with the same version. For each benchmark we generate the LLVM-IR, we dynamically compile it with and without using the TAFFO toolchain. The last stage of the dynamic compilation of the LLVM-IR uses the maximum code optimization level (`-O3`).

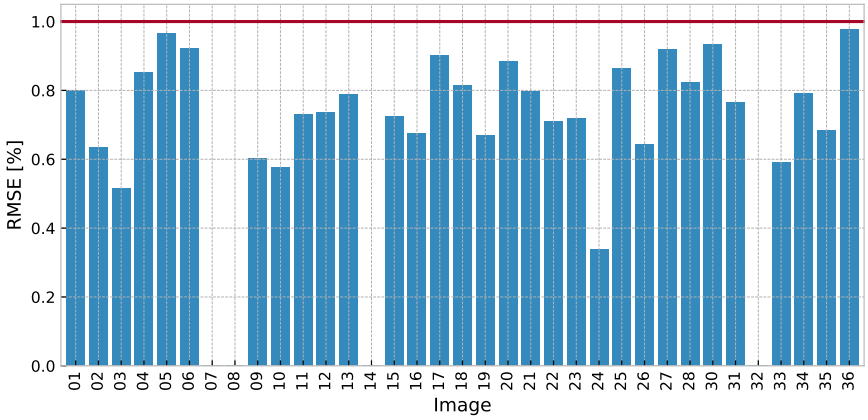
We collect time measures on the via the `clock_gettime` API of the Linux kernel. To limit the system noise we leave the machines unloaded. Furthermore, for each time measure we run the experiment 21 times and we consider only the median value.

5.5.3 *Result Discussion*

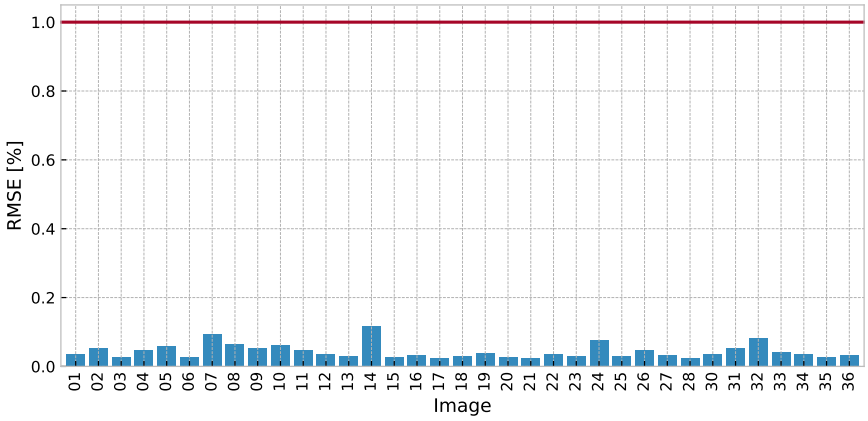
The function specialization allows the compiler to propagate as constants the values which are common to all the elements of the input class, including TAFFO range annotations.

FUNCTIONAL EVALUATION The TAFFO static evaluation of the mixed precision version guarantees that the error is under the 1% threshold for the *Sobel*, and the *K-means* benchmarks – for all input classes – and for the *Blackscholes* benchmark – only in the class with *high volatility* and *average ratio*. According to our approach, whenever the mixed precision version does not guarantee to a sufficient precision, the compiler should use the original version. In the rest of this section we discuss functional and performance evaluation for the *Blackscholes* benchmark limited to the case of the input class that passes the accuracy test. Figure 5.10 shows the *measured error* of the mixed precision versions with respect to the floating point versions. The measured error is always under the threshold.

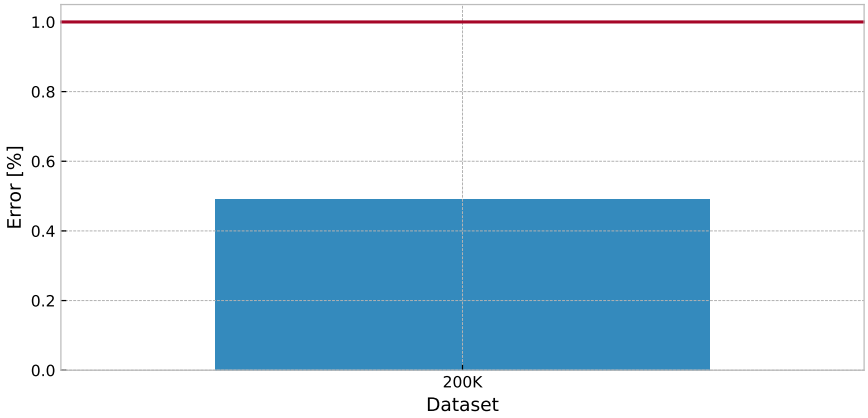
THE CONSTANT PROPAGATION CONTRIBUTION The dynamic compilation approach allows us to combine the constant propagation with the precision reduction approach. We measure the effect of



(a) K-means



(b) Sobel



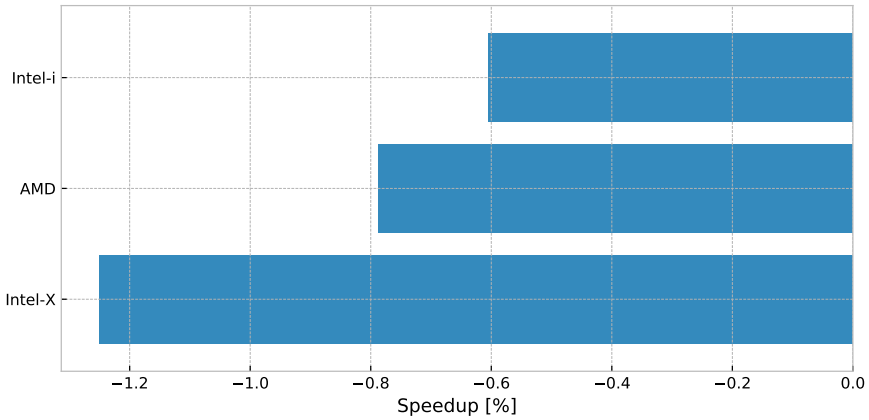
(c) Blackscholes

Figure 5.10: Measured Error of the mixed precision versions over the corresponding floating point implementation for each input batch.

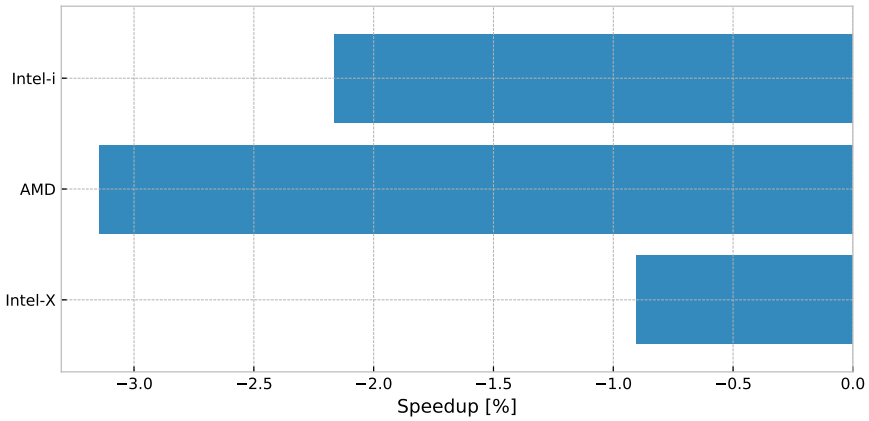
the constant propagation without precision reduction by comparing the dynamically compiled version against with constant propagation against the equivalent original version. Figure 5.11 shows minimal variations which range from 0.14% speedup to 3.3% slowdown. We consider these variations not significant and mostly due to system noise. However, the impact of the constant propagation is evident when applied to the mixed precision version. Figure 5.12 shows that for the *Sobel* benchmark there is a considerable speedup – up to 12% – while for other benchmarks the variations are not relevant, as in the original version. Although the speedup is achieved only in one benchmark, the constant propagation in the mixed precision versions is used also as a mean to propagate TAFFO annotations, which impact on the functional behavior of the applications.

DYNAMIC PRECISION REDUCTION In the rest of this section we compare the dynamically generated reduced precision with constant propagation version against the dynamically generated original precision version with constant propagation. Figure 5.13 shows the speedup achieved by the mixed precision version for each input batch. For the sake of clarity we report the aggregated value for the whole dataset in the case of the *Blackscholes* benchmark. Figure 5.14 summarizes the speedup information by averaging them for every benchmark. We notice impressive speedups on every benchmark – always greater than 100% except for the *Blackscholes* benchmark on the *AMD* architecture – up to more than 400% on the *AMD* architecture. The image processing applications (*K-means* and *Sobel*) have greater benefits on the *AMD* architecture, whilst the *Intel-X* architecture which feature the slower – but also the most floating-point optimized – processor architecture provides the lower speedup.

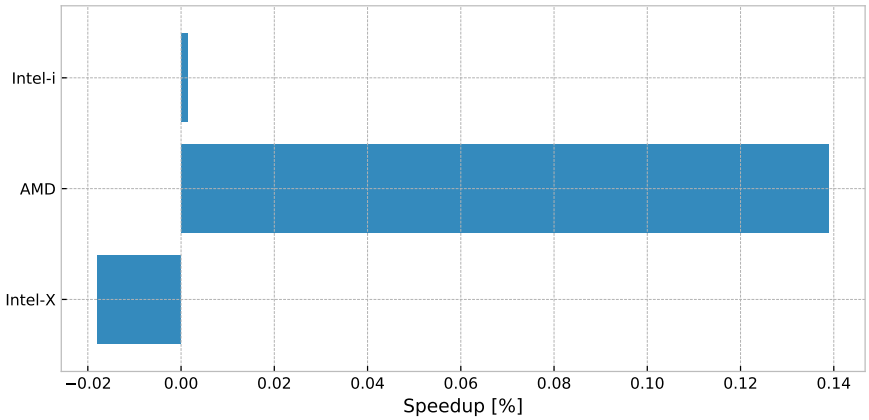
OVERHEAD DISCUSSION The dynamic precision tuning approach we propose implies an overhead of dynamic (re-)compilation that has to be paid every time we want to generate a new version, which means once every input class. We define as payback time the number of input batches that the application have to process absorb the overhead due to the additional compilation stage. In the context of the dynamic precision tuning approach we propose, this parameter represents also the minimal cardinality of each input class to expect a global speedup in the real-world application. We report in Table 5.2 for each benchmark on every architecture, the dynamic compilation time (inclusive of the TAFFO optimization stages), the time the application takes to process a single input batch, and the payback time. We notice that the payback time is similar between the *AMD* and the *Intel-i* architectures, while it is considerably higher on the *Intel-X* architecture. This difference is due to the compiler performance, which is one order of magnitude slower on the latter architecture.



(a) K-means

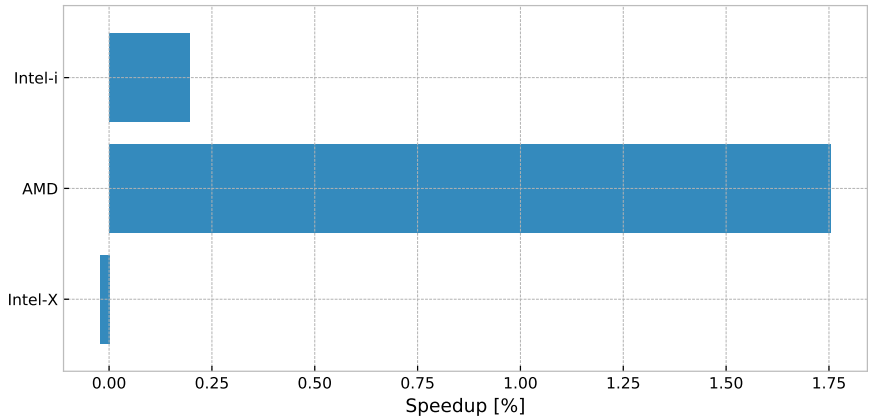


(b) Sobel

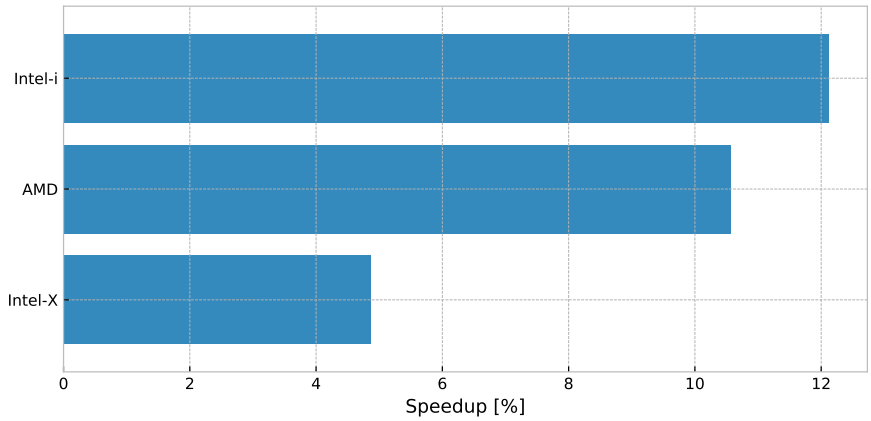


(c) Blackscholes

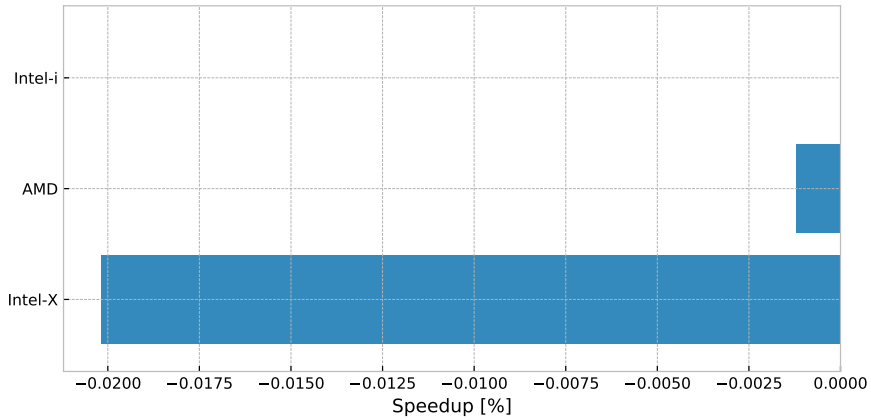
Figure 5.11: Measured speedup of the dynamically compiled floating point versions with constant propagation over the corresponding floating point implementation.



(a) K-means

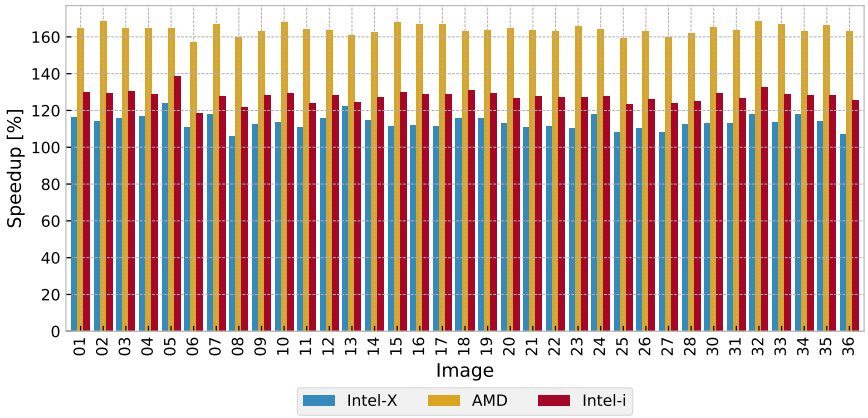


(b) Sobel

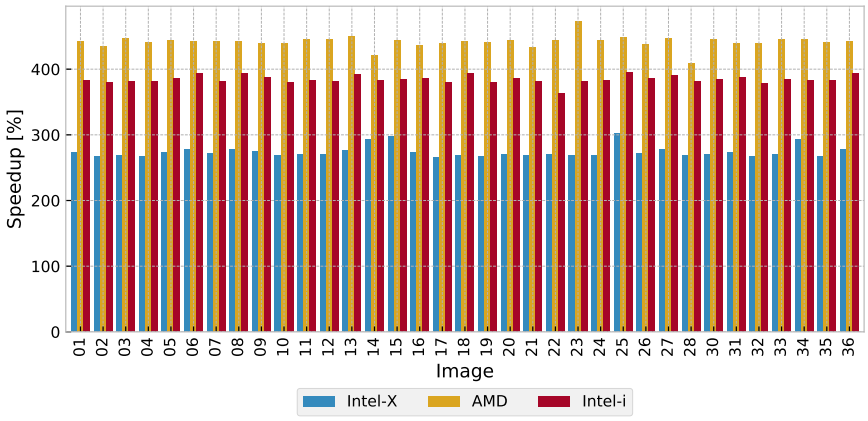


(c) Blackscholes

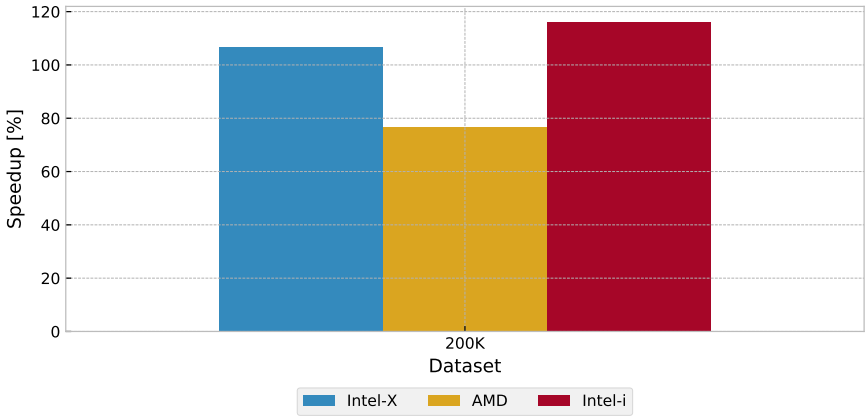
Figure 5.12: Measured speedup of the dynamically compiled mixed precision versions with constant propagation over the corresponding mixed precision implementation.



(a) K-means



(b) Sobel



(c) Blackscholes

Figure 5.13: Measured speedup ($T_{flt}/T_{fix} - 1$) of the mixed precision versions over the corresponding floating point implementation for each input batch.

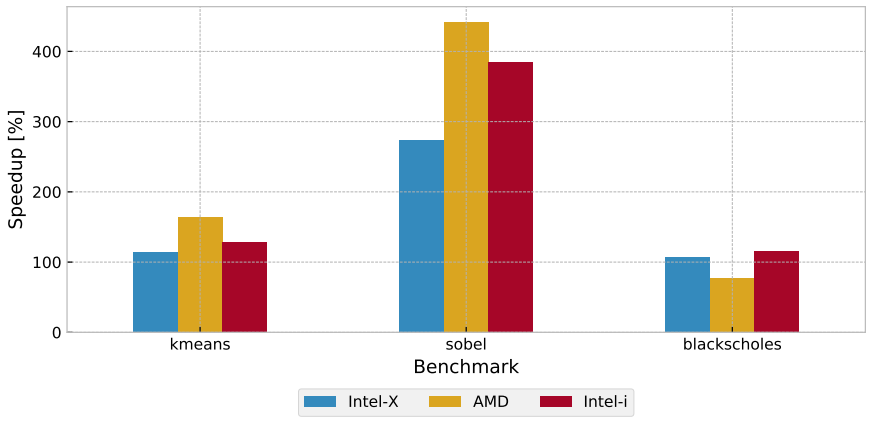


Figure 5.14: Measured speedup ($T_{flt}/T_{fix} - 1$) of the mixed precision versions over the corresponding floating point implementation geometrically averaged over all input batches.

Table 5.2: Dynamic recompilation time overhead.

	compilation time $[ms]$	application time $[ms]$		Payback $[\#batch]$	
K-means					
Intel-X	20005.7343	38.7263		466	
AMD	1274.9577	33.9045		24	
Intel-i	477.1484	10.7302		36	
Sobel					
Intel-X	29085.8714	21.1992		542	
AMD	1792.8637	15.9999		30	
Intel-i	680.7878	5.1892		41	
Blackscholes					
Intel-X	17609.8000	0.2005	$\cdot 10^{-3}$	8.23	$\cdot 10^{7}$
AMD	1132.2300	0.2456	$\cdot 10^{-3}$	5.98	$\cdot 10^{6}$
Intel-i	408.2600	0.0535	$\cdot 10^{-3}$	6.56	$\cdot 10^{6}$

6

CONCLUSIONS

In this thesis we discussed the problem of finding the best finite-precision data type given user-defined accuracy requirements. We presented a survey of tools and techniques to automatize the process of precision tuning process. We considered two different approaches to precision tuning: ahead of time tuning of a single precision mix configuration to be used during the whole program execution, and continuous adaptation of the precision mix configuration during the program execution. We name them respectively static and dynamic precision tuning.

We proposed an effective approach to apply static precision tuning, that is currently the most popular approach in the state of the art. We evaluated two different solutions: one source-to-source toolchain and one compiler-based toolchain. In particular, we re-purposed a tool – *GeCoS* – initially designed for hardware/software co-design, to target general purpose applications. Although we achieved performance improvements on a small set of high performance computing benchmarks, we encountered issues in the compilation process, as the compiler did not properly vectorize fixed point code. Then, we decided to move the code manipulation process from the source level into the compilation process, and we designed, and developed a compiler-based precision tuning solution (TAFFO). We evaluated TAFFO on a suite of high performance computing benchmarks and on a set of error-tolerant applications. The results we achieved highlight two significant improvements over the previous approach. First of all TAFFO allowed us to reduce the precision on complex applications with limited effort. Second, the compiler-based solution was able to leverage the compiler code transformation and optimization capabilities to generate a more efficient machine code. As a side note, we improved also the maintanability of the precision tuning solution by shipping it as a set of plugins for the LLVM compiler framework.

The effective adoption of dynamic precision tuning is still an open issue, for which we designed and suggested a solution from a compiler perspective. In particular, we envisioned, designed, and proposed a solution for this problem. We implemented a toolchain to automatically apply the application-independent components of our solutions and we provided guidelines to effectively implement the application-dependent components. The toolchain is composed of a dynamic compilation component (LIBVC), a compiler-level code manipulation pass (TAFFO), a set of static code analysis and verification passes (also part of TAFFO), and a tuning policy based on input classification. We

separately evaluated libVC and TAFFO on several high performance and embedded systems use cases. We later evaluated the whole dynamic precision tuning toolchain on a set of approximate computing benchmarks.

FUTURE WORKS Our solutions are optimized for the case of user-driven optimizations, which require the user to have domain knowledge on the application being tuned. This scenario is traditionally common in the high performance computing and emmbedded systems domains, where the end user is typically the application developer or someone close to them. However, it is increasingly frequent to differentiate concerns linked to the application development from those linked to the application optimization. We plan to improve the automation of the whole precision tuning approach by implementing a profile-based alternative to the manual annotation step. To this end we envision a profile run of the application kernel which automatically inserts in the source code the proper TAFFO annotations. This profiling phase enters the toolchain just a stage before the generation of LLVM-IR to be parsed by TAFFO both for the static and for the dynamic precision tuning approaches.

Additionally, we aim at improving the initial dynamic precision toolchain from a qualitative perspective. First, we want to improve the value range analysis and feedback estimation components of TAFFO to provide stricter bounds on the runtime values and on the output error, which are over-estimated by the conservative framework. This improvement can be applyed by using more sophisticated static analysis techniques that are able to handle loops.

An ultimate extension to the feedback estimation component would be the integration of the state-of-the-art tools to estimate the throughput of a program within the performance estimation model. An example of such tool is the recently developed LLVM *Machine Code Analyzer* (LLVM-MCA), which seems profitable for our use case as it benefits from the architecture model defined in the LLVM compiler infrastructure. However, LLVM-MCA still presents strong limitations – e.g. data dependencies in cross-loop iterations are not recognized, and the `x86_64` is the only fully supported architecture model. Although these issues prevent us from using LLVM-MCA on the benchmarks we target, we believe that the LLVM community will soon fill these gaps.

BIBLIOGRAPHY

[1] Giovanni Agosta, Stefano Crespi Reghizzi, Paolo Palumbo, and Martino Sykora. "Selective compilation via fast code analysis and bytecode tracing." In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. SAC '06. Dijon, France, 2006, pp. 906–911. ISBN: 1-59593-108-2. DOI: 10.1145/1141277.1141488.

[2] Jean-Marc Alliot, Nicolas Durand, David Gianazza, and Jean-Baptiste Gotteland. "Finding and Proving the Optimum: Cooperative Stochastic and Deterministic Search." In: *Proceedings of the 20th European Conference on Artificial Intelligence*. ECAI'12. Montpellier, France, 2012, pp. 55–60. ISBN: 978-1-61499-097-0. DOI: 10.3233/978-1-61499-098-7-55.

[3] Alexandra Angerd, Erik Sintorn, and Per Stenström. "A Framework for Automated and Controlled Floating-Point Accuracy Reduction in Graphics Applications on GPUs." In: *ACM Trans. Archit. Code Optim.* 14.4 (Dec. 2017), 46:1–46:25. ISSN: 1544-3566. DOI: 10.1145/3151032.

[4] Jason Ansel, Yee L. Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. "Language and compiler support for auto-tuning variable-accuracy algorithms." In: *International Symposium on Code Generation and Optimization (CGO 2011)*. 2011, pp. 85–96. DOI: 10.1109/CGO.2011.5764677.

[5] John Aycock. "A Brief History of Just-in-time." In: *ACM Computing Surveys* 35.2 (2003), pp. 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077.

[6] *BFLOAT16 – Hardware Numerics Definitions*. Tech. rep. Intel Corporation, 2018. URL: https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf.

[7] David H. Bailey. *A thread-safe arbitrary precision computation package (full documentation)*. Tech. rep. 2017.

[8] David H. Bailey, Hida Yozo, Xiaoye S. Li, and Brandon Thompson. *ARPREC: An arbitrary precision computation package*. Tech. rep. 2002.

[9] Protonu Basu, Samuel Williams, Brian Van Straalen, Leonid Oliker, Phillip Colella, and Mary Hall. "Compiler-based code generation and autotuning for geometric multigrid on GPU-accelerated supercomputers." In: *Parallel Computing* 64.Supplement C (2017). High-End Computing for Next-Generation Scientific Discovery, pp. 50 –64. ISSN: 0167-8191. DOI: 10.1016/j.parco.2017.04.002.

[10] Andrea R Beccari, Carlo Cavazzoni, Claudia Beato, and Gabriele Costantino. "LiGen: a high performance workflow for chemistry driven de novo design." In: *Journal of chemical information and modeling* 53.6 (2013), pp. 1518–1527. ISSN: 1549-9596. DOI: 10.1021/ci400078g.

[11] Siegfried Benkner, Franz Franchetti, Hans Michael Gerndt, and Jeffrey K Hollingsworth. "Automatic Application Tuning for HPC Architectures (Dagstuhl Seminar 13401)." In: *Dagstuhl Reports* 3.9 (2014), pp. 214–244. ISSN: 2192-5283. DOI: 10.4230/DagRep.3.9.214.

[12] Sylvie Boldo and Cesar Munoz. *A High-Level Formalization of Floating-Point Number in PVS*. Tech. rep. NASA Langley Research Center, 2006.

[13] Leo Breiman. "Bagging Predictors." In: *Machine Learning* 24.2 (1996), pp. 123–140. ISSN: 1573-0565. DOI: 10.1023/A:1018054314350.

[14] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. "RICH: Automatically Protecting Against Integer-Based Vulnerabilities." In: (Jan. 2007). DOI: 10.1184/R1/6469253.v1".

[15] Bryan Buck and Jeffrey K. Hollingsworth. "An API for Runtime Code Patching." In: *The International Journal of High Performance Computing Applications* 14.4 (2000), pp. 317–329. DOI: 10.1177/109434200001400404.

[16] Francesco Casella. "Simulation of large-scale models in modelica: State of the art and future perspectives." In: *LINKÖPING ELECTRONIC CONFERENCE PROCEEDINGS*. 2015, pp. 459–468.

[17] Daniele Cattaneo, Antonio Di Bello, Stefano Cherubin, Federico Terraneo, and Giovanni Agosta. "Embedded Operating System Optimization through Floating to Fixed Point Compiler Transformation." In: *21st Euromicro Conference on Digital System Design (DSD)*. Vol. 00. Prague, Czech Republic, 2018, pp. 172–176. ISBN: 978-1-5386-7377-5. DOI: 10.1109/DSD.2018.00042.

[18] Howard Chen, Jiwei Lu, Wei-Chung Hsu, and Pen-Chung Yew. "Continuous Adaptive Object-Code Re-optimization Framework." In: *Advances in Computer Systems Architecture*. Ed. by Pen-Chung Yew and Jingling Xue. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 241–255. ISBN: 978-3-540-30102-8. DOI: 10.1007/978-3-540-30102-8_20.

[19] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. "Evaluating Iterative Optimization Across 1000 Datasets." In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. Toronto, Ontario, Canada:

ACM, 2010, pp. 448–459. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806647.

[20] Stefano Cherubin. *fixedpoint*. https://github.com/skeru/fixedpoint. Accessed: 2018-07-04. 2017.

[21] Stefano Cherubin and Giovanni Agosta. "libVersioningCompiler: An easy-to-use library for dynamic generation and invocation of multiple code versions." In: *SoftwareX* 7 (2018), pp. 95–100. ISSN: 2352-7110. DOI: 10.1016/j.softx.2018.03.006.

[22] Stefano Cherubin, Giovanni Agosta, Imane Lasri, Erven Rohou, and Olivier Sentieys. "Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error." In: *Parallel Computing is Everywhere*. Vol. 32: Advances in Parallel Computing. International Conference on Parallel Computing (ParCo), Sep 2017. Bologna, Italy, 2018, pp. 297 –306. ISBN: 978-1-61499-842-6. DOI: 10.3233/978-1-61499-843-3-297.

[23] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. "TAFFO: Tuning Assistant for Floating to Fixed point Optimization." In: *IEEE Embedded Systems Letters* (2019), pp. 1–1. ISSN: 1943-0663. DOI: 10.1109/LES.2019.2913774.

[24] Wei-Fan Chiang, Mark S. Baranowski, Ian Briggs, and Zvonimir Rakamarić. *FPTuner: Rigorous Floating-Point Mixed-Precision Tuner*. https://github.com/soarlab/FPTuner. Accessed: 2018-08-20. 2017.

[25] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. "Rigorous Floating-point Mixed-precision Tuning." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France, 2017, pp. 300–315. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009846.

[26] A. Cohen and E. Rohou. "Processor virtualization and split compilation for heterogeneous multicore embedded systems." In: *Design Automation Conference*. 2010, pp. 102–107. DOI: 10.1145/1837274.1837303.

[27] Jeremy Cohen, Thierry Rayna, and John Darlington. "Understanding Resource Selection Requirements for Computationally Intensive Tasks on Heterogeneous Computing Infrastructure." In: *Economics of Grids, Clouds, Systems, and Services*. Ed. by José Ángel Bañares, Konstantinos Tserpes, and Jörn Altmann. Cham: Springer International Publishing, 2017, pp. 250–262. ISBN: 978-3-319-61920-0. DOI: 10.1007/978-3-319-61920-0_18.

[28] B. Jack Copeland. "The Church-Turing Thesis." In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2019. Metaphysics Research Lab, Stanford University, 2019.

[29] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTRÉE Analyzer." In: *Proceedings of the 14th European Conference on Programming Languages and Systems*. ESOP'05. 2005, pp. 21–30. ISBN: 978-3-540-31987-0. DOI: 10.1007/978-3-540-31987-0_3.

[30] Nasrine Damouche and Matthieu Martel. "Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs." In: *Automated Formal Methods*. Vol. 5. AFM 2017. 2017, pp. 63–76. DOI: 10.29007/j2fd.

[31] Eva Darulova. *Rosa, the real compiler*. https://github.com/malyzajko/rosa. Accessed: 2018-08-21. 2015.

[32] Eva Darulova, Einar Horn, and Saksham Sharma. "Sound Mixed-precision Optimization with Rewriting." In: *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS '18. Porto, Portugal, 2018, pp. 208–219. ISBN: 978-1-5386-5301-2. DOI: 10.1109/ICCPS.2018.00028.

[33] Eva Darulova and Viktor Kuncak. "Trustworthy Numerical Computation in Scala." In: *SIGPLAN Not.* 46.10 (Oct. 2011), pp. 325–344. ISSN: 0362-1340. DOI: 10.1145/2076021.2048094.

[34] Eva Darulova and Viktor Kuncak. "Sound Compilation of Reals." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA, 2014, pp. 235–248. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535874.

[35] Eva Darulova and Viktor Kuncak. "Towards a Compiler for Reals." In: *ACM Trans. Program. Lang. Syst.* 39.2 (Mar. 2017), 8:1–8:28. ISSN: 0164-0925. DOI: 10.1145/3014426.

[36] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. "Synthesis of Fixed-point Programs." In: *Proceedings of the Eleventh ACM International Conference on Embedded Software*. EMSOFT '13. Montreal, Quebec, Canada, 2013, 22:1–22:10. ISBN: 978-1-4799-1443-2.

[37] Eva Darulova et al. *daisy*. https://github.com/malyzajko/daisy. Accessed: 2018-08-21. 2018.

[38] Marc Daumas and Guillaume Melquiond. "Certification of Bounds on Expressions Involving Rounded Operators." In: *ACM Trans. Math. Softw.* 37.1 (Jan. 2010), 2:1–2:20. ISSN: 0098-3500. DOI: 10.1145/1644001.1644003.

[39] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08.

Budapest, Hungary, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0.

[40] Gaël Deest, Tomofumi Yuki, Olivier Sentieys, and Steven Derrien. "Toward Scalable Source Level Accuracy Analysis for Floating-point to Fixed-point Conversion." In: *International Conference on Computer-Aided Design (ICCAD)*. 2014, pp. 726–733.

[41] Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. "Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic." In: *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. 2016, pp. 55–62. DOI: 10.1109/ARITH.2016.31.

[42] Will Dietz, Peng Li, John Regehr, and Vikram Adve. "Understanding Integer Overflow in C/C++." In: *ACM Trans. Softw. Eng. Methodol.* 25.1 (2015), 2:1–2:29. ISSN: 1049-331X. DOI: 10.1145/2743019.

[43] Florent de Dinechin. *FloPoCo*. http://flopoco.gforge.inria.fr. Accessed: 2018-08-27. 2018.

[44] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo." In: *IEEE Design & Test of Computers* 28.4 (2011), pp. 18–27. ISSN: 0740-7475. DOI: 10.1109/MDT.2011.44.

[45] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. "Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal Compiler." In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. Cracow, Poland, 2014, pp. 187–193. ISBN: 978-1-4503-2926-2. DOI: 10.1145/2647508.2647521.

[46] Brian Fahs, Todd Rafacz, Sanjay J. Patel, and Steven S. Lumetta. "Continuous Optimization." In: *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 86–97. ISBN: 0-7695-2270-X. DOI: 10.1109/ISCA.2005.19.

[47] Marco Festa, Nicole Gervasoni, Stefano Cherubin, and Giovanni Agosta. "Continuous Program Optimization via Advanced Dynamic Compilation Techniques." In: *Proceedings of the 10th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 8th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*. PARMA-DITAM '19. Valencia, Spain, 2019, 2:1–2:6. ISBN: 978-1-4503-6321-1. DOI: 10.1145/3310411.3310415.

[48] Luiz Henrique de Figueiredo and Jorge Stolfi. "Affine Arithmetic: Concepts and Applications." In: *Numerical Algorithms* 37.1 (2004), pp. 147–158. ISSN: 1572-9265. DOI: 10.1023/B:NUMA.0000049462.70970.b6.

[49] Goran Flegar, Florian Scheidegger, and Vedran Novakovic. *FloatX*. https://github.com/oprecomp/floatx. Accessed: 2018-08-06. 2018.

[50] Antoine Floc'h et al. "GeCoS: A framework for prototyping custom hardware design flows." In: *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2013, pp. 100–105.

[51] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. "MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding." In: *ACM Trans. Math. Softw.* 33.2 (June 2007). ISSN: 0098-3500. DOI: 10.1145/1236463.1236468.

[52] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. "OpenModelica - A free open-source environment for system modeling, simulation, and teaching." In: *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*. 2006, pp. 1588–1595. DOI: 10.1109/CACSD-CCA-ISIC.2006.4776878.

[53] Grigori Fursin, Anton Lokhmotov, and Ed Plowman. "Collective Knowledge: towards R&D sustainability." In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'16)*. 2016, pp. 864–869.

[54] Davide Gadioli, Gianluca Palermo, and Cristina Silvano. "Application autotuning to support runtime adaptivity in multicore architectures." In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. IEEE. 2015, pp. 173–180.

[55] Altaf Abdul Gaffar, Oskar Mencer, and Wayne Luk. "Unifying bit-width optimisation for fixed-point and floating-point designs." In: *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 2004, pp. 79–88. DOI: 10.1109/FCCM.2004.59.

[56] Sicun Gao, Soonho Kong, and Edmund M. Clarke. "dReal: An SMT Solver for Nonlinear Theories over the Reals." In: *Automated Deduction – CADE-24*. 2013, pp. 208–214. ISBN: 978-3-642-38574-2.

[57] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. 2nd. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002. ISBN: 0133056996.

[58] GitHub Inc. *The state of the Octoverse: Top Languages over Time*. https://octoverse.github.com/projects. Online, accessed Feb 12, 2019.

[59] David Goldberg. "What Every Computer Scientist Should Know About Floating-point Arithmetic." In: *ACM Comput. Surv.* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163.

[60] Frédéric Goualard. *GAOL (Not Just Another Interval Library)*. http://frederic.goualard.net/#research-software. Accessed: 2018-08-26. 2001.

[61] Eric Goubault and Sylvie Putot. "Static Analysis of Numerical Algorithms." In: *Static Analysis*. Ed. by Kwangkeun Yi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 18–34. ISBN: 978-3-540-37758-0.

[62] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. *Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic*. Tech. rep. 2016.

[63] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. "Numerical validation in quadruple precision using stochastic arithmetic." 2018. URL: https://hal.archives-ouvertes.fr/hal-01777397.

[64] Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. "TruffleC: Dynamic Execution of C on a Java Virtual Machine." In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '14. Cracow, Poland, 2014, pp. 17–26. ISBN: 978-1-4503-2926-2. DOI: 10.1145/2647508.2647528.

[65] Hui Guo and Cindy Rubio-González. "Exploiting Community Structure for Floating-point Precision Tuning." In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands, 2018, pp. 333–343. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213862.

[66] Gurobi Company. *Gurobi Optimization*. http://www.gurobi.com. Accessed: 2018-08-26. 2018.

[67] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "MiBench: A free, commercially representative embedded benchmark suite." In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload*

Characterization. WWC-4. 2001, pp. 3–14. ISBN: 0-7803-7315-4. DOI: 10.1109/WWC.2001.990739.

[68] John Harrison. "A Machine-Checked Theory of Floating Point Arithmetic." In: *Theorem Proving in Higher Order Logics*. 1999, pp. 113–130. ISBN: 978-3-540-48256-7. DOI: 10.1007/3-540-48256-3_9.

[69] Laurent Hascoet and Valérie Pascual. "The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification." In: *ACM Trans. Math. Softw.* 39.3 (2013), 20:1–20:43. ISSN: 0098-3500. DOI: 10.1145/2450153.2450158.

[70] IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee. "IEEE Standard for Binary Floating-Point Arithmetic." In: *ANSI/IEEE Std 754-1985* (1985), pp. 1–14. DOI: 10.1109/IEEESTD.1985.82928.

[71] IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee. "IEEE Standard for Floating-Point Arithmetic." In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[72] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 1. 2018. Chap. 8.

[73] Fabienne Jézéquel and Jean-Marie Chesneaux. "CADNA: a library for estimating round-off error propagation." In: *Computer Physics Communications* 178.12 (2008), pp. 933–955. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2008.02.003.

[74] H. Keding, M. Willems, M. Coors, and H. Meyr. "FRIDGE: A Fixed-point Design and Simulation Environment." In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '98. Le Palais des Congrès de Paris, France, 1998, pp. 429–435. ISBN: 0-8186-8359-7.

[75] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. "Fixed-point optimization utility for C and C++ based digital signal processing programs." In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 45.11 (1998), pp. 1455–1464. ISSN: 1057-7130. DOI: 10.1109/82.735357.

[76] Adam B. Kinsman and Nicola Nicolici. "Finite Precision bit-width allocation using SAT-Modulo Theory." In: *2009 Design, Automation Test in Europe Conference Exhibition*. 2009, pp. 1106–1111. DOI: 10.1109/DATE.2009.5090829.

[77] Thomas Kistler and Michael Franz. "Continuous Program Optimization: A Case Study." In: *ACM Trans. Program. Lang. Syst.* 25.4 (July 2003), pp. 500–548. ISSN: 0164-0925. DOI: 10.1145/778559.778562.

[78] Bastian Koller, Nico Struckmann, Jochen Buchholz, and Michael Gienger. "Towards an Environment to Deliver High Performance Computing to Small and Medium Enterprises." In: *Sustained Simulation Performance 2015*. Springer, 2015, pp. 41–50. ISBN: 978-3-319-20340-9.

[79] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. "AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors." In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 47.9 (2000), pp. 840–848. ISSN: 1057-7130. DOI: 10.1109/82.868453.

[80] Michael O. Lam. *CRAFT: Configurable Runtime Analysis for Floating-point Tuning*. https://github.com/crafthpc/craft. Accessed: 2018-11-23. 2018.

[81] Michael O. Lam. *Shadow Value Analysis Library (SHVAL)*. https://github.com/crafthpc/shval. Accessed: 2019-03-27. 2018.

[82] Michael O Lam and Jeffrey K Hollingsworth. "Fine-grained floating-point precision analysis." In: *The International Journal of High Performance Computing Applications* 32.2 (2016), pp. 231–245. DOI: 10.1177/1094342016652462.

[83] Michael O. Lam, Jeffrey K. Hollingsworth, and G. W. Stewart. "Dynamic Floating-point Cancellation Detection." In: *Parallel Computing* 39.3 (2013), pp. 146–155. ISSN: 0167-8191. DOI: 10.1016/j.parco.2012.08.002.

[84] Michael O. Lam and Barry L. Rountree. "Floating-point Shadow Value Analysis." In: *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*. ESPT '16. Salt Lake City, Utah, 2016, pp. 18–25. ISBN: 978-1-5090-3918-0. DOI: 10.1109/ESPT.2016.10.

[85] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. "Automatically Adapting Programs for Mixed-precision Floating-point Computation." In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. Eugene, Oregon, USA, 2013, pp. 369–378. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465018.

[86] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California, 2004, pp. 75–. ISBN: 0-7695-2102-9.

[87] Hong Q. Le, J. A. Van Norstrand, B. W. Thompto, J. E. Moreira, D. Q. Nguyen, D. Hrusecky, M. J. Genden, and M. Kroener. "IBM POWER9 processor core." In: *IBM Journal of Research and*

Development (2018), pp. 1–1. ISSN: 0018-8646. DOI: 10.1147/JRD.2018.2854039.

[88] Dong-U. Lee, Altaf Abdul Gaffar, Ray C. C. Cheung, Oskar Mencer, Wayne Luk, and George A. Constantinides. "Accuracy-Guaranteed Bit-Width Optimization." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.10 (2006), pp. 1990–2000. ISSN: 0278-0070. DOI: 10.1109/TCAD.2006.873887.

[89] JunKyu Lee, Hans Vandierendonck, Mahwish Arif, Gregory D. Peterson, and Dimitrios S. Nikolopoulos. "Energy-Efficient Iterative Refinement using Dynamic Precision." In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2018), pp. 1–14. ISSN: 2156-3357. DOI: 10.1109/JETCAS.2018.2850665.

[90] Alberto Leva, Martina Maggio, Alessandro V. Papadopoulos, and Federico Terraneo. *Control-Based Operating System Design*. Institution of Engineering and Technology, 2013. ISBN: 1849196095, 9781849196093.

[91] Cedric Lichtenau, Steven Carlough, and Silvia M. Mueller. "Quad Precision Floating Point on the IBM z13." In: *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. Vol. 00. 2016, pp. 87–94. DOI: 10.1109/ARITH.2016.26.

[92] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation." In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA, 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034.

[93] Martina Maggio, Federico Terraneo, and Alberto Leva. "Task scheduling: a control-theoretical viewpoint for a general and flexible solution." In: *Transactions on Embedded Computing Systems* 13.4 (2014), pp. 1–22.

[94] Victor Magron, George Constantinides, and Alastair Donaldson. "Certified Roundoff Error Bounds Using Semidefinite Programming." In: *ACM Trans. Math. Softw.* 43.4 (Jan. 2017), 34:1–34:31. ISSN: 0098-3500. DOI: 10.1145/3015465.

[95] Victor Magron and Tillmann Weisser. *NLCertify: a Formal Nonlinear Optimizer: Project Files*. https://forge.ocamlcore.org/frs/?group_id=351. Accessed: 2019-06-07. 2017.

[96] Ramy Medhat. *Shadow Value Analysis Library (SHVAL)*. https://github.com/ramymedhat/shval. Accessed: 2018-11-23. 2017.

[97] Ramy Medhat, Michael O. Lam, Barry L. Rountree, Borzoo Bonakdarpour, and Sebastian Fischmeister. "Managing the Performance/Error Tradeoff of Floating-point Intensive Applications." In: *ACM Trans. Embed. Comput. Syst.* 16.5s (2017), 184:1–184:19. ISSN: 1539-9087. DOI: 10.1145/3126519.

[98] Guillaume Melquiond. https://gforge.inria.fr/projects/gappa/. Accessed: 2018-10-16. 2018.

[99] Daniel Menard, Daniel Chillet, François Charot, and Olivier Sentieys. "Automatic Floating-point to Fixed-point Conversion for DSP Code Generation." In: *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES '02. Grenoble, France, 2002, pp. 270–276. ISBN: 1-58113-575-0. DOI: 10.1145/581630.581674.

[100] Harshitha Menon and Michael O. Lam. *ADAPT: Algorithmic Differentiation for Floating-Point Precision Tuning*. https://github.com/LLNL/adapt-fp. Accessed: 2019-03-27. 2018.

[101] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. "ADAPT: Algorithmic Differentiation Applied to Floating-point Precision Tuning." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. Dallas, Texas, 2018, 48:1–48:13.

[102] Paul S. Miner. *Defining the IEEE-854 Floating-Point Standard in PVS*. Tech. rep. NASA Langley Research Center, 1995.

[103] Asit K Mishra, Rajkishore Barik, and Somnath Paul. "iACT: A software-hardware framework for understanding the scope of approximate computing." In: *Workshop on Approximate Computing Across the System Stack*. WACAS. 2014.

[104] Sparsh Mittal. "A Survey of Techniques for Approximate Computing." In: *ACM Computing Surveys* 48.4 (Mar. 2016), 62:1–62:33. ISSN: 0360-0300. DOI: 10.1145/2893356.

[105] David Monniaux. "The Pitfalls of Verifying Floating-point Computations." In: *ACM Transactions on Programming Languages and Systems* 30.3 (2008), 12:1–12:41. ISSN: 0164-0925. DOI: 10.1145/1353445.1353446.

[106] Ramon E Moore. *Interval analysis*. Vol. 4. Prentice-Hall Englewood Cliffs, NJ, 1966.

[107] Ramon E Moore et al. *Introduction to interval analysis*. Siam, 2009.

[108] Mariano Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. "Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis." In: *Computer Safety, Reliability, and Security*. Springer International Publishing, 2017, pp. 213–229. ISBN: 978-3-319-66266-4.

[109] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. "Definitions and Basic Notions." In: *Handbook of Floating-Point Arithmetic*. 2018, pp. 15–45. ISBN: 978-3-319-76526-6. DOI: 10.1007/978-3-319-76526-6_2.

[110] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2012. ISBN: 161197206X, 9781611972061.

[111] Ricardo Nobre, Luís Reis, João Bispo, Tiago Carvalho, João M. P. Cardoso, Stefano Cherubin, and Giovanni Agosta. "Aspect-Driven Mixed-Precision Tuning Targeting GPUs." In: *Proceedings of the 9th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 7th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*. PARMA-DITAM '18. Manchester, United Kingdom, 2018, pp. 26–31. DOI: 10.1145/3183767.3183776.

[112] Andres Nötzli and Fraser Brown. "LifeJacket: Verifying Precise Floating-point Optimizations in LLVM." In: *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. SOAP 2016. Santa Barbara, CA, USA, 2016, pp. 24–29. ISBN: 978-1-4503-4385-5. DOI: 10.1145/2931021.2931024.

[113] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. "JIT Technology with C/C++: Feedback-directed Dynamic Recompilation for Statically Compiled Languages." In: *ACM Trans. Archit. Code Optim.* 10.4 (Dec. 2013), 59:1–59:25. ISSN: 1544-3566. DOI: 10.1145/2541228.2555315.

[114] William G. Osborne, Ray C. C. Cheung, José G. F. Coutinho, Wayne Luk, and Oskar Mencer. "Automatic Accuracy-Guaranteed Bit-Width Optimization for Fixed and Floating-Point Systems." In: *2007 International Conference on Field Programmable Logic and Applications*. 2007, pp. 617–620. DOI: 10.1109/FPL.2007.4380730.

[115] Sam Owre, John M. Rushby, and Natarajan Shankar. "PVS: A prototype verification system." In: *International Conference on Automated Deduction*. CADE-11. 1992, pp. 748–752. ISBN: 978-3-540-47252-0.

[116] Michael Paleczny, Christopher Vick, and Cliff Click. "The Java™ hotspot Server Compiler." In: *Symposium on Java™ Virtual Machine Research and Technology Symposium*. Vol. 1. JVM'01. Monterey, California, 2001, pp. 1–1.

[117] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. "Automatically Improving Accuracy for Floating Point Expressions." In: *SIGPLAN Not.* 50.6 (2015), pp. 1–11. ISSN: 0362-1340. DOI: 10.1145/2813885.2737959.

[118] Yu Pang and Katarzyna Radecka. "Optimizing imprecise fixed-point arithmetic circuits specified by Taylor Series through Arithmetic Transform." In: *2008 45th ACM/IEEE Design Automation Conference*. 2008, pp. 397–402. DOI: 10.1145/1391469.1391574.

[119] Yu Pang, Katarzyna Radecka, and Zeljko Zilic. "An Efficient Hybrid Engine to Perform Range Analysis and Allocate Integer Bit-widths for Arithmetic Circuits." In: *Proceedings of the 16th Asia and South Pacific Design Automation Conference*. ASPDAC '11. Yokohama, Japan, 2011, pp. 455–460. ISBN: 978-1-4244-7516-2. DOI: 10.1109/ASPDAC.2011.5722233.

[120] Karthick Nagaraj Parashar, Daniel Menard, and Olivier Sentieys. "Accelerated Performance Evaluation of Fixed-Point Systems With Un-Smooth Operations." In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.4 (Apr. 2014), pp. 599–612.

[121] Heejoung Park et al. "Design and Implementation and On-Chip High-Speed Test of SFQ Half-Precision Floating-Point Adders." In: *IEEE Transactions on Applied Superconductivity* 19.3 (2009), pp. 634–639. ISSN: 1051-8223. DOI: 10.1109/TASC.2009.2019070.

[122] Douglass Stott Parker. *Monte Carlo Arithmetic: exploiting randomness in floating-point arithmetic*. Tech. rep. University of California (Los Angeles). Computer Science Department, 1997.

[123] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[124] Pedro Pinto, Tiago Carvalho, João Bispo, and João M. P. Cardoso. "LARA As a Language-independent Aspect-oriented Programming Approach." In: *Proceedings of the Symposium on Applied Computing*. SAC '17. Marrakech, Morocco, 2017, pp. 1623–1630. ISBN: 978-1-4503-4486-9. DOI: 10.1145/3019612.3019749.

[125] Dan Quinlan. "ROSE: COMPILER SUPPORT FOR OBJECT-ORIENTED FRAMEWORKS." In: *Parallel Processing Letters* 10.02n03 (2000), pp. 215–226. DOI: 10.1142/S0129626400000214.

[126] Daniel A. Reed and Jack Dongarra. "Exascale Computing and Big Data." In: *Communications of the ACM* 58.7 (June 2015), pp. 56–68. ISSN: 0001-0782. DOI: 10.1145/2699414.

[127] Manuel Richey and Hossein Saiedian. "A new class of floating-point data formats with applications to 16-bit digital-signal processing systems." In: *IEEE Communications Magazine* 47.7 (2009), pp. 94–101. ISSN: 0163-6804. DOI: 10.1109/MCOM.2009.5183478.

[128] Victor Hugo Rodrigues Raphael Ernani andSperle Campos and Fernando Magno Quintão Pereira. "A fast and low-overhead technique to secure programs against integer overflows." In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2013, pp. 1–11. DOI: 10.1109/CGO.2013.6494996.

[129] Krzysztof Rojek. "Machine learning method for energy reduction by utilizing dynamic mixed precision on GPU-based supercomputers." In: *Concurrency and Computation: Practice and Experience* 0.0 (2018), pp. 1–12. DOI: 10.1002/cpe.4644.

[130] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. "ASAC: Automatic Sensitivity Analysis for Approximate Computing." In: *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES '14. Edinburgh, United Kingdom, 2014, pp. 95–104. ISBN: 978-1-4503-2877-7. DOI: 10.1145/2597809.2597812.

[131] Cindy Rubio-Gonzalez and Cuong Nguyen. *Shadow Execution*. https://github.com/corvette-berkeley/shadow-execution. Accessed: 2019-03-27. 2015.

[132] Cindy Rubio-Gonzalez and Cuong Nguyen. *Precimonious*. https://github.com/corvette-berkeley/precimonious. Accessed: 2018-08-25. 2016.

[133] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. "Precimonious: Tuning Assistant for Floating-point Precision." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado, 2013, 27:1–27:12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503296.

[134] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. "Floating-point Precision Tuning Using Blame Analysis." In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas, 2016, pp. 1074–1085. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884850.

[135] Max Sagebaum, Tim Albring, and Nicolas R. Gauger. "High-Performance Derivative Computations using CoDiPack." In: *CoRR* abs/1709.07229 (2017). arXiv: 1709.07229. URL: http://arxiv.org/abs/1709.07229.

[136] Markus Schordan. *Typeforge*. https://github.com/rose-compiler/rose-develop/tree/master/projects/typeforge. Accessed: 2019-04-01. 2019.

[137] Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R Beccari, Luca Benini, João Bispo, Radim Cmar, João MP Cardoso, Carlo Cavazzoni, Jan Martinovič, et al. "AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems: the ANTAREX Approach." In: *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. DATE '16. Dresden, Germany, 2016, pp. 708–713. ISBN: 978-3-9815370-6-2.

[138] Cristina Silvano, Giovanni Agosta, Stefano Cherubin, Davide Gadioli, Gianluca Palermo, Andrea Bartolini, Luca Benini, Jan Martinovič, Martin Palkovič, Kateřina Slaninová, et al. "The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems." In: *Proceedings of the ACM International Conference on Computing Frontiers*. CF '16. Como, Italy: ACM, 2016, pp. 288–293. ISBN: 978-1-4503-4128-8. DOI: 10.1145/2903150.2903470.

[139] Cristina Silvano et al. "The ANTAREX tool flow for monitoring and autotuning energy efficient HPC systems." In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. SAMOS '17. Pythagorion, Greece, 2017, pp. 308–316. DOI: 10.1109/SAMOS.2017.8344645.

[140] Cristina Silvano et al. "Autotuning and Adaptivity in Energy Efficient HPC Systems: The ANTAREX Toolbox." In: *Proceedings of the 15th ACM International Conference on Computing Frontiers*. CF '18. Ischia, Italy, 2018, pp. 270–275. ISBN: 978-1-4503-5761-6. DOI: 10.1145/3203217.3205338.

[141] Cristina Silvano et al. "The ANTAREX domain specific language for high performance computing." In: *Microprocessors and Microsystems* 68 (2019), pp. 58 –73. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2019.05.005.

[142] N. Simon, D. Menard, and O. Sentieys. "ID.Fix-infrastructure for the design of fixed-point systems." In: *University Booth of the Conference on Design, Automation and Test in Europe (DATE)*. Vol. 38. 2011. URL: http://idfix.gforge.inria.fr.

[143] Alexey Solovyev. *FPTuner: Rigorous Floating-Point Mixed-Precision Tuner*. https://github.com/soarlab/FPTaylor. Accessed: 2019-06-06. 2018.

[144] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. "Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions." In: *FM 2015: Formal Methods*. Ed. by Nikolaj Bjørner and Frank de Boer. 2015, pp. 532–550. ISBN: 978-3-319-19249-9.

[145] Jorge Stolfi and Luiz Henrique de Figueiredo. "An Introduction to Affine Arithmetic." In: *Trends in Applied and Computational Mathematics* 4.3 (2003), pp. 297–312. ISSN: 2179-8451. DOI: 10.5540/tema.2003.04.03.0297.

[146] Sun Microsystems Java team. *The Java™ HotSpot Virtual Machine, v1.4.1*. Tech. rep. 2006.

[147] Arjun Suresh, Erven Rohou, and André Seznec. "Compile-time Function Memoization." In: *Proceedings of the 26th International Conference on Compiler Construction*. CC 2017. Austin, TX, USA, 2017, pp. 45–54. ISBN: 978-1-4503-5233-8. DOI: 10.1145/3033019.3033024.

[148] Giuseppe Tagliavini. *FlexFloat*. https://github.com/oprecomp/flexfloat. Accessed: 2018-08-06. 2018.

[149] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. "A transprecision floating-point platform for ultra-low power computing." In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, pp. 1051–1056. DOI: 10.23919/DATE.2018.8342167.

[150] Michele Tartara and Stefano Crespi Reghizzi. "Continuous Learning of Compiler Heuristics." In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 46:1–46:25. ISSN: 1544-3566. DOI: 10.1145/2400682.2400705.

[151] Federico Terraneo, Alberto Leva, Silvano Seva, Martina Maggio, and Alessandro Vittorio Papadopoulos. "Reverse Flooding: Exploiting Radio Interference for Efficient Propagation Delay Compensation in WSN Clock Synchronization." In: *2015 IEEE Real-Time Systems Symposium*. RTSS. San Antonio, Texas, USA, 2015, pp. 175–184. DOI: 10.1109/RTSS.2015.24.

[152] Laura Titolo, Mariano Moscato, Marco Feliu, and Cesar Muñoz. *PRECISA: Program Round-off Error via Static Analysis*. https://github.com/nasa/PRECiSA. Accessed: 2018-10-09. 2017.

[153] Laura Titolo, Mariano Moscato, Marco Feliu, and Cesar Muñoz. *PRECISA: Program Round-off Error via Static Analysis*. http://precisa.nianet.org. Accessed: 2018-10-09. 2017.

[154] Laura Titolo, Marco A. Feliú, Mariano Moscato, and César A. Muñoz. "An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs." In: *Verification, Model Checking, and Abstract Interpretation*. Springer

International Publishing, 2018, pp. 516–537. ISBN: 978-3-319-73721-8.

[155] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem." In: *Journal of Math* 58 (1936), pp. 345–363. DOI: 10.1112/plms/s2-42.1.230.

[156] V Vasilev, Ph Canal, A Naumann, and P Russo. "Cling – The New Interactive Interpreter for ROOT 6." In: *Journal of Physics: Conference Series* 396.5 (2012), p. 052071.

[157] Vlad Vergu and Eelco Visser. "Specializing a Meta-Interpreter: JIT Compilation of DynSem Specifications on the Graal VM." In: *Proceedings of the 15th International Conf. on Managed Languages & Runtimes*. ManLang '18. Linz, Austria, 2018, 16:1–16:14. ISBN: 978-1-4503-6424-9. DOI: 10.1145/3237009.3237018.

[158] Joannès Vermorel and Mehryar Mohri. "Multi-armed Bandit Algorithms and Empirical Evaluation." In: *Machine Learning: ECML 2005*. 2005, pp. 437–448. ISBN: 978-3-540-31692-3.

[159] Jean Vignes. "Discrete Stochastic Arithmetic for Validating Results of Numerical Software." In: *Numerical Algorithms* 37.1 (2004), pp. 377–390. ISSN: 1572-9265. DOI: 10.1023/B:NUMA.0000049483.75679.ce.

[160] Nelson H. Weiderman and Nick I. Kamenoff. "Hartstone Uniprocessor Benchmark: Definitions and experiments for real-time systems." In: *Real-Time Systems* 4.4 (1992), pp. 353–382. ISSN: 1573-1383. DOI: 10.1007/BF00355299.

[161] Robert P. Wilson et al. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers." In: *ACM SIGPLAN Notices* 29.12 (Dec. 1994), pp. 31–37. ISSN: 0362-1340. DOI: 10.1145/193209.193217.

[162] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. "Approximate Computing: A Survey." In: *IEEE Design Test* 33.1 (2016), pp. 8–22. ISSN: 2168-2356. DOI: 10.1109/MDAT.2015.2505723.

[163] Amir Yazdanbakhsh et al. "AxBench: A Multiplatform Benchmark Suite for Approximate Computing." In: *IEEE Design Test* 34.2 (2017), pp. 60–68. ISSN: 2168-2356. DOI: 10.1109/MDAT.2016.2630270.

[164] Serif Yesil, Ismail Akturk, and Ulya R. Karpuzcu. "Toward Dynamic Precision Scaling." In: *IEEE Micro* 38.4 (2018), pp. 30–39. ISSN: 0272-1732. DOI: 10.1109/MM.2018.043191123.

[165] Tomofumi Yuki. "Understanding PolyBench/C 3.2 kernels." In: *International workshop on Polyhedral Compilation Techniques (IMPACT)*. 2014.

[166] Universite de Versailles St-Quentin-en Yvelines. https://github.com/verificarlo/verificarlo. Accessed: 2019-06-07. 2018.

[167] Andreas Zeller and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input." In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200. ISSN: 0098-5589. DOI: 10.1109/32.988498.

[168] W. Ziegler, R. D'Ippolito, M. D'Auria, J. Berends, M. Nelissen, and R. Diaz. "Implementing a "one-stop-shop" providing SMEs with integrated HPC simulation resources using Fortissimo resources." In: *eChallenges e-2014 Conference Proceedings*. 2014, pp. 1–11.

A

DYNAMIC COMPILATION INSIGHTS

A.1 ADDING JIT COMPILATION TO LIBVC

The content of this Section has been presented in PARMA-DITAM '19 workshop. Valencia, Spain. Jan 2019. [47]

Dynamic compilation techniques allow to compile source code into executable code after the software itself has been deployed. Whenever important runtime conditions or checkpoints are met, a dynamic re-configuration of the software system may entail a dynamic (re-)compilation of its source code to apply a different set of optimization which better fit the incoming workload. Dynamic compilation can be performed via ad-hoc software hypervisors, via dynamic generation and loading of software libraries, or via the integration of a compiler stack within the adaptive software system itself.

The former approach has difficulty in maintaining both the hypervisor and the code to use it. This approach requires a deep knowledge of the hypervisor system, and an accurate configuration over the software system.

The dynamic generation and loading of software libraries are platform-dependent solutions that requires fine tuning of the compiler configuration at deploy time. Moreover, this approach requires to access and to manage additional persistent memory space to handle the dynamically-generated shared objects. This problem has a non-trivial impact on HPC infrastructures, as such systems usually aims at minimizing the access to persistent memory due to its intrinsic high-latency. Although LIBVC APIs [21] simplify the configuration of compiler settings, the limitation given by the memory access of a compiler invokation still persists. The JIT paradigm does not suffer from these problems since it does not create any persistent object and it integrates the full compiler functionalities within the adaptive application.

PRINCIPLES OF JUST-IN-TIME COMPILATION In JIT compilation, a fragment of code (usually a function or method) is only compiled when it is first executed. The main issue with JIT compilation is that, at each new function encountered, a compilation latency is experienced, leading to potentially large start-up times.

JIT compilers combine some properties of static compilation, typically the performance of the generated code, with other properties which are typical of interpreters, such as the ability to leverage runtime knowledge about the program (runtime constant, control flow) or portability. In the context of continuous optimization, JIT compilation is a key enabler: it replaces compiled code fragments with new versions tuned to different parameters rather than recompiling the

entire program. It is worth noting that, while a JIT generally does not preserve the compiler code, this is not a limitation in the context of HPC systems. Here, a given application may run over a long time, but may be invoked only a few times, thus making the persistance of the optimized code less relevant.

A.1.1 *Providing JIT APIs via LLVM*

In this work, we leverage the LIBVC framework, which provides APIs to enable dynamic compilation and re-use of code versions. It features three different implementation of compiler APIs: `SystemCompiler`, `SystemCompilerOptimizer`, and `ClangLibCompiler`.

The first two solutions need to be configured to properly interact with external compilers already deployed on the host machine. The latter implements the *Compiler as a library* paradigm, and therefore needs LLVM to be installed in the host machine. We extend LIBVC with `JITCompiler`, an additional implementation for the `Compiler` interface, with the aim of providing true JIT compilation capabilities.

To pursue continuity with the original implementation of LIBVC, we base our implementation of `JITCompiler` on the LLVM compiler framework. In particular, we support the generation of LLVM-IR bitcode files, the optimization of such intermediate representation, and the compilation of it into executable code. Whilst the bitcode generation and optimization are implemented similarly to the *Clang as a library* paradigm, the generation of executable code is extremely different and it represents the core of the `JITCompiler` approach. To achieve such improvement we leverage LLVM's On-Request-Compilation (ORC) APIs. Our new LIBVC compiler implementation allows us to parse the bitcode files and to elevate them to their in-memory representation. The freshly imported LLVM Module objects are then passed as arguments to the core LLVM ORC API handle `addModule`, which actually starts the JIT compilation process. Similar APIs allow us to fetch one or more symbols from the jitted code and use them in the same way LIBVC provides function pointers to the host code, so they will be able to access the dynamically compiled version of the code via indirect function calls. Like the other compiler implementations defined in LIBVC, we provide a mechanism to control the memory footprint of the dynamic compilation framework by exposing knobs to offload and to reload from the memory the dynamically compiled modules. This approach is fully compliant with the state-of-the-art LIBVC paradigm to perform continuous optimization.

A.1.2 *Evaluation*

We evaluate our implementation of the LLVM-based JIT compiler by comparing its capabilities with the alternatives which are already

available within LIBVC [21]. Our solution exposes the same APIs as the other compiler implementations within LIBVC. Thus, the usability of this compiler, and its fitness to continuous optimization use cases corresponds to those of its alternatives. In this section, we focus on two main performance indicators, namely **code quality** and **compilation time**. To this end, we compare our solution against the other LIBVC compiler implementations over the well-known PolyBench/C benchmark suite [165]. We use this benchmark suite as a proxy to validate the effectiveness of our approach as it is particularly representative of the typical HPC workload. In fact, it contains kernels from several application domains, such as linear algebra, data mining, and image processing, which are the core emerging areas in the HPC world.

The platform to run the experiments is representative of modern supercomputer nodes. The selected hardware is a NUMA node with two Intel Xeon E5-2630 V3 CPUs (@3.2 GHz) for a total of 16 cores, with hyper threading enabled and 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration. The operating system is Ubuntu 16.04 with version 4.4.0 of the Linux kernel. The experiments run with the machine completely unloaded from other user processes.

We base our tests on the official LIBVC testing setup for PolyBench/C[1]. We configured LIBVC to use the following compilers:

SYSTEMCOMPILER default host compiler (`gcc` version 5.4.0)

SYSTEMCOMPILEROPTIMIZER `clang` and `opt`, version 6.0.1

CLANGLIB `libclang` version 6.0.1

JITCOMPILER our solution, based on LLVM version 6.0.1

We configure each compiler to run with the same set of compiler options, i.e. we specify only the optimization level `-O3`. We rely on the MEDIUM dataset size, and on the default initialization routines which are defined by the PolyBench/C benchmark suite for each kernel. All kernels use the IEEE-754 floating point double precision data type for the computation.

CODE QUALITY We expect to have the same code quality given by the same compiler with the same compiler options. To verify this assertion we compile and run the PolyBench/C benchmarks and we report in Figure a.2 runtime and compilation time for each benchmark. The run time of the different code versions generated by LIBVC compilers is not significantly influenced by the chosen compiler alternative. On the contrary, this choice becomes relevant for the compilation time. Figure a.1 bar chart highlights how the `JITCompiler` compiler performance outstands the other clang-based alternatives. This is confirmed

1 PolyBench/C benchmark suite integrated with LIBVC https://github.com/skeru/polybench_libVC

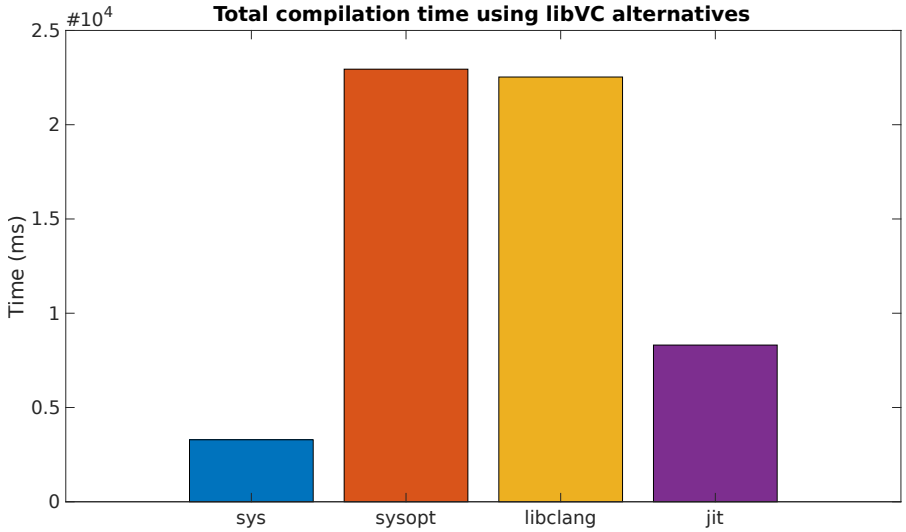


Figure a.1: Compilation time of each kernel of the PolyBench/C benchmark suite with different implementation alternatives from libVC.

by every benchmark in the polybench suite on IEEE-754 double precision type data. Thus, the code quality is not significantly influenced by the compiler choice.

COMPILATION TIME The main overhead that applies to continous program optimization via dynamic compilation is given by the compilation time. As the code quality is not influenced by the chosen compiler, the conditions that trigger a re-compilation task do not change. The compilation time of the single kernel is not sufficiently significative per se. Thus, we compare the compilation time of each compiler implementation when it is asked to compile the whole set of PolyBench/C benchmarks. Figure a.1 underlines the improved performance of the `JITCompiler` when compared with other clang-based compilers. The `SystemCompiler` is based instead on the `gcc` compiler technology, which performs better in terms of compilation time. Figure a.2 confirms that this trend holds for each kernel.

a.1.3 *Other JIT implementations*

The history of JIT compilation is almost as old as computer science. Aycook summarizes early works on JIT compilation techniques starting from the 1960s in his survey [5]. In modern times, the Sun Microsystems Java HotSpot Virtual Machine [146] started to extensively use JIT technologies by running both an interpreter and a compiler, the latter invoked on hot-spots [116]. In [1], an evaluation of several techniques for *hot spot* detection is presented.

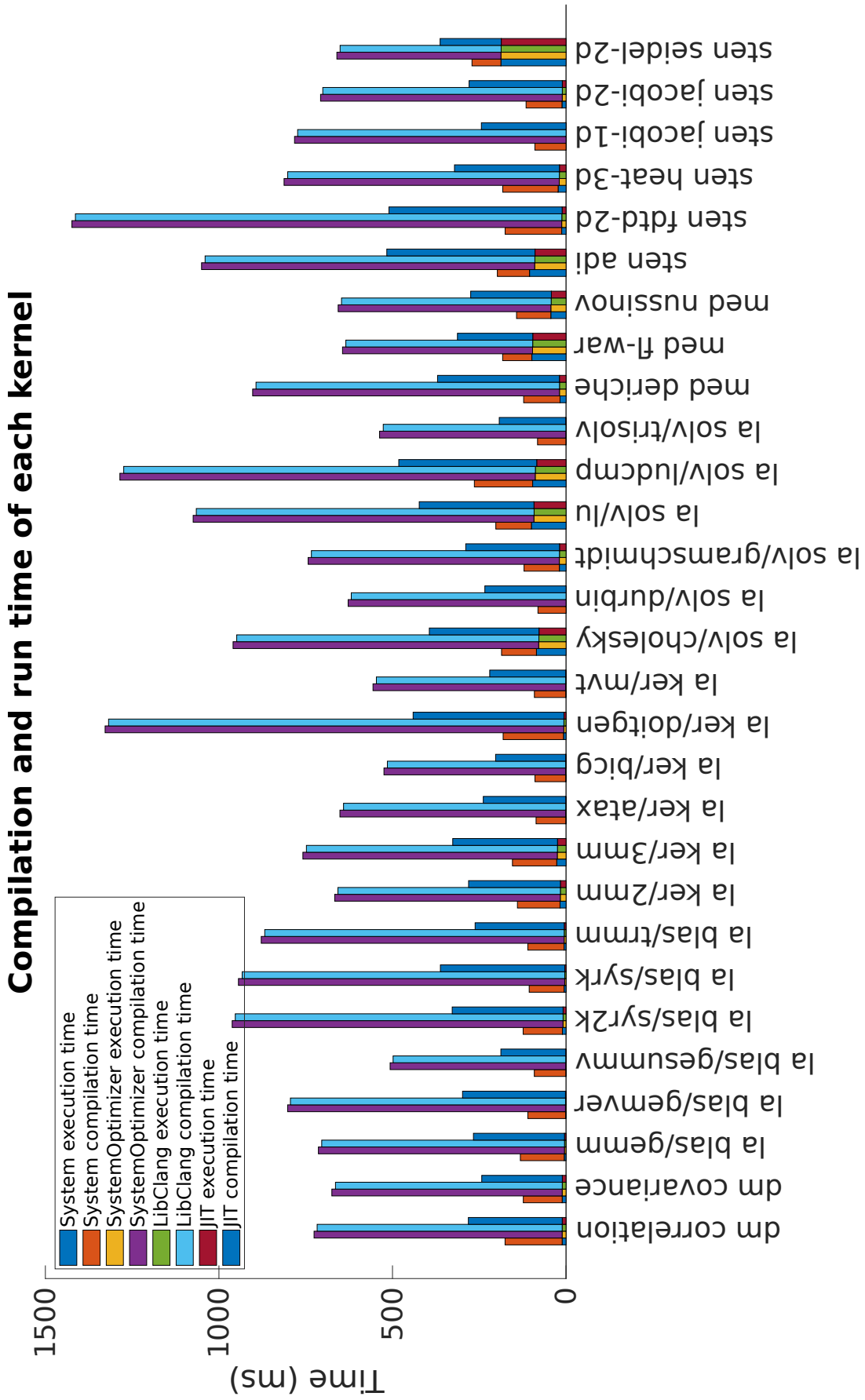


Figure a.2: Compile time and run time of each kernel of the PolyBench/C benchmark suite with different implementation alternatives from LIBVC.

More recently, the *Graal Project*[2] aims at leveraging the JIT technologies to improve the performance of several interpreted, bitcode-interpreted, and compiled languages. In particular, the *GraalVM*[3] is the most relevant outcome of this project. It embraces JIT compilation as the key-stone of the framework in order to enable several speculative optimizations [45]. *GraalVM* has been used to provide JIT support to a wide range of programming languages, from domain specific languages [157] to popular languages, such as JavaScript and C/C++ [64]. However, it relies on LLVM-based solutions to dynamically compile C/C++ source code to LLVM-IR (`clang` front end), and to interpret LLVM-IR (`lli` interpreter). Thus, *GraalVM* can be considered technologically equivalent to the LLVM compiler toolchain we evaluated via LIBVC.

Another effort to enable dynamic compilation over C/C++ code is represented by `Cling` [156] — the `clang`-based C++ interpreter. Since this project implements the Read-Eval-Print-Loop (REPL) paradigm over C++ source code, it is the closest effort to a pure C++ JIT compiler in the state-of-the-art. In the rest of this section we further discuss `Cling` and its usage. Finally, we provide a comparative analysis between our proposed solution and `Cling`.

CLING `Cling` originates from the need to process a vast amount of data with the capabilities of a compiled language with a strong focus on code efficiency, such as C++. It was developed as a part of CERN's data processing framework ROOT. `Cling`'s core objective is to provide a fast and interactive way to run applications able to access experimental results generated by the high-energy-physics research community. A structural analysis of `Cling` highlights the 3 fundamental parts it is composed of:

INTERPRETER which implements the parsing, JIT compilation, and evaluation of native C++ capabilities

META PROCESSOR which provides a command line interface to send commands to the interpreter

USER INTERFACE which provides the interactive prompt, and an exception handling mechanism

`Cling`'s interpreter is based on the version 5.0.0. of LLVM, and on its C++ frontend Clang. While our `JITCompiler` leverages the `IRCompileLayer` class to provide jitting, `Cling` uses the `LazyEmitLayer` one. In the former case the class JIT compiles the LLVM Module as soon as this is passed to the `addModule` API. In the latter case, instead, it explicitly waits the symbol to be called to begin compiling.

2 Oracle Graal project https://openjdk.java.net/projects/graal/
3 GraalVM https://www.graalvm.org

Given `Cling`'s intrinsic REPL-based implementation, to measure its performance we exploit the bash's `time` utility to extract the precise execution time lapse of the evaluation of a single kernel. To exclude the overhead due to other `Cling`'s components initialization, such as the User Interface, we run the Interpreter several times with no instructions but the `.q` (exit) directive. In this way, we collect the average time lapse of the setup and the tear down routines of `Cling`.

Moreover, we separate compilation and execution times, which are indistinguishable due to the lazyness of `Cling`'s jitting strategy, by running the interpreter multiple times. This procedure allows us to infer the single compilation-only time lapse for each kernel.

COMPARATIVE ANALYSIS We compare `Cling` with our JIT implementation within LIBVC by scheduling the run of each kernel from the PolyBench/C benchmark suite. We rely on the same hardware and software setup described in Section a.1.2. We measure execution time and the (re-)compilation overhead. Since `Cling` does not support any code optimization level, we slightly modify the previously-described experimental setup to allow a fair comparison between `Cling` and `JITCompiler`. In particular, we disable any optimization in our `JITCompiler` via the `-O0` compiler option. We collect data using the `MEDIUM` dataset size preset and the IEEE-754 floating point single precision data type for the computation.

In Figure a.3 we compare `JITCompiler` from LIBVC with the external tool `Cling`. As expected, the compiling phase is a more time requesting task than a single execution of the PolyBench/C benchmark suite. Figure a.3 clearly shows that the compiling plus execution time of the `JITCompiler` outperforms the `Cling` interpreter even when the former does not schedule any optimization, i.e. when it runs with `-O0` optimization level.

Figure a.4 better highlights the saving coming from the use of `JITCompiler`, as it shows the overall amount of time spent compiling and running the full PolyBench/C benchmark suite.

Figure a.5 shows the comparison bewteen the compilation time plus the execution time of `Cling` compared with the equivant of the `JITCompiler` when the latter is using the `-O3` optimization level. Running the compiled code several times allows us to analyze the actual speedup given by the `JICompiler`. Although the total compilation time, as seen in Figure a.4, does not show astonishing differences, with an increased number of invocation of the same kernel the `JITCompiler` optimization capabilities — which are paid by longer compilation times — allows the system to reach better performance.

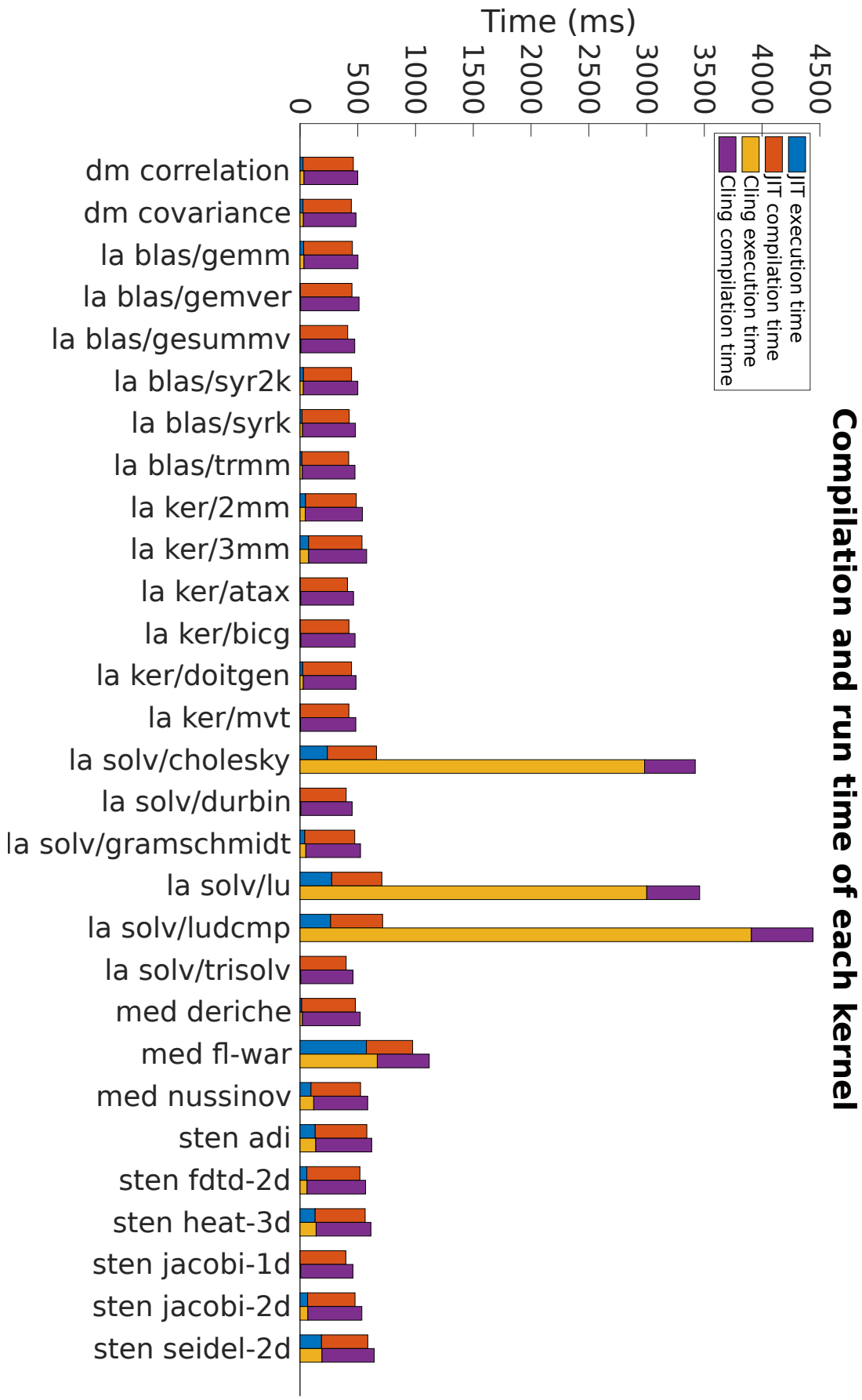


Figure a.3: Compilation and execution time of each kernel of the PolyBench/C benchmark suite with `Cling` and with `JITCompiler`.

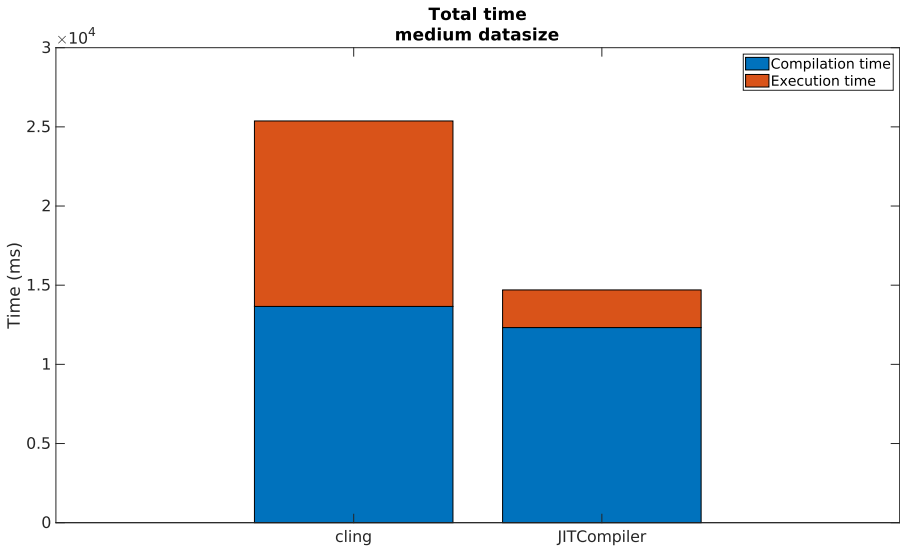


Figure a.4: Compilation and execution time of the overall PolyBench/C benchmark suite with `Cling` and with `JITCompiler`.

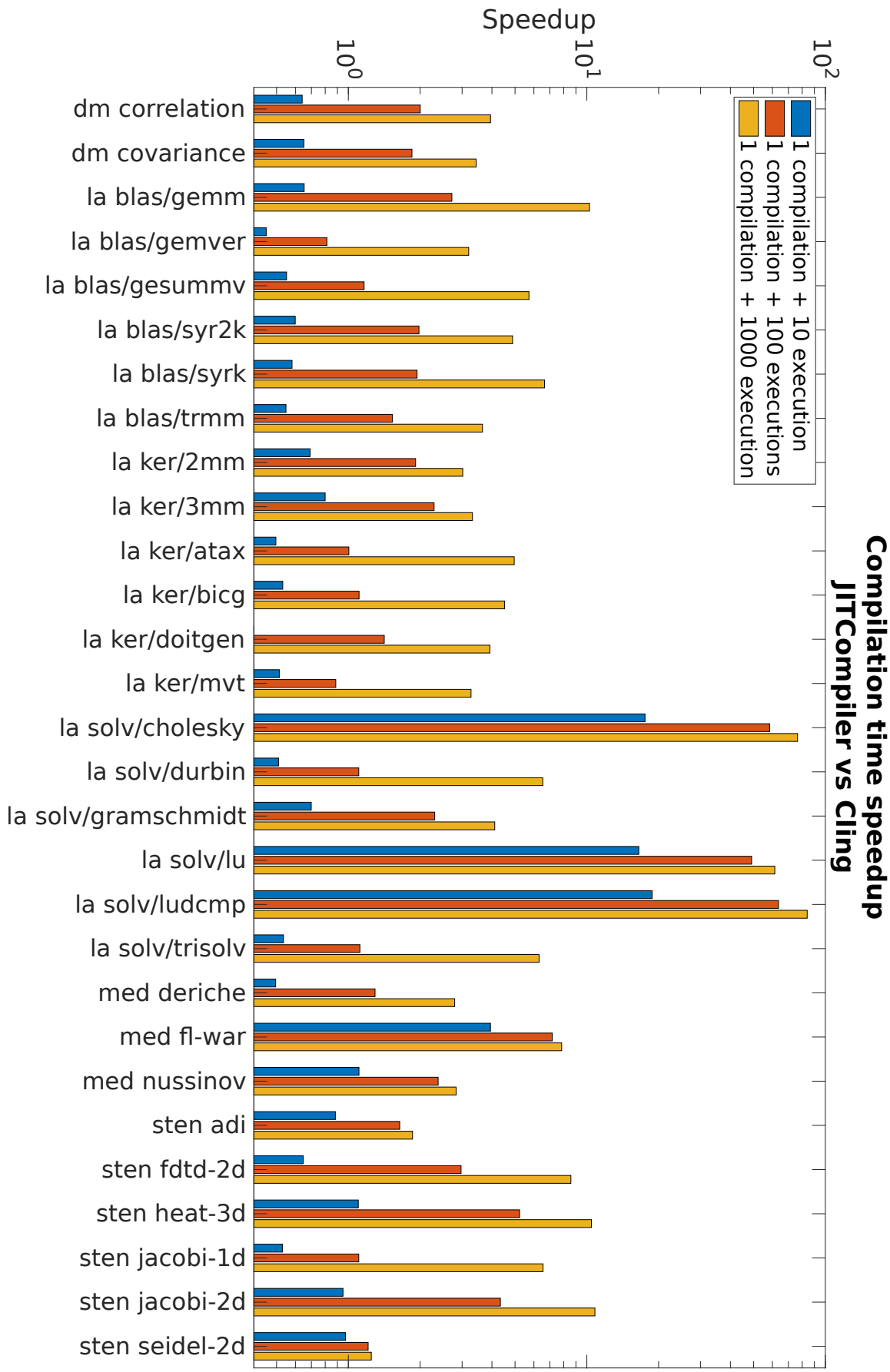


Figure a.5: Speedup achieved by using `JITCompiler` instead of `Cling` with 10, 100, and 1000 invocations of the kernel.

DECLARATION

This page concludes the thesis. Hope you found it interesting to read. Now let's go out and have some drinks.

Milano, June 2019

Stefano Cherubin