A COMPUTATIONAL APPROACH TO CUSTOM DATA REPRESENTATION FOR HARDWARE ACCELERATORS

A COMPUTATIONAL APPROACH TO CUSTOM DATA REPRESENTATION FOR HARDWARE ACCELERATORS

BY
ADAM B. KINSMAN, B.Eng., M.A.Sc.
APRIL 2010

A THESIS
SUBMITTED TO THE SCHOOL OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE
DOCTOR OF PHILOSOPHY

McMaster University

DOCTOR OF PHILOSOPHY (2010)
(Electrical and Computer Engineering)

MCMASTER UNIVERSITY
Hamilton, Ontario, Canada

TITLE:	A Computational Approach to Custom Data Representation for Hardware Accelerators
AUTHOR:	Adam B. Kinsman, B.Eng. Engineering Physics, M.A.Sc. Electrical and Computer Engineering, McMaster University, Canada
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	xiv, 150

Abstract

This thesis details the application of computational methods to the problem of determining custom data representations when building hardware accelerators for numerical computations. A majority of scientific applications which require hardware acceleration are implemented in IEEE-754 double precision. However, in many cases the error tolerance requirements of the application are much less than the accuracy which IEEE-754 double precision provides. By leveraging custom data representations, a more resource efficient hardware implementation arises thereby enabling greater parallelism and thus higher performance of the accelerator.

The existing custom representation methods are unable to guarantee robust representations while at the same time adequately supporting ill-conditioned operators. Support for both of these scenarios is necessary for accelerating scientific calculations. To address this, we propose the use of a computational method based on Satisfiability-Modulo Theory (SMT). By capturing a calculation as a set of constraints, an SMT instance can be formulated which provides meaningful bounds even in the presence of ill-conditioned operators. At the same time, the analytical nature of SMT satisfies the need for robustness. Utilizing block vector arithmetic, our SMT approach is extended to provide scalability to large instances involving vector calculus which arise in scientific calculations. Atop this foundation, a unified error model is proposed which deals simultaneously with absolute and relative error, thereby providing the means of supporting both fixed-point and custom floating-point data types. Iterative algorithm analysis is leveraged to derive constraints for the SMT method. The application of the method to several scientific algorithms is discussed by way of case studies.

Acknowledgements

As in any significant endeavour, to try to list all the people who in some way have impacted my life during the course of this undertaking would require far too many pages and surely still leave people out. At the same time, doubtless there are those whose effect in my life was un-mistakable, and they deserve special mention here.

I would like to express my thanks to the the administrative and technical staff of the ECE department at McMaster, who work hard to keep things running smoothly at a practical level, maintaining an environment which enables research to be done. I am appreciative also to the many faculty who influenced me during my time at McMaster in both personal and academic ways - in particular, Dr. Alex Jeremic and Dr. Shahin Sirouspour have given generously of themselves in serving on my supervisory committee, and have helped in identifying and defining the motivation for this work.

For sharing in both the joys and sorrows of this adventure, I am thankful to friends, both personal and professional. Special mention is surely deserved by former and current students in the Computer Aided Design and Test Research group of McMaster - Bai Hong Fang, Qiang Xu, David Lemstra, David Leung, Ehab Anis, Kaveh Elizeh, Mark Jobes, Roomi Sahi, Zahra Lak and Jason Thong - who have shared in countless hours of stimulating discussion, proofreading papers and coffee-break philosophizing. Among this group are also Henry Ko and Phil Kinsman, with whom the history goes back so far that I can say we've grown up together. I am also deeply indebted to my supervisor Dr. Nicola Nicolici for his unswerving commitment, inspiring vision and patient mentorship which have shaped me in a profound and lasting way. I cannot imagine this thesis being possible without his tireless efforts.

To my brothers Josh, Matt and Phil and my sisters-in-law Jenn, Beth, Alex and Amanda, I am thankful for constant emotional support and for many time-outs to relax and reconnect with the world. My father and mother in-law Bruce and Jane, and my dad and mom Bruce and Jan have also provided much support, and I am deeply grateful for their wisdom in helping me make tough decisions. To my wife Pamela for her patience, dedication, love and support I owe more than all the rest put together - I could not have done this without her. She has sustained me, and only through the joy that she and our children bring to my life, has my sanity been maintained.

Above all I praise God for placing in my life so many supportive people, for providing for me and for upholding me. Through Him, I have come to understand myself much better, for I am weak but He is strong.

List of Abbreviations

AA	Affine Arithmetic,
ASIC	Application Specific Integrated Circuit,
CAD	Computer-Aided Design,
CC	Clock Cycle,
CG	Conjugate Gradient,
CPU	Central Processing Unit,
DFP	Davidon-Fletcher-Powell,
DSP	Digital Signal Processing,
EDA	Electronic Design Automation,
ESD	Energy Spectral Density,
FF	Flip-flop,
FFT	Fast Fourier Transform,
FLOPS	Floating-point Operations Per Second,
FPGA	Field Programmable Gate Array,
GIA	Generalized Interval Arithmetic,
GPGPU	General-Purpose computation on GPUs,
GPU	Graphics Processing Unit,
HDL	Hardware Description Language,
IA	Interval Arithmetic,
ILP	Integer-Linear Programming,
IP	Intellectual Property,
LTI	Linear Time-Invariant,
LUT	Look-Up Table,

NOC	Network-On-Chip,
NRE	Non-Recurrent Engineering,
OS	Operating System,
PC	Personal Computer,
RTL	Register Transfer Level,
SAT	Boolean satisfiability problem,
SMT	Satisfiability-Modulo Theory,
SVD	Singular-Value Decomposition,
VHDL	Very-high-speed-integrated-circuit HDL,
VLSI	Very Large Scale Integration,

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the early days of the transistor roughly 60 years ago, exponential scaling has driven an increase in integration levels to enable modern circuits with billions of transistors in a single device, and operating frequencies in the low gigahertz (GHz) range [13]. To deal with the inherent complexity of designing such circuits, an entire ecosystem of computer-aided design (CAD) tools and design intellectual property (IP) has evolved.

Over the same period, computers have been in a symbiotic relationship with applications growing similarly in complexity, enabling problems of growing difficulty and scale to be tackled by computers. In this chapter, discussion begins with a general description of how computers are employed for solving problems, eventually leading to the justification for this work in Section 1.5.

1.1 Computation to solve problems

One of the primary ways in which computers have improved our problem solving capacity is their ability to carry out with speed and precision tasks which human beings may find too repetitive, tedious, error prone or which involve overwhelming amounts of data. A straightforward example of this is the use of computers to sort, search and filter databases involving Terabytes (TB, $= 10^{12}$ bytes) of information [17]. Going beyond mere record keeping, in general terms computers are used to solve problems by manipulating data according to a set of rules. In the physical sciences and engineering in particular, where analysis based

on mathematical theory has seen a great deal of success, computers have been extensively employed.

One explanation of the success of computers in science and engineering is the emphasis placed by those fields on creating models of the systems which they study. A reliable model of a system will enable one to reason about the system and make predictions about its behaviour. In this way, a model enables transference of some of the physical experimentation required to draw conclusions into the domain of abstract reasoning. The role of computers is to carry out in an automated way the reasoning related to the model, and once set up a computer can often be reused to perform many virtual experiments, providing more information at reduced cost compared to physical experimentation. Physical experimentation however will always be required to verify the conclusions drawn from the model and to inform/refine the model itself.

A poignant example of this virtualization of experimentation is the use of electronic analog computers for solving differential equations [72], a technique which was employed before programmable digital computers became ubiquitous. The equations which govern the behaviour of electronic circuit elements such as inductors, capacitors and resistors bear striking resemblance to equations which govern many physical phenomena e.g., mass-spring systems or fluid-flow systems. The similarity in the underlying mathematics enables a compact, inexpensive electronic system which models a bulky, costly physical system to replace it.

While virtual experiments can provide more information with less cost than physical ones, they do not come for free - there is a cost associated with creating a computer to reason about a given model, and the conclusions drawn are only as reliable as the models themselves. In general, more complex phenomena require more sophisticated models, which in general require more computational effort and/or setup cost. The next subsection examines these issues of computational effort and setup cost.

1.1.1 Computational effort and cost

The inherent tradeoff between flexibility and cost exists in many avenues of life, and electronic systems are not excluded. When a dedicated electronic system is constructed to serve

a solitary purpose, assumptions can be made about its task and environment leading to a simpler, more efficient implementation. Making a system more flexible often involves supporting scenarios which violate some assumptions, and as such any efficiency gains coming from those assumptions are lost. On the other hand what is gained from increased flexibility is reuse which impacts cost. While a dedicated system can be more efficient than a general purpose system at a particular task, usually it can do nothing but that task. A more flexible system will be less efficient for the same particular task but will be able to perform a number of tasks at similar efficiency.

In light of this tradeoff, the last few decades of evolution in electronics reveals a steady stream of applications implemented at first in dedicated electronic systems moving to ever more general and programmable platforms such as general purpose processors [64] - even as they are followed by applications of increasing complexity [31]. Seen from another point of view, applications emerge which seek electronic implementation, and the technology advancements arising during the course of that implementation have the two pronged effect of 1) enabling the same application to be solved on a more general platform and 2) extending the reach of dedicated systems to reach previously unsolvable problems. It is primarily through the first effect that cost is reduced, by moving to a general platform, the development and manufacturing costs of that platform are shared among all the applications which use that platform, and thus are lower than for a dedicated platform.

This pattern is exemplified particularly clearly in the transitioning of multimedia from analog to digital. Early electronic audio processing using dedicated analog systems gradually migrated to using dedicated digital signal processing circuitry, then to programmable digital systems as the capabilities of digital circuitry expanded thanks to Moore's law [90]. Today, audio processing in software is almost trivial even on commodity general purpose hardware. Similarly (but lagging by a number of years) was the transition for video, originally managed using analog circuitry, now processed on mainstream personal computers.

1.2 The case for acceleration

In addition to the migration from dedicated to general purpose platforms discussed in the previous section, what has also become clear over these few decades is the existence of

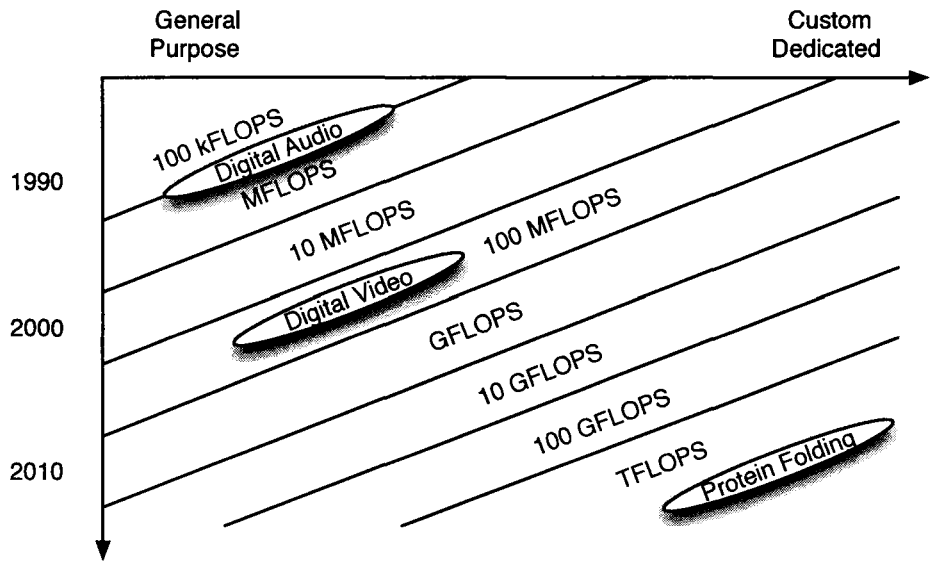


Figure 1.1: Computational thresholds for applications.

computational thresholds, i.e., barriers of complexity imposing limits on what problems are feasible with a certain capacity to carry out computations, commonly referred to as "compute power" (as distinct from power as it relates to energy consumption). These compute power thresholds are the reason for the lag in time between digitization of audio and of video, the computational threshold for video is higher than for audio. Figure 1.1 depicts in very general terms the evolution of compute power across the spectrum of platforms (from general purpose to dedicated), as well as computational requirements for some applications.

It is worth noting that the tasks in the upper-most computational capacity ranges (100 GFLOPS and TFLOPS), while being recognized today as important problems may not have even been conceived of in a similar graph from a decade ago. This carries the important point alluded to above that technology advancement with a specific application in mind has implications in other unanticipated applications. Furthermore, while Figure 1.1 focusses on the case where the higher efficiency of dedicated platforms is leveraged for the sole purpose of increased compute capacity, the efficiency may improve other aspects of performance.

Having established that more computational capacity can always be made use of, we must find ways of making this greater computational power available. Historically for general purpose computing, increased compute power came from two sources. On one hand, individual devices were capable of more operations per second through advancement in process technology (bringing higher clock rates), and architectural innovation (reducing execution overhead). On the other hand was parallelism, integrating many microprocessor devices into a much larger, more powerful supercomputer. While in the past the former has been the primary focus for increasing computational capacity, there has been a recent shift to relying on the latter to provide the compute power for the ever increasing complexity of applications [4].

The concept of parallelism for creating more powerful computers, and the means of implementing it will be discussed in more depth in Chapter 2, but are summarized here. Currently, there are three main directions: multi-core central processing units (CPUs), graphics processing units (GPUs) and field-programmable gate array (FPGA)-based hardware accelerators. In terms of integration levels, there are tens of processing engines in each multi-core CPU; hundreds of them in a GPU and thousands in each FPGA, as of today.

The differences between these three platforms lie in the amount of design effort required to map an application to each platform, and also the maximum achievable performance. Using the existing design methods and tools, implementing an application in FPGAs requires about three times more implementation effort than multi-core CPUs and about two times more effort than GPUs [92]. Figure 1.2 shows how FPGA-based acceleration can make a difference. Although in the audio/video processing fields, CPUs are sufficient today, there is a growing adoption of GPUs in the fields of computer graphics and quantitative finance, for example. There are however, fields such as biomedical sciences (medical equipment for remote surgery [81] or gene sequencing [50]) and environmental sciences (oil/gas exploration [34] or weather simulation [31]) where the added compute power brought by high-end FPGAs facilitates the much-needed acceleration.

In order to make this much-needed acceleration accessible, what is required is sophisticated computer-aided design (CAD) tool support. One problem which contributes to the higher design effort for FPGAs and as such needs to be addressed is the choice of a suitable numerical representation format as discussed in the next section.

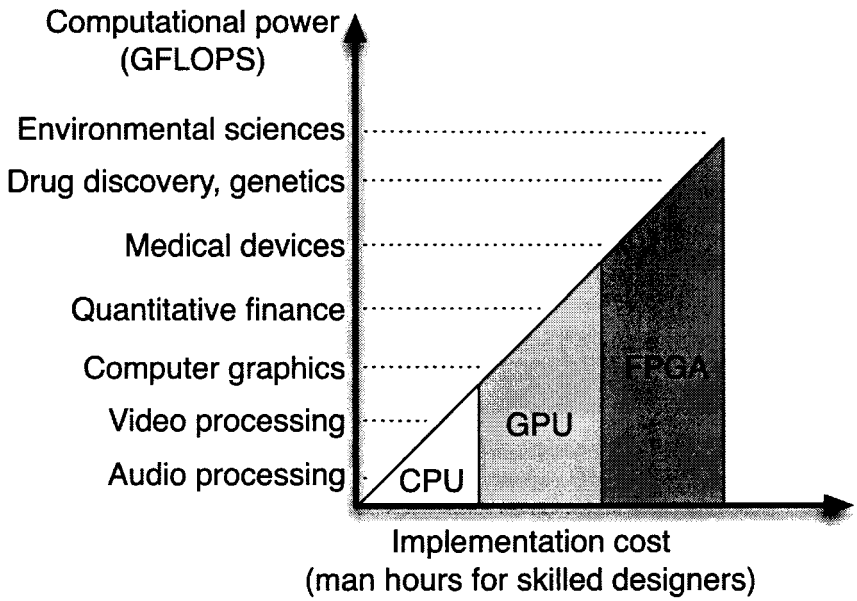


Figure 1.2: Computational capacity/requirements and development time for state of the art platforms/applications.

1.3 The need for custom representations

While the previous sections have established the role of computers in problem solving and provided motivation for using accelerators to increase the problem solving capacity of computers, they have focussed primarily on the operational aspect of computer operation. This section examines another, equally important aspect of how the data on which the computer operates is represented.

1.3.1 Symbolic vs. numerical computing

The representation of the data which computers are used to process generally exists under two main paradigms: symbolic and numerical computing. In symbolic computing, the data which is processed and rules by which it is processed are both abstract, and are derived

from the theory governing the problem which the computer is working to solve. Numerical computing on the other hand, uses rules based on arithmetic to manipulate data which are quantities representing aspects of the problem under consideration.

To understand the difference between symbolic and numerical computing, consider a geometric series $\sum_{j=0}^{j=n-1} ar^j$. In symbolic computing the data would be the expression and its variables while the rules would be derived from algebra, application of which should lead eventually to the expression $a\frac{1-r^n}{1-r}$. In numerical computing however, the rules are derived from arithmetic, but the problem has no clear meaning without explicit numerical values for a, r and n. Once specified, application of the rules means performing the exponentiation, multiplication and summation over all j to produce the final result.

Due to key advantages (i.e., completeness, compactness, exactness) in some scenarios, symbolic computation packages have been developed such as computer algebra systems, e.g., Maple [85], Mathematica [128], and Maxima [97]. However, the majority of science/engineering problems today are solved using numerical techniques for two primary reasons. First, the precision of the input parameters to a problem as well as the required accuracy of the solution are typically limited. This is especially true in science and engineering which involve inexact measurements, and any uncertainty could frustrate the exactness, or complicate the analysis, of a symbolic solution. Second, although not the case in the geometric series example above, numerical representations may be more compact (efficient) for problems of current-day complexity, leading to reduced computation times especially in light of the tolerance created by inexactness.

1.3.2 Representation of real numbers

Upon deciding to perform calculations numerically, the next decision is what format to use to describe numerical quantities. Since digital computers are discrete in nature, representation of discrete sets such as the integers is natural. The set of real numbers on the other hand is continuous and infinite, requiring approximation to be represented using the discrete, finite resources of the computer. A number of schemes have been devised to manage the error arising from this approximation as discussed next.

Continued fractions

One approach to approximating the real numbers within the finite, discrete scope of digital computers is the use of continued fractions [124]. A continued fraction representation for a number x is a sequence of integers a_i such that:

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \ddots}},$$

where the sequence a_i is finite for rational numbers and infinite for irrational numbers. While some of its mathematical properties (e.g. truncation yields best rational approximation) make it a favourable choice in theory, practical implementations encounter some difficulties. For one, human beings are not used to these representations so any user interaction with computers involves translation, at relatively high cost due to numerous divisions. Also, arithmetic operations on continued fractions are complex [124], and the varying representation length of different values can cause storage and manipulation complications.

Rational representation

Rational representations can be thought of as a simplified version of continued fraction, instead of a sequence with nested fractions, there is instead a pair of numbers (numerator and denominator) and a single fraction [87]. This solves some of the practical issues raised above, i.e., only a single division is required to convert to a human-readable format, and the size is more uniform over all representable numbers. However, manipulation can still cause problems in that straightforward operations could cause the numerator and denominator to grow without bound, but finding the best approximation with numerator and denominator within a certain range is by no means trivial (continued fractions provide such a means). At the same time this method has seen adoption, particularly in situations where the application can lend some insight to the ranges involved, and where values are primarily rational.

Fixed-point

Pushing the rational representation one step further, we can fix the denominator and consider only the numerator giving rise to a fixed-point representation [131]. In digital systems specifically, the choice of denominator is typically restricted to a power of 2, making conversion from one denominator to another (on a binary platform) the straightforward operation of shifting left or right. Consider the rational number $19/3$, using denominator of $8 = 2^3$. This reduces to representing the integer nearest $19/3 \times 2^3$, which is $51 = (1)2^5 + (1)2^4 + (0)2^3 + (0)2^2 + (1)2^1 + (1)2^0$ so the fixed-point representation would be 110011. Since the implicit denominator of 2^3 produces a shift so that $19/3 \approx (1)2^2 + (1)2^1 + (0)2^0 + (0)2^{-1} + (1)2^{-2} + (1)2^{-3}$, a "binary point" can be placed between the 3^{rd} and 4^{th} bits from the right with the number 110.011 resulting. This divides the representation into I *integer bits* on the left representing the part of the number (in absolute value) ≥ 1, and F *fraction bits* on the right representing the part of the number (in absolute value) < 1. In essence, I limits the range of representable numbers and F limits the resolution. To extend the representation to negative numbers, 2's complement is used where $-x$ is represented as $2^I - x$, and the range of representable numbers is -2^{I-1} to $2^{I-1} - 2^{-F}$. To summarize then, for a 16 bit fixed-point number with 5 integer bits and 11 fraction bits, the resolution is $2^{-11} \approx 4.88 \times 10^{-4}$, the range is -16 to 15.9995, and $19/3$ would be represented 00110.01010101011, while $-17/7$ would be 11101.10010010010.

This format provides a bound on absolute error incurred at each operation, and as such is very attractive in terms of precision however, the dynamic range is severely limited. As a result, this method has received a great deal of attention in applications with well bounded numerical ranges (for example digital signal processing) and will be discussed in more depth in Chapter 2

Floating-point

While fixed-point representations are compelling in terms of precision, careful management of the implicit denominators is required to address the dynamic range limitations. Unfortunately, this can limit the flexibility and reuse of any platform which uses it. This accounts for its success in dedicated systems with well understood numerical patterns, and

its shortcomings in more general purpose platforms. Expanding the dynamic range has involved the observation that bounding relative error rather than absolute error in calculations will often suffice, the reason for the emergence of scientific notation when doing calculations by hand. In this format a number x is commonly represented (or approximated) as $\textit{significant digits} \times \textit{base}^{\textit{exponent}}$. The case of scientific notation uses 10 as the base, computers in general use base 2. The scaling by $\textit{base}^{\textit{exponent}}$ is analogous to moving the "point" of fixed-point giving rise to the name floating-point [41].

In the specific case of base 2, the significant digits (also called *significand* or *mantissa*) are scaled to between 1.0 inclusive and 2.0 exclusive, i.e. [1.0..2.0). For example, $19/3$ as above can be represented as $1.583... \times 2^2$. If the exponent is represented on 4 bits we have 0010, and the mantissa on 11 bits we have (1.)10010101011, where the (1.) is implicit (not stored). Extension to negative numbers is done through a sign bit indicating positive or negative, and for the case above of 1 sign, 4 exponent and 11 mantissa bits enables representation of numbers within the range $\approx \pm[2.94 \times 10^{-39}, 6.80 \times 10^{38}]$ to within relative error of $\approx 4.89 \times 10^{-4}$.

The bounded relative error behaviour of floating-point numbers makes it particularly suitable for use as a representation format, especially for scientific applications. The convenience of relative error for measurement and control in the physical sciences and engineering also makes floating-point a natural choice. Moreover, the nature of many scientific applications provides contained output error for bounded input error. Finally, the dynamic range issue is addressed as resources for floating- vs. fixed-point to provide the same dynamic range are $O(log(log(range)))$ vs. $O(log(range))$. These advantages have enabled floating-point arithmetic to be successfully deployed in computing machines.

1.3.3 Standardization of floating-point support

As a result of the advantages of floating-point representation discussed above, it has been favoured for numerical computing applications, based primarily on software libraries (e.g., [36, 95]). In response, computer hardware makers (seeking competitive advantage) provided dedicated hardware supporting floating-point arithmetic to improve performance for numerical tasks. Independent hardware makers frequently adopted different choices for not

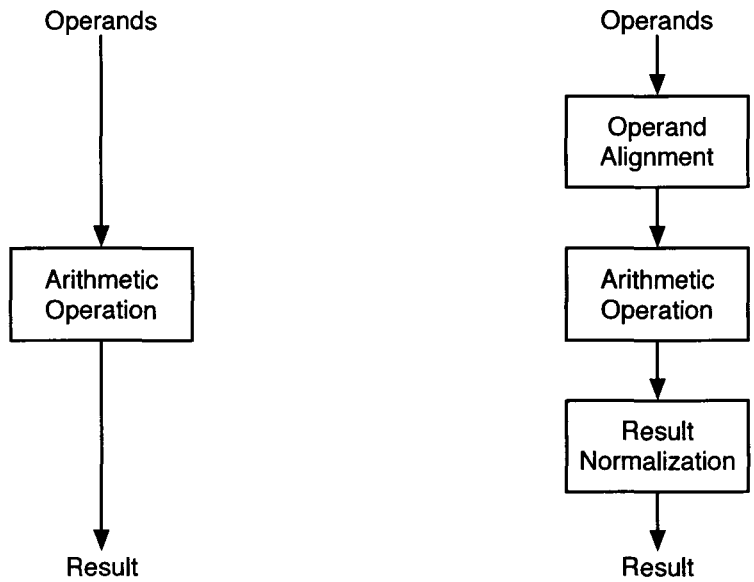
only size of exponent and mantissa fields, but even the base. In order to foster portability of software across hardware platforms, as well as consistency and reproducibility of results obtained from numerical programs, the need for a standard floating-point representation became clear [59].

The culmination of this need for a standard was the IEEE-754 [56] standard for floating-point representations, describing single precision (on 32 bits: 1 sign + 8 exponent + 23 mantissa) and double precision (on 64 bits: 1 sign + 11 exponent + 52 mantissa) formats. One important aspect arising from this standardization is the handling of corner cases, i.e. values or operations resulting in values outside or very nearly outside the set of numbers properly represented by the format. As [59] points out, this matter was largely ignored prior to standardization, with each hardware manufacturer making their own decisions. As also pointed out by [59], corner case control is imperative for maintaining portability of software and reliability of results obtained from software. For this reason, the standard provides options for a number of scenarios which are under programmer control (having defaults also assigned by the standard), so that a programmer with knowledge of the application may dictate the appropriate behaviour for a given corner case scenario.

1.3.4 Custom precision floating-point

The previous section has shown that the capacity to manage a broad set of corner cases and the programmability to dictate behaviour are very important for general purpose hardware meant to execute a plethora of numerical programs requiring myriad features and behaviours. By contrast, an application-specific hardware solution needs only be concerned with the corner case behaviour relevant to the particular numerical task it implements. This important difference provides room for custom precision floating-point representations.

Most obviously, if an application of interest does not require the full dynamic range provided by IEEE-754, a custom representation may use smaller widths for the exponent and mantissa fields. More importantly however, support for the many modes dictated IEEE-754 take a significant toll when implemented in hardware, both in performance (maximum clock rate) and area, especially because of the programmability requirement. Simply freezing the corner case behaviour (removing the programmability requirement) would already



(a) A fixed-point arithmetic operation. (b) A floating-point arithmetic operation.

Figure 1.3: Fixed and floating-point arithmetic operations.

bring implementation cost reduction. Even more than this however is that for custom representations the boundaries marking regions of corner case behaviour are themselves flexible. For example, adding a single bit to the exponent field effectively squares the range of representable numbers to eliminate the need for any overflow handling whatsoever.

Leveraging this added degree of freedom, units tailored to the representation requirements of a given application can be designed to be smaller than fully standard compliant IEEE-754 arithmetic units. By tailoring representation requirements not only to the application as a whole, but even the specific stage of calculation, meaningful implementation cost and performance savings can be attained as discussed in the next section.

1.4 Cost reduction and performance gain

In order to understand how implementation cost and performance gain are affected by choice of representation, consider Figure 1.3 which shows in a very general sense arithmetic operations on fixed and floating-point operands. Since fixed-point numbers are essentially unencoded (they do not contain control information), arithmetic operations on them such as addition or multiplication are relatively direct (Figure 1.3(a)). For floating-point operations however, the exponents of the operands must be decoded, the two operands properly aligned, the operation performed, and the number re-encoded into a proper floating-point representation (Figure 1.3(b)).

The difference in terms of performance can be see more quantitatively in Table 1.1, based on data drawn from [77]. The table compares field-programmable gate-array (FPGA) implementations of multiplication and addition for 32-bit floating- and fixed-point operands (8 bits exponent and 23 bits mantissa for the floating-point). The comparison is made in terms of latency in clock cycles (CCs) and implementation cost in flip-flops (FFs) and lookup tables (LUTs) the basic implementation units which make up an FPGA. What differs between the two data types is that while fixed-point calculations use the data directly as operands, floating-point operations require the operands to be scaled before the operation can be performed, and the result must be normalized.

Observing the table, the impact of these differences can be seen. For multiplication, the variation between fixed- and floating-point is minimal. The mantissa of the result comes from the multiplication of the operand mantissas, and the exponent is essentially the sum of the operand exponents, with the smaller multiplication (23 vs 32 bits) balancing out the addition for the exponents. In light of Figure 1.3, the alignment and normalization units from 1.3(b) are relatively small compared to other operations, and are offset by how much smaller the operation unit is compared to 1.3(a) because of the 23 bit mantissa instead of the 32 bit fixed-point value.

For addition on the other hand, the differences are significant. In this case, the floating-point operation requires checking conditions on operand overlap, and a shift of anywhere between 0 and 22 positions may be required to align the operands. In contrast, for fixed-point nothing more than a simple adder is required. In terms of latency, the case is similar.

Table 1.1: Comparing area/performance for floating vs. fixed-point [77].

Operation	Area-LUTs		Area-FFs		Performance-CCs	
	Floating	Fixed	Floating	Fixed	Floating	Fixed
Multiplication	≈ 750	≈ 750	≈ 1000	≈ 750	≈ 30	≈ 30
Addition	> 600	< 40	> 600	< 40	> 10	$= 1$

For multiplication, the clock cycles required are essentially the same, while for addition more clock cycles are required for pipelining the large barrel shifters required to align the numbers before performing the addition. Referring again to Figure 1.3, in this case the alignment and normalization units are more costly to implement than for multiplication.

What is clear from this example is that representation can have a significant impact on area and performance of individual calculation units. This impact is amplified by as many calculation units as are employed together to achieve parallelism. In simple terms, if a calculation unit is smaller, more can be fit onto an acceleration platform (like an FPGA) thereby increasing the parallelism, and if it is faster the overall throughput is further increased. With this in mind, Figure 1.4 depicts in general terms the potential gains of moving away from the IEEE-754 double precision standard. By fully leveraging any slack in precision requirements, resource cost of arithmetic units can be reduced, leading in the end to greater parallelism and with it increased computational capacity.

What is also the case regarding representation impact on area and latency is that for a fixed choice of architecture and implementation technology, a direct relationship between representation and cost/latency can be identified. For example, the choice of a combinational multiplier vs. a sequential one for a given FPGA device family will yield a certain (complex) tradeoff between maximum clock frequency, number of clock cycles latency to the result and number of LUTs required. These choices are influenced by many factors which are independent from the representation choice, and can vary significantly from one application to another. Furthermore, they are usually settled (or very nearly) when custom

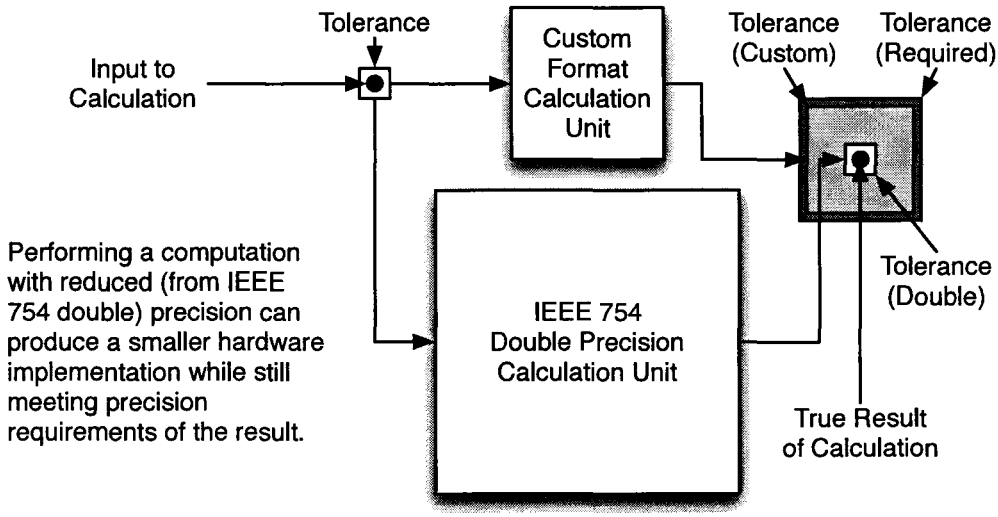


Figure 1.4: Contrasting standardized double vs. custom precision floating-point .

representations are derived. For this reason, cross platform and cross architecture applicability of a custom representation method must provide flexibility to deal with a wide array of design scenarios.

Support for these varying scenarios can be provided by making abstraction of the underlying implementation platform by use of models relating performance and cost to custom representation (bit-width). A simple example is a combinational multiplier with operands of size n bits exhibiting an approximate n^2 area cost. Similarly, a ripple carry adder has a roughly linear relationship between operand size and delay. In a specific implementation technology, these numbers can be more precisely quantified. Aggregate cost models for full architectures can be formed by combining models for the functional units they contain, and greater accuracy of the model can be attained by adding more detail to the model.

Abstracting the implementation in this way brings flexibility to the custom representation approach, allowing the same methodology to be targeted toward different architectures

and platforms. In this way, the improved performance which is the goal of custom representations transcends the implementation platform. Through this abstraction, custom representations are identified solely by architecture matched bit-widths, obtained with respect to architecture and platform specific cost models. That is, with a set of bit-widths relating to a defined architecture and platform, the hardware implementation follows directly. It is the burden of determining these parameters (bit-widths) that is focus of this thesis.

1.5 Problem statement

In this chapter, we have established the motivation for accelerating scientific application and identified the key role to this played by custom data representations. In this section, the necessary features for deriving custom numerical representations for applications in the scientific computing domain are described and the organization of the remainder of the thesis is summarized.

1.5.1 Robustness requirement

A fundamental requirement of any representation which is to be used for scientific computing is robustness - how far the correctness of the results can be trusted. Evidence for the need for this feature can be seen in the lengths which IEEE-754 goes to in providing support for indicating when a numerical problem arises (e.g. division by zero) and to correct for such problems if possible. While in some application domains even catastrophic numerical errors have little impact in terms of real repercussions, this cannot be taken for granted in scientific applications. For example, a multimedia decoding system having non-robust numerical support may lead to a corrupted media stream which, although potentially diminishing overall user experience, is of far less consequence than a virtual surgery system where numerical mistakes may translate to loss of human life.

1.5.2 Ill-conditioned operator requirement

The second feature which must be supported is the ability to deal with potentially ill-conditioned operators and/or singularities during numerical processing. As will be detailed

in Chapter 2, many existing approaches to custom representation deal only with linear, time-invariant (LTI) systems having favourable numerical properties. When such methods are applied to scientific applications involving division (even potentially by zero) for example, the representations determined are likely to not bring any resource savings at all, if a representation can even be conclusively derived. Given that singularities arising from calculations such as division and trigonometric functions, ill-conditioned operators are a reality in many scientific computing, support for such situations must be present.

1.5.3 Iterative method requirement

Many scientific applications - especially in the state of the art - rely on iterative procedures such as Newton's method [12] for root finding to reach a result. In particular, a large class of applications which solve discretized partial differential equations create large sparse linear systems, which are commonly solved iteratively using the Conjugate Gradient algorithm [113]. Examples of domains and applications include: in medicine for virtual surgery simulation with haptic feedback [81], in aerospace for non-destructive testing using computational fluid dynamics [46] and in nuclear physics for fusion reactors [22]. As Chapter 2 will discuss, many existing works like those mentioned above deal only with non-iterative applications. Because iterative procedures may involve a large and varied number of iterations, scalable support must be provided to draw conclusions about their numerical representation requirements.

1.5.4 Hardware efficiency requirement

Addressing the above requirements would not be of much use if the performance gains sought in developing custom representations are lost due to poor pairing with the implementation technology which is utilized. The impact on overall system performance that choice of custom representation will have is tightly coupled to architecture and implementation technology, and these factors will vary significantly between applications. Influencing factors can include types and constraints for memory and dedicated processing units (e.g. embedded multipliers in FPGAs), as well as choice of sequential or combinational implementation of calculation units. The result is that custom representations which are

favourable to one architecture/implementation technology may be inefficient under another.

As such, an effective custom representation methodology should be sufficiently modular that it can make abstraction of these details. It should support external feedback on the performance vs. area cost vs. representation choice tradeoff and user definable objectives related to performance, area and error tolerance.

1.5.5 CAD methodology requirement

Even while supporting all the above requirements, a methodology for determining custom data representations is of little use if it cannot be effectively accessed by designers as a part of a larger CAD tool flow. In light of this, some CAD methodology requirements arise. In order to facilitate seamless integration, designer intervention should be minimized. In the ideal case, the entire process which the designer would undertake manually to derive custom representations should be automated.

To accomplish this, plug-and-play interfaces for all interactions with the rest of the tool flow are necessary. Specifically, a front end which supports languages in wide use for scientific computing software (e.g., MATLAB, C) is needed. A back end which generates automatically hardware descriptions for the custom calculation units in a variety of hardware description language (HDL) formats (e.g., Verilog, VHDL) would also be needed to maintain implementation technology independence. Further required to abstract from implementation technology is an interface for integrating hardware cost models (as discussed in Section 1.5.4).

1.6 Thesis organization

The remainder of the thesis is organized as follows. Chapter 2 provides a survey of the existing methods for automated data representation in light of the requirements discussed above. Following this, Chapter 3 deals with Satisfiability-Modulo Theories (SMT), the underlying computational framework we use to address the data representation problem. The concepts needed to comprehend SMT solvers are introduced, and a range refinement

algorithm [65, 68] is proposed to address the range aspect of bit-width allocation. Improvements over existing techniques are demonstrated through application of the method to case studies characteristic to scientific computing.

With this computational technique in place, Chapter 4 builds upon it, adding support for dealing with large abstract data types (e.g., vectors and matrices) to provide scalability to large problems [66]. Vectors are initially represented in terms of their magnitude accompanied by a loss of directional correlation information. This loss of information is addressed through the use of block vectors which enable a smoother tradeoff between problem complexity and bounds quality. An algorithm is proposed for navigating this tradeoff, and the method is applied to the computational method of Chapter 3 as well as existing techniques and demonstrated on a set of case studies.

Built atop this scalable computational framework, Chapter 5 describes the full application of the method for determining custom representation for an iterative scientific application [67]. After dealing with formation of constraints for precision expressions (as opposed to just range in Chapters 3 and 4), an analysis methodology for iterative algorithms is presented. The proposed analysis techniques are applied to iterative case studies with scientific calculation characteristics. Finally, Chapter 6 provides concluding remarks and avenues of future work.

Chapter 2

Background and prior work

In this chapter, the various approaches to delivering acceleration as discussed in Section 1.2 will be discussed, leading eventually to the adoption of field-programmable gate array (FPGA) based accelerators. Building on this, existing CAD support for FPGAs is discussed with a particular focus on numerical representation and in light of the requirements set up in Section 1.5.

2.1 Acceleration through parallelism

As mentioned in Section 1.2, a direct means of extending the problem-solving reach of computers is to perform more computations at a time (instead of just reducing the time per computation) by coordinating a number of individual computers so they work together. In such a setup, connections between individual processors allow them to share data and results, and a small piece of the overall compute task is tackled by each processor which works in parallel with all other processors giving rise to the term *parallelism*. Overall performance depends in general upon the processing power of the individual processors vs. complexity of the individual subtasks, as well as the communication capacity of the interconnections vs. the amount of data which must be passed between processors [30].

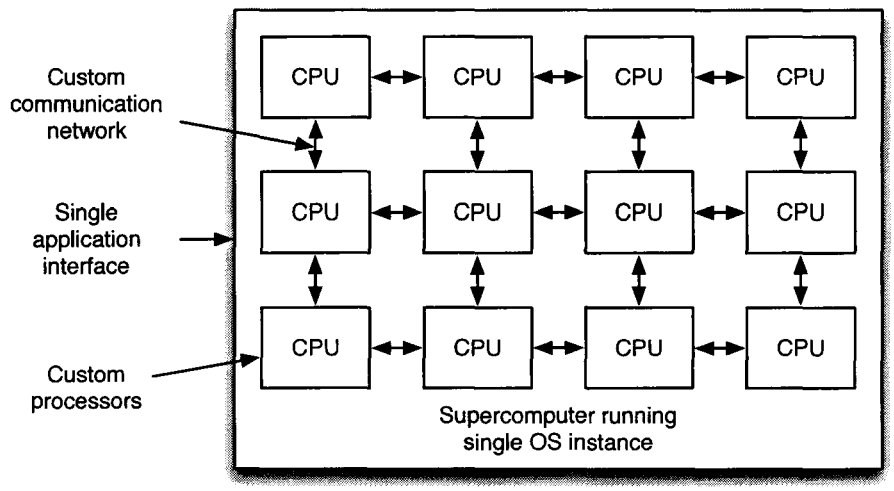
As it turns out, aside from a relatively small class of problems known as "embarrassingly parallel", partitioning a large problem so as to attain the best performance on a given

supercomputer is far from trivial [30]. This arises from data dependencies within applications, causing one step in the computation to block others. While much research has been done on automatic parallelization, results obtained manually which leverage an understanding of the data dependencies specific to an application are almost universally superior [3].

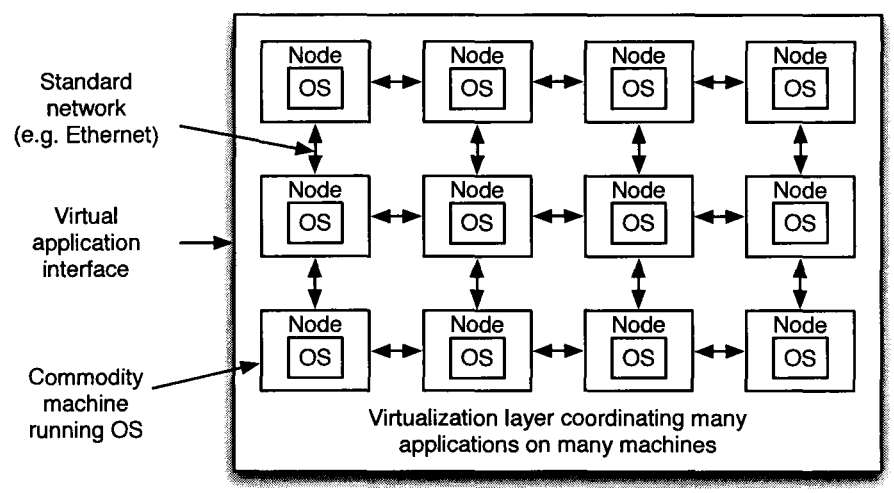
2.1.1 Parallelism via cluster computing

The obvious approach to parallelism of simply connecting numerous individual computer devices together is probably one of the earliest ways in which supercomputers were constructed. Beginning with early dedicated supercomputers, for example *IBM 7030 Stretch* [55] and *Cray-1* [70], management of resources was typically under a single application instance paradigm, using a centralized interface as depicted in Figure 2.1(a). With Moore's law [90] driving evolution in process technology, personal desktop machine compute power rose accordingly, typically providing equivalent computing power to decade earlier supercomputers. Adoption of each generation of personal desktop machine brings cost reduction through economies of scale, reducing desktop computer power to a commodity.

This commoditization of desktop computing power, along with advancement and standardization of computer network technology and protocols, has led to the more modern variant of distributed computing in grids (Beowulf cluster [40], IBM Roadrunner [6]). In such a setup, many individual standalone machines (called nodes), each running an operating system instance, are networked (usually densely) together, enabling all the machines to collaborate on one or many problems at once, a scenario shown in Figure 2.1(b). One advantage of this type of platform comes through abstraction of the node hardware. Since each node's OS instance can take care of local system tasks, a virtualization layer can be created for the application, which can handle issues such as heterogeneity of the nodes or fault tolerance/load balancing. Node hardware can range from server machines (IBM Roadrunner [6]) to low cost personal computer (PC) hardware running Linux (Beowulf cluster [40]) to even gaming consoles [76, 78] or personal computers of volunteers connected through the Internet such as in the SETI@Home [123] and Folding@Home [117] projects.



(a) Supercomputer (custom processors and network, single operating system instance and application interface).



(b) Cluster computer (virtualization software on commodity machines with separate operating system instances).

Figure 2.1: Parallelism via supercomputers and grid/cluster based computers.

Despite the benefits and past successes of cluster/grid computing, it does suffer some shortcomings. One drawback of escalating concern is cooling requirements and power consumption for large clusters made up of hundreds of thousands of nodes, which easily reach into the range of tens to hundreds of kilowatts. Size and setup cost for such a system are prohibitive for small organizations having only occasional needs for extensive compute power, and leasing time on supercomputers can also be expensive and unacceptably non-deterministic. Such organizations are also unlikely to benefit from volunteered compute power such as in SETI@Home or Folding@Home, both because of lack of participant goodwill and because of sensitivity of data.

Combined with the drawbacks above, the desire to bring greater amounts of compute power in-field (e.g., arctic seismic analysis [71]) leaves cluster/grid computing at a loss. Likewise, applications with form-factor (real-time, energy, weight, size) constraints such as deployed/embedded systems pose similar challenges with examples being real-time signal capture and processing for communications (e.g. mobile phones [127]) and diagnostics/visualization (e.g., medicine [121]). One answer to this problem follows the historically successful strategy of further integration as discussed next.

2.1.2 Parallelism via multicore

The cluster based approach to parallelism has played out relatively successfully over the last decade or so largely by riding Moore's law, whereby each new generation of commodity processor was able to operate at a faster clock speed, as well more memory could be integrated per device and network speeds were increasing. As such, clusters with more, faster nodes could be built, bringing higher computational throughput. Recently however, the escalating capabilities of single processors have begun to wane because of diminishing returns on three fronts: memory, instruction level parallelism and power [4]. In answer, microprocessor vendors have for the last 5 years (at least) pursued *multicore processors*.

Figure 2.2 illustrates the multicore concept, note the resemblance to Figure 2.1(a) if the custom processors are replaced with general processor cores, the custom communication network is replaced by an on-chip communication network, and the entire system is integrated onto a single device as opposed to being built out of individual components. In

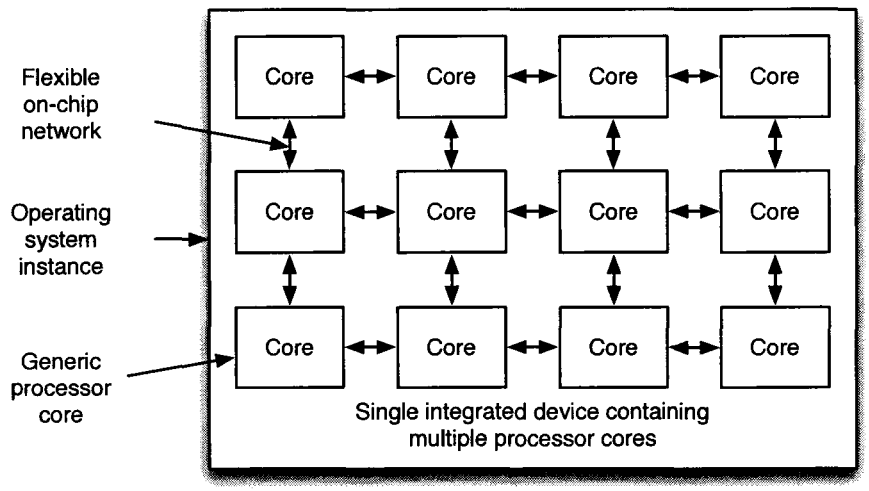


Figure 2.2: Parallelism via multicore devices.

addition, the role of the single (OS) instance of Figure 2.1(a) can in theory be filled by a traditional OS ported to run on such a multicore architecture. The network carries data and directives between the processors thereby enabling them to collaborate and choices range from dedicated custom bus architectures [69] to general network-on-a-chip (NOC) [39]. The processors themselves carry out tasks on the data, and can be special purpose or general, even on the same chip as for the Cell Broadband Engine [44].

What is attractive about this model of computation is the resemblance it bears to both the supercomputer model and at the same time traditional single processor machines, but with the advantages of power, latency and physical space savings brought by the integration. However, what does not carry over is the performance gains which traditionally came for free due to higher clock rates in each new processor generation. Challenges which face multicore integrated devices are presented in [4], with the central ones being related to power, memory and instruction level parallelism. In terms of power, the number of devices which can now be integrated into a single device coupled with the high switching

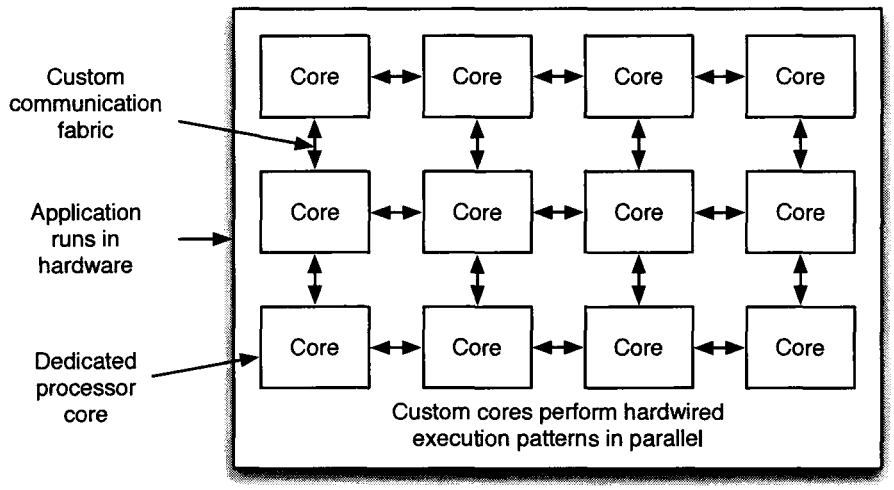


Figure 2.3: Parallelism via customized ASICs.

frequency results in difficulties both in getting sufficient power on to the chip, as well dissipating heat out of the chip. In terms of memory, with limited bandwidth for moving data on/off of chip keeping a growing number of increasingly more powerful processing units busy presents challenges [130]. Finally in terms of instruction level (fine-grained) parallelism, the majority has been exploited already through the evolution of microprocessor architectural innovations (e.g. branch prediction, out-of-order execution, speculation) [51]. New avenues of instruction level parallelism come with diminishing returns, and coarse-grained parallelism in applications must now be found and exploited. *This important shift in the source of performance gain has reopened interest in customization of the engines which are placed in parallel, a topic highlighted in the next section.*

2.1.3 Parallelism via ASICs

Alongside the evolution in general purpose processors based systems from custom platforms to grids to multicore, some isolated application domains have pursued greater computational power by custom-designing individual processor hardware. Such custom-designed hardware is implemented as application specific integrated circuits (ASICs). Taking this approach, the speedup comes from crafting the processor to be particularly efficient for the (usually narrow) set of execution patterns specific to that application domain.

This strategy has been particularly successful in the graphics processing application domain, largely as a result of two factors. First, the highly data-intensive nature of graphics processing relieves some of the hardware design complexity thus lending feasibility to the prospect of building dedicated accelerators. Second, the significant non-recurrent engineering (NRE) costs were financed relatively early on by consumers with high-end gaming interest who were willing to pay a premium for performance, catalyzing the cycle of increasing adoption and reducing cost. The culmination of this cycle over the last couple decades is the relegation of graphics processing units (GPUs) to the realm of commodity hardware.

A similar phenomenon has occurred in the digital signal processing (DSP) domain which, on the design side, shares the data-intensive nature of graphics processing. The economic motivation however came primarily from the embedded systems domain, specifically mobile multimedia where power efficiency was the important objective. Reducing computational effort wasted on execution overhead provides greater energy efficiency and thus longer battery life. Similarly to GPUs, consumer adoption led to large manufacturing volumes driving device cost down.

Aside from the decades old graphics and digital signal processing domains, this strategy is still in use today. A notable example is the development of a molecular dynamics supercomputer known as Anton [112]. This platform consists of dedicated chips specifically designed to be efficient for performing molecular dynamics calculations, joined by a custom connection infrastructure designed to be most efficient for the data traffic patterns exhibited in molecular dynamics calculations.

While the potential of application specific supercomputers stands well above anything

achievable with general purpose supercomputing platforms, the obvious drawback is the significant cost to develop such a machine. While custom processor design improves performance, it reduces flexibility which narrows the scope of applicability, and eliminates the opportunity to amortize the design cost over many applications. As technology advancements allow new generations of general purpose platforms to match older generation dedicated hardware, custom platforms representing significant investments can be obsoleted perhaps before even recovering their NRE costs.

It is on this point that the importance of the recent shift to performance through parallelism over performance through clock speed mentioned in the previous section hinges. In the past, the performance gains from adopting a new generation processor came so cheaply that the enormous (by contrast) development costs of dedicated hardware could not be justified despite substantially better performance. The increased design effort required for software on multicore vs. traditional CPUs has closed the development cost gap between general purpose software and dedicated hardware platforms, making custom hardware worth considering in light of the potential performance benefit.

On a related note, there has been recent interest in repurposing GPUs as a multicore platform, in order to leverage the maturity of the hardware, a movement known as "general purpose computing on GPUs" (GPGPU) [84]. The maturity of the technology has produced current day GPUs with hundreds of cores capable of performing IEEE-754 compliant floating-point operations at high rates (GHz) and, while lacking the sophisticated control features of modern day microprocessors, can often deliver higher performance by virtue of the parallelism. Recognizing that many scientific computations can be broken down into calculations which GPUs can handle very quickly, significant effort has been invested both to 1) directly map applications to GPUs ([18, 20, 79]) and 2) develop tools/compilers to assist/automate the mapping process ([29, 57]).

Despite this interest in reusing GPUs, the fact remains that GPUs are domain specific and while they perform excellently for calculations which can be made to resemble graphics processing, they cannot compete (performance wise) in domains where the calculations look very different. In such domains, application specific hardware will tend to provide better performance (in terms of both computational throughput and power efficiency, e.g. [45]), but with higher development cost which has been some of the motivation for using

GPUs. The next section looks at a trend which significantly closes the development cost gap, significantly raising the viability of application specific hardware processors.

2.1.4 Parallelism via FPGAs

It has been shown in the previous sections that a paradigm shift has occurred where performance gains are now derived from increased parallelism rather than from increased clock speed. Furthermore we have seen that dedicated hardware systems deliver higher performance than reconfigurable software platforms but at increased cost. While the shift from clock speed to parallelism driven performance gains has closed the gap in development cost between multicore software and dedicated hardware. At the same time, in the last decade field-programmable gate-array (FPGA) technology advancements have significantly reduced the performance gap to ASICs. The reduction of this gap on both sides has created an opportunity for research into *reconfigurable computing platforms* [122] - particularly those based on FPGAs.

The attraction of FPGA-based reconfigurable computing platforms is their ability to provide "not-much-less-than" ASIC performance for "not-much-more-than" multicore development effort. While FPGAs have higher logic delays and lower integration capacities than ASICs (accounting for the lower performance), the physical platform can be reconfigured and therefore used over many applications better amortizing the already lower NRE costs. At the same time, while architecture design requires more effort than software design as for GPUs and multicore, maturity of FPGA tool support makes the development cost gap smaller than for ASICs.

While the superior performance potential of FPGA-based platforms has been recognized, so too has the fact that development cost remains a roadblock to adoption [45]. To address this, much investigation has been done as of late into improving tool support to further reduce development effort. Figure 2.4 (an extension of Figure 1.2) illustrates the aim of such research, to bring the higher performance gains associated with FPGAs at the lower development cost associated with GPUs. It is this broad category to which this thesis belongs, providing automation support for reducing the design effort of mapping applications onto FPGAs, and the next section looks more in depth at existing tool support.

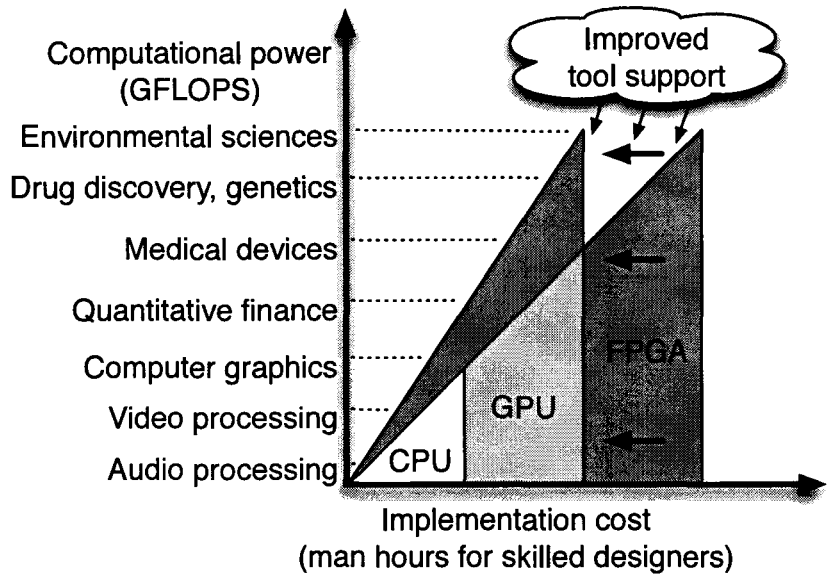


Figure 2.4: Reduced design effort through better tool support.

2.2 CAD support for FPGAs

The previous section has discussed the acceleration benefits of dedicated hardware which are made cost feasible through FPGA platforms. In order to leverage this acceleration while not suffering an inordinate increase in development cost (also discussed in the previous section), substantial research has been done in providing CAD support to lower development complexity for said platforms. In particular, a fair amount of effort has surrounded raising the abstraction of design entry, through so called behavioural synthesis [115]. Direct synthesis of hardware from a behavioural model in C or SystemVerilog relegates the difficult control intensive state machine design tasks to the CAD tool, thereby improving designer productivity. However, state of the art behavioural synthesis tools are still unable to produce designs as efficient as those created by a skilled designer implementing a design at the register transfer level (RTL) [9, 110].

Behavioural synthesis tools have found greatest acceptance in application domains and environments where design is feature driven rather than performance driven, such as multimedia. In feature driven design, competitive advantage comes from feature set and quick time to market, with efficiency being a secondary concern. On the other hand, in performance driven design, efficiency is the primary concern, for example in many embedded systems, which have form factor constraints. In this case, tool support still exists but the focus is different, with the tool being operated by a skilled designer working at the RTL.

As identified in Chapter 1, choice of numerical representation is a problem of significance for more efficiently using resources for better performance. Manual solution to this problem has been estimated to account for 25-50% of the design time in some scenarios [88]. Thus, to improve productivity tool support is necessary, forming the motivation for this thesis. In this section an overview of the problem is provided, as well as existing approaches which have been applied to address this problem.

2.2.1 Problem aspects

Before discussing specific approaches to solving the bit-width allocation problem, this section summarizes the aspects of the problem which various approaches seek to address. The first aspect involves the fact that discovering the minimum number of bits necessary to accurately represent an intermediate variable from a calculation is a two part problem. Both the range and precision required must be determined, from which can be inferred the required number of *exponent* and *mantissa* bits in floating-point, or *integer* and *fraction* bits in fixed-point.

The second aspect deals with cost models for both error and hardware. The goal of works in this category is to provide easily calculable yet reasonably accurate estimates of impact on numerical quality and resource requirements for a given choice of representation scheme. Put another way, such approaches provide the means to make a statement such as: for a choice of representations for the intermediate variables in a dataflow, here is the numerical deviation from infinite precision (true value) and the hardware resources required to implement the dataflow. The need for reasonable accuracy and easy calculation involves the third aspect described in the next paragraph.

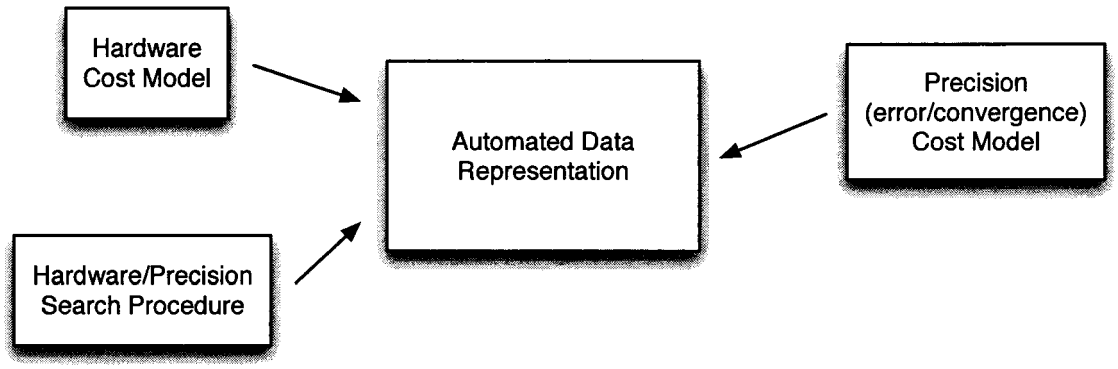


Figure 2.5: Summary of aspects which existing works address.

The final aspect involves search procedures and metrics for navigating the solution space of possible representation choices across all intermediate variables. Using the models from the previous paragraph at the core of the search, these procedures will propose a representation scheme, evaluate the error/hardware cost (through the models) and update the representation scheme in order to improve a metric which reflects the overall goal of the search (i.e. error optimization, hardware optimization or a hybrid).

While the first aspect is independent of the latter two (meaning that models and searches can apply to range and/or precision), the latter two are related, but essentially orthogonal (meaning that an improved cost model brings benefit to essentially any search procedure). Many works contribute to one or multiple of these aspects, which are summarized in Figure 2.5. Apart from the aspects of the problem however, there are also requirements which solutions must satisfy to be useful in the context of scientific computing. Existing approaches to solving the problem in light of these requirements (identified in Section 1.5) are discussed next.

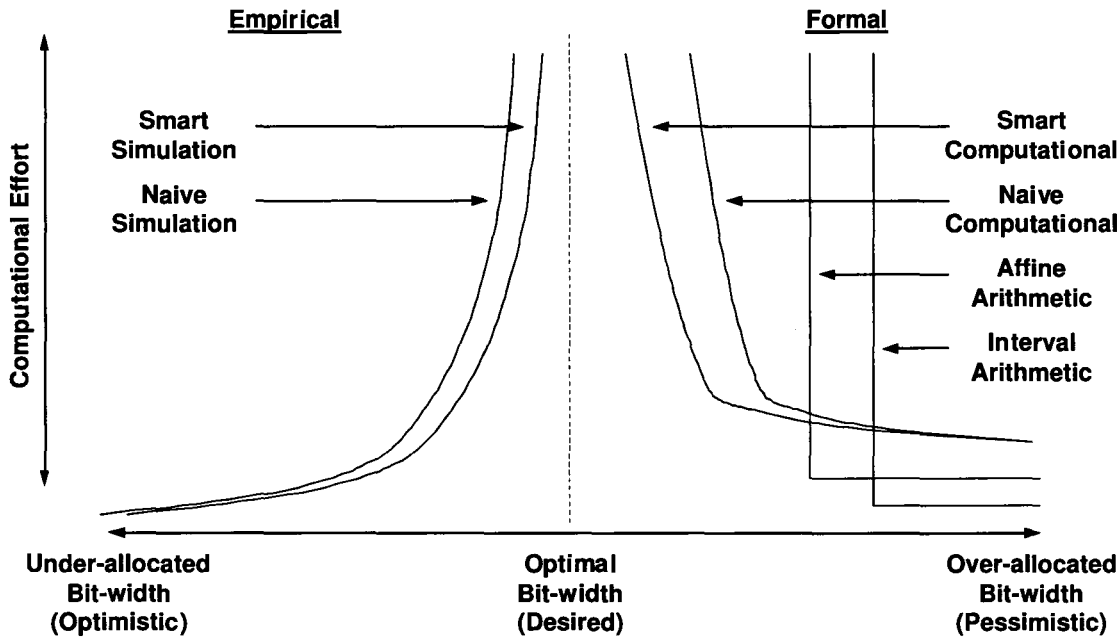


Figure 2.6: Overview of approaches to bit-width allocation.

2.2.2 Existing approaches

In Section 1.5, the primary requirements which must be met by a design method in order that it will produce representations viable for scientific computing were presented. The first of these requirements is robustness and aligns well with the categorization of existing methods between analytical (formal) which can guarantee robustness and simulation based (empirical) which cannot. Figure 2.6 summarizes the landscape of approaches which will be discussed in more detail below. Two important points can be seen, first that empirical methods on the left produce tight but non-robust representations, and investing greater simulation effort enables getting nearer to the optimal. Second is that formal approaches produce pessimistic yet robust bit-widths, but existing methods reach a limit where no further improvement is possible despite extra compute power. The computation methods proposed in this thesis however enable a tradeoff between bit-width and computational

effort like for simulation, but while still maintaining robustness.

Simulation based approaches rely on a representative input data set and work by comparing the outcome of simulation of the reduced precision system to that of the "infinite" precision system, "infinite" being approximated by "very high" - e.g., double precision floating-point on a general purpose machine. The statistics which arise from the simulation provide insight on the precision and range of the intermediate variables and a number of techniques have been proposed along these lines, which are mostly differentiated by 1) search algorithms and metrics which are relatively independent from the error bounds estimation, or 2) how information is extracted from the simulation.

In terms of search algorithms and metrics, the error estimation as discussed below is leveraged to decide how to update a target precision scheme. Because the search and error bounding facets are strongly decoupled, while many of the approaches mentioned here were proposed within a simulation error bounding framework, analytical error is used by some and may potentially be used by all of them. For example, integer linear programming (ILP) and mixed-ILP are used by [28] and [26, 27] respectively. Also, genetic algorithm based searches are used by [48, 125], with many variations existing [15, 16, 19, 25, 47, 73, 75, 120]. What is common to all these is that while essentially independent from the error estimation, there are tight links between the hardware cost models and the search procedures, and both of these can be strongly influenced by the implementation technology. Also in contrast to the independence from the specific method of error estimation, good error feedback from the estimation procedure is essential to effectively guide the search. Impact of choice of error estimation on estimate quality is discussed next.

While the search decides how to update a potential choice of representation based on feedback from the error estimator, the error methods deal with deriving error bounds from a potential representation scheme posited by the search. Some of the error methods above operate by replacing the standard data types in an implementation language with augmented ones designed to carry more information. A class for C++ is provided by [14], similarly for C by [21, 74] and MATLAB [86]. In addition to replacing data types, function and library replacements can also be used, a particular example being automatic differentiation which augments the standard operators so that the derivative is calculated on execution as well. This method is used in simulation by [37] and [38] to collect sensitivity as well as

range data during simulation. Simulation environments have also been proposed, such as [126] modification of source code for simulation, Fixify [7] and FRIDGE [63] and some perform automated conversion from a high level dataflow into custom representations like [83] from SystemC and [5, 93] from MATLAB. A similar conversion from MATLAB is done by [114], but targets the simulation more directly at error behaviour by creating a difference system between the full and reduced precision systems. In this way, much less data and simulation time are required as the simulation effort is not diluted in extracting system statistics instead of error behaviour statistics.

Simulation based methods have found significant adoption in the digital signal processing (DSP) application domain, as well as some embedded systems applications, due to some common properties. For example, many DSP systems can be characterized very well (in terms of both their input and output) using statistics such as expected input distribution, input correlation, signal to noise ratio and bit error rate. This enables efficient stimuli modelling providing a framework for simulation, especially if error (noise) is already a consideration in the system (as is often the case for DSP [101]). Also, given the real-time nature of many DSP/embedded systems applications, the potential input space may be restricted enough to permit very good coverage during simulation.

In contrast to the above, for general scientific computing stimuli characterization is often not as extensive as for DSP, and there is often minimal error consideration provided. Furthermore, robustness which is generally not necessary for DSP applications, is necessary for scientific computing. At the same time, statistical methods can easily miss minute yet important regions of the simulation space entirely [60]. As a result, situations not covered by the simulation stimuli can lead to overflow or error excitation conditions which ultimately can lead to incorrect and/or unreliable calculation results. Because of these differences between general scientific computing applications and the DSP/embedded systems application domain, simulation based methods cannot be relied upon for scientific computing. On the other hand, despite the fact that they tend to produce less compact data representations, analytical approaches deliver provable limits for range and precision requirements from which robust representations can be derived. The most straightforward analytical method is known as *range* or *interval* arithmetic and is described next.

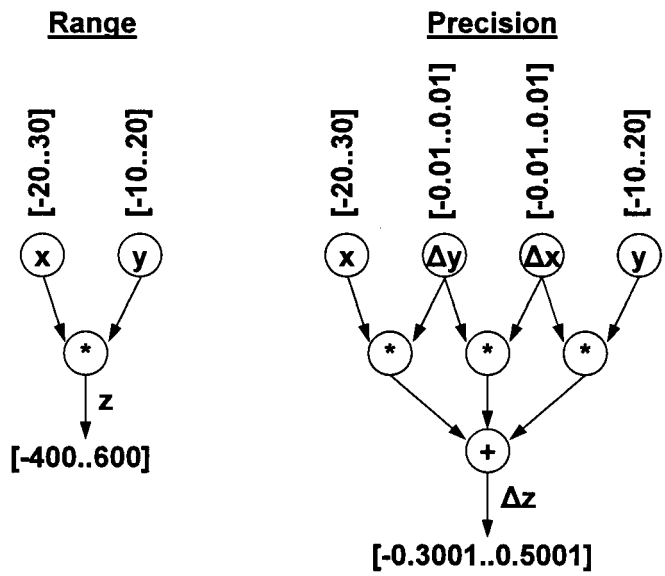


Figure 2.7: Example of interval arithmetic (IA) operation.

Interval Arithmetic

Interval arithmetic (IA), first proposed by Moore [91] is the most straightforward approach to the problem of determining bounds on values within a calculation. It operates by establishing worst case bounds at each step of the calculation. Each variable is replaced by an interval e.g. $x \rightarrow \{x|x_L \leq x \leq x_H\}$ and interval analogues of the basic operations are defined. For example, the $+$ operation on intervals $\{x|x_L \leq x \leq x_H\} + \{y|y_L \leq y \leq y_H\}$ produces the interval $\{z|(x_L + y_L) \leq z \leq (x_H + y_H)\}$. Intervals are then propagated through an entire calculation, producing reliable bounds at each stage including the output.

With a means of calculating reliable bounds in hand, limits for the range aspect of the data representation problem can be obtained directly from the application of IA. To adapt IA for obtaining limits for the precision aspect, each variable is supplemented with a perturbation variable which is propagated through the operation of interest to obtain a precision analogue of the calculation. For example, the multiplication $z = xy$ would be

transformed:

$$\begin{aligned}(z+\Delta z) &= (x+\Delta x)(y+\Delta y)\\ xy+\Delta z &= xy+y\Delta x+x\Delta y+\Delta x\Delta y\\ \Delta z &= y\Delta x+x\Delta y+\Delta x\Delta y\end{aligned}$$

yielding an uncertainty variable Δz for the variable z. Following this procedure, an uncertainty expression can be derived for each intermediate variable in an entire calculation, similar to the way traditional uncertainty analysis has been performed in the physical sciences [111]. Under this model, input uncertainties of any form (including quantization noise) can be propagated through the data path. Furthermore, quantization occurring at any point throughout the calculation is addressed by simply injecting the resultant quantization noise into the appropriate perturbation variable. The application of IA provides an error model, as per the representation problem aspects discussed in Section 2.2.1.

Figure 2.7 illustrates the process for simple multiplication $z = xy$, for $-20 \leq x \leq 30$ and $-10 \leq y \leq 20$. The left half of the figure shows application of IA directly to the multiplication yielding a range for z of $-400 \leq z \leq 600$. On the right hand side of the figure is the precision calculation, under the assumption of uncertainties of ± 0.01 for both x and y, yielding uncertainty in z of $+0.5001, -0.3001$.

With the IA evaluation of the multiplication mapping intervals to intervals, clearly complex calculations can be tackled by simply performing IA for each operation, feeding intervals obtained in one stage of the calculation into the next. This however is where IA begins to experience difficulty. While it so happens that the ranges obtained in Figure 2.7 are in fact tight, *this is only due to the lack of interdependencies between intermediates in the calculation*. IA is incapable of retaining any correlation information between variables and as a result overestimates the range when correlations are involved, a phenomenon known as range inflation. To make matters worse, each subsequent stage of the calculation further compounds the overestimation, exacerbating the range inflation effect. Some attempts to mitigate this have been undertaken by recognizing that the degree of overestimation is correlated to the size of the interval, giving rise to so called multi-interval analysis [8]. This approach separates each input variable into a number of intervals instead of just one and independent IA is performed on each sub-interval and the results across all sub-intervals

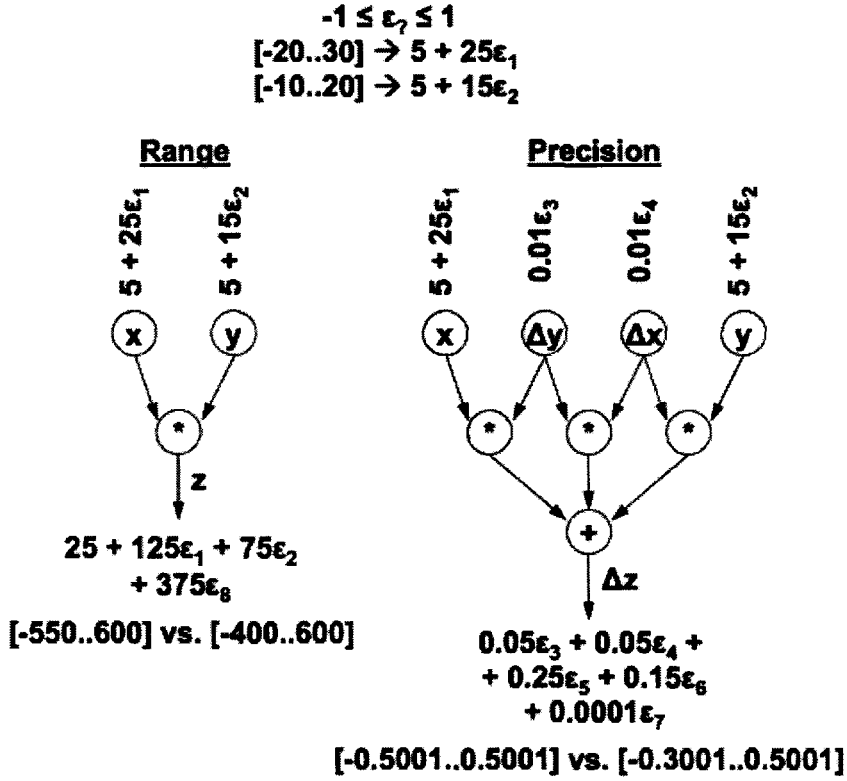


Figure 2.8: Example of affine arithmetic (AA) operation.

are merged. While over inflation is reduced, the method does not scale to the point of eliminating it. To address this, a means of keeping track of correlations is required, and such a method is detailed next.

Affine Arithmetic

Keeping track of correlations between variables enables those correlations to cancel when recombined at later points in a calculation, thus mitigating range inflation. One approach to this which retains first order approximations is known as *affine arithmetic (AA)* [119]. In AA, a fixed interval is replaced by an affine expression in a variable (e.g., ε) over the range

$[-1..1]$, and affine expressions are propagated and compounded throughout the calculation ([80] provides a good summary of affine approximations for common operations). This enables dependencies which operate in opposing directions to cancel each other out when combined, thereby reducing overestimation of the resultant range.

Figure 2.8 illustrates the concept using the same example calculation from Figure 2.7. The ranges of $\{x| -20 \leq x \leq 30\}$ and $\{y| -10 \leq y \leq 20\}$ are mapped to affine expressions $x = 5 + 25\varepsilon_1$ and $y = 5 + 15\varepsilon_2$, where $-1 \leq \varepsilon_1, \varepsilon_2 \leq 1$. As before, the left half of the figure shows the range aspect, and the expression for z is obtained as:

$$(5 + 25\varepsilon_1) \times (5 + 15\varepsilon_2) = 25 + 125\varepsilon_1 + 75\varepsilon_2 + 375\varepsilon_1\varepsilon_2.$$

Due to the cross term however, this expression is not itself affine and as such could not be propagated as is under the AA paradigm. This can be dealt with in a number of ways, one of which is to note that $\varepsilon_1 \times \varepsilon_2$ can itself take on values $[-1..1]$, and so to assign a new epsilon term - specifically ε_8 in the figure.

Having now an affine expression instead of a simple interval for each intermediate, the limits on any intermediate can be obtained by pushing the affine expression to its maximum and minimum e.g., for the lower limit set each $\varepsilon_?$ in the expression with a negative coefficient to $+1$ and each a positive one with -1, and vice-versa for the upper limit. The left half of the figure demonstrates the limits obtained in this way to be $[-550..600]$. Examining the right side of the figure, the same process as before has been employed to derive the precision expressions, and the affine expressions for the inputs are propagated through just as for the range, along with the numerical limits.

It is important to note that while the numerical bounds on z and Δz produce larger ranges than those obtained by IA, what differs is in the corresponding expression. With AA, dependency information relating to x and y is retained, leaving the chance for them to cancel at a later point. Many approaches to the data representation problem use AA at their core, and they are differentiated primarily in how they deal with non-affine terms which arise due to non-affine operations, as well as how to recast complex affine expressions to simpler ones when they become unwieldy.

While AA has been proven to generate tighter ranges than IA, and therefore more compact data representations [98, 100, 106], retention of correlations is still limited to the first

order (linear correlations). As a result, whenever strongly non-affine operations (i.e., with high curvature) occur, AA is not able to keep up with the degree of the correlations and as a result can severely overestimate ranges. A particular case of this is division, which also causes problems due to *range inversion* - the property of division that large numbers in the denominator are mapped to larger numbers in the quotient and vice-versa. Yet another problem with division is if range overestimation extends the range of the denominator to include zero. Consider the following example:

$a \in [0.01, 100]$	$a \leftarrow 50.005 + 49.995\varepsilon_1$	*IA vs. AA*
$1/a \in [0.01, 100]$	$1/a \leftarrow 50.005 - 0.0049995\varepsilon_1 + 99.980001\varepsilon_2$	$[0.01, 100]$ *vs.* $[-49.98, 149.99]$
$(a)(1/a) = 1$	$(a)(1/a) \approx 2500.4 + 2499.8\varepsilon_1 + 4999.5\varepsilon_2 - 0.125\varepsilon_3 + 4998.5\varepsilon_4$	$[10^{-4}, 10^4]$ *vs.* $[-9998, 14998]$

where affine approximations for $1/a$ and $(a)(1/a)$ from [80] are used. The range for the reciprocal $1/a$ is not overestimated too badly at about twice the width of the true interval. When this reciprocal is multiplied by the original a, the result should be 1, but the AA expression yields a much larger range, worse even than IA. While the examples earlier excused slight overestimation of wider ranges produced by AA over IA in favour of retaining the correlations, this case does not experience any of that benefit. Notice that the only ε from the $(a)(1/a)$ expression actually correlated to a is ε_1 with coefficient 2499.8 or about 20% of the resultant range. The problems actually arise as soon as the calculation of the reciprocal, where it is clear that the resultant range is dominated by the newly introduced ε_2, uncorrelated to ε_1. What makes this overestimation more serious is that a strictly positive interval (recognized even by IA) is now approximated as one containing zero, meaning no subsequent division may use this variable as the denominator. While the dominant application domains for AA tend to use division seldom [33] - accounting partly for the success of AA in said applications - the division operation is a staple of scientific computing and as such must be addressed in a satisfactory way, and such is a goal of this thesis.

Symbolic and polynomial techniques

While the goal of IA and AA has been to retain correlations between variables to a greater degree, some symbolic methods have been proposed as well to address this shortcoming. In [24] error expressions are generated through perturbation and linearized for simplification. Symbolic differentiation is compared to AA as well as AA with scalar coefficients replaced by intervals, known as general interval analysis (GIA) in [23] and partial derivatives are used by [125] while symbolic noise models are used by [1, 129]. While in many cases these approaches can outperform IA and AA, when the dataflow becomes complex, scalability can suffer badly and a tradeoff between error bound tightness and run-time must be reached, such as made possible by the linearization in [24]. However, in some cases techniques which pertain specifically to a given class of calculations may be useful, such is the case with [116] which deals with polynomials. Because of their importance and ubiquity, polynomials are the target of many approaches as detailed below.

In addition to IA and AA, as well as the symbolic methods above, which are meant to be applicable to any operation (no matter how poorly performing), there is a set of approaches which focus directly on polynomials and use their properties derive more suitable implementations in high level synthesis. Leveraging the properties of fixed-point representations in this context (which resemble finite fields), more efficient representations can be derived. Approaches are diverse and include arithmetic transform [102, 107, 108], vanishing polynomials [43] and factorization with multivariate Grobner basis [104] and common subexpression elimination [53, 54].

All these methods operate by replacing a polynomial with another which is equivalent (within desired accuracy bounds) over the limited region of interest in the inputs. They have been applied to fixed-point data types only, and extensions beyond polynomials are made through Taylor series (which is itself polynomial). Although Taylor series approximations for large, complex dataflows may not be feasible over desired dynamic range in some scientific applications, these methods do offer robustness. Furthermore, Taylor series approximation of functions (e.g. a small block) occurs frequently in scientific computing, and the methods above can explore the extra dimension of error in the function approximation and thereby perhaps achieve smaller bit-widths. As a result, they can be used in a

complementary way with automated data representation for scientific applications. A simple example is to use such methods to obtain a functional unit for a trigonometric function with clearly defined error behaviour which can be used as a black-box unit for applications analyzed under the approach described in this thesis.

Support for iterative methods

While the above formal methods have been applied extensively in the digital signal processing (DSP) domain, it has been emphasized that adoption has been limited in scientific computing. Ill-conditioned operators such as division near zero are essentially untreated, and only strictly linear iterative methods in the context of DSP have had even a cursory treatment (e.g. infinite impulse response filters [33, 80]). Lack of support for these two key characteristics of scientific calculations has significantly hindered the use of analytical methods for custom representations in this domain.

As a result, empirical methods have historically been used to obtain custom representations for numerical methods. The simulation based nature of empirical methods allows them to support virtually any application, since the range and precision information is derived directly from its execution. For example, a recent application of real-time finite element method modelling for haptic feedback [81] used this approach to obtain custom representations. These custom representations were required to achieve a high enough degree of parallelism to satisfy the real-time constraint.

At the same time however, the need for robust representations has been shown, and simulation based methods are unable to support this robustness. Furthermore, when iterative methods are used to replace exact solutions to difficult problems, the run-time can be very long [113]. In this case, the already extensive times required by simulation based methods are amplified due to the execution of each case requiring a longer time. On top of this, more cases are required to achieve good coverage of calculation scenarios, because variables in each iteration should be considered unique - scaling the complexity of the calculation with the number of iterations until termination.

2.3 Summary

In this chapter different approaches to accelerating computational tasks have been presented. Among these, custom and reconfigurable hardware accelerators based on FPGAs have been identified as providing a favourable performance/cost tradeoff. The implementation effort for porting applications to these FPGA-based accelerators is eased by CAD support, and one key step in the design process to enable higher performance is the assignment of custom data representations.

Existing work on this custom representation step has been on one of two fronts. On one front are simulation based methods which require extensive execution times and do not guarantee robustness. On the other front are existing analytical techniques which focus primarily on linear time invariant (LTI) systems such as in DSP, and do not provide support or ill-conditioned operators or iterative methods.

Since all three of these properties (robustness requirements, ill-conditioned operators and iterative methods) are characteristic of general scientific computing applications, the existing approaches cannot provide support for deriving custom representations when porting scientific applications to FPGA-based hardware accelerators. The rest of this thesis details the method proposed to satisfy these criteria.

Chapter 3

Satisfiability-Modulo Theories for the range problem

This chapter begins with a motivational example highlighting the shortcomings of existing approaches to solving the range determination aspect of the automated representation problem . Thereafter, the computational framework of Satisfiability-Modulo Theories (SMT) is introduced, including solver operation. Building on this foundation, SMT is applied at the core of the range refinement method [65, 68] targeted at the range aspect of automated representation. Application of the method is demonstrated on a number of case studies.

3.1 Motivation

Let $\mathbf{d}$ and $\mathbf{r}$ be vectors $\in \mathbb{R}^4$, where for both vectors, each component lies in the range $[-100, 100]$. Suppose we have:

$$z = \frac{z_1}{z_2} = \frac{\mathbf{d} \cdot \mathbf{r}}{1 + \|\mathbf{d} - \mathbf{r}\|^2}$$

and we want to determine the range of z for integer bit-width allocation (i.e., solve the range problem). Table 3.1 shows the ranges obtained from simulation, affine arithmetic and the proposed method. Notice that simulation underestimates the range by 2 bits after 540 seconds, $\approx 5\times$ the execution time of the proposed method (98 seconds). This happens because only a very small but still important fraction of the input space where $\mathbf{d}$ and $\mathbf{r}$

Table 3.1: Motivational example.

Var.	Empirical		Formal			
	Simulation		Affine		Proposed	
	Range	Bits	Range	Bits	Range	Bits
z_1	[-3.7e4,3.7e4]	17	[-4e4,4e4]	17	[-4e4,4e4]	17
z_2	[1,1.4e5]	18	[-8e4,1.6e5]	18	[0,1.6e5]	18
z	[0,1e4]	14	∞	–	[-864,4e4]	16

are identical (to reduce z_2) and large (to increase z_1) will maximize z. In contrast, the formal methods always give hard bounds but because the affine estimation of the range of the denominator contains zero, affine arithmetic cannot provide a range for the quotient z. *It will be shown that this scenario is handled correctly by the method proposed in this chapter which maintains all the benefits of analytical (formal) methods while at the same time visibly tightening the range of the operands. The key to this is the application of the recent developments in SAT-Modulo Theory and details of its operation are discussed next.*

3.2 Fundamentals of SAT-Modulo Theories

Since the SAT-Modulo theory is an extension of the concept of Boolean SAT, this section begins with a refresher of the Boolean satisfiability problem. Building on this, extensions to other logic systems are described in Section 3.2.2, and basic solver principles are explained in Section 3.2.3.

3.2.1 Boolean SAT refresher

Boolean satisfiability (SAT) is a well known problem which seeks to answer whether for a given set of clauses (disjunctions) in a set of literals (boolean variables and their complements), there exists an assignment of those variables such that all the clauses are (their conjunction is) true. The variant of SAT where all clauses contain 3 literals (3SAT) is

known to be NP-complete [103]. As an example of a SAT instance, take Boolean variables x, y and z and clauses

$$\{\bar{x},y\},\{\bar{y},z\},\{\bar{x},\bar{z}\},\{x,y,z\},\{x,y,\bar{z}\},\{x,\bar{y},\bar{z}\}$$

which comprise an unsatisfiable instance because no assignment produces at least one true literal in each clause. To prove this consider that if we assign $x = 1$, propagating yields a reduced instance:

$$\{y\},\{\bar{y},z\},\{\bar{z}\}$$

since each clause containing x is true regardless of the other literals, and the value of the clauses containing $\bar{x}$ depends only on the other literals in those clauses. However the first and third of these remaining clauses necessitate $y = 1$ and $z = 0$ for satisfiability, but these assignments invalidate the second clause. From this we can infer that no satisfying assignment exists having $x = 1$. Considering now $x = 0$, the same reasoning as above can be used to reduce the instance to:

$$\{\bar{y},z\},\{y,z\},\{y,\bar{z}\},\{\bar{y},\bar{z}\}.$$

Now, assigning either $y = 0$ or $y = 1$ yields $\{z,\},\{\bar{z}\}$ which is a contradiction and thus unsatisfiable since y must be either 0 or 1 and neither is satisfiable, we can conclude that no satisfying assignment exists having $x = 0$. Repeated application of this reasoning on x indicates the entire instance is unsatisfiable.

Many Boolean SAT solver implementations (e.g. zChaff [105] and MiniSat [32]) operate along the same lines as the above example, applying two step recursion: a *Decision step* where a variable is selected upon which to branch, and a *Propagation step* which applies the result of the decision step to all affected clauses, inferring values (using the rules of Boolean logic) for other variables as appropriate. When a contradiction arises (a clause with all literals decidedly untrue), the solver backtracks and reverses an assignment. At any point where all variables are decided (or inferred) and there is no contradiction, the problem is solved and the satisfying assignment is precisely the sequence of decisions (or inferences) on the variables. If on the other hand, the entire assignment space has been covered, leading to a contradiction in each case, the instance is concluded to be unsatisfiable. In this case, the cover of the assignment space serves as a proof of unsatisfiability.

3.2.2 Extending to other logics

By extending the Boolean SAT concepts of the previous section to other first-order logic systems, SAT-Modulo theories (SMT) arise. Under the theory of real numbers, Boolean variables are replaced with real variables and clauses are replaced with constraints. This gives rise to instances such as: does there exist an assignment of $x, y, z \in \mathbb{R}$ for which $x > 10$, $y > 25$, $z < 30$ and $z = x + y$. For this example there is not i.e., this instance is *unsatisfiable*.

As the above example reflects, instances are given in terms of variables with accompanying ranges and constraints. As in the case of Boolean SAT, the solver attempts to find an assignment on the input variables (inside the ranges) for which all the constraints are satisfied. Also like Boolean SAT, most implementations follow a 2-step model in which: 1) the *Decision step* selects a variable, splits its range into two, and temporarily discards one of the sub-ranges then 2) the *Propagation step* infers ranges of other variables from the newly split range. Unsatisfiability of a subcase is proven when the range for any variable becomes empty which leads to backtracking (evaluation of a previously discarded portion of a split). Again, akin to Boolean SAT, the solver proceeds in this way until it has either found a satisfying assignment or unsatisfiability has been proven over the entire specified domain. The next section provides more detailed insight into SMT solver operation, particularly as it pertains to the problem of range refinement.

3.2.3 Solver operation

As the previous section has highlighted, the operation of SMT solvers is closely analogous to that of Boolean SAT solvers. Where the SMT solver operation does differ significantly from Boolean SAT is in how the range inferences for variables are made. Variable values are no longer restricted to 0 or 1 as in Boolean SAT, and any reasoning system rooted in the logic over which the solver operates (e.g., the real numbers) may be used. Due to its simplicity which enables very fast inferences (which is crucial for fast solver operations), interval arithmetic (IA, as detailed in Section 2.2.2) has been adopted for some state-of-the-art solvers for variable range inference.

Figure 3.1 shows how range inference based on IA proceeds for the addition operator. Note that unlike basic IA which only supports forward propagation (inferring c' from a, b),

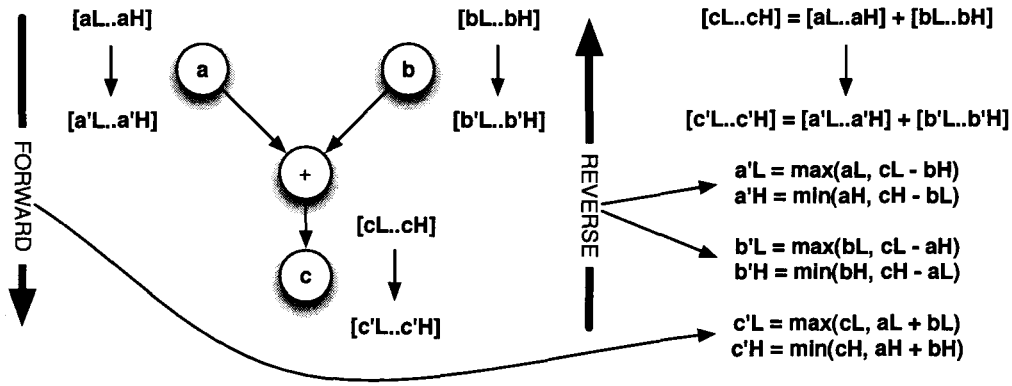


Figure 3.1: Inferring intervals of variables for the addition operator.

reverse propagation (inferring a' from b,c and b' from a,c) is also supported here. This allows information from deeper in the datapath (e.g., if there were a constraint on c) to provide information on values earlier in the datapath (e.g. a and b).

When dealing with an entire instance (dataflows + constraints, as opposed to a single operation as in Figure 3.1), the inference is performed iteratively. Figure 3.2 illustrates the concept for a simple instance:

$$-100 \leq x \leq 100 \quad -100 \leq y \leq 100$$
$$z = \frac{xy}{x^2+y^2} \qquad z > 0.6.$$

In the figure, each node labeled (a) to (m) is either a variable, a constant (interval) or an operation - each corresponding to a part of the instance above. For example, node (f) near the middle of the figure corresponds to the addition in the denominator of the expression for z above, and node (m) on the right hand side of the figure corresponds to the constant 0.6 in the constraint $z > 0.6$ above.

Associated with each node is a sequence of intervals numbered (1) to (7). These indicate how the interval associated with that node evolves through each iteration of inference (1-7 for this example). Nodes for constants and input variables have their ranges known before entering the first iteration (they are defined as part of the instance), and all the rest are

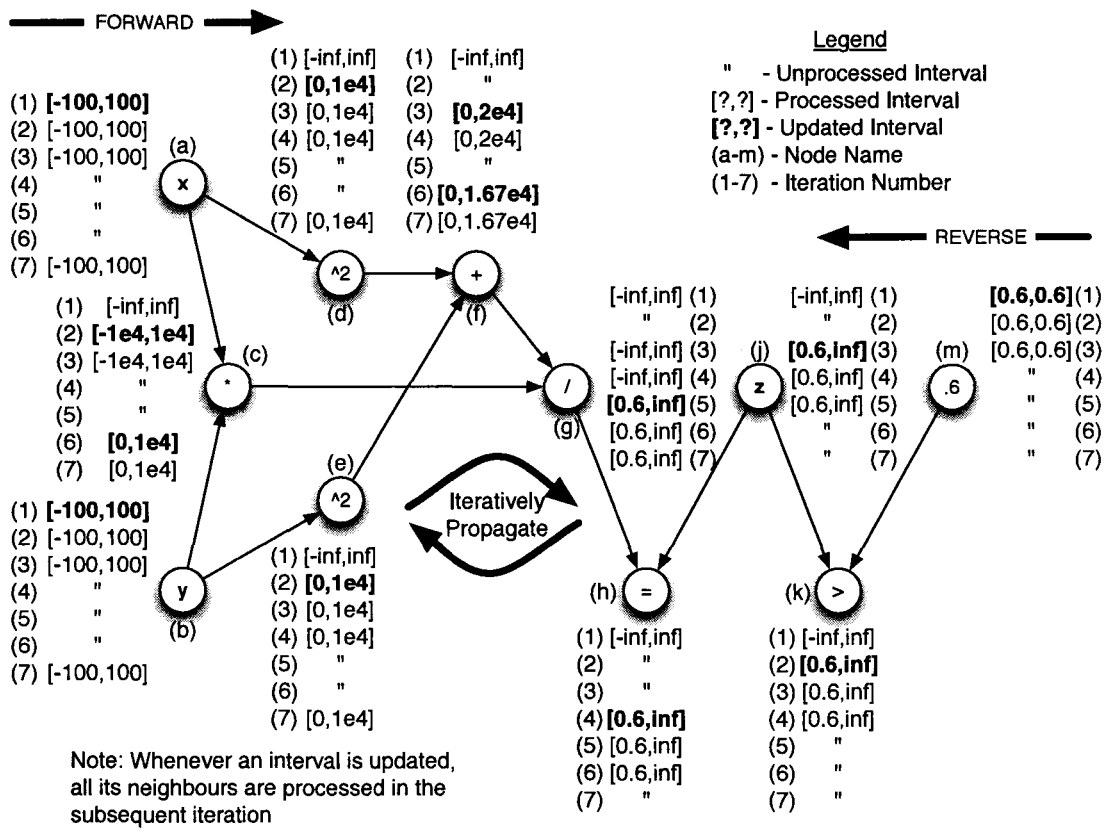


Figure 3.2: Inferring intervals in a full dataflow.

assigned $[-\infty, \infty]$ ([-inf,inf] in the figure). In each iteration, not all nodes are processed as any node whose neighbours have not changed will not change. These unprocessed nodes are indicated in the figure by a quotation mark (") in place of the interval to show no actual processing. For nodes that are actually processed (at least one neighbour changed in the last iteration), processing may lead to the same range, or the range may be reduced, which is indicated in the figure by bold-face intervals.

The inference of intervals for an entire instance is then the culmination of the iterations (of which there are seven for this example). At the onset, all intervals are set to $[-\infty, \infty]$, and then known intervals from input variables and constants are applied (x, y, 0.6) and those nodes are "marked" to indicate their ranges have contracted. This is shown in the figure in the set of intervals with label (1) over all the nodes. In the next iteration, we can see that nodes $(f), (g), (h)$ and (j) are not updated (as indicated by the ") none of their neighbours are "marked" (contain a bold interval) so there is nothing to drive a change for them.

All the other nodes - i.e., $(a), (b), (c), (d), (e)$ and (k) - are processed due to having a marked neighbour. The node processing is just interval inference for a single operation, the process illustrated in Figure 3.1 for addition. To reiterate that process: each operand's interval is the intersection of its current interval with its inferred interval assuming the intervals for the other operands. If any of the intervals contracts (as it does for $(c), (d), (e)$ and (k) in iteration 2), the node is marked so that the interval changes can be propagated to the neighbours in the next iteration.

The above procedure of iterative forward and backward interval inference on an entire instance will eventually terminate (the intervals either contract or remain the same, and a machine-epsilon can be used to avoid infinitesimal advances) yielding one of two outcomes. If an empty interval can be established at any node, we can immediately infer emptiness of every interval and thus the instance is *unsatisfiable*. The alternative result is a set of intervals (one for each node) which bound the space (most likely loosely) in which all the constraints are satisfied, providing a (most likely overestimated) interval for each intermediate. The latter is the case for our example, at the end of the seventh iteration none of the nodes are marked, so the remaining intervals bound the set of (x, y, z) triplets which satisfy the constraints. All iterations from first assignment to stabilization (1 to 7 in our example) forms one inference referred to in the context of a solver as the *propagation step*.

Decision step

We have now established above the necessary machinery to carry over what is simple binary logic reasoning in the Boolean domain into the domain of extended logics (where SMT resides) in forming the *propagation step*. To see the need for the other kind of step (*decision* step), consider that in the specific case of Figure 3.2, we have termination with a set of intervals at iteration 7. At the same time however, for this example we can derive analytically the range of z through the substitutions $x = r\cos(\theta)$ and $y = r\sin(\theta)$ yielding:

$$z = \frac{xy}{x^2+y^2} = \frac{r^2\cos(\theta)\sin(\theta)}{r^2\cos^2(\theta)+r^2\sin^2(\theta)} \quad z = \cos(\theta)\sin(\theta) \quad z = \frac{1}{2}\sin(2\theta)$$

which is limited in range to $-0.5 \leq z \leq 0.5$ and thus based on this fact, the instance is clearly unsatisfiable ($[0.6,\infty] \cap [-0.5..0.5] = \emptyset$). The disparity between this true interval of $[-0.5..0.5]$ and the interval reported by the propagation arises from data dependencies which the IA based inference cannot retain beyond one operation. Through the decision step, the solver more closely examines smaller partitions of the assignment space to search for inconsistencies which are masked by the lack of dependence retention exhibited by IA.

Because the central contribution to automated data representation of this thesis is built upon SMT, in order to concretize the concepts of SMT instances and solver operation an example is provided in Figure 3.3. Building on the same instance used earlier of $z = \frac{xy}{x^2+y^2}$, Figure 3.2 becomes one *Propagation* step from Figure 3.3, and the large boxes indicate computed intervals for (an important subset of) nodes within the calculation instance. These ranges are computed at each node of the decision graph at the centre of the figure illustrating the sequence of *Decision* steps for a search where the branching and backtracking protocols are taken for granted for the sake of the example. In practice, both branching and backtracking decisions are complex, with a number of options having varying impacts on search efficiency. Due to their complexity, they have been abstracted so as not to distract from the main purpose of the example which is to show how decision and propagation steps are employed to decide an instance.

Each node in Figure 3.3 represents a subspace of assignments over all intermediate nodes of Figure 3.2, and a decision splits the interval of one of those intermediates into a number of sub-intervals - one per child node - the union of which is the original interval. An

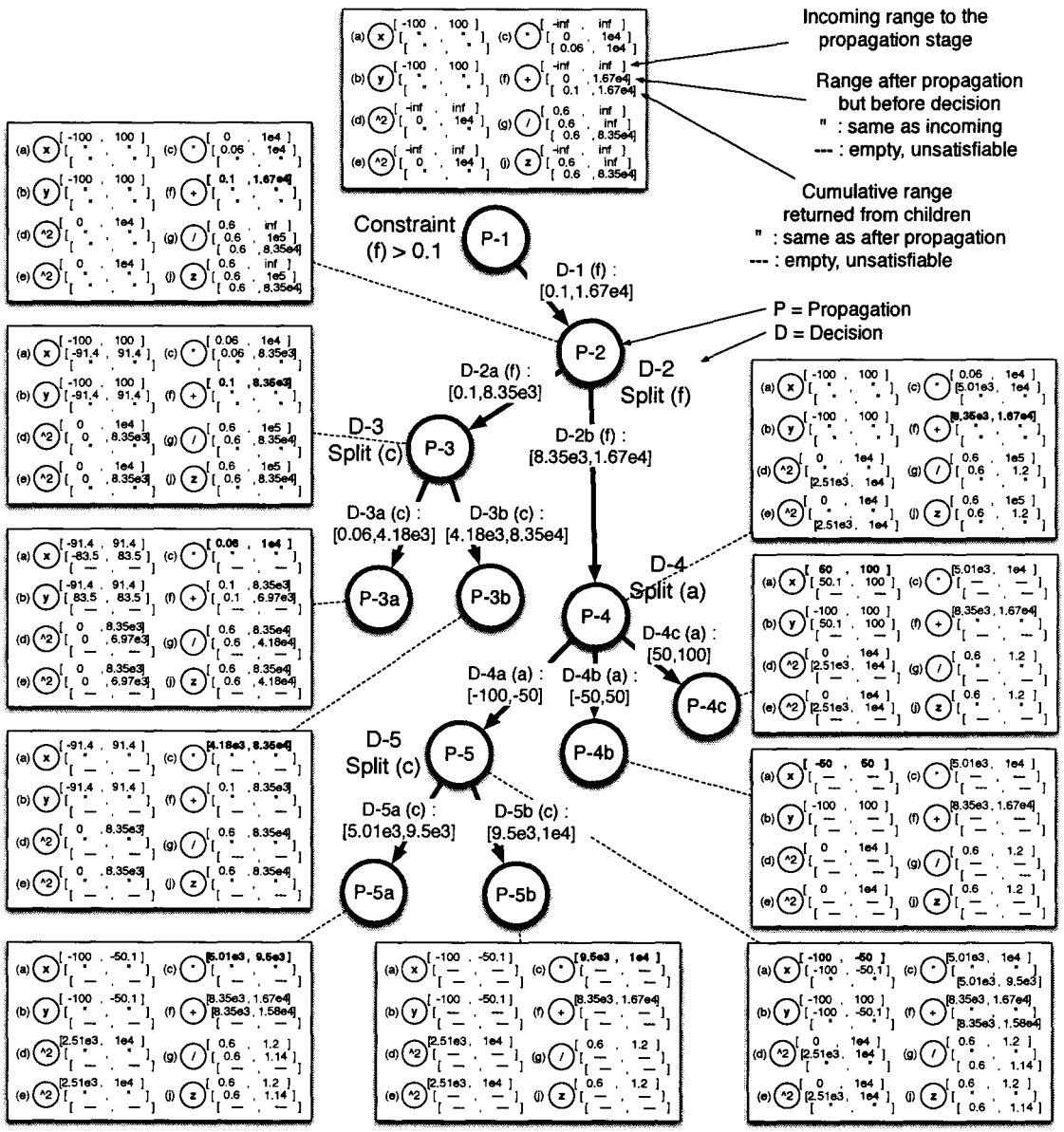


Figure 3.3: SMT solver example.

exception to this occurs when additional constraints are provided via the solver, in which case the union of sub-intervals is the interval of interest. Taking for granted decisions in Figure 3.3 for the sake of the example, decision 1 illustrates the above mentioned constraint type of decision. In this case, we only care about solutions with (f) ≥ 0.1 At each node, the edge along which the node is entered contains a search sub-interval for a particular intermediate of the instance (the one on which branching has occurred), and upon entering the node the following is done:

1. Assign intervals of the parent to each intermediate, except the one indicated by the incoming edge, use the search sub-interval there. These intervals are captured for each search node in the first of the triplet associated with each intermediate ((a)-(j)).

2. Propagate the search sub-interval through the calculation using the procedure outlined in Figure 3.2. The resulting intervals are captured for each search node in the second of the triplet associated with each intermediate ((a)-(j)).

3. If not branching further (this is a leaf), return interval resulting from the propagation for each intermediate. Further branching is guaranteed not to occur if the propagation of the previous step returns empty intervals (i.e. unsatisfiable).

4. If branching further, decide upon a new intermediate and how to split it, form a child for each sub-interval and recurse to each child.

5. Once all children have been visited, return for each intermediate the union of the intervals returned for that intermediate over the traversals to all the children.

Traversal of the entire tree (returning from the root node) when every single leaf node has empty intervals will yield the union of empty intervals and therefore unsatisfiability of the instance. Otherwise, the instance is declared satisfiable and delivers (potentially reduced) intervals for each intermediate within which a satisfying assignment will lie.

Each of these steps above is reflected in the figure. Steps 1 and 2 can be seen in each box, where the intervals in the first of each triplet are copied from the second triplet for the same intermediate of the parent, except the search sub-interval (indicated in bold for each box) informed by the incoming edge, and the second of each triplet is obtained through

propagation. Step 3 is shown in the boxes for nodes labeled as *Propagation 3a, 3b, 5a, 5b, 4b and 4c* - with *4b* and *5b* unsatisfiable. While no explicit branching mechanism is described, Step 4 is embodied in all the edges in the search graph - in particular note that *Decision 4* splits (a) into 3 parts covering (a) instead of two parts like all other splits. Finally, Step 5 can be seen in all boxes (except the leaf nodes) as well, but is most clearly seen in the lower bounds for third triplets of (d) and (e) of the *Propagation 4* node. Of the three children of this node, two resulted in lower limits of 2.51e3 for (d) and (e), and one resulted in unsatisfiable, so 2.51e3 is passed up the tree as the new lower limit.

Having now described the fundamentals necessary to understand the operation of SMT solvers, two points stand to be made before turning to their role in automated data representation. First, as mentioned above, there are many schemes which may be employed for deciding whether to branch or backtrack, and when branching where to split and how. Research into this topic belongs to the field of SMT solver design and is beyond the scope of this thesis. In fact, in order to focus on the automated representation problem, as proof of concept we invoke an off-the-shelf solver [35, 96] for the work in this chapter, as well as Chapter 2. In Chapter 5, both this off-the-shelf solver as well as a custom, in-house developed solver are used.

The second important point is that when a set of constraints defining a dataflow is augmented with a constraint such as $z > 0.6$ (nodes (j),(k),(m) in Figures 3.2 and 3.3), the result of the decision problem indicates reliability of the constant as a bound (upper bound of 0.6 on z in this case). Thus SMT can be used as a bounds checking engine, which is exactly its role in the automated data representation problem, the topic to which we turn in the next section.

3.3 Range refinement using SMT

Building on the framework of the previous section, an SMT engine can be used to prove or disprove validity of a bound on a given expression by checking for satisfiability. This section details how such bounds proving is accomplished and how it can be used as the core of a procedure addressing the range determination problem.

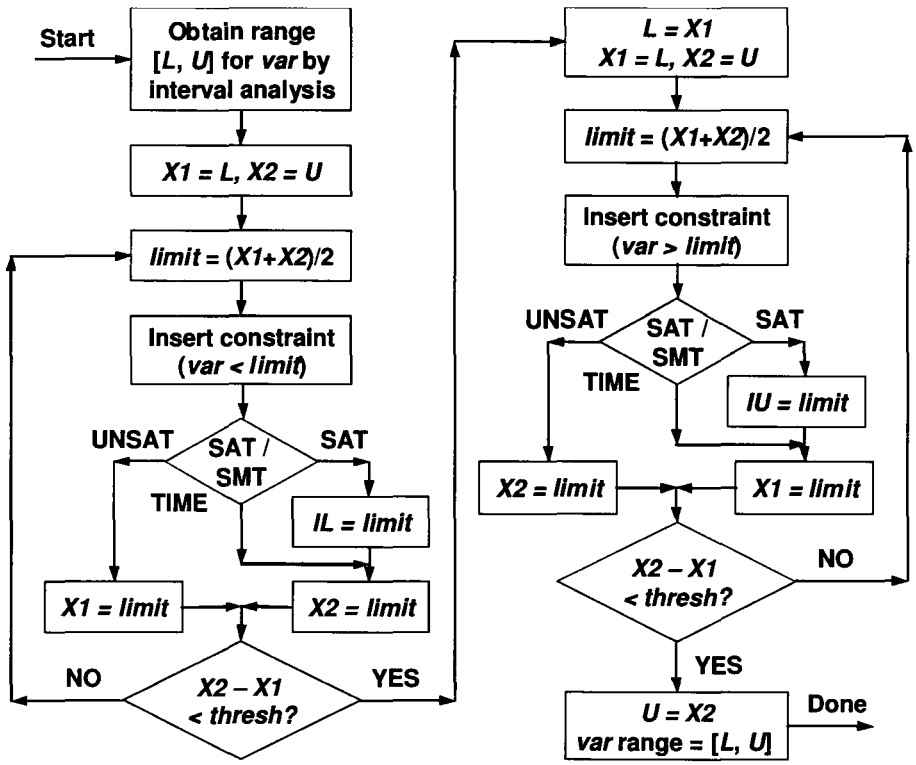


Figure 3.4: SAT/SMT range refinement of *var*.

Figure 3.4 illustrates the binary search method employed for range analysis on an intermediate variable: *var*. Note that each SMT instance evaluated contains the inserted constraint ($var < limit$ or $var > limit$), like nodes (j),(k) and (m) of Figures 3.2 and 3.3. The loop on the left of the figure (between "$limit = (X1 + X2)/2$" and "$X2 - X1 < thresh?$") narrows in on the lower bound L, maintaining $X1$ less than or equal to the true (and as yet unknown) lower bound. Each time satisfiability is proven (SAT path), $X2$ is updated while $X1$ is updated in cases of unsatisfiability (UNSAT path), until the gap between $X1$ and $X2$ is less than a user specified threshold (the role of the TIME path and the values IL and IU will be discussed below in Section 3.3.2). Subsequently, the loop on the right performs the same search on the upper bound U, maintaining $X2$ greater than or equal to the true upper

bound. Since the SMT solver works on the full calculation, all interdependencies among variables are taken into account (via the SMT constraints) so the new bounds successively remove over-estimation in the original bounds resulting from the use of interval arithmetic.

Algorithm 3.1 RangeRefine

```
1: BaseSMTFormulation = CreateSMTInstance(CalculationSteps)
2: for (each var in InputVarList) do
3:    Copy range of var from InputVarRanges into BaseSMTFormulation
4: end for
5: for (each var in IntermediateVarList) do
6:    Refine range of var (Figure 3.4)
7:    update IntermediateVarRanges for var
8:    update BaseSMTFormulation with new range of var
9: end for
10: return IntermediateVarRanges
```

The overall range refinement process, Algorithm 3.1 operates on the dataflow named *CalculationSteps* (with input variables in *InputVarList* having ranges *InputVarRanges*) to produce refined ranges *IntermediateVarRanges* of the the intermediate variables in *IntermediateVarList*. It is worth noting that while currently *IntermediateVarList* is ordered according to first use in the dataflow, the ordering can impact SMT solver runtime - a problem left for future exploration. The process uses the steps of the calculation and the ranges of the input variables as constraints to set up the base SMT formulation (lines 1 and 2), like Figure 3.2 without nodes (j),(k) and (m). It is this base formulation into which *Insert constraint:* from Figure 3.4 inserts, and the form of the constraint is like nodes (j), (k) and (m) from Figure 3.2 (with $>$ for upper and $<$ for lower bounds). It iterates through the intermediate variables (line 5) applying Figure 3.4 (line 6) to obtain a refined range for that variable. Once all variables have been processed the algorithm returns ranges for the intermediate variables (line 10).

3.3.1 Dealing with division

As discussed in Section 2.2, non-affine functions with high curvature cause problems for AA, and while these are rare in the context of DSP (as confirmed by [33]) they occur

frequently in scientific computing and is particularly problematic due to range inversion (quotient increases as divisor decreases). While AA tends to give reasonable (but still overestimated) ranges for compounded multiplication since product terms and the corresponding affine expression grow in the same direction, this is not the case for division. Furthermore, both IA and AA are unequipped to deal with divisors having a range that includes zero.

Use of SMT mitigates these problems. Even for SMT solvers which do not support divisions directly, divisions can be re-written as multiplication constraints, another advantage of the constraint centric formulation. Furthermore, an additional constraint can be included which restricts the divisor from coming near zero (like Decision 1 from Figure 3.3). Since singularities such as division by zero result from the underlying math (i.e., are not a result of the implementation) their effects do not belong to range/precision analysis and SMT provides convenient circumvention during analysis. This technique is equally applicable to other kinds of singularities such as those arising from logarithms or trigonometric functions. In such cases, the restriction constraints define the operating conditions over which robustness is guaranteed.

3.3.2 Consideration of run-time

While leveraging the mathematical structure of the calculation enables SMT to provide much better run-times than using Boolean SAT (where the entire data-path and numerical constraints are modelled by clauses obfuscating the mathematical structure), run-time may still become unacceptable as complexity of the calculation under analysis grows. To address this, a timeout is used to cancel the inquiry if it does not return before timeout expiry. This is the meaning of the "TIME" path in Figure 3.4. Once timeout occurs, the bounds obtained for that variable will not necessarily be tight however, because the timeout path feeds into the satisfiability path, robustness will still be maintained - i.e., to assume satisfiable gives pessimistic bounds. The purpose of the IL and IU values are to keep track of the proven inside bounds - $[L,U]$ must contain $[IL,IU]$.

It should be noted that for IA and AA, once the range of an intermediate variable is overestimated all subsequent variables which depend on it will also have overestimated

ranges due to the use of forward propagation only. A clear advantage of using SMT is that this is no longer the case, the SMT solver views the entire calculation as a single instance. As a result, information inferred about the range of a particular variable can be used to refine the range of a variable that precedes it in the data-path, an effect which can be observed in Figure 3.3. Finally, by using a variable timeout, the tradeoff between run-time and the tightness of the variables' bounds can be user controlled, as informed by the $[IL, IU]$ range. If this range is much smaller than $[L, U]$, it is worthwhile to search in more depth for that variable (by increasing the timeout) and furthermore, the work done to determine $[L, U]$ using a particular timeout can be reused for a larger timeout by starting (for example) with a search range for L of $[L, IL]$ instead of $[L, U]$ (similarly for U).

3.4 Case studies and results

Given that the target application domain for this method is hardware acceleration for scientific computing, we seek to address specifically the problem of division which is known both to be common in scientific calculations and to cause problems for existing methods. For this reason, in this section we detail a few case studies involving division, as well as one non-affine example from DSP, and we compare the results of basic AA and the proposed SMT approach applied to these case studies.

The experiments in this section were carried out on 1.5 GHz Pentium 4 with 512 MB of RAM running Gentoo Linux, using the freely available *HySAT* implementation [35, 96] as the core SMT solver. Ranges were obtained using (unless otherwise specified) a timeout of 2 seconds, resulting in run-times for all cases on the order of 100 seconds. It is worth noting that the same time that although the affine experiments are in some cases faster, speed does not matter when support for the calculation is not provided by AA. In all cases the number of bits needed for range $[L, U]$ has been taken as $\lceil \log_2(U - L) \rceil$ to facilitate evaluation of the actual span of the numbers, taking into account also how they are centred. In almost all cases however, the result is identical to taking $\lceil \log_2(\max(|L|, |U|)) \rceil + a$, where a is 0 if L and U have the same sign, and 1 otherwise.

Table 3.2: Affine vs. SAT-Modulo for energy spectral density.

	Affine		SAT-Modulo	
Output	Range	Bits	Range	Bits
0	[-1835008 , 2097152]	22	[-1 , 2097153]	22
1	[-2373666 , 2635814]	23	[-1 , 1984106]	21
2	[-2269321 , 2531463]	23	[-1 , 1790022]	21
3	[-2373666 , 2635814]	23	[-1 , 2052757]	21
4	[-1835008 , 2097152]	22	[-1 , 2097153]	22
5	[-2373666 , 2635814]	23	[-1 , 1957096]	21
6	[-2269321 , 2531463]	23	[-1 , 1790023]	21
7	[-2373666 , 2635814]	23	[-1 , 2029555]	21

3.4.1 Energy spectral density

One application involving non-linearity which appears frequently in DSP is the calculation of energy spectral density (ESD) for a signal [11]. ESD can be obtained as:

$$\Phi(\omega) = \frac{1}{2\pi} F(\omega) F^*(\omega)$$

where $F(\omega)$ indicates the Fourier Transform of the signal of interest, or the Fast Fourier Transform (FFT) for discrete signals. Since the FFT itself is affine, AA provides exact bounds on all intermediate variables however, the ESD involves magnitude of a complex number (non-affine) leading to range overestimation.

In this experiment an 8-point FFT has been used, with each of the 8 inputs a complex number in $[-128, 128] + [-128, 128]i$. While both AA and SMT provide exact bounds on all intermediate variables in the FFT calculation, AA overestimates the magnitude (non-affine). Table 3.2 shows the ranges obtained from both AA and SMT when applied to each of the 8 outputs of the ESD calculation (to obtain these ranges a solver timeout of 5 seconds was used). Note that AA ranges are centered close to zero while SMT ranges start near zero which is correct as only positive values would be expected.

Clearly for this calculation, AA provides good estimates of the ranges and thus the bit-widths, since only one level of non-affine calculations occurs. In light of the inclusion of

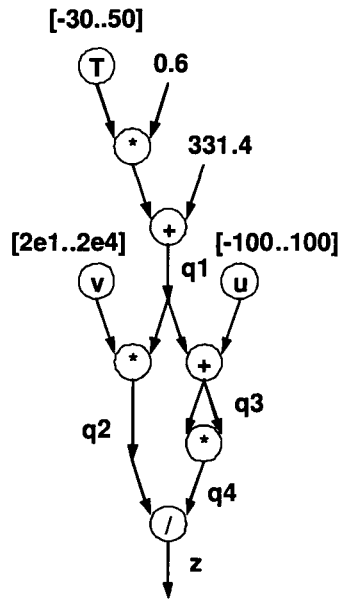


Figure 3.5: Data dependencies for Doppler effect case study.

a large range of numbers below zero, subsequent calculations relying on the ESD can be expected to already begin experiencing range inflation, especially as more variable inter-dependencies arise. Consider also that division does not occur, as it does in the following examples.

3.4.2 Doppler effect

The Doppler effect is the apparent change in frequency observed when a sound source is in motion with respect to an observer [111]. For a given emitted frequency v and a relative speed of u between the source and observer, the perceived frequency will be $v' = cv \div (c+u)$ where c is the speed of sound in the medium. If the medium is air and we wish to know how the rate of frequency change with respect to the relative speed u depends on

Table 3.3: Affine vs. SAT-Modulo for Doppler.

	Affine		SAT-Modulo	
Output	Range	Bits	Range	Bits
q_1	[313 , 362]	6	[313 , 362]	6
q_2	[-473252 , 7228000]	23	[6267 , 7228000]	23
q_3	[213 , 462]	8	[213 , 462]	8
q_4	[25363 , 212890]	18	[45539 , 212890]	18
z	[-80,229]	9	[0 , 138]	8

temperature, we have:

$$z = \frac{dv'}{du} = \frac{-(331.4+0.6T)v}{(331.4+0.6T+u)^2}$$

using the approximation for the speed of sound in air $c \approx 331.4 + 0.6T$, T in degrees Celsius.

The overall calculation for this case study was broken into intermediate calculations as follows:

$$q_1 = 331.4+0.6T \quad q_2 = q_1 v$$
$$q_3 = q_1 + u \quad q_4 = q_3^2 \quad z = q_2/q_4$$

and the parameters that were used were:

- temperature: $-30°C \leq T \leq 50°C$,
- audible frequencies: $20Hz \leq v \leq 20000Hz$,
- relative speed: $-100m/s \leq u \leq 100m/s$.

This breakdown results in a fairly deep data-path as illustrated in Figure 3.5. Since AA will tend to overestimate ranges of non-affine operations at each point, deep data-paths can result in accumulated overestimations eventually becoming significant.

Observing Table 3.3, note that AA and SMT provide comparable ranges for all variables except for z where the division occurs and where the bit-width is overestimated by 1 bit.

Despite the fact that this calculation has fewer levels of intermediate variables than the aforementioned ESD, and the fact that all upper bounds from AA were exact, the resultant range was still overestimated, a prime example of the result of range inversion mentioned in Section 3.3.1.

3.4.3 Analytic center

The analytic center of a set of inequality constraints is the point which maximizes the distance (as defined by some distance metric) from all the constraint boundaries, and it has applications in convex optimization. In particular (following an example from [10]) solving for the analytic center when using a distance metric based on a logarithmic penalty function will give rise to calculations such as:

$$z[i] = d[i]\mathbf{a}[i] \cdot (\mathbf{C} - \mathbf{P}) \qquad d[i] = \frac{1}{b[i] - \mathbf{a}[i] \cdot \mathbf{C}}$$

where $\mathbf{C}$ is the analytic center of the inequality constraints defined by $\mathbf{a}[i], b[i]$, and $z[i]$ reflects a penalty with respect to $\mathbf{a}[i], b[i]$ if $\mathbf{C}$ were moved to $\mathbf{P}$.

Taking the specific case of 3 inequality constraints in $\mathbb{R}^2$, we can expand these vector equations into scalar ones:

$$q_1[i] = b[i] - (a_x[i]C_x + a_y[i]C_y) \qquad q_2[i] = \frac{1}{q_1[i]}$$

$$z[i] = q_2[i]\left(a_x[i]\left(C_x - P_x\right) + a_y[i]\left(C_y - P_y\right)\right)$$

for $i \in \{1,2,3\}$. Under ranges of $C_x, C_y, P_x, P_y \in [-100, 100]$, and all $a_x[i], a_y[i], b[i] \in [-10, 10]$, bit-widths obtained by using AA and SMT are compared in Table 3.4.

Notice in the table that AA is unable to determine a range for $q_2[i]$, and as a result any intermediate variables that depend on $q_2[i]$. The reason for this is in the division of which $q_2[i]$ is the result, the range of the denominator contains 0 resulting in an infinite range for the result. While the same issue in principle can occur for SMT, SMT provides a convenient mechanism for ignoring this scenario while in the case of AA, adding such constraints is inconvenient (when possible) and destroys the affine correlations, leading to range explosion as happens in IA.

Table 3.4: Affine vs. SAT-Modulo for analytic center.

	Affine		SAT-Modulo	
Output	Range	Bits	Range	Bits
$q_1[i]$	[-2010 , 2010]	12	[-2010 , 2010]	12
$q_2[i]$	∞	–	[-101 , 101]	8
$z[i]$	∞	–	[-300557, 301544]	20

Disregarding this region around the singularity is permitted since the singularity occurs also in double and even infinite precision, and thus is not relevant in determining the precision required for representing the "well behaved" parts of the calculation. It is up to the algorithm implementer to set the limits on how close (based on the application precision requirements) to the singularity should be considered as part of the normal calculation; in this case study we have used the constraint $q_1^2[i] \geq 0.0001$.

Note finally for this case study that if we substitute the range obtained for $q_2[i]$ by SMT into the affine formulation as a free variable, although we get a larger range of $[-400400, 400400]$ for $z[i]$, the number of bits required is the same. At the same time, propagation of this larger range can lead to range overestimation deeper in the calculation as it does in the next case study.

3.4.4 Euclidian projection

Another case study having applications in convex optimization/analysis is Euclidian projection of a point or set of points onto a hyperplane, for instance to reduce the dimension of a problem [10]. Given a hyperplane defined by $\mathbf{a} \cdot \mathbf{x} + b = 0$, the projection of a point $\mathbf{x0}$ is given by:

$$P(\mathbf{x0}) = \mathbf{x0} + \frac{b - \mathbf{a} \cdot \mathbf{x0}}{\mathbf{a} \cdot \mathbf{a}} \mathbf{a}$$

Table 3.5: Affine vs. SAT-Modulo for Euclidian projection.

	Affine		SAT-Modulo	
Output	Range	Bits	Range	Bits
q_1	[-3001 , 3001]	13	[-3001 , 3001]	13
q_2	[-3011 , 3011]	13	[-3011 , 3011]	13
q_3	[0 , 300]	9	[0 , 301]	9
q_4	∞	–	[-3033 , 3019]	13
z	∞	–	[-465 , 489]	10

If we choose the values for this case study to belong to $\mathbb{R}^3$, the vector equations can be expanded into scalar equations in a similar way as in Section 3.4.3:

$$\begin{aligned} q_1 &= a_i x_i + a_j x_j + a_k x_k \\ q_2 &= a_i^2 + a_j^2 + a_k^2 \\ q_3 &= b - q_1 \\ q_4 &= \frac{q_3}{q_2} \\ z_i &= x_i + q_4 a_i \\ z_j &= x_j + q_4 a_j \\ z_k &= x_k + q_4 a_k \end{aligned}$$

using $\mathbf{x}$ in place of $\mathbf{x0}$, with ranges $x_i, x_j, x_k \in [-100, 100]$ and $a_i, a_j, a_k, b \in [-10, 10]$.

Note how in Table 3.5, AA starts off strong giving tight ranges for q_1,q_2 and q_3. However, as in Section 3.4.3, the inclusion of 0 in the range for the denominator of a division causes an indeterminate range for q_4 and subsequent variables from AA. By applying the same constraint as in that case, SMT is again able to provide meaningful ranges. What differs however from the case study of Section 3.4.3 is that when the range for q_4 as obtained through SMT is substituted into the affine model as an input, AA returns the range of $z[i]$ as $[-30424, 30424]$ requiring 16 bits for representation instead of 10. This confirms the assertion that the main tool AA employs to determine tighter ranges is the propagation of

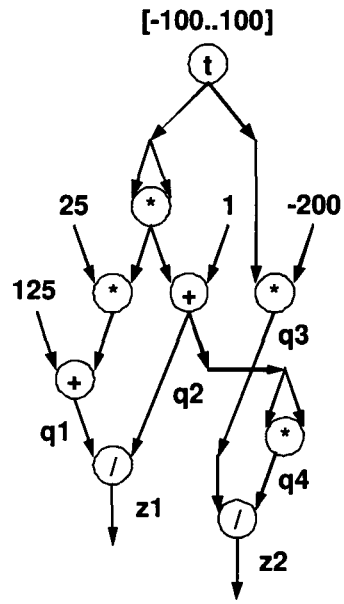


Figure 3.6: Data dependencies for rational function case study.

first order variable dependencies, and when those dependencies do not survive the division, ranges of subsequent variables are naturally overestimated.

3.4.5 A rational function

This case study employs a rational function such as those which arise when fitting curves to experimental data [118]. Consider the following function and its derivative:

$$z_1 = \frac{25t^2+125}{t^2+1} \qquad z_2 = \frac{dz_1}{dt} = \frac{-200t}{(t^2+1)^2}$$

over the range $-100 \leq t \leq 100$. It is worth noting that in addition to being common in scientific computing, such calculations may also arise in an embedded system, e.g., as a part of the model used for prediction/control.

Table 3.6: Affine vs. SAT-Modulo for a rational function.

	Affine		SAT-Modulo	
Output	Range	Bits	Range	Bits
q_1	[125 , 250125]	18	[124 , 250126]	18
q_2	[1 , 10001]	14	[0 , 10002]	14
q_3	[-20000 , 20000]	16	[-20001 , 20001]	16
q_4	[-24999999 , 100020001]	27	[0 , 100020008]	27
z_1	[-250 , 369]	10	[24 , 126]	7
z_2	∞	–	[-67 , 67]	8

This case study utilizes only one free variable, $-100 \leq t \leq 100$ leading to strong correlations between all intermediates:

$$\begin{aligned} q_1 &= 25t^2 + 125 \\ q_2 &= t^2 + 1 \\ z_1 &= q_1/q_2 \\ q_3 &= -200t \\ q_4 &= q_2^2 \\ z_2 &= q_3/q_4 \end{aligned}$$

illustrated in the multiple fanouts / reconvergences in the data-path of Figure 3.6. Table 3.6 shows the evolving ranges; as before z_1 suffers because of the division. Notice as well that AA cannot provide bounds for z_2 because the range of the divisor (q_4) includes zero, according to the affine approximation. The problem here however is not an underlying singularity which has to be excluded as in the previous 2 case studies, instead this inclusion of zero arises from accumulated successive overestimation. Even using SMT lower bound of $q_4 \geq 1$, the resultant range will be $z_2 \in [-20000, 20000]$ requiring 16 bits, 8 more than allocated using the SAT-Modulo approach. Finally, in many places when AA bounds are tight, SAT-Modulo differs from AA by 1. The real difference gets inflated to 1 by application of the floor(ceiling) function converting the obtained lower(upper) bound into an integer.

Table 3.7: Affine vs. SAT-Modulo for Newton's method.

	Affine		SAT-Modulo	
Output	Range	Bits	Range	Bits
z_1	[-1205360 , 1170360]	22	[-1205361 , 1135361]	22
z_2	[-5753 , 35769]	16	[1 , 35769]	16
z_3	∞	–	[-39 , 38]	7
z	∞	–	[-69 , 72]	8

3.4.6 Newton's method

The final case deals with root-finding using Newton's method [12] applied to a polynomial. Given a polynomial

$$f(x) = c_3x^3 + c_2x^2 + c_1x + c_0$$

roots can be obtained by using Newton's method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

If we consider a single iteration, this results in:

$$\begin{aligned} z_1 &= c_3x^3 + c_2x^2 + c_1x + c_0 \\ z_2 &= 3c_3x^2 + 2c_2x + c_1 \\ z_3 &= \frac{z_1}{z_2} \\ z &= x - z_3 \end{aligned}$$

For the sake of readability, the fully expanded intermediates have been omitted. The numerator and denominator polynomials (z_1 and z_2 respectively) above were expanded in intermediate steps using Horner's method [12] to reflect a potential hardware implementation. The range of x used was [-100,100] and the coefficient ranges were:

$$c_0 \in [-10,10] \quad c_1 \in [7.5,8.5] \quad c_2 \in [-3.75,-3.25] \quad c_3 \in [0.833,1.167]$$

Table 3.7 shows the results for the major intermediates (the ones which have been omitted had identical bit-widths), where as before the quotient z_3 cannot be calculated due to the

inclusion of zero within the range of the divisor z_2. Just as in the case of Section 3.4.5, no actual singularity exists, and thus the SMT method does not require any additional constraint. If as before we use the bound from SMT (1.83 which has been floored to 1 in the table), we end up with $z_3 \in [-658668, 620416]$ and $z \in [-658768, 620516]$ requiring 21 integer bits each, at least 13 more than necessary per signal. This reinforces the fact that even when measures are taken with AA to avoid the singularity, the correlations are lost.

3.4.7 Key points of case studies

The previous sections have presented 6 case studies taken from a variety of topics within the large field of scientific computing, which have been presented in this way to emphasize a few key points about the range determination problem within bit-width allocation.

Summarized below are the main points highlighted by each case study:

- Section 3.4.1: Insertion of a single non-affine calculation into a data-path plants a seed of overestimation which other calculations will augment;
- Section 3.4.2: Division is particularly detrimental as an affine approximation resulting from range inversion whereby small numbers map to large ones and vice-versa;
- Section 3.4.3: When the true ranges of an intermediate contain zero, SMT provides a convenient mechanism for excluding singularity points from consideration during range determination;
- Section 3.4.4: In addition to merely excluding singularities, meaningful ranges taking the exclusion into account must be determined otherwise range overestimation is inevitable;
- Section 3.4.5: Tight non-affine interdependencies caused by diverging/reconverging paths in the overall data-path will eventually outpace first-order approximations;
- Section 3.4.6: Serious range overestimation can lead to inclusion of 0 in the range for a variable where it does not actually belong, which will result in complete range indeterminacy if such variables become the denominator of a division.

In light of the results presented in the previous sections and the high occurrence of these scenarios in general scientific computing, computational range refinement based on SMT provides a reliable, practical method as the basis for automated bit-width allocation, especially for scientific computing applications. Now, in the next section, we briefly discuss the tradeoff between SMT timeout and accuracy of bounds.

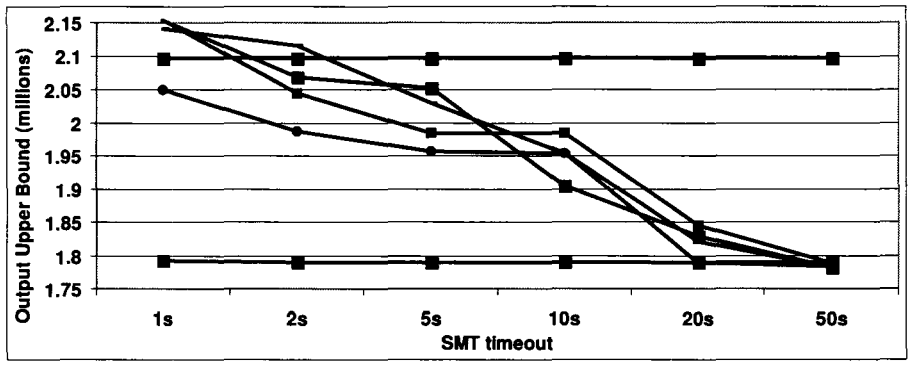
3.4.8 Run-time/accuracy tradeoff

Figure 3.7 shows how the range and number of bits for the 8 ESD outputs from the case study of Section 3.4.1 vary as timeout increases. In Figure 3.7(a), the ranges for the outputs are shown in no particular order, although it can be clearly inferred from the results of Table 3.2 that outputs 0 and 4 must be in the top line of the figure (since they both require 22 bits). At the same time, Figure 3.7(b) shows the bits required for each output, again in no particular order. While it may be difficult to discern between the graphs, the following is important to note: between 1 and 2 second timeout at least one bit-width is reduced; between 2 and 5 second timeout, at least one additional bit-width is reduced; after 5 second timeout, no more bit-widths are reduced.

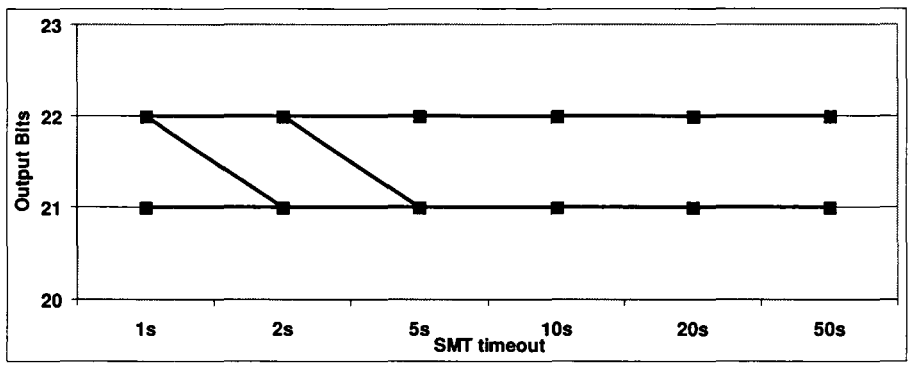
Note first that the bounds are always hard (robust), and thus decrease as the SMT solver is allowed to run for longer. The SMT assumes satisfiability when terminated, and for short timeouts, extra bit-width ends up being allocated. Notice also however that as the timeout is increased the range comes down very slowly (Figure 3.7(a)), while the number of bits is met with much less effort (Figure 3.7(b)). This is natural considering that the number of bits required scales logarithmically with the range which must be represented. As a result, the range must be halved in order to reduce a single bit. The main point however is that while determining the precise range of a variable can take a large amount of time, getting into the right power of 2 range will generally occur much earlier.

3.5 Summary

This chapter has highlighted the problem of bit-width allocation as important to the success of hardware acceleration, and illustrated the shortcomings of state-of-the-art methods in the



(a) ESD Output Ranges.



(b) ESD Output Bit-widths.

Figure 3.7: Effect of timeout on range/bit-width.

scientific computing application domain. By formulating range determination as a decision problem, computational methods can be applied - specifically SAT-Modulo theory - to the solution of this problem. Through six case studies the effectiveness of this approach has been demonstrated in dealing with the difficulties presented by scientific numerical algorithms to obtain 1) robust bit-widths necessary for scientific applications which simulation is unable to provide, 2) tighter bounds than those obtained through the use of affine arithmetic, and meaningful bit-widths even in cases where the affine result is indeterminate (division by 0). The next chapter builds on top of this foundation, providing a means of scaling the approach to larger instances.

Chapter 4

Scalability through block-vector formulations

In the previous chapter, computational methods based on SAT-Modulo Theory have been introduced and applied to the range determination aspect of the automated representation problem. While the use of computational methods was shown to provide robust and tight bounds for ranges, it was also observed that run-times could easily escalate as instance size grows as evidenced by the need to introduce a timeout path into the range search. The sharp rise in SMT solver runtime which accompanies an increased number of variables could render this method infeasible for many scientific calculations of practical relevance which may involve vectors and matrices of hundreds or even thousands of elements. To address this issue, a means of enabling scalability to large problem instances is described in this chapter based on block vector formulations [66].

4.1 Bit-width allocation in vector calculus

Before moving into the details of this algorithmic approach to bit-width allocation for operations in complex vector calculus, a discussion on using uniform bit-width within a vector is presented first. After this, representation of complex numbers is treated, followed by presentation of the vector magnitude and block vector magnitude models.

4.1.1 Uniform vector bit-width

In order to leverage the vector structure of the calculation under analysis and thereby reduce the complexity of the bit-width problem to the point of being tractable, the underlying principle of this work involves sacrificing the independence (in terms of range/bit-width) of the vector components as scalars. At the same time, hardware implementations of vector based calculations typically exhibit the following characteristics:

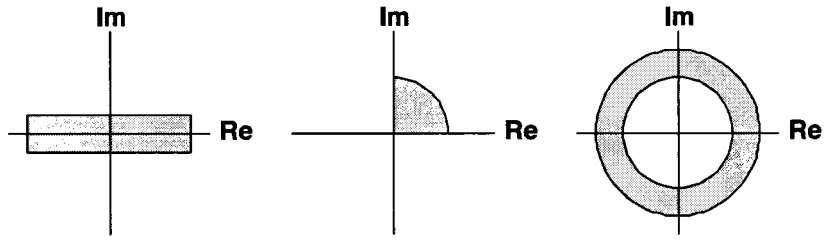
- Vectors are typically stored in memories already having the same number of data bits at each address;
- Datapath calculation units used in vector calculations must be allocated to accommodate the full extent of bit-widths which arise across the elements of the vectors to which they apply;

Based on these two key observations, the same number of bits already tend to be used implicitly for all elements within a vector simply as a side effect of common hardware design choices. This fact is exploited by the bit-width allocation problem to reduce its complexity, thus leading in some cases to tighter bit-widths (as will be confirmed in Section 4.2). As a result, the techniques laid out in this section impose uniform bit-widths within a vector, i.e., all the elements within a vector will use the same representation. However, each distinct vector will still have a uniquely determined bit-width.

4.1.2 Representation of complex numbers

Since the vector calculus we wish to support permits the vector elements to be complex numbers, it is necessary to discuss different representations for complex numbers which may be adopted, noting that all formats considered reduce in the end to a pair of binary numbers for each complex number.

Figure 4.1 shows different ways in which the possible values a complex number may take on can be restricted. In Figure 4.1(a) the imaginary part of the number is clearly more tightly bounded than the real part. Figures 4.1(b) and 4.1(c) show polar bounding of the angle and magnitude respectively. As will become clear throughout this chapter, the key aspect on which the effectiveness of these techniques hinges is the degree of interdependence



(a) Rectangular. (b) Angular. (c) Magnitude.

Figure 4.1: Example bounding constraints put on complex numbers.

between different elements within a vector, or so called *directional correlations*. This relationship stands also for the real and imaginary components of complex numbers and can be used to inform the decision of how to represent the complex numbers.

Note that in any situation where polar representation of complex numbers is used (e.g. Figures 4.1(b) and 4.1(c)), almost certainly we will want to have different bit-widths for the magnitude and angle components, since angle is bounded between 0 and 2π whereas magnitude may be arbitrarily large. Even in the case of rectangular representation, situations such as in Figure 4.1(a) would warrant the use of independent representations for the real and complex values since (in this specific case) the range of the imaginary part is much smaller and would therefore require fewer bits.

The remaining case of rectangular representations where the real and imaginary parts have similar ranges may be good candidates for using uniform bit-width across real and imaginary part by merging the vector of real components with the vector of imaginary components (i.e. represent the complex vector as a single real vector of double the size). Consider for example a complex matrix multiplication $\mathbf{y} = A\mathbf{x}$. If $\mathbf{x}$ has $\mathbf{x_r}$ and $\mathbf{x_i}$ as vectors of all the real and imaginary parts of $\mathbf{x}$ respectively (i.e. $\mathbf{x_r} = Re(\mathbf{x})$ and $\mathbf{x_i} = Im(\mathbf{x})$ and $\mathbf{x} = \mathbf{x_r} + i\mathbf{x_i}$) then we can form a new vector $\mathbf{x'}$ by interleaving elements from $\mathbf{x_r}$ and $\mathbf{x_i}$, which is the same as replacing every element x_k of $\mathbf{x}$ with the two elements $[Re(x_k), Im(x_k)]$. Applying the same process to $\mathbf{y}$ to form $\mathbf{y'}$, and replacing each element $A_{i,j}$ in A with the

2x2 matrix:

$$\begin{bmatrix} Re(A_{i,j}) & -Im(A_{i,j}) \\ Im(A_{i,j}) & Re(A_{i,j}) \end{bmatrix}$$

to create A' yields a purely real matrix multiplication $\mathbf{y}' = A'\mathbf{x}'$ on vectors twice the size which can replace the complex one. The form of the 2x2 matrix results from the fact that $Re(A\mathbf{x}) = Re(A)Re(\mathbf{x}) - Im(A)Im(\mathbf{x})$ and $Im(A\mathbf{x}) = Im(A)Re(\mathbf{x}) + Re(A)Im(\mathbf{x})$. It is worth noting that specific structure of the vector is not important to the method, the choice of whether to interleave real and imaginary parts of the vector can be made in accordance with whichever representation produces the most suitable hardware architecture. In other words, for uniform bit-width between real and imaginary parts, it is fair to consider the merged vector as interleaved real/imaginary or concatenated real/imaginary, so long as there is consistency in all places where the vector is used (e.g. the form of the matrix is different between the two).

The advantage of doing this replacement would be to reduce the complexity of the model since the complex matrix multiplication would remain a single statement involving two vectors and a matrix instead of two statements involving two matrices and four vectors, but at the cost of reduced directional information. As the next section shows, this also arises in applying the proposed model, and Section 4.1.4 shows how to recover some directional information, since operations such as pointwise complex multiplication can exhibit strong directional correlation.

4.1.3 Vector magnitudes

Whether complex or real, the approach to dealing with problems specified in terms of vectors centres around the fact that no element of a vector can have absolute value (magnitude) greater than the vector magnitude i.e., for a vector $\mathbf{x} \in \mathbb{C}^n$:

$$\mathbf{x} = (x_0, x_1, ..., x_{n-1})$$
$$||\mathbf{x}|| = \sqrt{\mathbf{x}^*\mathbf{x}}$$
$$|x_i| \leq ||\mathbf{x}|| \ , \ 0 \leq i \leq n-1$$

where * denotes taking the complex-conjugate transpose. Starting from this fact, we can create from the input calculation a vector-magnitude model which can be used to obtain

Table 4.1: Magnitude bounding operations.

Name	Vector Operation	Magnitude Model
Dot Product	$\mathbf{x}\cdot\mathbf{y}$	$\|\|\mathbf{x}\|\|\|\|\mathbf{y}\|\|\cos(\theta_{xy})$
Outer Product	$\mathbf{x}\mathbf{y}^T$	$\|\|\mathbf{x}\|\|\|\|\mathbf{y}\|\|$
Addition	$\mathbf{x}+\mathbf{y}$	$\sqrt{\|\|\mathbf{x}\|\|^2+\|\|\mathbf{y}\|\|^2+2\mathbf{x}\cdot\mathbf{y}}$
Subtraction	$\mathbf{x}-\mathbf{y}$	$\sqrt{\|\|\mathbf{x}\|\|^2+\|\|\mathbf{y}\|\|^2-2\mathbf{x}\cdot\mathbf{y}}$
Matrix Multiplication	$A\mathbf{x}$	$\sigma\|\|\mathbf{x}\|\|$ $\|\sigma_A\|^{min} \leq \sigma \leq \|\sigma_A\|^{max}$
Pointwise Product	$\mathbf{x}\circ\mathbf{y}$	$\varepsilon\|\|\mathbf{x}\|\|\|\|\mathbf{y}\|\|$ $0 \leq \varepsilon \leq 1$

bounds on the magnitude of each vector. These bounds can then be adopted as element bounds and from them the required uniform bit-width for that vector can be inferred.

Creating the vector-magnitude model involves replacing elementary vector operations with equivalent operations bounding vector magnitude, Table 4.1 contains the specific operations used in this chapter. When these substitutions are made, the number of variables for which bit-widths must be determined can be significantly reduced as well as the number of expressions, especially for large vectors.

The entries of Table 4.1 arise either as basic identities within, or derivations from, vector arithmetic. The first entry is simply one of the definitions of the standard inner product, and the outer product expression comes from the identity $(\sum_i |x_i|^2)(\sum_j |y_j|^2) = \sum_{ij}(|x_i||y_j|)^2$. The addition and subtraction entries are resultant from the parallelogram law [2]. The matrix multiplication entry is based on knowing the singular values σ_i of the matrix (further description below), and the pointwise product comes from: $\sum_i |x_i|^2|y_i|^2 \leq (\sum_i |x_i|^2)(\sum_i |y_i|^2)$.

When dealing with complex matrix multiplication, the values of σ_A^{min} and σ_A^{max} relate to the singular value decomposition (SVD) of $A = U\Sigma V^*$ and can be obtained by examining A^*A and AA^*. These matrices are guaranteed to be normal and hence to have a diagonalization producing eigenvalues. If the eigenvalues are arranged according to absolute value,

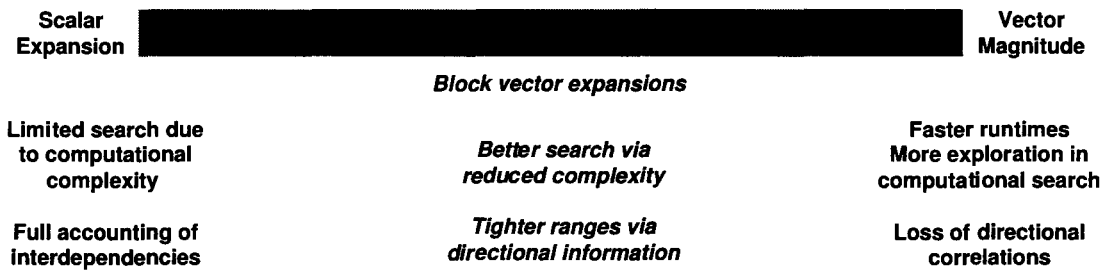


Figure 4.2: Goal of block vector representations.

$|\sigma_A^{min}|$ is the square root of the eigenvalue with the smallest value, and $|\sigma_A|^{max}$ is the square root of the eigenvalue with the largest absolute value. These values give limits on how the matrix will scale a vector during matrix multiplication.

While significantly reducing the complexity of the range determination problem, the drawback to using this method is that directional correlations between vectors are virtually unaccounted for. For example, vectors $\mathbf{x}$ and $A\mathbf{x}$ are treated as having independent directions, while in fact the true range of vector $\mathbf{x}+A\mathbf{x}$ may be restricted due to the interdependence. In light of this drawback, the next section proposes a means of restoring some directional information without reverting entirely to the scalar expansion based formulation. Incidentally, the method below can also recover directional information which may be lost during replacement of complex vectors with real ones as described in Section 4.1.2.

4.1.4 Directionality via block vectors

As discussed in the previous section, bounds on the magnitude of a vector can be used as bounds on the elements, with the advantage of significantly reduced computational complexity. In essence, the vector structure of the calculation (which would be obfuscated by expansion to scalars) is leveraged to speed up range exploration. These two methods of vector-magnitude and scalar expansion form the two extremes of a spectrum of approaches, as illustrated in Figure 4.2. At the scalar side, there is full description of interdependencies, but much higher computational complexity which limits how thoroughly one can search for the range limits. At the vector-magnitude side directional interdependencies are almost

completely lost but computational effort is significantly reduced enabling more efficient use of range search effort. A tradeoff between these two extremes is made accessible through the use of block vectors, which this section details.

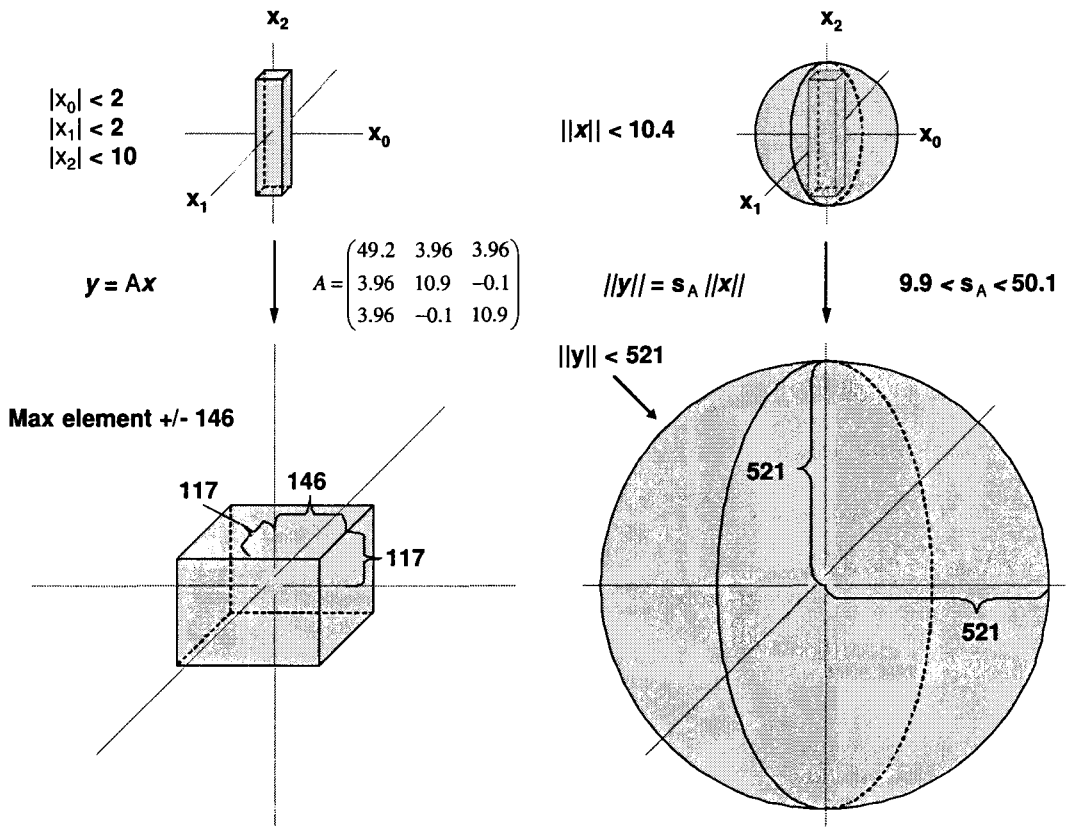
Simply put, expanding the vector-magnitude model to include some directional information amounts to expanding from the use of one variable per vector (recording magnitude) to multiple variables per vector, but still fewer than the number of elements per vector. Two natural questions arise: what information to store in those variables and, if multiple options of similar compute complexity exist, then how to choose the option that can lead to tighter bounds.

Consider as an example a simple 3x3 matrix multiplication $\mathbf{y} = A\mathbf{x}$, where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^3$ and $\mathbf{x} = [x_0, x_1, x_2]^T$. Figure 4.3(a) shows an example matrix A (with $9.9 \leq \sigma_A \leq 50.1$) as well as constraints on the component ranges of $\mathbf{x}$ (upper part of the figure). If the transformation which the matrix multiplication represents is applied, the result will be a scaled/skewed/rotated version of the original cuboid into a parallelepiped. While not depicted exactly, the resultant parallelepiped is fully and tightly contained by the box shown in the lower half of the figure. This indicates that the largest magnitude element is ≈ 146, and under the assumption of uniform bit-widths each element in the vector would require 9 range bits. Since these calculations were carried out exactly on the full scalar model, they are absolute minimum bounds, thus at least 9 bits are definitely required, and are in fact sufficient.

Moving beyond the direct calculation method of Figure 4.3(a) is Figure 4.3(b) wherein the vector-magnitude approach is applied: $||\mathbf{x}||$ is bounded by $\sqrt{2^2 + 2^2 + 10^2} = 10.4$, and the bound on $||\mathbf{y}||$ is obtained as $\sigma_A^{max}||\mathbf{x}||$ which for this example is $50.1 \times 10.4 \approx 521$.

The vector-magnitude estimate of ≈ 521 can be seen to be relatively poor over the true component bounds for matrix multiplication calculated to be ≈ 146. The inflation results from dismissal of correlations between components in the vector. To address this loss of correlation, block vector partitioning is proposed which is discussed next. The same vector magnitude model is applied, but to multiple partial vectors formed by breaking the original vector.

Figure 4.4(a) shows one possible partitioning scenario, where partitioning is performed around the component with the largest range, i.e. $\mathbf{x_0} = [x_0, x_1]^T, \mathbf{x_1} = x_2$. The input range



(a) Scalar model.

(b) Vector magnitude model.

Figure 4.3: Vector matrix multiplication example: scalar vs. vector magnitude.

bounds now translate into a circle (bounded magnitude) in the $[x_0, x_1]$-plane and a range on the x_2-axis. The corresponding partitioning of the matrix is:

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \quad \begin{bmatrix} \mathbf{y_0} \\ \mathbf{y_1} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} \mathbf{x_0} \\ \mathbf{x_1} \end{bmatrix}$$

where $10.4 \leq \sigma_{A_{00}} \leq 49.7$ and $0 \leq \sigma_{A_{01}}, \sigma_{A_{10}} \leq 3.97$ and simply $A_{11} = \sigma_{11} = 10.9$. Using block vector arithmetic [42], which bears a large degree of resemblance to standard vector

arithmetic, it can be shown that

$$\mathbf{y_0} = A_{00}\mathbf{x_0} + A_{01}\mathbf{x_1}$$
$$\mathbf{y_1} = A_{10}\mathbf{x_0} + A_{11}\mathbf{x_1}.$$

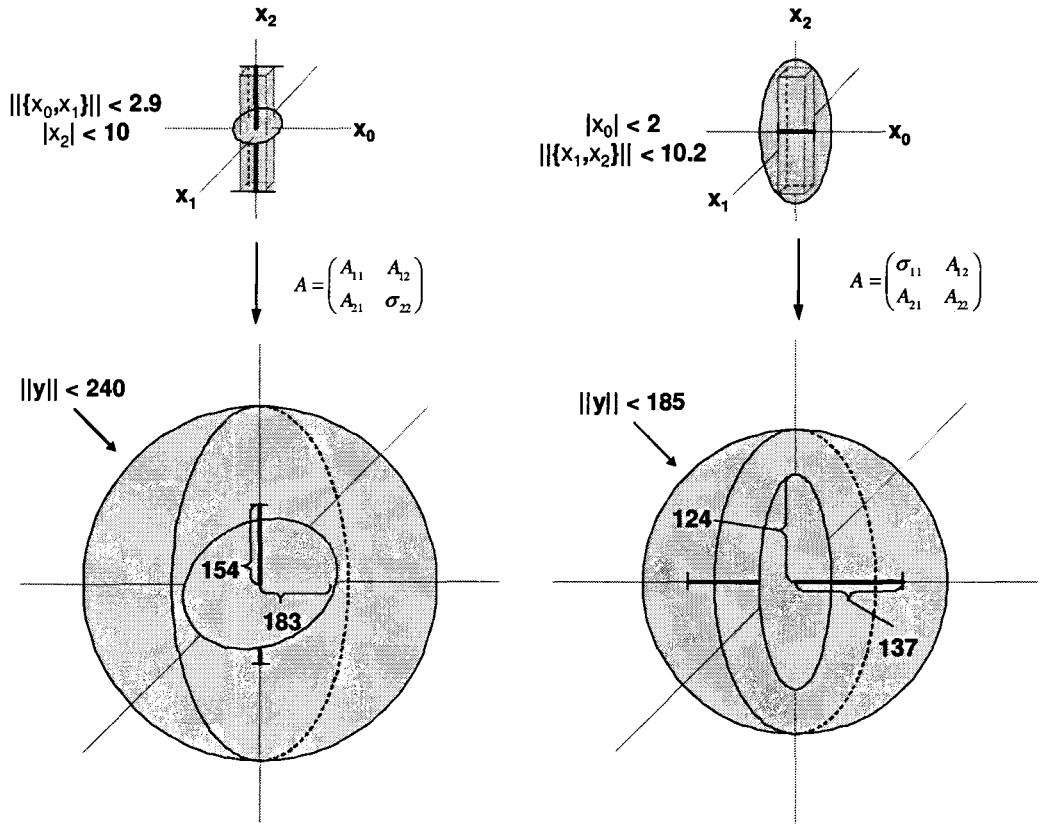
Expanding each of these equations using vector-magnitude in accordance with the operations from Table 4.1, we obtain:

$$||\mathbf{y_0}||^2 = (\sigma_{A_{00}}||\mathbf{x_0}||)^2 + (\sigma_{A_{01}}||\mathbf{x_1}||)^2 + 2\sigma_{A_{00}}\sigma_{A_{01}}||\mathbf{x_0}||||\mathbf{x_1}||$$
$$||\mathbf{y_1}||^2 = (\sigma_{A_{10}}||\mathbf{x_0}||)^2 + (\sigma_{A_{11}}||\mathbf{x_1}||)^2 + 2\sigma_{A_{10}}\sigma_{A_{11}}||\mathbf{x_0}||||\mathbf{x_1}||$$

As Figure 4.4(a) shows, applying the vector-magnitude calculation to the partitions individually amounts to expanding the circle in the $[x_0, x_1]$-plane, as well as expanding the x_2 range, after which these two magnitudes can be recombined by taking the norm of $[\mathbf{y_0}, \mathbf{y_1}]^T$ to obtain the overall magnitude $||\mathbf{y}|| = \sqrt{||\mathbf{y_0}||^2 + ||\mathbf{y_1}||^2}$. By applying the vector-magnitude calculation to the partitions individually, the directional information about the component with the largest range is taken into account. This yields $||\mathbf{y_0}|| \leq\approx 183$ and $||\mathbf{y_1}|| \leq\approx 154$, thus producing the bound $||\mathbf{y}|| \leq\approx 240$.

Next to Figure 4.4(a), Figure 4.4(b) shows an alternative way of partitioning the vector, this time with respect to the basis vectors of the matrix A. Decomposition of the matrix used in this example reveals the direction associated with the largest σ_A to be very close to the x_0 axis. Partitioning in this way (i.e., $\mathbf{x_0} = x_0, \mathbf{x_1} = [x_1, x_2]^T$) results in the same equations as above for partitioning around x_2, but with different values for the sub-matrices: $A_{00} = \sigma_{00} = 49.2$ and $0 \leq \sigma_{A_{01}}, \sigma_{A_{10}} \leq 5.61$ and $10.8 \leq \sigma_{A_{11}} \leq 11.0$. As the figure shows, we now have range on the x_0-axis and a circle in the $[x_1, x_2]$-plane which expand according the same rules (with different numbers) as before yielding this time: $||\mathbf{y_0}|| \leq\approx 137$ and $||\mathbf{y_1}|| \leq\approx 124$ producing a tighter overall magnitude bound $||\mathbf{y}|| \leq\approx 185$.

Given the larger circle resulting from this choice of expansion, it may seem surprising the bounds obtained are tighter in this case. However, consider that in the x_0 direction (very close to the direction of the largest σ_A), the bound is overestimated by $2.9/2 \approx 1.5$ in the partitioning from Figure 4.4(a), while it is exact in the partitioning from Figure 4.4(b). Contrast this to the overestimation of the magnitude in the $[x_1, x_2]$-plane of only about 2% in which case it makes sense that this partitioning provides the tighter bound. On the other hand, different basis vectors for A could reverse the situation leading to better bounds



(a) Partitioning on $\mathbf{x}_2$.

(b) Partitioning on $\mathbf{x}_0$.

Figure 4.4: Effect of partitioning on range overestimation.

from the other partitioning. Clearly the decision of how to partition a vector can have significant impact on the quality of the bounds. Consequently, the next section discusses an algorithmic solution for making this decision.

4.1.5 Partition selection

Recall that in the cases from Figures 4.4(a) and 4.4(b), the magnitude inflation is reduced, but for two different reasons. Considering this fact from another angle, it can be restated

that the initial range inflation (which we are reducing by moving to block vectors) arises as a result of two different phenomena; 1) inferring larger ranges for individual components as a result of one or a few components with large ranges as in the middle case or 2) allowing any vector in the span defined by the magnitude to be scaled by the maximum σ_A even though this only really occurs along the direction of the corresponding basis vector of A.

In the case of magnitude overestimation resulting from one or a few large components, the impact can be quantified using range inflation: $f_{vec}(\mathbf{x}) = ||\mathbf{x}||/||\hat{\mathbf{x}}||$, where $\hat{\mathbf{x}}$ is $\mathbf{x}$ with the largest component removed. Intuitively, if all the components have relatively similar ranges, this expression will evaluate to near unity. On the other hand, removing a solitary large component range will produce a value larger than (and farther from) the value 1, as in the example from Figure 4.4(a) of $10.4/2.9 = 3.6$

Turning to the second source of inflation based on σ_A, we can similarly quantify the penalty of using σ_A over the entire input span by evaluating the impact of removing a component associated with the largest σ_A. If we define α as the absolute value of the largest component of the basis vector corresponding to σ_A^{max}, and $\hat{A}$ as the block matrix obtained by removing that same component's corresponding row and column from A, we can define: $f_{mat}(A) = \alpha\sigma_A^{max}/\sigma_{\hat{A}}^{max}$, where $\sigma_{\hat{A}}^{max}$ is the maximum across the σ^{max} of each sub-matrix of $\hat{A}$. The α factor is required to weight the impact of that basis vector as it relates to an actual component of the vector which A multiplies. When σ_A^{max} is greater than the other σ_A, f_{mat} will increase (Figure 4.4(b)) to $0.99 \times 49.2/11.0 = 4.4$. Also note that the partition with the greater value ($4.4 \geq 3.6$) produces the tighter magnitude bound ($185 < 240$). Finally, it is worth mentioning that if the full singular-value decomposition (SVD) is available, the partitioning is much more straightforward - these metrics are used primarily to provide partitioning guidance when the SVD is unavailable.

Algorithm 4.1 shows the steps involved in extracting magnitude bounds (from which bit-widths can be derived) of a vector based calculation. It takes as input the specification in terms of the calculation steps (*VectorCalculation*), input variables and their ranges (*InputVarList* and *InputVarRanges* respectively) as well as intermediate variables (*IntermediateVarList*). From this input, *VectorMagnitudeModel* on Line 1 creates the base vector-magnitude model as in Section 4.1.3. The algorithm then proceeds by successively

Algorithm 4.1 VectorMagnitude

```
1: VecMagModel = VectorMagnitudeModel(VectorCalculation, InputVarList,
   InputVarRanges, IntermediateVarList)
2: UpdatedMagnitudes=DetermineRanges(VecMagModel)
3: Flag=TRUE
4: while (Flag) do
5:    VecMagModel = Partition(VecMagModel)
6:    IntermediateMagnitudes = UpdatedMagnitudes
7:    UpdatedMagnitudes=DetermineRanges(VecMagModel)
8:    Gain = max(UpdatedMagnitudes, IntermediateMagnitudes)
9:    if (Gain < GainThresh) then
10:      Flag=FALSE
11:   end if
12:   if (size(VecMagModel) > SizeThresh) then
13:      Flag=FALSE
14:   end if
15: end while
16: return IntermediateMagnitudes
```

partitioning the model (Line 5) and updating the magnitudes (Line 7) until either no significant tightening of the magnitude bounds occurs (as defined by *GainThresh* on Line 9) or until the model grows to become too complex (as defined by *SizeThresh* on Line 12). Note that the *DetermineRanges* function on Lines 2 and 7 is based upon existing range analysis techniques. Even interval or affine arithmetic could be used here, however we employ the computational method from Chapter 3. The *Partition* function (Line 5) utilizes the impact functions f_{vec} and f_{mat} to determine a good candidate for partitioning. Computing the f_{vec} function for each input vector is inexpensive, while the f_{mat} function involves two largest singular(eigen) value calculations for each matrix. It is worth noting however that a partition will not change the entire *Model*, and thus many of the f_{vec} and f_{mat} values can be reused across multiple calls of *Partition*.

With a means in hand of obtaining a block vector partitioning that provides a balance between SMT instance complexity and magnitude overestimation, the next section applies this method to a few case studies which demonstrate its operation.

4.2 Case studies

This section details 5 case studies which demonstrate application of the vector based methods discussed in Section 4.1. Two of these case studies are extended from Chapter 3 (Sections 4.2.1 and 4.2.2) where originally they used scalar equations only. Results for the case studies compare affine arithmetic operating on the vector model and the scalar expansion to the computation method of from Chapter 3 built on top of HySAT [35], again applied to both the vector model and scalar expansion. Experiments were carried out on the same platform as before, a 1.5 GHz Pentium 4 with 512 MB of RAM running Gentoo Linux. In all experiments, run-times for the experiments were on the order of 10's of minutes for the computational (SMT) scalar expansions, minutes for the vector and block vector model based on SMT and seconds for all affine arithmetic based experiments. Conversion of ranges [L,U] to bits was done according to the formula: $bits = \lceil \log_2(max(|L|,|U|)+1) \rceil + a$ where $a = 0$ if it is a scalar range (not vector magnitude) and L and U have the same sign, otherwise $a = 1$.

4.2.1 Analytic center

As we have seen in Section 3.4.3, the analytic center of a set of inequality constraints maximizes a distance metric from all constraint boundaries. Using the same as before from convex optimization, when using a distance metric based on a logarithmic penalty function; solving for the analytic center will give rise to calculations such as those below [10].

$$z[i] = d[i]\mathbf{a}[i] \cdot (\mathbf{C} - \mathbf{P}) \qquad d[i] = 1/(b[i] - \mathbf{a}[i] \cdot \mathbf{C})$$

The inequality constraints are defined by $\mathbf{a}[i], b[i]$ and $\mathbf{C}$ is the analytic center. The values $z[i]$ reflect a penalty of moving $\mathbf{C}$ to $\mathbf{P}$ with respect to $\mathbf{a}[i], b[i]$. For the specific case of 5 inequality constraints in $\mathbb{R}^3$, the vector equations can be expanded into scalar equations ($i \in \{1,2,3,4,5\}$):

$$q_1[i] = b[i] - (a_x[i]C_x + a_y[i]C_y + a_z[i]C_z) \qquad q_2[i] = 1/q_1[i]$$

$$z[i] = q_2[i]\,(a_x[i]\,(C_x - P_x) + a_y[i]\,(C_y - P_y) + a_z[i]\,(C_z - P_z))$$

Table 4.2: Affine vs. SAT-Modulo for vector and scalar analytic center.

	Scalar Affine		Scalar SAT-Modulo	
Output	Range	Bits	Range	Bits
$q_1[i]$	[-3010 , 3010]	13	[-3011 , 3011]	13
$q_2[i]$	∞	–	[-101 , 101]	8
$z[i]$	∞	–	[-3.6e5 , 3.6e5]	20
	Vector Affine		Vector SAT-Modulo	
Output	Range	Bits	Range	Bits
$q_1[i]$	[-3010 , 3010]	13	[-3011 , 3011]	13
$q_2[i]$	∞	–	[-101 , 101]	8
$q_3[i]$	∞	–	[-347 , 347]	10
$z[i]$	∞	–	[-6.1e5 , 6.0e5]	21

Consider ranges of $C_x, C_y, C_z, P_x, P_y, P_z \in [-100, 100]$, and $a_x[i], a_y[i], a_z[i], b[i] \in [-10, 10]$. The equivalent vector-magnitude model is as follows:

$$q_1[i] = b[i] - ||\mathbf{a}||||\mathbf{C}||\cos(\theta_{aC}) \qquad q_2[i] = 1/q_1[i]$$

$$q_3[i] = \sqrt{||\mathbf{C}||^2 + ||\mathbf{P}||^2 - 2||\mathbf{C}||||\mathbf{P}||\cos(\theta_{CP})}$$
$$z[i] = q_2[i]||\mathbf{a}||q_3[i]\cos(\theta_{aq_3})$$

for $0 \leq ||\mathbf{a}|| \leq 17.4$ and $0 \leq ||\mathbf{C}||, ||\mathbf{P}|| \leq 173.3$.

Table 4.2 shows ranges and equivalent bit-widths for the analytic center calculation when using affine arithmetic and the SAT-Modulo Theory (SMT) approach. Because in this and in the next case study no matrix multiplications exist, the block vectors are not required (only the vector-magnitude results are shown). Up to q_2, the vector-magnitude model gives identical results to the scalar expansion, but overestimates z, a result of losing the correlations. Note also that the singularity is circumvented by adding the constraint $q_1^2 \geq 0.0001$, which is convenient in the SMT approach, but for which no mechanism exists in affine arithmetic.

4.2.2 Euclidian projection

A second convex optimization/analysis based case study coming again from Chapter 3 is Euclidian projection of a point/points onto a hyperplane. In Section 3.4.4 we saw that for a hyperplane $\mathbf{a} \cdot \mathbf{x} + b = 0$, a point $\mathbf{x0}$ will have a projection:

$$P(\mathbf{x0}) = \mathbf{x0} + \frac{b - \mathbf{a} \cdot \mathbf{x0}}{\mathbf{a} \cdot \mathbf{a}} \mathbf{a}$$

If we consider for the case study $\mathbf{x}$ substituted for $\mathbf{x0}$ and $\mathbf{a}, \mathbf{x} \in \mathbb{R}^5$, once again the vector equations can be expanded into scalar equations:

$$q_1 = \sum_{i=0}^{4} a_i x_i; \;\; q_2 = \sum_{i=0}^{4} a_i^2$$
$$q_3 = b - q_1; \;\; q_4 = q_3 / q_2$$
$$z_i = x_i + q_4 a_i, \;\; i \in \{0,1,2,3,4\}$$

where $-100 \leq x_i \leq 100$ and $-10 \leq a_i \leq 10$ for $i \in \{0,1,2,3,4\}$ and $-10 \leq b \leq 10$. The vector-magnitude model can also be formulated:

$$q_1 = ||\mathbf{a}|| ||\mathbf{x}|| \cos(\theta_{ax}); \;\; q_2 = ||\mathbf{a}||^2$$
$$q_3 = b - q_1; \;\; q_4 = q_3 / q_2$$
$$||\mathbf{z}|| = \sqrt{||\mathbf{x}||^2 + (q_4 ||\mathbf{a}||)^2 + 2||\mathbf{x}|| q_4 ||\mathbf{a}|| \cos(\theta_{ax})}$$

with $0 \leq ||\mathbf{x}|| \leq 223.7$ and $1 \leq ||\mathbf{a}|| \leq 22.4$.

Table 4.3 shows ranges and equivalent bit-widths for Euclidian projection under the conditions described above. Similarly to analytic center, the vector model keeps up well with the scalar model for the first intermediates and in fact, as of q_4, the vector model actually surpasses the scalar model to provide *better* results, since it can be more thoroughly searched due to lower computational complexity. Note also that singularity avoidance is unnecessary due to the $||\mathbf{a}|| \geq 1$ constraint in the specification. Nonetheless, since no mechanism exists in affine arithmetic for capturing such a constraint, it cannot handle the division.

Table 4.3: Affine vs. SAT-Modulo for vector and scalar Euclidian projection.

	Scalar Affine		Scalar SAT-Modulo	
Output	Range	Bits	Range	Bits
q_1	[-5000 , 5000]	14	[-5001 , 5001]	14
q_2	[0 , 500]	9	[0 , 501]	9
q_3	[-5010 , 5010]	14	[-5011 , 5011]	14
q_4	∞	–	[-348 , 348]	10
z_i	∞	–	[-646 , 630]	11
	Vector Affine		Vector SAT-Modulo	
Output	Range	Bits	Range	Bits
q_1	[-5003 , 5003]	14	[-5003 , 5003]	14
q_2	[-114 , 501]	9	[0 , 501]	9
q_3	[-5013 , 5013]	14	[-5013 , 5013]	14
q_4	∞	–	[-235 , 235]	9
z_i	∞	–	[-270 , 271]	10

4.2.3 Davidon-Fletcher-Powell formula

Moving out of the realm of convex optimization, a formula that has been applied in the quasi-Newton method as a multidimensional generalization of the secant method for non-linear optimization is the Davidon-Fletcher-Powell (DFP) formula [94]. The method maintains at each step of optimization a local estimate of the Hessian of the objective function, and proceeds by successively updating this estimate. The DFP formula solves the secant equation with the least modification to the current Hessian estimate, while maintaining symmetry and positive definiteness of the Hessian.

Calculations such as this one can arise in embedded systems which must make decisions based on very complex models. For this case study we have focussed on the update equation for the inverse Hessian:

$$H_{k+1} = H_k - \frac{H_k \mathbf{y_k} \mathbf{y_k}^T H_k}{\mathbf{y_k}^T H_k \mathbf{y_k}} + \frac{\mathbf{s_k} \mathbf{s_k^T}}{\mathbf{y_k}^T \mathbf{s_k}}$$

which takes as input the current (for iteration k) estimate of the inverse Hessian matrix H_k,

the step vector $\mathbf{s_k}$, and the change in the gradient of the objective across the step $\mathbf{y_k}$ (a vector). Since the true ranges of these values depend so heavily on the context, we have picked up arbitrary element ranges of [-100..100] for elements of H_k which has eigenvalues in the range [0..100], and [-10..10] for $\mathbf{s_k}$ and $\mathbf{y_k}$, for a problem of dimension 3. These values suffice to highlight the difference between the range determination methods in question, which is our goal, rather than to explicitly solve the range problem in a specific context.

The vector calculations for DFP are as follows:

$$\begin{array}{lll} \mathbf{q1} = H_k \mathbf{y_k} & q4 = \mathbf{y_k}^T \mathbf{q1} & q7 = \mathbf{y_k}^T \mathbf{s_k} \\ \mathbf{q2}^T = \mathbf{y_k}^T H_k & Q5 = Q3/q4 & Q8 = Q6/q7 \\ Q3 = \mathbf{q1q2}^T & Q6 = \mathbf{s_k s_k}^T & \end{array}$$

$$H_{k+1} = H_k - Q5 + Q8$$

Note that $Q3$ and $Q6$ are both outer products yielding matrices while $q4$ and $q7$ are both inner products yielding scalars.

Ranges and equivalent bit-widths for the intermediate variables of this calculation are shown in Table 4.4 for scalar expansion and the vector magnitude model, where $[]_i j$ has been used to indicate element bounds. Notice first that as before, affine arithmetic is unable to resolve the division, however insertion of constraints like before ($q4^2 \geq 0.0001, q7^2 \geq 0.0001$) enable SMT to still provide meaningful ranges. Note that while scalar affine cannot actually proceed any further than $[Q5]_{ij}$, if we borrow the missing ranges for the divisions $(q4, q7)$ from the scalar SMT results, the other intermediate ranges given by affine are the same as the scalar SMT.

Turning to the vector results, notice that, as above, the affine and SMT results track together until the division. If the ranges are borrowed, as before, the ranges continue to track until $\mathbf{H^{k+1}}$, where affine arithmetic grossly overestimates the range. This happens because in this case the vector magnitude model is more sensitive to the correlations, which are lost when borrowing the ranges from the SMT vector results.

Finally for this case study, note the difference between the SMT scalar and SMT vector results. For $Q6$ and $Q8$, the range for SMT scalar is slightly tighter than for SMT vector, but for all other variables SMT vector gives the same or tighter ranges. Unlike in the case study from Section 4.2.2, this is not a result of the SMT vector being able to more thoroughly

Table 4.4: Affine vs. SAT-Modulo for vector and scalar Davidon-Fletcher-Powell.

	Scalar Affine		Scalar SAT-Modulo	
Output	Range	Bits	Range	Bits
$[q1]_i$	[-3e3 , 3e3]	13	[-3e3 , 3e3]	13
$[q2]_i$	[-3e3 , 3e3]	13	[-3e3 , 3e3]	13
$[Q3]_{ij}$	[-9e6 , 9e6]	25	[-4.01e6 , 9.01e6]	25
$q4$	[0 , 9e4]	17	[0 , 9e4]	17
$[Q5]_{ij}$	∞	–	[-9.01e9 , 9.01e9]	35
$[Q6]_{ij}$	[-100 , 100]	8	[-101 , 101]	8
$q7$	[-300 , 300]	10	[-301 , 301]	10
$[Q8]_{ij}$	∞	–	[-1.01e4 , 1.01e4]	15
$[H_{k+1}]_{ij}$	[-9.1e8 , 9.1e8]	31	[-9.1e8 , 9.1e8]	31
	Vector Affine		Vector SAT-Modulo	
Output	Range	Bits	Range	Bits
$\|\|\mathbf{q1}\|\|$	[0 , 1740]	12	[0 , 1741]	12
$\|\|\mathbf{q2}\|\|$	[0 , 1740]	12	[0 , 1740]	12
$\|\|Q3\|\|$	[0 , 3.03e6]	23	[0 , 3.03e6]	23
$q4$	[-3.03e4 , 3.03e4]	16	[-3.03e4 , 3.03e4]	16
$\|\|Q5\|\|$	∞	–	[0 , 3.03e8]	30
$\|\|Q6\|\|$	[0 , 303]	10	[0 , 303]	10
$q7$	[-303 , 303]	10	[-303 , 303]	10
$\|\|Q8\|\|$	∞	–	[0 , 3.03e4]	16
$\|\|H_{k+1}\|\|$	[0 , 9.17e16]	58	[0 , 4.29e8]	30

search the solution space. Rather, the results from SMT scalar are in fact tight based on the specification, but the scalar specification is unable to capture the eigenvalue constraint on H_k. This illustrates another very useful feature of the vector magnitude model, the ability to capture some specification aspects rooted in the vector structure of the calculation, which are awkward or impossible to represent in the scalar expansion.

4.2.4 Conjugate Gradient method

The fourth case study is based on a single iteration of the Conjugate Gradient method for solving linear systems of equations [113]. This method has many applications, examples include finite element method analysis and solutions to partial differential equations. A single (in fact the first) iteration can be formulated as follows. Given a matrix A, and a vector $\mathbf{b}$ with an initial guess for $\mathbf{x}$ (which solves $A\mathbf{x} = \mathbf{b}$) of $\mathbf{x_0}$ then let:

$$A = \begin{bmatrix} 5.665 & 2.630 & 1.088 \\ 2.630 & 9.624 & 2.647 \\ 1.088 & 2.647 & 6.329 \end{bmatrix}$$

$$\begin{array}{ccc} \mathbf{r} = \mathbf{b} & \mathbf{d} = \mathbf{x_0} & \mathbf{q} = A\mathbf{d} \\ \alpha = \frac{\mathbf{r}^T\mathbf{r}}{\mathbf{d}^T\mathbf{q}} & \mathbf{r}' = \mathbf{r} - \alpha\mathbf{q} & \beta = \frac{\mathbf{r}'^T\mathbf{r}'}{\mathbf{r}^T\mathbf{r}} \\ & \mathbf{d}' = \mathbf{r}' + \beta\mathbf{d} & \end{array}$$

and $\mathbf{r}', \mathbf{d}'$ feed into the next iteration.

Taking constraints on the inputs as $0.1 \leq \mathbf{x_0} \leq 104$ and $0.1 \leq \mathbf{b} \leq 260$, Table 4.5 shows the ranges of the intermediates obtained through affine arithmetic and SMT. In this case, due to the increased complexity of the scalar formulation caused by the matrix multiplication, the scalar method over-estimates the ranges (this is because of the timeouts due to the problem size). While there is no guarantee that the ranges obtained through the vector method are optimal, they are significantly better than the scalar ones once again because the reduced complexity of the formulation enables more thorough search of the solution.

The significant reduction in bit-width from the vector method in this case study arises largely because the eigenvalues of the matrix are fairly uniformly distributed, and the input ranges of the vectors are uniform over the elements. This algorithm also has inherently weak directional interdependencies between intermediate variables, they are more strongly correlated in terms of magnitude, which further accounts for the success of the vector-magnitude approach. Because of this however, no significant gains are made by applying the block-vector approach, unlike the next study.

Table 4.5: Affine vs. SAT-Modulo for vector and scalar Conjugate Gradient.

	Scalar Affine		Scalar SAT-Modulo	
Output	Range	Bits	Range	Bits
$[d]_i$	[-104 , 104]	8	[-104 , 104]	8
$[r]_i$	[-260 , 260]	10	[-260 , 260]	10
$[q]_i$	[-1550 , 1550]	12	[-1075 , 1075]	12
rTr	[0 , 2.03e5]	19	[0 , 7.74e4]	17
dTq	[-1.38e5 , 3.72]	20	[0 , 1.44e5]	18
α	∞	–	[0 , 7.78e8]	30
$[r']_i$	[-1.21e12 , 1.21e12]	42	[-1.02e11 , 9.40e10]	38
$r'Tr'$	[-2.06e24 , 2.70e24]	83	[0 , 2.08e22]	75
β	∞	–	[0 , 2.23e17]	58
$[d']_i$	[-2.32e19 , 2.32e19]	66	[-8.24e18 , 7.33e18]	64
	Vector Affine		Vector SAT-Modulo	
Output	Range	Bits	Range	Bits
$\|\|\mathbf{d}\|\|$	[0.1 , 104]	8	[0.1 , 104]	8
$\|\|\mathbf{r}\|\|$	[0.1 , 260]	10	[0.1 , 260]	10
$\|\|\mathbf{q}\|\|$	[0.42 , 1300]	12	[0.42 , 1300]	12
$\mathbf{r}^T\mathbf{r}$	[0 , 6.77e4]	18	[0 , 6.77e4]	18
$\mathbf{d}^T\mathbf{q}$	[0 , 1.36e5]	19	[0 , 1.36e5]	19
α	∞	–	[0 , 1.60e6]	22
$\|\|\mathbf{r}'\|\|$	∞	–	[0 , 2.01e6]	22
$\mathbf{r}'^T\mathbf{r}'$	∞	–	[0 , 4.04e12]	43
β	∞	–	[0 , 5.97e7]	27
$\|\|\mathbf{d}'\|\|$	∞	–	[0 , 4.18e7]	27

4.2.5 FFT based correlation

The final case study is based on a fast method for computing correlation via sum-of-square-differences for 2-dimensional (2D) data, as applied to object tracking. An application and its dataflow are detailed in [49], which has been reproduced below. The inputs f, g are 2D arrays of data values, referred to by [49] as the search and reference window respectively and the final correlation result *ssd* is:

$$ssd = \mathscr{F}^{-1}\{(sinc \circ \mathscr{F}\{f \circ f\}) - 2(\mathscr{F}\{f\} \circ \mathscr{F}\{g\}^*)\}$$

where recall from Table 4.1 that $\circ$ is the element-wise product of arrays, *sinc* is the 2D sinc function of appropriate size, and $\mathscr{F}$ is computed using the Fast Fourier Transform (FFT).

Two points are of primary interest in Table 4.6. First, note that the vector method overestimates the range for $\mathscr{F}\{ssd\}$, this is due to the strong directional correlation of the pointwise matrix produce. However, the range for *ssd* is actually smaller, due to the same phenomenon of the previous case study, i.e., the scalar instance becomes too complex that it cannot be feasibly searched and thus overestimates the bit-width. In the bottom portion of the table, block vectors have been implemented leading to tightening of the range of the range for $\mathscr{F}\{ssd\}$, while the range of *ssd* is unaffected due to the directional independence of the FFT in the final step.

4.3 Summary

This chapter has shown how to deal with vectors when allocating bit-widths for hardware accelerators, which ultimately will impact both area and performance of the accelerator. Formal approaches are required for scientific algorithms to guarantee robustness and thus correctness, however runtimes of formal methods can scale unacceptably. We have introduced two primary techniques to manage the runtime of our formal approach without compromising the quality of the solution: the vector-magnitude model and the block-vector model. Both of these can deal with abstract data types, including real and complex valued. The raised level of abstraction this method provides also opens the door to analyzing and leveraging vector specific representations such as [52]. There are far-reaching benefits to

Table 4.6: Affine vs. SAT-Modulo for vector and scalar FFT-based correlation.

Output	Scalar Affine		Scalar SAT-Modulo	
	Range	Bits	Range	Bits
$f \circ f$	[0 , 6.56e4]	17	[0 , 6.56e4]	17
$F = \mathscr{F}\{f\}$	[-512 , 1024]	12	[-512 , 1024]	12
$\mathscr{F}\{f \circ f\}$	[-1.64e5 , 2.63e5]	20	[-1.32e5 , 2.63e5]	20
$F \circ G^*$	[-2.63e5 , 1.05e6]	22	[-2.63e5 , 1.05e6]	22
$\mathscr{F}\{ssd\}$	[-1.93e6 , 1.05e6]	22	[-1.99e6 , 5.25e5]	22
ssd	[-1.06e8 , 9.23e7]	28	[-1.01e8 , 8.01e7]	28
Output	Vector Affine		Vector SAT-Modulo	
	Range	Bits	Range	Bits
$f \circ f$	[0 , 4.20e6]	24	[0 , 4.20e6]	24
$F = \mathscr{F}\{f\}$	[0 , 2.05e3]	13	[0 , 2.05e3]	13
$\mathscr{F}\{f \circ f\}$	[0 , 4.20e6]	24	[0 , 4.20e6]	24
$F \circ G^*$	[0 , 4.20e6]	24	[0 , 4.20e6]	24
$\mathscr{F}\{ssd\}$	[0 , 7.55e7]	28	[0 , 7.55e7]	28
ssd	[0 , 7.55e7]	28	[0 , 7.55e7]	28
Output	Block Vector Affine		Block Vector SAT-Modulo	
	Range	Bits	Range	Bits
$f \circ f$	[0 , 4.20e6]	24	[0 , 4.20e6]	24
$F = \mathscr{F}\{f\}$	[0 , 2.05e3]	13	[0 , 2.05e3]	13
$\mathscr{F}\{f \circ f\}$	[0 , 4.20e6]	24	[0 , 4.20e6]	24
$F \circ G^*$	[0 , 2.10e6]	23	[0 , 2.10e6]	23
$\mathscr{F}\{ssd\}$	[0 , 3.78e7]	27	[0 , 3.78e7]	27
ssd	[0 , 7.55e7]	28	[0 , 7.55e7]	28

the implementation flow, because this is a key step before RTL synthesis which has traditionally lacked automation for robust solutions, which are required to migrate scientific applications to hardware.

Chapter 5

Custom floating-point for scientific calculations

In the previous two chapters, computational methods based on SMT have been employed to establish bounds on intermediate variables within a calculation, and data abstractions have been employed to reduce SMT solver instance complexity. Both of these have focused on the range aspect of the problem to facilitate explanations and to not detract from the point of each chapter through the greater level of detail required to consider the precision aspect. Now that a scalable computational bounding framework has been developed, this chapter builds on top of it to address the issue of precision for scientific computing calculations [67], including iterative methods.

5.1 Method

This section as a whole details the approach to solving the custom-representation bit-width allocation problem for general scientific calculations. An error model to unify fixed and floating-point representations is given in Section 5.1.1 and Section 5.1.2 shows how a calculation is mapped under this model into SMT constraints. Following this, Section 5.1.3 discusses how to break the problem for iterative calculations down using an iterative analysis phase (described in Section 5.1.4) into a sequence of analyses on direct calculations (described in Section 5.1.5).

5.1.1 Fixed/floating-point error model

Fixed- and floating-point representations are typically treated separately for quantization error analysis, given that they represent different modes of uncertainty: absolute and relative respectively. Absolute error analysis applied to fixed-point numbers is exact however, relative error analysis does not fully represent floating-point quantization despite being relied on solely in many situations. The failure to exactly represent floating-point quantization arises from the finite exponent field which limits the smallest magnitude, non-zero number which can be represented. In order to maintain true relative error for values infinitesimally close to zero, the quantization step size between representable values in the region around zero would have to be zero itself. Zero step would of course not be possible since neighbouring representable values would be the same. Therefore, real floating-point representations have quantization error which behaves in an absolute way (like fixed-point representations) in the region of this smallest representable number. For this reason, when dealing with floating-point error tolerances, consideration must be given to the smallest representable number, as illustrated in the following example.

Relative Error Example. *Consider the addition of two floating-point numbers:* $a+b=c$ *with* $a,b \in [10,100]$*, each having a 7 bit mantissa yielding relative error bounded by* $\approx 1\%$*. The value of c ranges over* $[20,200]$*, and straightforward relative error analysis for c gives relative error of* 1%*, and in this case the result is reliable.*

Consider now different input ranges: $a,b \in [-100,100]$*, but still the same tolerances (7 bits* $\approx 1\%$*). Simplistic relative error analysis should still give the same result of* 1% *for c however, the true relative error is unbounded. This results from the inability to represent infinitesimally small numbers, i.e. the relative error in* a,b *is not actually bounded by* 1% *for all values in the range. A finite number of exponent bits limits the smallest representable number, which then becomes the step size at zero, leading to unbounded relative error for any value between this step size and zero.*

Unbounded relative error near zero is not a problem for the operands only, but also for the result c. In the specific case of $a = 100 \pm 1, b = -99 \pm 0.99$ *both operands have clearly bounded relative error however, we obtain* $c = 1 \pm 1.99$ *which has nearly* 200% *relative*

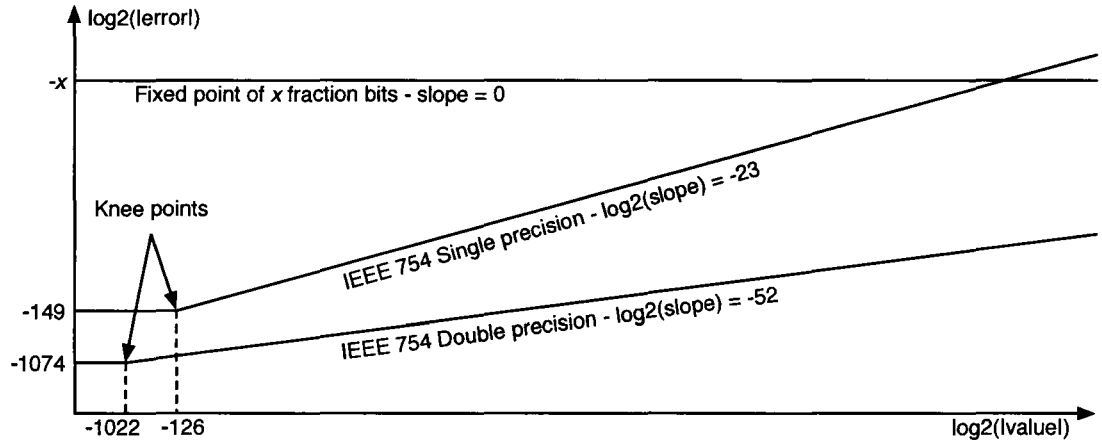


Figure 5.1: Unified fixed/floating-point error model characterizing data type by knee and slope.

error. This phenomenon is known as catastrophic cancelation *[41] and (like in the previous paragraph) grows without bound as the result c is brought closer to zero.*

In both cases within the given example, the problems could be dealt with by utilizing absolute error analysis however, this may produce wasteful representations having much tighter tolerances than necessary for large numbers. For instance, absolute error of 0.01 for a, b (7 fraction bits) leads to absolute error of 0.02 for c, which translates into a relative error bound of 0.01% for $c = 200$ (which could be unnecessarily tight) and 100% for $c = 0.02$ (which may be unacceptably loose). This impasse arising from the tension between relative error near zero and absolute error far away from zero forms the motivation for a hybrid error model.

Figure 5.1 shows a unified absolute/relative error model which can simultaneously model fixed- and floating-point numbers by providing a means of restricting when absolute or relative error applies. Error in this model is quantified in terms of two numbers (knee and slope as described below) instead of just one (as in the case of absolute and relative error analysis). Put very simply, the *knee point* divides the range of the value of a variable into absolute and relative error regions, and the slope indicates the relative error bound in

the relative region (above and to the right of the knee point). Below and to the left of the knee point the absolute error is bounded by the value *knee*, the absolute error at the knee point (which sits at the value knee / slope). We thus define:

- *knee* : absolute error bound below the knee-point
- *knee point* = knee / slope : value at which error behaviour transitions from absolute to relative
- *slope* : fraction of value which bounds the relative error in the region above the knee point.

As shown in Figure 5.1, the model can embody the error behaviour of both fixed-point and floating-point types such as IEEE-754 single and double precision. Of more significance than just being able to capture the error behaviour of existing representations however, is the ability to provide error constraints to the bit-width allocation process which are more more descriptive than basic absolute or relative error bounds. Through this model, a designer can specify explicitly the desired error behaviour of the system, potentially opening the door to greater optimization than is possible considering only absolute or relative error alone (both of which are subsumed by this model). Also, under this model bit-width allocation is no longer fragmented between fixed- and floating-point procedures. Instead, custom representations are derived from knee and slope values obtained subject to application constraints/optimization objectives. How to construct precision constraints for an SMT formulation will be discussed in Section 5.1.2, while translation of the aforementioned application objectives into such constraints is the subject of the subsequent Sections 5.1.3 and 5.1.4.

5.1.2 Forming precision constraints

The role of precision constraints is to capture the precision behaviour of intermediate variables, supporting (in our context) the hybrid error model discussed above in Section 5.1.1. The concept of a precision expression was introduced briefly in Chapter 2, in Section 2.2.2 for the sake of facilitating discussion of previous work. Here more detail is provided on

expressions, as well as how constraints are formed for the SMT instance. We define four symbols relating to an intermediate (e.g., x):

1. the unaltered variable x stands for the ideal abstract value in infinite precision,
2. $\bar{x}$ stands for the finite precision value of the variable,
3. Δx stands for accumulated error due to finite precision representation, so $\bar{x} = x + \Delta x$,
4. δx stands for the part of Δx which is introduced during quantization of x.

By using these variables we can use the SMT instance to keep track symbolically of the finite precision effects, and place constraints where they are needed to satisfy objectives from the calculation specification. In order to keep the size of the precision (e.g., Δx) terms from growing explosively, atomic precision expressions can be derived at the operator level. Consider for example division and replace the infinite precision expression $z = x/y$ with:

$$\begin{aligned} \bar{z} &= \bar{x}/\bar{y} \\ \bar{z}\,\bar{y} &= \bar{x} \\ (z+\Delta z)(y+\Delta y) &= x+\Delta x \\ zy + z\Delta y + y\Delta z + \Delta y\Delta z &= x+\Delta x \\ y\Delta z + \Delta y\Delta z &= \Delta x - z\Delta y \\ \Delta z &= \frac{\Delta x - z\Delta y}{y+\Delta y}. \end{aligned}$$

What is shown above describes the operand induced error component of Δz, measuring the reliability of z given that there is uncertainty in its operands. If next z were cast into a finite precision data type, this quantization results in $\Delta z = \frac{\Delta x - z\Delta y}{y+\Delta y} + \delta z$, where δz captures the effect of the quantization. The type of quantization (its behaviour) can be specified by what kind of constraints are placed on δz, which is discussed in more detail below.

This same process shown above for division can be applied to many different operators, scalar and vector alike. Table 5.1 shows expressions and accompanying constraints for common operators. In particular the precision expression for square root as shown in the table highlights a principal strength of the computational approach, that constraints rather than assignments are used. If only forward assignments were allowed, the precision

Table 5.1: Precision expression counterparts for common operators.

Operator	Infinite Precision (Expression)	Finite Precision (Constraint)
Addition	$z = x + y$	$\Delta z = \Delta x + \Delta y + \delta z$
Subtraction	$z = x - y$	$\Delta z = \Delta x - \Delta y + \delta z$
Multiplication	$z = xy$	$\Delta z = x\Delta y + y\Delta x + \Delta x\Delta y + \delta z$
Division	$z = x/y$	$\Delta z = \frac{\Delta x - z\Delta y}{y + \Delta y} + \delta z$
Square root	$z = \sqrt{x}$	$(\Delta z)^2 + 2(z - \delta z)\Delta z = \Delta x - (\delta z)^2 + 2z\delta z$
Vector inner product	$\mathbf{z} = \mathbf{x}^\mathbf{T}\mathbf{y}$	$\Delta\mathbf{z} = \mathbf{x}^\mathbf{T}(\Delta\mathbf{y}) + (\Delta\mathbf{x})^\mathbf{T}\mathbf{y} + (\Delta\mathbf{x})^\mathbf{T}(\Delta\mathbf{y}) + \delta\mathbf{z}$
Matrix-vector product	$\mathbf{z} = A\mathbf{x}$	$\Delta\mathbf{z} = A(\Delta\mathbf{x}) + (\Delta A)\mathbf{x} + (\Delta A)(\Delta\mathbf{x}) + \delta\mathbf{z}$

expression would be more complicated, involving use of the quadratic formula. Because of this feature of SMT instances however, constraints such as this one for square root are permissible.

Beyond the operators of Table 5.1, constraints for other operators can be derived under the same process shown for division, or operators can be compounded to produce more complex calculations. As an example, consider the quadratic form $\mathbf{z} = \mathbf{x}^\mathbf{T}A\mathbf{x}$, for which error constraints can be obtained by explicit derivation (starting from $\bar{\mathbf{z}} = \bar{\mathbf{x}}^\mathbf{T}A\bar{\mathbf{x}}$) or by compounding an inner product with a matrix multiplication $\mathbf{x}^\mathbf{T}(A\mathbf{x})$. Furthermore, constraints of other types can be added, for instance to capture the discrete nature of constant coefficients for exploitation similarly to the way they are exploited in [108].

When capturing an entire dataflow into constraints for an SMT formulation, careful consideration needs to be given to the δ terms. In each place they are used they must capture error behaviour for the entire part of the calculation to which they apply. For example, a matrix multiplication which is implemented using multiply accumulate units at the matrix/vector element level may quantize the sum throughout the accumulation of a single element of the result vector. In this case it is not sufficient for the $\delta\mathbf{z}$ for matrix multiplication from Table 5.1 to simply capture the final quantization of the result vector into a memory location, but the internal quantization effects throughout the operation of the

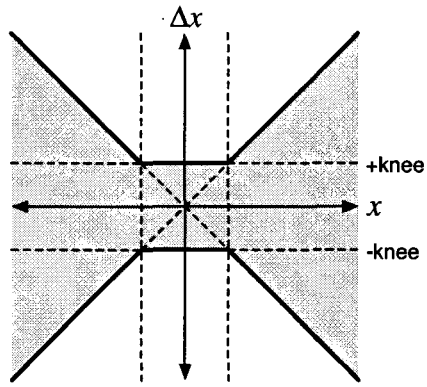
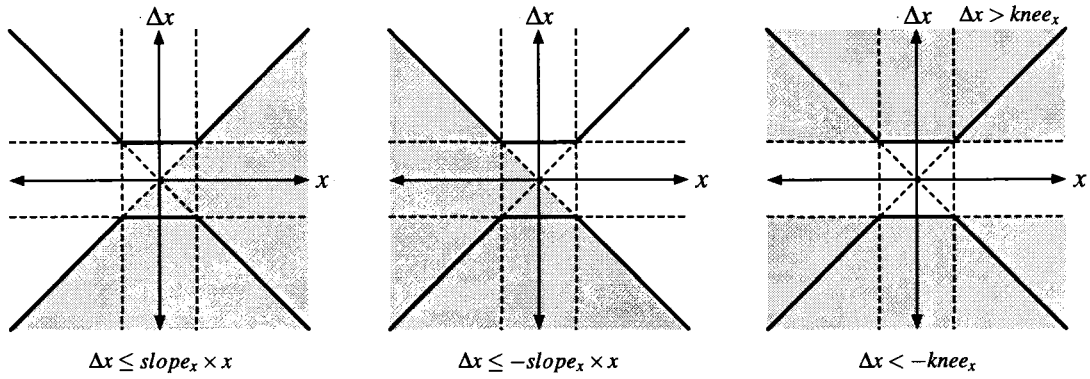


Figure 5.2: Error region for a custom floating-point number.

matrix multiply unit. What is useful about this setup is that it allows off-the-shelf hardware units (such as matrix multiply) to be modelled directly within the tool, so long as there is a clear description of the error behaviour.

The final point of importance regarding δ terms is how they are constrained to encode quantization behaviour. In the case of quantization into a fixed-point data type, the range is simply bounded by the rightmost fraction bit. For example, if a fixed-point data type with 32 fraction bits is used and quantization is done by truncation (floor), the constraint would be $-2^{-32} < \delta x \leq 0$. Similarly for rounded and ceiling, the constraints would be $-2^{-33} \leq \delta x \leq 2^{-33}$ and $0 \leq \delta x < 2^{-32}$ respectively. These constraints are loose (and therefore robust) in the sense that they assume no correlation between quantization error and value, when in reality correlation does exist. If the nature of the correlation is known, it can be captured into constraints with an accompanying increase in solver complexity. The tradeoff between tighter error bounds and increased solver complexity has to be evaluated on an application by application basis. What is important here is that the SMT framework provides the descriptive power necessary to capture the correlation if it is known.

In contrast to fixed-point quantization, constraints for floating-point numbers are more complex because error cannot anymore be divorced from value. Figure 5.2 shows the error region of a custom floating-point representation. The use of Δ indicates either potential



(a) $C1 : \Delta x \leq slope_x \times x.$ (b) $C2 : \Delta x \leq -slope_x \times x.$ (c) $C3 : \Delta x > knee_x$ $C4 : \Delta x < -knee_x.$

Figure 5.3: Partial error regions and their associated constraints.

input error or tolerable output error of a representation. In the discussion which follows the conclusions drawn apply equally to quantization if Δ is replaced with δ. Possibly the simplest constraint to describe this region indicated in Figure 5.2 would be:

$$abs(\Delta x) \leq max(slope_x \times abs(x), knee_x)$$

provided that the $abs()$ and $max()$ functions are supported by the SMT solver. In the event that they are not supported, and noting that another strength of the SMT framework is the capacity to handle both numerical and Boolean constraints, a varying set of potential constraints exists to represent this region.

Figure 5.3 shows numerical constraints which generate Boolean values and the accompanying region of the error space in which they are true. From this, a number of constraints can be formed which isolate the region of interest. For example, note that the unshaded region of Figure 5.2, where the constraint should be false, is described by:

$$\left\{\left(\overline{C1}\ AND\ \overline{C2}\ AND\ C3\right)\ OR\ \left(C1\ AND\ C2\ AND\ C4\right)\right\}$$

noting that $\overline{C1}$ refers to Boolean complementation and not finite precision as used elsewhere in this chapter and that *AND* and *OR* refer to the Boolean relations. The complement of

this produces a pair of constraints which define the region in which we are interested (note that $C3$ and $C4$ are never simultaneously true):

$$\{\overline{C1}\ OR\ \overline{C2}\ OR\ \overline{C4}\} \qquad \{C1\ OR\ C2\ OR\ \overline{C3}\}$$

$$\left\{\begin{array}{c}(\Delta x > slope_x \times x)\ OR\\ (\Delta x > -slope_x \times x)\ OR\\ (\Delta x \geq -knee_x)\end{array}\right\} \quad \left\{\begin{array}{c}(\Delta x \leq slope_x \times x)\ OR\\ (\Delta x \leq -slope_x \times x)\ OR\\ (\Delta x \leq knee_x)\end{array}\right\}$$

While the one (custom, in house developed) solver used for our experiments does not support the *abs*() and *max*() functions, the other (off-the-shelf solver, HySAT - [35, 96]), does provide this support. Even in this case however, the above Boolean constraints can be helpful alongside the strictly numerical ones. Because proof of unsatisfiability is what is required to give robustness (as shown in Chapter 3), providing extra constraints can help to speed the search by leading to a shorter proof. It also should be noted that when different solvers are used, any information known about the solver's search algorithm can be leveraged to set up the constraints in such a way to maximize search efficiency. Finally, while the above discussion relates specifically to our error model, it is by no means restricted to it - any desired error behaviour which can be captured into constraints is permissible. This further highlights the flexibility gained by using the computational approach. The next subsection now deals with partitioning an iterative calculation into pieces which can be analyzed by the framework which is now in place.

5.1.3 Iterative calculation partitioning

Numerical schemes for solving scientific problems can in general be divided into two main categories: 1) *direct* where a finite number of operations leads to the exact result and 2) *iterative* where the application of one iteration refines an estimate of the final result which is converged upon by repeated iteration. Figure 5.4 shows one way how an iterative calculation may be broken into sub-calculations which are direct, as well as the data dependency

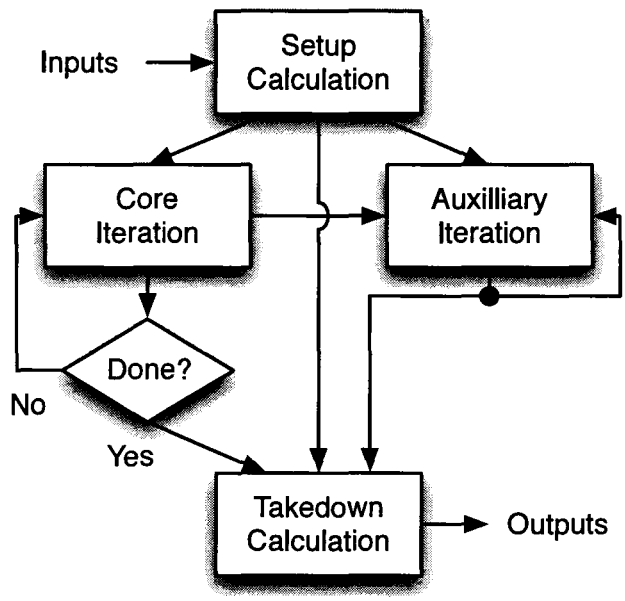


Figure 5.4: A generalized view of the flow of data within an iterative calculation.

relationships between the different sub-calculations. In particular, the *Setup, Core, Auxilliary* and *Takedown* blocks of Figure 5.4 represent direct calculations, and *Done?* may also involve some computation but produces (directly) an affirmative/negative answer. The *Setup* calculations provide (based on the inputs) initial iterates, which are updated by *Core* and *Auxillary* until the termination criterion encapsulated by *Done* is met, at which point post-processing to obtain the final result is completed by *Takedown*, which may take information from *Core, Auxillary* and *Setup*. What distinguishes the *Core* iteration from *Auxilliary* is that *Core* contains only the intermediate calculations required to decide convergence. That is, the iterative procedure will still operate and terminate the same way for the same set of inputs if the *Auxilliary* calculations are removed. The reason for this distinction is twofold: 1) convergence analysis for *Core* can be handled in more depth by the solver when calculations that are spurious (with respect to convergence) are removed, and 2) error requirements of the two parts may be different, thus needing different error analysis with the potential of leading to higher quality solutions.

Under the partitioning scheme of Figure 5.4, the top level flow for determining representations is shown in Figure 5.5. The left half of the figure depicts the partitioning and iterative analysis phases of the overall process, while the portion on the right shows bounds estimation and representation search applied to direct analysis. The intuition behind Figure 5.5 is as follows:

- At the onset, precision of inputs and precision requirements of outputs are known
- Direct analysis on the *Setup* stage with known input precision provides precision of inputs to the iterative stage
- Direct analysis on the *Takedown* stage with known output precision requirements provides output precision requirements of the iterative stage
- Between the above, and iterative analysis leveraging convergence information, forward propagation of input errors and the backward propagation of output error constraints provide the conditions for direct analysis of the *Core* and *Auxilliary* iterations.

Building on the robust computational foundation of SMT that has been established during Chapters 3 and 4, Figure 5.5 shows how SMT is leveraged through formation of the SMT constraints for bounds estimation. Also shown is the representation search which selects and evaluates candidate representations based on feedback from the hardware cost model and the bounds estimator. The reason for the bidirectional relationship between iterative and direct analysis is to retain the forward/backward propagation which marks the SMT method and thus to retain closure between the range/precision details of the algorithm inputs and the error tolerance limits imposed on the algorithm output. Over the next two sections, these iterative and direct analysis blocks will be elaborated in more detail.

5.1.4 Analysis for iterative calculations

As outlined above, the role of the iterative analysis part of the overall flow is to close the gap between forward propagation of input error, backward propagation of output error constraints and convergence constraints over the iterations. Even while the plethora of techniques which have been developed throughout the history of numerical analysis to

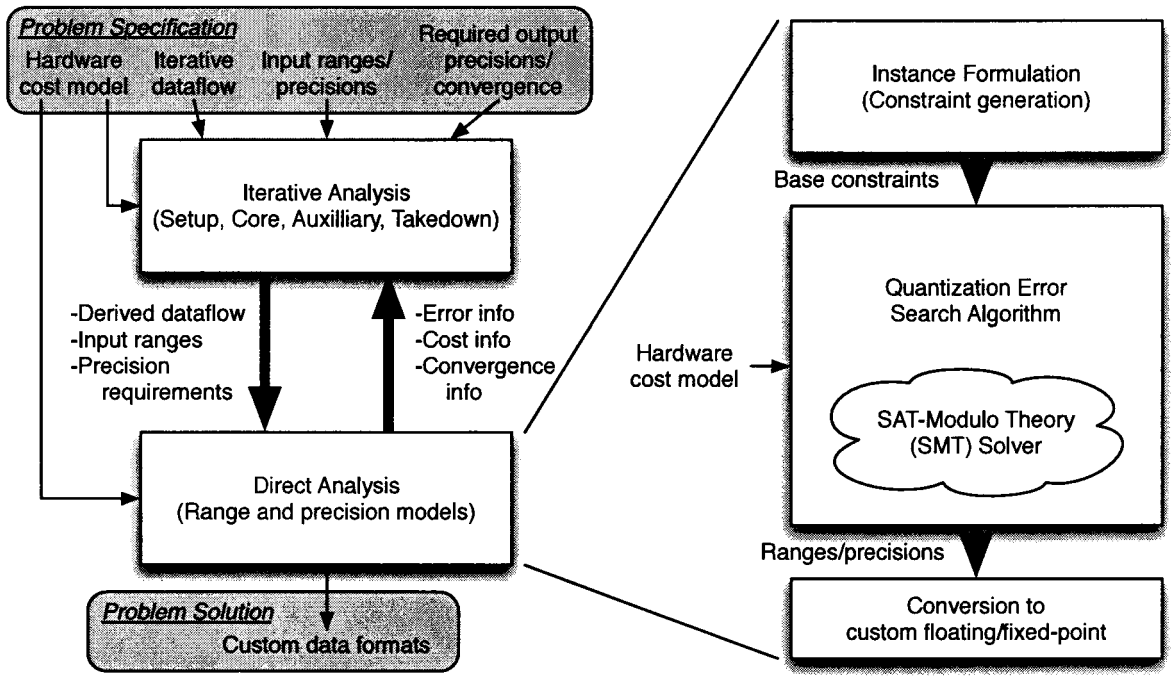


Figure 5.5: Conceptual flow for solving the bit-width allocation problem for iterative numerical calculations.

provide detailed convergence information on iterative algorithms give a testament to the complexity of this problem and the lack of a one-size-fits all solution, we are not completely without recourse. In particular, one of the best aides that can be provided in our context of bit-width allocation is a means of extracting constraints which encode information specific to the convergence/error properties of the algorithm under analysis.

A simplistic approach to analysis of the iterative portion would be to merely "unroll" the iteration, as shown in Figure 5.6. A set of independent solver variables and dataflow constraints are created for the intermediates of each iteration with the output variables of one iteration fed into the input of the next. While this approach is attractive in terms of both automation (replication of iteration dataflow is easy) and of capturing the data correlations across iterations, the resulting instance is very large. As a result, solver run-times can

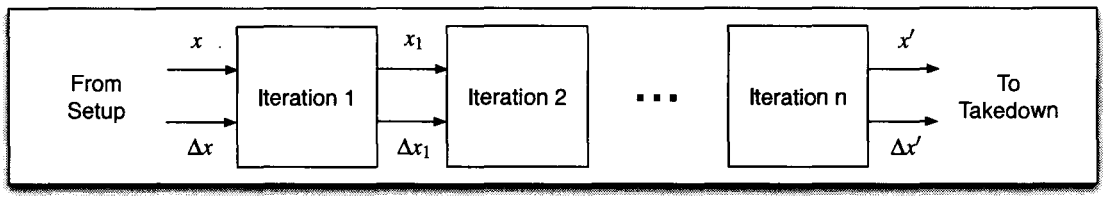


Figure 5.6: Iterative analysis by iteration unrolling.

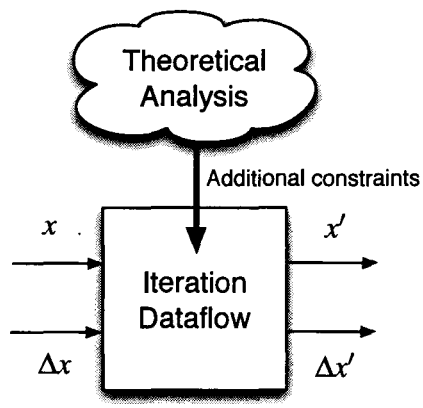


Figure 5.7: Iterative analysis using information from theoretical analysis.

explode, providing in reality very little meaningful feedback due to solver timeouts (Section 3.3.2). Furthermore, termination may be conditional leading to a variable number of iterations. This can resist being captured into constraints and increase instance complexity.

For the reasons above, it is preferred to have an instance for the iterative part based on the dataflow of a single iteration, augmented with some extra constraints as shown in Figure 5.7. The challenge then becomes determining those constraints which will ensure desired behaviour over the entire iterative phase of the numerical calculation. Note also that very large instances of direct calculations could be partitioned into more manageable sub-calculations under a similar procedure. As alluded to earlier, a rich source of information for such constraints is the theoretical analysis of the algorithm of interest. While some general facts surrounding iterative methods do exist (e.g., Banach fixed point theorem

[58]), the guidance provided by such facts to the SMT search is limited by their generality. In addition, even these general facts may tie in algorithm specific details (e.g., the definition of the metric of the space for the Banach fixed point theorem). Finally, the algorithm designer/analyst ought (in general) to have greater insight into the subtleties of the algorithm than the algorithm user, including assumptions under which it operates correctly.

To summarize the iterative analysis process, consider that application of SMT is essentially done to automate and accelerate reasoning about the algorithm. If any reasoning done offline by a person with deeper intuition about the algorithm can be captured into constraints, it will save the solver from trying to derive that reasoning independently (if even it could). In light of the application specific nature of this analysis process, it is best illustrated through example as done in Section 5.2. Before that however, direct analysis is described, since it is at the core of the method with *Setup*, *Takedown* and the result of the iterative analysis phase all being managed through direct analysis.

5.1.5 Direct calculation precision

Having shown that an iterative algorithm can be broken down into separate pieces to be analyzed which are direct calculations, here we discuss in more detail this direct analysis which is depicted in the right half of Figure 5.5, which consists of three main stages. The constraints for the base formulation come from the dataflow and its precision counterpart involving δ terms, formed as discussed in Section 5.1.2. This base formulation includes constraints for known bounds on input error and requirements on output error. This feeds into the core of the analysis which is the error search across the δ terms (discussed immediately below), which produces in the end range limits (as in Chapter 3) for each intermediate, as well as for each δ term indicating quantization points within the dataflow which map directly onto custom representations. Table 5.2 shows robust conversions from ranges to bit-widths, with I, E, F, M and s representing integer, exponent, fraction, mantissa and sign respectively. In each case, the sign bit is 0 if x_L and x_H have the same sign, 1 otherwise. Note that vector-magnitude bounds must first be converted to element-wise bounds before applying the rules in the table, using $||\mathbf{x}||_2 < X \rightarrow -X < [\mathbf{x}]_i < X$, and $||\delta\mathbf{x}||_2 < X \rightarrow -X/\sqrt{N} < [\delta\mathbf{x}]_i < X/\sqrt{N}$ for vectors with N elements.

Table 5.2: Converting ranges to bit-widths for fixed and custom floating types.

Type	Constraints	Bits Required	Total
Fixed	$x_L < \bar{x} < x_H$ $\delta x_L < \delta x < \delta x_H$	$I = \lceil log_2 \lceil max(\lvert x_L \rvert, \lvert x_H \rvert) + 1 \rceil \rceil$ $F = -\lfloor log_2 \lceil min(\lvert \delta x_L \rvert, \lvert \delta x_H \rvert) \rceil \rfloor$	s + I + F
Float	$x_L < \bar{x} < x_H$ $\delta x \leq max \begin{pmatrix} slope_x \lvert x \rvert, \\ knee_x \end{pmatrix}$	$M = -\lfloor log_2 (slope_x) \rfloor$ $E_H = \lceil log_2 \lceil max(\lvert x_L \rvert, \lvert x_H \rvert) \rceil \rceil$ $E_L = \lfloor log_2 knee_x \rfloor + M - 1$ $E = \lceil log_2 (E_H - E_L + 1) \rceil$	s + M + E

With the base constraints set up as discussed above forming an instance for the solver, execution of the solver becomes the means of evaluating a chosen quantization scheme. The error search that forms the core of the direct analysis utilizes this error bound engine by plugging into the *knee* and *slope* values for each δ constraint. Selections for *knee* and *slope* are made based on feedback from both the hardware cost model and the error bounds obtained from the solver. In this way, the quantization choices made as the search runs are meant to evolve toward a quantization scheme which satisfies the desired end error/cost tradeoff. With the machinery now in place to move from specification to custom data types, application of the method to some case studies is presented next.

5.2 Case studies

As a means both of further illustrating the method as well as evaluating its effectiveness, a few scenarios have been explored: the two-operand scalar addition from the relative error example of Section 5.1.1, an iterative Newton-Raphson division scheme and Newton's method root finding scheme, with comparative analysis having shown throughout the thesis that existing techniques cannot support division. After these, in Section 5.3, a case study based on the Conjugate Gradient method is presented. The platform for the experiments used the same SMT solver implementation as in the previous two chapters (HySAT [35, 96]), supplemented with a custom developed solver. The machine used was a dual core

3.00GHz Intel Pentium IV with 1GB of RAM running Ubuntu Linux 9.10. Execution times ranged up to low 10's of minutes except for the Conjugate Gradient example where execution times were in the 10's of hours.

As per discussion in Section 2.2.1, the hardware cost model and search procedure are treated as aspects which are separate from error bounding, the focus of this thesis. Regardless, in order to evaluate the method a search method must be in place. As such, we have employed a simple metric where cost is directly proportional to number of bits. Based on this metric, a greedy search proceeds as follows:

1. For each quantized intermediate x, set $slope_x = 0$, $knee_x = KTEST_x$ and set *slope* and *knee* to zero for all other intermediates. Using $KTEST_x = 0$ as a starting point, adjust $KTEST_x$ until the output error requirements for the calculation are just met. This establishes inner (minimum resource) bounds on the *knee* for each intermediate.

2. Repeat the above step, but for $knee_x = 0$ and $slope_x = STEST_x$ which will establish inner bounds on the *slope* for each intermediate.

3. Using for each variable $knee = KTEST$ and $slope = STEST$ determined in the last two steps as a starting point, check if the output error is within the tolerance afforded by the specification, if so terminate.

4. Check for each variable the effect on output error of reducing *knee* by a factor of 2 and of reducing *slope* by a factor of 2. Adopt the choice which reduces the error the most and go to the previous step.

The inner bounds established through steps 1, and 2 above are actually upper bounds on the respective *knee* and *slope* values, which produce minimum resources, and *knee* can also be used as the minimum number of fraction bits required for fixed-point representations. The intuition behind the greedy search is to try to find the shortest path to bring the output error inside of the tolerance requirements of the specification. Furthermore, if error tolerance violation is occurring solely because of a variable's error behaviour in the region below the knee point (absolute error behaviour), adjusting slope should not affect it. The reverse is also true. In addition, if error from one variable is so large that it overwhelms error from other variables, more precision will be allocated to that variable.

It should be noted the overall hardware result depends heavily on the suitability of the hardware cost models and associated search procedure, and how well coupled they are. As discussed in Chapter 2, one of the motivations of the abstraction between these aspects and the estimation of error bounds is how much they vary based on the implementation technology. It has been stressed that regardless of the choice of cost model and search, they cannot be effective without error bounds which are tight enough to be informative, such has been the motivation of this thesis. In light of this, the main purpose of the above (potentially inefficient) search and metric is to provide the support necessary to highlight the capacity of SMT based error bounding to deal with scientific calculations. With that in mind, we turn to the case studies.

5.2.1 Two operand addition

The first and simplest case study is two operand addition with a relative error constraint, following the setup from the relative error example of Section 5.1.1. The calculation is direct: $a+b=c$ between two floating-point numbers $a,b \in [10,100]$, and quantization is applied to the inputs a,b as well as the result c. For output error constraints of $slope_c \leq 1\%$ and $knee_c \leq 0.1$ the tool returns $slope_a = slope_b = 1.25 \times 10^{-3}, knee_a = knee_b = 1.25 \times 10^{-2}$ and $slope_c = 2.5 \times 10^{-3}, knee_c = 5.0 \times 10^{-2}$. These translate into $s = 1$ sign $+ m = 10$ mantissa $+ e = 4$ exponent bits for a and b, and $s = 1$ sign $+ m = 9$ mantissa $+ e = 4$ exponent bits for c. Because the resulting range of $c \in [20,200]$ does not include $knee_c$, relative error limits are guaranteed over the entire range.

Note that the example from Section 5.1.1 upon which this case is based provides (for the sake of clarity in the example) only forward relative error analysis without considering quantization effects. Since quantization injects additional error into the calculation, the accumulation of those errors must amount in the end to less than the tolerance. This is the reason for 10 mantissa bits for a and b and 9 for c which give slope less than the 1% required by the specification. Furthermore, because no rounding mode is specified, no assumption is made in the tool to retain robustness and as a result, slopes from output error tolerance specifications and those reported back from quantization points are adjusted down by a factor of 2, to ensure that any quantization mode that might be chosen supports the error

tolerance requirements.

Moving to the second part of the experiment where the input ranges are changed to $a,b \in [-100,100]$ the effect of the catastrophic cancellation highlighted in Section 5.1.1 can be seen immediately. The slope values returned by the tool indicate smaller errors at the extremes of the ranges than the knee values, which themselves depend on the knee value constraint imposed externally on c. In essence, the only error which can be guaranteed in this case (since values could cancel to zero) is absolute error, which is controlled by the knee value. For a constraint of $knee_c \leq 0.1$ (regardless of the $slope_c$ constraint) the tool indicates a fixed-point representation with 7 fraction bits for a,b and 6 fraction bits for c. If the $knee_c$ constraint is reduced four-fold, 2 more bits are required on each of a,b and c. The selection of fixed-point by the tool for this scenario indicates that it is aware (through the formalism of the SMT instance) of the catastrophic cancelation effect and is indeed generating robust bit-widths, as that is the only way to ensure the required precision on the output over the entire range.

5.2.2 Newton-Raphson division

The previous example serves to clearly demonstrate the robustness of this bit-width allocation method, as well as the concepts of slope and knee, and the necessary transition from floating to fixed-point under increasingly tight error tolerances, while being simple enough to evaluate intuitively. However, for this same reason it is of little practical significance. In this section, the effectiveness of the method as applied to iterative calculations is illustrated through a case study based on *Newton-Raphson division.*

Newton-Raphson division enables the calculation of a quotient $Q = N/D$, where $N,Q \in \mathbb{R}, D \in \mathbb{R}, D \neq 0$. The quotient is calculated iteratively without using a division explicitly, instead only subtraction and multiplication. It is useful for hardware implementations where it is undesirable to allocate resources for a standalone divider.

In actuality, the problem is broken into $2^e N \times (1/2^e D)$ where Newton's method is used to obtain the reciprocal of $2^e D$ and $e \in \mathbb{I}$ is selected so that $1 < 2^e D \leq 2$ where convergence of Newton's method can be effectively predicted. The function $f(X) = 1/X - 2^e D$ having a root at $1/(2^e D)$ is selected for the iteration, and applying Newton's method gives the

iteration $X_{k+1} = X_k(2 - DX_k)$. To summarize the procedure:

1. Shift D,N by e so that $1 < 2^e D \leq 2$
2. Starting from $X_0 = 0.5$,
3. Iterate $X_{k+1} = X_k(2 - DX_k)$ until $X_{k+1} - X_k < \varepsilon$
4. Obtain $Q = NX_{k+1}$

where ε is roughly the desired relative error of the result. As will be elaborated below, this example also serves as useful for illustrating the different components of the iterative flow from Figure 5.4.

Steps 1 and 2 above constitute the *Setup calculation* calculation portion of the iterative method (from Figure 5.4), while the first half of step 3 is the *Core iteration* and step 4 is the *Takedown calculation*. The second half of step 3 is the termination or *Done* condition, and there is no *Auxiliary Iteration* as the entire iteration is required to decide convergence. In terms of architecture, a potential hardware implementation would involve a priority encoder structure for step 1, and one or two multipliers would be required for step 3 depending on whether sharing is permitted, for DX_k and the subtraction result multiplied by X_k. A subtractor would also be required in step 3 (for $2 - DX_k$), and the convergence test also requires a subtractor (which can be dedicated or shared with step 3) plus comparison with a constant. Finally step 4 requires a multiplier which can be dedicated, or again shared with the one from step 3.

The iterative analysis phase for the algorithm is in essence done implicitly, subtly indicated by the statement of $1 < 2^e D \leq 2$ *where convergence of Newton's method can be effectively predicted.* Theoretical analysis of Newton's method using $f(X) = 1/X - D$ having a root at $X = D$ yields the following facts:

- f has unique root at $X = D$,
- f has non-zero derivative at $X = D$,
- f is continuously differentiable around, $X = D$,
- f has a second derivative at $X = D$.

The above conditions are sufficient to guarantee quadratic convergence by which we approximate the error after k iterations as 2^{-2k}. Furthermore, the sequence of iterations will converge monotonically to the result. With these parts of the iterative analysis in place, we can setup and perform the direct analyses.

Let the inputs N and D to the algorithm have range $[1 \times 10^{-3}, 1 \times 10^{3}]$, with no error in N and D. Also let there be a tolerance constraint of $slope_{\Delta Q} \leq 1 \times 10^{-3}$ and $knee_{\Delta Q} \leq 1 \times 10^{-6}$. Note that this $knee_{\Delta Q}$ constraint shifts the knee point out of the range of Q; thus the error is controlled by relative error over the entire range of Q.

The setup calculation is trivial since X_0 is fixed, D is assumed to be without error and is shifted to between 1.0 and 2.0. The *Takedown Calculation* however is a little more involved and is used to backward propagate the output error constraint. The *Takedown Calculation* constraints are as follows.

$$Q = NX$$
$$0.5 \leq X < 1$$
$$1 \times 10^{-3} \leq N \leq 1 \times 10^{3}$$
$$\Delta Q = (\Delta N)X + N(\Delta X) + (\Delta N)(\Delta X) + \delta Q$$
$$\Delta N = 0$$
$$abs(\Delta X) \leq max(slope_X abs(X), knee_X)$$
$$abs(\Delta Q) \leq max((1 \times 10^{-3}) abs(Q), 1 \times 10^{-6})$$
$$abs(\delta Q) \leq max(slope_{\delta Q} abs(Q), knee_{\delta Q})$$

The multiplication implies to X the relative error constraint of Q (plus a couple extra bits due to rounding mode robustness as in the addition case study), but we know that the range of X is limited to $0.5 \leq X < 1$ so that the relative error constraint can be replaced with the same absolute error giving $\Delta X \leq 1.25 \times 10^{-4}$ (13 fraction bits).

Based on the iterative analysis above, the convergence is stronger than the error requirement for the final iteration as derived above. As the iterations proceed, we can consider that X_k is assigned $\overline{X_{k-1}}$, and because there is only a single iterated variable, no divergence between iterated variables occurs. As a result, we can assume $\Delta X = 0$ in our instance and due to the quadratic error convergence we know that $\varepsilon = 6.10 \times 10^{-5}$, and $\Delta X' < 6.10 \times 10^{-5}$

meets the error criteria. The constraints are below:

$$Y = 2 - DX$$
$$X' = XY$$
$$0.5 \leq X < 1$$
$$0.5 \leq X' < 1$$
$$\Delta Y = -(\Delta D)X - D(\Delta X) - (\Delta D)(\Delta X) + \delta Y$$
$$\Delta X' = -(\Delta X)Y + X(\Delta Y) + (\Delta X)(\Delta Y) + \delta X'$$
$$\Delta X = 0$$
$$\Delta X' < 6.10 \times 10^{-5}$$

Fixed-point data types result due to the limited dynamic range of the variables involved. One way of thinking of this is that the exponent for shifting the floating-point type which would guarantee relative error containment is implicitly encoded in the shift which happens before entering the iterative phase, to bring D into the range [1.0,2.0].

5.2.3 Newton's method root finding

While the last two examples have been sufficiently small to argue that exhaustive simulation or existing analytical methods are equal to the task, this example again based on Newton's method defies both methods. It is in fact the same example taken from Chapter 3, in Section 3.4.6 - where it was addressed only for the range problem across a single iteration. Here on the other hand, the full method is employed to determine the root of a polynomial: $f(x) = a_3x^3 + a_2x^2 + a_1x + a_0$. The Newton iteration:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

yields the iteration:

$$x_{n+1} = x_n - z_n, z_n = \frac{z1_n}{z2_n} = \frac{a_3x_n^3 + a_2x_n^2 + a_1x_n + a_0}{3a_3x_n^2 + 2a_2x_n + a_1}.$$

Such calculations can arise in embedded systems for control when some characteristic of the system is modelled regressively using a 3^{rd} degree polynomial. Ranges which are used

in Section 3.4.6 are given below:

$$x_0 \in [-100, 100] \quad a_0 \in [-10, 10]$$

$$a_1 \in [\tfrac{15}{2}, \tfrac{17}{2}] \quad a_2 \in [\tfrac{-15}{4}, \tfrac{-13}{4}] \quad a_3 \in [\tfrac{5}{6}, \tfrac{7}{6}]$$

It has been emphasized that in the context of scientific computing, robust representations are required, and this may be an important criterion for many embedded systems applications which have a scientific motivation. For a problem even of modest size such as this one, if the range of each variable is split into 100 parts, the input space for simulation based methods would be $100^5 = 10^{10}$. This immense simulation may have to be performed multiple times to evaluate error ramifications of each quantization choice. On top of this, a simulation of this magnitude is still not truly exhaustive and thus cannot substantiate robustness. Together these facts clearly invalidate the use of simulation based methods. As we have also seen in Section 3.4.6, existing formal methods based on interval arithmetic and affine arithmetic also fail to produce usable results due to the inclusion of 0 in the denominator of z_n whereas the computational technique provides tighter ranges.

This example bears some similarity to the previous one in that it is iterative, relying on Newton's method. However, while the previous example did not require any divisions as a part of the dataflow (in fact its purpose was to implement division *without* using a divider), this method exemplifies some of the core challenges which scientific calculations invoke - potential singularities and ill-conditioning, which are the stumbling blocks for existing analytical bit-width allocation methods [33, 80, 99, 106] when applied to numerical methods (see Section 2.2.2).

The setup for this experiment in terms of input variable ranges is given above, and perfect representation of the inputs is assumed (i.e., $\Delta x_0 = \Delta a_j = 0$). Furthermore, there is no *Setup, Takedown* or *Auxiliary Iteration*, only the *Core Iteration*. Setting the tolerance limits for iteration output x_{n+1} as 0.01 for the slope and 0.001 for the knee, the tool returned fixed-point bit-widths for the intermediates in the numerator and denominator on 16 to 20 bits fractional part, but floating-point types for $z1$ and $z2$ themselves. This is interesting because the dynamic range of these variables is larger due to the 3^{rd} degree polynomials, while the division between them (for which relative error representation is convenient) is contracting, bringing the dynamic range back down to a span suitable for fixed-point

implementation. The ability of the computational based SMT to provide meaningful bit-widths for this iterative numerical algorithm demonstrates the potential SMT has to enable robust data representations for iterative scientific algorithms.

5.3 Conjugate Gradient case study

As a final example to tie together concepts from throughout this thesis, this example analyzes the Conjugate Gradient method [113] for solving linear systems of equations. While as in the previous example the Conjugate Gradient method was treated earlier, it was again only for ranges over a single iteration, and for only a 3×3 matrix in Section 4.2.4. In this section however, its (Jacobi) pre-conditioned extension is addressed in an environment of much greater practical significance and complexity - an application for haptic interaction with deformable bodies as presented in [81, 82].

5.3.1 Summary of the application

In order to provide the context of the application and to properly understand the formal results presented here, the work in [82] is briefly overviewed. In the application, a deformable body is modelled using the finite element method (FEM), and a large linear system can be derived from localized force-distance equations over this FEM model, which the Conjugate Gradient method is in turn applied to solve. In order to provide a sufficiently true-to-reality simulation with their setup, the FEM mesh consists of 1000's of nodes, generating vectors with length on the order of 10^3 to 10^4. The corresponding matrix is sparse and the haptic feedback system imposes a time constraint requiring solution of this large sparse system within 1-2ms.

To address this significant computational demand imposed by the above conditions, a hardware accelerator using a custom-floating-point numerical representations called *dynamically scaled fixed-point* was developed by the authors of [82]. This was done using an empirical approach wherein adequate limits on the precision requirements were estimated using Monte Carlo simulations. A reproduction of the algorithm and required bit-widths from [82] using their notation are provided in Algorithm 5.1 and Table 5.3 respectively.

Algorithm 5.1 Preconditioned Conjugate Gradient from [82]

1: $\mathbf{u} = \mathbf{init}$;
2: $\mathbf{r} = \mathbf{b} - \mathbf{Ku}$;
3: $\mathbf{z} = \mathbf{P}^{-1}\mathbf{r}$;
4: $\mathbf{d} = \mathbf{z}$;
5: $cntr = 1$;
6: $zr = \mathbf{z}^{\mathbf{T}}\mathbf{r}$;
7: **while** $(cntr < \#m)$ **do**
8: $\quad \alpha = zr/(\mathbf{d}^{\mathbf{T}}\mathbf{Kd})$;
9: $\quad \mathbf{u} = \mathbf{u} + \alpha\mathbf{d}$;
10: $\quad \mathbf{rn} = \mathbf{r} - \alpha\mathbf{Kd}$;
11: $\quad \mathbf{zn} = \mathbf{P}^{-1}\mathbf{rn}$;
12: $\quad zrn = \mathbf{zn}^{\mathbf{T}}\mathbf{rn}$;
13: $\quad \beta = zrn/zr$;
14: $\quad \mathbf{d} = \mathbf{zn} + \beta\mathbf{d}$;
15: $\quad \mathbf{r} = \mathbf{rn}$;
16: $\quad \mathbf{z} = \mathbf{zn}$;
17: $\quad zr = zrn$;
18: $\quad cntr = cntr + 1$;
19: **end while**
20: **return** $\mathbf{u}$;

Table 5.3: Required bit-widths for Algorithm 5.1 as determined by [82]

Vectors		Matrix	Scalars	
$\mathbf{b,r,u,Kd}$	$\mathbf{d,z,P^{-1}}$	$[\mathbf{K}]_{ij} \neq 0$	$\mathbf{d^T Kd, r^T r}$	α, β
36-bit	18-bit	18-bit	64-bit	18-bit

Case 2 of the Monte Carlo experiment from [82] (Section III-B) involved a mesh with 1144 points, creating a matrix $\mathbf{K}$ of 3432×3432 with condition number ($\kappa = \lambda_{max}/\lambda_{min}$) 8899. The experiment consisted of simulation of 50 random initial points (**init**) with a deviation of 10% from the actual solution, as justified by the haptic environment. Over the 50 simulations the normalized error of the solution vector was recorded, which was calculated as:

$$err_{\mathbf{u}} = \frac{||\mathbf{u} - \mathbf{u}_{true}||}{||\mathbf{u}_{true}||}$$

where $\mathbf{u}_{true}$ is obtained using double precision solution to the system of equations. For each of the 50 simulations, 25 Conjugate Gradient iterations were applied and the resultant value of $err_{\mathbf{u}}$ was on average ≈ 0.004 and in the worst case ≈ 0.009, with standard deviation ≈ 0.002. In addition, deviation from the result obtained from an IEEE-754 double precision floating-point implementation was negligible over the simulations. Due to these errors falling well within acceptable tolerances for maintaining the reality of the application, as well as the convenience of fixed latency calculation in a real-time system, [82] indicates the number of iterations $\#m$ from Algorithm 5.1 can be fixed at 20. While this analysis has provided relatively compelling motivation for the choice of bit-widths used in the functioning prototype system, the next section provides a formal analysis, revealing the non-robustness of the empirical approaches.

5.3.2 Formal analysis and robust representations

Applying the methodology presented thus and shown in Figure 5.5, and restricting our analysis to linear system based on a constant matrix $\mathbf{K}$ (as in [82]), the algorithm can

be partitioned into *Setup, Takedown, Core Iteration* and *Auxiliary Iteration* of Figure 5.4 (Section 5.1.3). While the *Takedown Calculation* is empty (given that the iteration directly produces the solution vector $\mathbf{u}$), lines 1 to 6 constitute the setup phase. For the iterative part, we will depart from the fixed number of iterations used by [82] to use convergence criteria from [113] of $zrn < \varepsilon^2 zr_0$, where zr_0 is zr at line 6. Under this condition, line 9 becomes the *Auxiliary Iteration* (because it is not needed to decide convergence) and all other lines from 7 to 19 constitute the *Core Iteration*, which will be the focus of the discussion for the remainder of this section.

Direct dataflow

Before setting up a direct calculation instance for analyzing the *Core Iteration*, we must determine constraints for its operation. The simplest first step is to write value and precision constraints based on Algorithm 5.1 as well as the discussion on precision expressions from Section 5.1.2 for the iterative part of the dataflow as follows:

$$
\begin{aligned}
\mathbf{q} &= K\mathbf{d} & \Delta\mathbf{q} &= K(\Delta\mathbf{d}) + \delta\mathbf{q} \\
dKd &= \mathbf{d}^{\mathbf{T}}\mathbf{q} & \Delta dKd &= (\Delta\mathbf{d})^{\mathbf{T}}\mathbf{q} + \mathbf{d}^{\mathbf{T}}(\Delta\mathbf{q}) + (\Delta\mathbf{d})^{\mathbf{T}}(\Delta\mathbf{q}) + \delta dKd \\
\alpha &= \frac{zr}{dKd} & \Delta\alpha &= \frac{(\Delta zr) - \alpha(\Delta dKd)}{dKd + (\Delta dKd)} + \delta\alpha \\
\mathbf{un} &= \mathbf{u} + \alpha\mathbf{d} & \Delta\mathbf{un} &= \Delta\mathbf{u} + (\Delta\alpha)\mathbf{d} + \alpha(\Delta\mathbf{d}) + (\Delta\alpha)(\Delta\mathbf{d}) + \delta\mathbf{un} \\
\mathbf{rn} &= \mathbf{r} - \alpha\mathbf{q} & \Delta\mathbf{rn} &= \Delta\mathbf{r} - (\Delta\alpha)\mathbf{q} - \alpha(\Delta\mathbf{q}) - (\Delta\alpha)(\Delta\mathbf{q}) + \delta\mathbf{rn} \\
\mathbf{zn} &= P^{-1}\mathbf{rn} & \Delta\mathbf{zn} &= P^{-1}(\Delta\mathbf{rn}) + \delta\mathbf{zn} \\
zrn &= \mathbf{zn}^{\mathbf{T}}\mathbf{rn} & \Delta zrn &= (\Delta\mathbf{zn})^{\mathbf{T}}\mathbf{rn} + \mathbf{zn}^{\mathbf{T}}(\Delta\mathbf{rn}) + (\Delta\mathbf{zn})^{\mathbf{T}}(\Delta\mathbf{rn}) + \delta zrn \\
\beta &= \frac{zrn}{zr} & \Delta\beta &= \frac{(\Delta zrn) - \beta(\Delta zr)}{zr + (\Delta zr)} + \delta\beta \\
\mathbf{dn} &= \mathbf{zn} + \beta\mathbf{d} & \Delta\mathbf{dn} &= \Delta\mathbf{zn} + (\Delta\beta)\mathbf{d} + \beta(\Delta\mathbf{d}) + (\Delta\beta)(\Delta\mathbf{d}) + \delta\mathbf{dn}
\end{aligned}
$$

This dataflow represents one iteration on the iterated variables $zr, \mathbf{d}, \mathbf{r}, \mathbf{u}$ and their accompanying finite precision errors $\Delta zr, \Delta\mathbf{d}, \Delta\mathbf{r}, \Delta\mathbf{u}$, under the assumption (as in [82]) that the matrices K and P are exactly represented in their finite precision forms (which we have adopted directly). In order to ensure that upon algorithm completion, the result is correct (within the tolerances prescribed by the problem specification), conditions are required. It

would also be beneficial to solver instance complexity if the scope of these conditions was limited to a single iteration. Such constraints arise from theoretical iterative analysis and are added to those above.

Iterative analysis

Along these lines, we can set up a target number of iterations based on convergence analysis. Under the setup of [82] the deviation in $\mathbf{u}$ (i.e. error term $\mathbf{e}_{(0)}$) at the onset is (rightly) considered safely bounded by 10% of the entire space of $\mathbf{u}$, and suppose we want to ensure that an execution of the algorithm will reduce it by $100\times$ to 0.1% of the space. Using the same matrix from [82] (case 2 of Section III-B) with condition number $\kappa = 8.873\text{e}3$, and using the convergence analysis provided by [113] we can have:

$$\frac{||\mathbf{e}_{(\mathbf{i})}||_K}{||\mathbf{e}_{(\mathbf{0})}||_K} \leq 2\left[\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^i + \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i\right]^{-1} \leq 0.01$$

which can be solved numerically to obtain $i \approx 250$.

With a target iteration count in hand we can establish necessary conditions on a single iteration to limit loss of precision across algorithm iterations during its operation. The derivation of the Conjugate Gradient method proceeds under the assumption that in any iteration (i), the residual is an exact image of the current solution vector over the transform defined by the matrix i.e.,

$$\mathbf{b} - K\mathbf{u}_{(i)} = \mathbf{r}_{(i)}$$

which is an invariant across the iterations, and is in fact what guarantees that when the algorithm terminates, the $\mathbf{u}$ which results is indeed the solution to the problem. In finite precision however, this relationship is not guaranteed to hold but instead we have:

$$||\overline{\mathbf{b}} - \overline{K}\overline{\mathbf{u}}_{(i)} - \overline{\mathbf{r}}_{(i)}|| < \varepsilon$$

and to examine one iteration we can derive:

$$\begin{aligned} ||(\overline{\mathbf{b}} - \overline{K}\overline{\mathbf{u}}_{(i)} - \overline{\mathbf{r}}_{(i)}) - (\overline{\mathbf{b}} - \overline{K}\overline{\mathbf{u}}_{(i+1)} - \overline{\mathbf{r}}_{(i+1)})|| &< \varepsilon_{(i)} \\ ||\overline{\mathbf{b}} - \overline{K}\overline{\mathbf{u}}_{(i)} - \overline{\mathbf{r}}_{(i)} - \overline{\mathbf{b}} + \overline{K}\overline{\mathbf{u}}_{(i+1)} + \overline{\mathbf{r}}_{(i+1)}|| &< \varepsilon_{(i)} \\ ||\overline{K}\overline{\mathbf{u}}_{(i+1)} - \overline{K}\overline{\mathbf{u}}_{(i)} + \overline{\mathbf{r}}_{(i+1)} - \overline{\mathbf{r}}_{(i)}|| &< \varepsilon_{(i)} \end{aligned}$$

and substituting the quantized variables with precision expressions:

$$\begin{aligned} ||(K+\Delta K)\left[(\mathbf{u}_{(i+1)}+\Delta\mathbf{u}_{(i+1)})-(\mathbf{u}_{(i)}+\Delta\mathbf{u}_{(i)})\right]+(\mathbf{r}_{(i+1)}+\Delta\mathbf{r}_{(i+1)})-(\mathbf{r}_{(i)}+\Delta\mathbf{r}_{(i)})|| &< \varepsilon_{(i)} \\ ||(K+\Delta K)\left[(\mathbf{u}_{(i+1)}-\mathbf{u}_{(i)})+(\Delta\mathbf{u}_{(i+1)}-\Delta\mathbf{u}_{(i)})\right]+(\mathbf{r}_{(i+1)}-\mathbf{r}_{(i)})+(\Delta\mathbf{r}_{(i+1)}-\Delta\mathbf{r}_{(i)})|| &< \varepsilon_{(i)} \end{aligned}$$

Proceeding under the assumption that the representation for matrix K is exact (as in [82]), $\Delta K = 0$. Examining the expression inside the norm on the left hand side and making substitutions from the precision dataflow:

$$\begin{aligned} K[(\mathbf{un}-\mathbf{u})+(\Delta\mathbf{un}-\Delta\mathbf{u})] &+ (\mathbf{rn}-\mathbf{r})+(\Delta\mathbf{rn}-\Delta\mathbf{r}) \\ K[((\mathbf{u}+\alpha\mathbf{d})-\mathbf{u})+(\Delta\mathbf{un}-\Delta\mathbf{u})] &+ ((\mathbf{r}-\alpha\mathbf{q})-\mathbf{r})+(\Delta\mathbf{rn}-\Delta\mathbf{r}) \\ \begin{pmatrix} K\alpha\mathbf{d}+K(\Delta\alpha)\mathbf{d}+K\alpha(\Delta\mathbf{d}) \\ +K(\Delta\alpha)(\Delta\mathbf{d})+K\delta\mathbf{un} \end{pmatrix} &+ \begin{pmatrix} -\alpha\mathbf{q}-(\Delta\alpha)\mathbf{q}-\alpha(\Delta\mathbf{q}) \\ -(\Delta\alpha)(\Delta\mathbf{q})+\delta\mathbf{rn} \end{pmatrix} \\ \begin{pmatrix} K\alpha\mathbf{d}+K(\Delta\alpha)\mathbf{d}+K\alpha(\Delta\mathbf{d}) \\ +K(\Delta\alpha)(\Delta\mathbf{d})+K\delta\mathbf{un} \end{pmatrix} &+ \begin{pmatrix} -\alpha K\mathbf{d}-(\Delta\alpha)K\mathbf{d}-\alpha(\Delta\mathbf{q}) \\ -(\Delta\alpha)(\Delta\mathbf{q})+\delta\mathbf{rn} \end{pmatrix} \\ \begin{pmatrix} K\alpha(\Delta\mathbf{d}) \\ +K(\Delta\alpha)(\Delta\mathbf{d})+K\delta\mathbf{un} \end{pmatrix} &+ \begin{pmatrix} -\alpha(K(\Delta\mathbf{d})+\delta\mathbf{q}) \\ -(\Delta\alpha)(K(\Delta\mathbf{d})+\delta\mathbf{q})+\delta\mathbf{rn} \end{pmatrix}. \end{aligned}$$

Recombining with the right hand side now,

$$\begin{aligned} ||K\delta\mathbf{un}-\alpha\delta\mathbf{q}-(\Delta\alpha)\delta\mathbf{q}+\delta\mathbf{rn}|| &\leq \varepsilon_{(i)} \\ ||K\delta\mathbf{un}-(\alpha+\Delta\alpha)\delta\mathbf{q}+\delta\mathbf{rn}|| &\leq \varepsilon_{(i)} \\ ||K\delta\mathbf{un}-\overline{\alpha}\delta\mathbf{q}+\delta\mathbf{rn}|| &\leq \varepsilon_{(i)} \end{aligned}$$

Examining this expression in the context of Conjugate Gradient, notice that the quantization choice for only the solution vector (**un**) and the residual (**rn**) as well as the matrix multiplication (**q**) affect the invariant. Intuitively this makes sense because **d** and α define a search direction and step size, and regardless of the choice of them, so long as both **u** and **r** are updated in a way consistent with each other, the invariant will hold.

Since some violation of the invariant will occur for finite precision, we seek to bound it using the target iterations determined above. Transforming the derived expression into space of the solution vector gives $||\delta\mathbf{un}-\overline{\alpha}K^{-1}\delta\mathbf{q}+K^{-1}\delta\mathbf{rn}|| \leq \varepsilon_{\mathbf{u}}$ which gives a constraint on how far the invariant can be broken as seen from **u**'s point of view. If we consider

that after the targeted 250 iterations we desire magnitude of deviation (absolute) between **u** and **r** of less than 0.01, then $\varepsilon_{\mathbf{u}} \leq 0.01/250 = 4\text{e-}5$ will meet this objective. Therefore the constraint which we add to our formulation to guarantee desired coherence between **u** and **r** at the end of the algorithm is:

$$||K\delta\mathbf{un} - \overline{\alpha}\delta\mathbf{q} + \delta\mathbf{rn}|| \leq 4 \times 10^{-5}.$$

This expression has been fixed for a given number of iterations to ease the complexity of the instance and thereby not overburden the solver. In actuality, given a powerful enough solver the above could be skipped, leaving the target number of iterations as a free variable to be explored by the solver, constrained by the convergence formula from [113]. By doing this analysis symbolically offline, we save the solver having to independently learn this constraint (if it can), or in the worst case work it out from scratch each time it would be of use. It is worth noting also that this constraint is not unique, in that other constraints may equally satisfy the accuracy requirements imposed on **u**. What matters from the robustness point of view however is that they do in fact guarantee the desired error conditions for **u**.

In addition to ensuring that the **u** and **r** variables remain consistent, we need a condition which keeps the search from getting off track and negatively impacting convergence. Without more detail surrounding the necessary conditions for proper operation of the algorithm in the context of its application, it is difficult to form a clear constraint. As $\mathbf{d}, \mathbf{dn}$ are implemented by [82] using dynamic scaling, the are essentially custom floating-point in that application, and as a result we adopt a bound on relative error introduced in each iteration. This bound is the same as the bound on quantization which can be guaranteed by the use of 18 bits mantissa as in [82], $||\Delta\mathbf{dn}|| < (2.2 \times 10^{-4})||\mathbf{dn}||$. In this case we ignore the absolute error region since no behaviour is specified for it by the application.

Robust representations

Augmenting the dataflow constraints with the two constraints above of $||K\delta\mathbf{un} - \overline{\alpha}\delta\mathbf{q} + \delta\mathbf{rn}|| \leq 4 \times 10^{-5}$ and $||\Delta\mathbf{dn}|| < (2.2 \times 10^{-4})||\mathbf{dn}||$ provides the instance for direct analysis of the iterative part of the calculation. When full ranges based on quantizations from [82]:

$$||\mathbf{init}|| \in [0, 5.25 \times 10^5] \quad ||\mathbf{b}|| \in [0, 6.88 \times 10^{10}] \quad \lambda_{\mathbf{K}} \in [22.7, 2.02 \times 10^5]$$

Table 5.4: Bitwidths required for fixed-point intermediate variables.

Variable	Integer	Fraction	Sign	Total
u	0	36	1	37
r	5	39	1	45
un	0	36	1	37
rn	5	39	1	45

with the corresponding error inputs set to 0 (since these values are accepted in their quantized form) are used, the resulting bit-widths are extremely large, going into the hundreds of bits. Section 5.3.3 will discuss in detail the reasons behind this, which have to do in part with the behaviour of the calculation in some corners of the solution space. By constraining the input space more tightly, the required bit-widths can be reduced.

To constrain the input space, consider that the purpose of the *Setup* step is to use the initialization vector **init** to reduce the residual as much as possible before beginning. Therefore, instead of allowing **init** and **b** to occupy the entire space of possibilities then constrain the deviation from the true solution, **init**, we can symbolically remove the starting point, effectively solving $K(\mathbf{u} - \mathbf{init}) = (\mathbf{b_{new}} - \mathbf{b_{current}})$, and constrain $||\mathbf{u}||$. Using a more restricted range of motion within one haptic frame than above, the update to **u** is bounded by $||\mathbf{u}|| < 10^{-4}$. In this space, the residual will be bounded by $||\mathbf{r}|| < 3.2 \times 10^{1}$. Under the assumption that the algorithm reduces the residual in each iteration [113] and that the **u** in the problem is now the error, we can also write $||\mathbf{rn}|| < ||\mathbf{r}||$, and $||\mathbf{un}|| < ||\mathbf{u}||$. For the same reason we can consider that the updates do not go outside this space marked off by the constraints so the updates are also contained with it: $||\alpha\mathbf{q}|| < ||\mathbf{r}||$ and $||\alpha\mathbf{d}|| < ||\mathbf{u}||$.

In essence the above changes transform the problem to the origin of the **u** space which significantly reduces the dynamic range of the variables involved. This works in this setup since ideally in each haptic frame the solution to $K\mathbf{u} = \mathbf{b}$ should be known and linearity allows easy solution to a new system, very nearby due to the short timescale. Transformation back to the initial system involves only an addition. While in infinite precision arithmetic these two problems are identical, in finite precision the smaller dynamic ranges can affect bit-widths significantly. The bit-widths obtained for the intermediate variables under these

Table 5.5: Bitwidths required for floating-point intermediate variables.

Variable	Exponent	Mantissa	Sign	Total
zr	7	22	0	29
d	5	20	1	26
q	6	44	1	51
dKd	6	39	0	45
α	8	41	0	49
zn	6	25	1	32
β	3	22	0	25
zrn	7	22	0	29
dn	5	20	1	26

conditions are given in Tables 5.4 (fixed-point) and 5.5 (floating-point).

These assignments should guarantee numerically correct results over the operation of the datapath under the constraints which have been imposed, over the region of the input space which has been selected. Interestingly, the selection of fixed-point for **u**, **un** and **r**, **rn** makes sense for maintaining consistency between the solution and the residual as was discussed in Section 5.3.2. Also, the relative size of the mantissas for the other variables shows the rough degree of dependency which the output **dn** experiences on the respective quantization error. For instance, 22 bits for β indicates a weaker (or less amplified) impact on the error for **dn** than 44 bits for **q**. Another interesting fact about this strong dependence on **q** is that it influences **dn** through a subtraction in **rn** which comes through **zn** and β.

While the results make sense in terms of how they relate to the expected influence of intermediate variables upon the output error, they represent hardware requirements of nearly single precision floating-point in some cases, and between single and double precision floating-point for others. It is only fair to note that at the same time, these requirements are derived for a significantly restricted region of the input space where intuitively it would be expected to be feasible with significantly narrower bit-widths, and indeed [82] provides smaller bit-widths. The next section examines this issue in more depth and provides some perspective on these seemingly inflated bit-widths.

5.3.3 Perspective on formal and empirical findings

While the results presented in the previous section represent significantly higher resource requirements than those presented in [82], even at the same time as they address operation over a significantly smaller region of the input space than in [82], the primary difference between the two comes down to robustness.

To highlight this, recall that in Section 5.3.1 the empirical results were summarized to be $err_{\mathbf{u}} \approx 0.004$ on average and ≈ 0.009 with standard deviation ≈ 0.002 when 25 iterations were performed, using an init vector differing from the solution by less than 10% of its space. However, based on the theoretical analysis of the algorithm utilized for the iterative analysis phase, it is known that convergence is slowed when the vector **b** takes on a direction having equal weights of the eigenvectors of K. Taking a 10% deviation along a direction in **u** which produces such a **b** creates a scenario where the normalized error magnitude $err_{\mathbf{u}} = 0.083$ after the same 25 iterations. Note that this represents $\approx$35 standard deviations from the worst case, and $\approx$40 from the average, under the statistics generated from simulation. In excess of 100 iterations are required to bring the normalized residual magnitude to the same range identified through simulation. As the number of iterations increases, the residual obtained using custom representation deviates from that obtained using double precision floating-point so that after 225 iterations, the custom representation normalized residual magnitude is $\approx 1.73 \times 10^{5}$ and the double precision normalized residual magnitude is $\approx 2.71 \times 10^{-8}$.

Taking this case a step further, the effect can be exacerbated by observing that floating-point types can lose accuracy due to cancelation, when large numbers are subtracted. The principle extends to vectors so a large **init** vector which must be reduced down to the solution **u** will experience the most cancellation. In this case, after simulating 225 iterations, the custom representation $err_{\mathbf{u}}$ is 5.55×10^{4}, marking a significant departure from the double precision $err_{\mathbf{u}}$ which is 29.2. The reason for this is the increased absolute error due to cancellation of relative error (custom floating) variables, which accumulates over a number of iterations.

What may be even more surprising is that double precision too is not immune to the effect. Using the same K matrix, using the same direction considered above of an evenly

weighted sum of eigenvectors represented by $\mathbf{v}$, we can set up $\mathbf{b} = 10^{-10}\mathbf{v}$ and $\mathbf{init} = 10^{10}\mathbf{v}$ giving an initial residual 1.21×10^{16}. After 350 iterations, the residual as calculated by the iterations is $\approx 1.09 \times 10^{-1}$ while the true residual as calculated using the resulting $\mathbf{u}$ vector (i.e. $K\mathbf{u} - \mathbf{b}$) is 7.96. Despite the fact that all variables stay well within the representable ranges of double precision numbers over the course of the entire simulation, the algorithm deviates significantly from the true solution. This brings to light the important fact that while serving the general needs of numerical computation well, double precision itself is not a substitute for infinite precision, and there remains a need for error analysis even using this ample representation [61, 62].

The above representational counter examples serve as important reminders of the lack of robustness which simulation, as well as assumed high precision data types provide. It is worth noting that the sample size used in [82] already required significant simulation time on state of the art platforms, and even if the size were increased 10-fold, there would be no guarantee of detecting the corner cases discussed here which invalidate a potential choice of representation. Furthermore, even the very modest feasible region of the input which is under consideration in Section 5.3.2 consists of somewhere on the order of 2^{3000} points, clearly infeasible to be handled through simulation. At the same time however, no counter example such as the ones above can be brought for lower precision types than the ones indicated in Tables 5.4 and 5.5 within this restricted input space. Furthermore, double precision has been used extensively and reliably in scientific applications for quite some time, and a functioning system has been prototyped for [82] which has demonstrated correct operation. The natural question arises, how can balance be brought to this situation?

The key to resolving these seemingly disparate circumstances is twofold: it involves being more precise about robustness from both the side of the tool and the side of the application. On the tool side, overestimation of ranges arises from abstraction of vector types and timeouts due to an inability to terminate the SMT search for large problem instances. On the problem side, extra precision may be allocated to support corners of the solution space which may be of little or no consequence.

The solution to the second problem identified above requires increased understanding of the application, and support for transforming such knowledge into constraints to exempt corners of the SMT search space from consideration. In a very small way, the constraints

used in some Chapter 3 case studies for restricting the denominator of a division away from zero present an example. Support for behaviour exactly at zero should not need to be provided, and indeed support near zero may not be necessary. At the same time, understanding how a calculation gets into the vicinity of a zero denominator (as an example) may lend insight into undesirable behaviour of the calculation. This ties into the solution of the first problem identified above, which requires greater solver sophistication and capacity, and would assist in solving both the problems identified above. Improving solver support will not only enable tighter bounds for large instances, but at the same time will give more detailed and informative feedback on potential regions for corner cases, where deeper investigation may be required. In general, by using the appropriate constraints the solver can be made to identify regions which stress a representation to its extremes, and that information can be used either to substantiate the need for a given representation (if the region is important to the application), or as guidance to set up constraints exempting that region (if it is unimportant to the application).

5.4 Summary

This chapter has built upon the SMT framework established over the last two chapters, to include support for precision, and iterative methods. An error model for unified custom floating/fixed-point representation has been provided, which deals with absolute and relative error over their respective ranges instead of over the entire regions as it has been the case in the past. Proof of concept has been established through small iterative case studies characteristic to scientific calculations. Challenges in dealing with practical problem setups have been identified and support for tackling these challenges is close at hand. Once overcome, scalable automated representation can be leveraged to accelerate existing scientific applications, improving their performance. Even beyond what exists however, custom data representations can facilitate emergence of new applications through increased parallelism on reconfigurable platforms.

Chapter 6

Concluding remarks

There is a steady shift in the computing industry toward deploying more parallelism per device, while using lower operating frequencies. In terms of parallelism provided by the number of arithmetic units, the FPGAs are outperforming their competing technologies, such as multi-core processors or graphics processing engines, especially for high-end scientific applications; however this comes at an increased implementation effort. The key to addressing this bottleneck is to improve the design methods for rapidly prototyping custom computing architectures in FPGAs, because there is a lack of tools and methods that can help reduce the size of the hardware (and hence boost the speed-up) while at the same time provide robust data representations.

The massive parallelism is dependent on the capacity to deal with potentially ill conditioned calculations, representation of abstract data types, and support for iterative methods. An essential challenge lies in understanding how to automatically scale the operands within the algorithmic dataflow to guarantee precision requirements are met while not over-allocating resources and therefore compromising on parallelism.

To this end, this thesis proposes what is to best of our knowledge the first application of computational methods to the bit-width allocation problem. Through this computational approach based on Satisfiability-Modulo Theories (SMT), as well as the proposed extensions to support large abstract data types and iterative methods, we have developed a framework that can help designers with building custom yet robust data representations for mapping iterative scientific calculations onto hardware.

6.1 Future work

While the adoption of computational methods marks a significant departure from the general wisdom surrounding the solution of this problem, a new beginning holds many exciting opportunities for exploration. With the role of CAD tool support as an enabling technology to improve designer productivity, the usefulness of a tool is influenced beyond how well it operates by how tightly it is integrated to the rest of the tools in the design flow of which it is a part. With this in mind, there are three main avenues along which this thesis could be built upon in the future:

- Improving efficiency of the SMT solver,
- Strengthening links to the application,
- Strengthening links to the implementation,

and each of these points is elaborated below.

6.1.1 SMT solver efficiency

Given the central dependence of the entire bit-width allocation process on the quality of the bounds which are produced by the SMT solver, an obvious direct means of improving the quality of the bit-widths which are obtained is to improve solver efficiency. Based on Section 3.2.3, the solver operation has been shown to consist of two main facets: decisions and propagations.

On the propagation side, interval arithmetic is currently employed due to its simplicity and fast calculation which is important in the context of a solver to enable fast evaluation at solver search branches, facilitating deep searches. At the same time, the quality of the bounds which it produces at each propagation step can degrade badly when the intervals are large. As a result, the solver may have to traverse deeper (make more levels of decisions) than if tighter bounds could be established at each propagation. Furthermore, vectors evaluated by this interval arithmetic through the methods proposed in Chapter 4 experience amplification of any overestimation which interval arithmetic produces.

A starting point to solving this problem is to explore both better bounding methods within the solver as well as tighter integration between all the levels where bounding occurs. Specifically, the block vector methods of Chapter 4 could be far more effective if they were placed within the solver itself - thereby allowing the solver to break the vector into blocks in a different way in each region that the search explores. In addition, alternatives to interval arithmetic could be explored for bounding during propagation, an example being ellipsoid calculus [109]. On top of all this, simulation based methods could be coupled to the solver to explore very quickly "inner bounds" indicating optimistic bit-widths which can be used as a starting point to be refined using the computational techniques herein.

On the search side, the use of internal search algorithms at the solver's core which are designed for general purpose SMT solving could be made more efficient by targeting directly common patterns emerging from bit-width allocation specific searches. Knowing that a particular kind of bounding happens commonly, extra support could be provided to reduce the amount of branching which must be done to obtain these bounds. Also, the way in which an interval is split when branching could be tailored to bit-width allocation, an example being ranges for floating-point numbers, which could be divided logarithmically. Furthermore, links between a variable and its associated precision expression could be established so that when examining a particular error expression, branching on the variables which influence it most strongly could lead to tight bounds more quickly.

6.1.2 Links to the application

When providing CAD tool support for a design problem, it is also important that the tool is well integrated to the rest of the flow. As such, a tool requiring a great deal of effort in order to leverage its benefits has a reduced impact on designer productivity. Directions of exploration on this front relate primarily to Chapter 5.

Significant productivity gains could be made by improving support for the iterative analysis component which extracts instances for direct analysis, primarily based on a theoretical analysis of the algorithm. While this process has been exclusively manual for this thesis, some automation could conceivably be provided, with great potential benefit to designer productivity. Beginning with domain specific analyses, a framework could be

provided which leverages specific knowledge in that domain to automatically create constraints to augment the iterative part of the dataflow.

With such frameworks in place for several domains, application links could be further strengthened by providing infrastructure for automatic code analysis. Providing the capability of analyzing existing software source code would enable reuse of existing code bases for porting to acceleration platforms. At the same time it would shorten the turnaround for new applications to be accelerated by enabling design and verification to be done at a higher level of abstraction. In addition, verification itself would be more tightly coupled and by making feedback on robustness of the representation more accessible, it could begin to play a role in the design process itself.

As Chapter 5 has shown, even double precision floating-point is not immune to the effects of finite precision. Having feedback on where the precision limits are most stressed could provide the designer the opportunity to evaluate whether the effect results from insufficient precision and/or poor choice of algorithm. On the other hand, conditions which are indicated as stressing the precision may be artificial so that they will never be encountered in practice. This feedback can also be useful to the designer, as would support for automatically generating constraints to exclude such the discovered scenario from consideration. Lastly, these input space corners can be an excellent starting point for validation because they will explore design corners.

Along the same lines, constraint support is currently only for hard constraints i.e., *all* constraints must be satisfied at *all* times. While this environment is suitable for scientific calculations and indeed necessary in some cases, it may be overly restrictive in other cases. In DSP, error tolerance criteria may be more descriptive when given in terms of statistics across a multitude of samples (e.g. signal to noise ratio) as opposed to hard error bounds on the calculation of any given sample. Such a characterization of error can be useful in scientific computing as well (e.g., based on [129]), for example applications based on the Monte Carlo method [89]. Clearly defining and providing solver support for these types of constraints can loosen the restriction of hard constraints, adding a degree of freedom to be exploited in searching for representations.

6.1.3 Links to the implementation

In addition to the CAD side, links to the implementation are important. It has been shown that the bit-width allocation problem can be abstracted from the implementation through the use of appropriate hardware/error tradeoff cost models, which are integrated into the flow in Chapter 5. The end result of this flow is then a set of acceptable quantization behaviours which, when applied, will still satisfy the specified error tolerance requirements. This set of quantization behaviours maps directly onto a choice of custom representations. The natural extension is to directly generate hardware descriptions for the custom representation calculation units.

While the map between the quantization behaviour and the representation is clear, having such support in place opens many interesting avenues for further pursuit. For example, being able to automatically generate hardware can be augmented with generation of the simulation and verification support (i.e. testbenches) which would be informed by what the corner cases are in terms of precision. At the same time, automatic generation of hardware assertions for constraints used in the dataflow would also be useful for verification, as they could monitor for unsupported arithmetic behaviour during operation. With automatic links to hardware, the bit-width allocation process could also be more tightly integrated with architectural synthesis. As an alternative to the current methodology where an architecture is decided and then the representation needs are established, choice of precision could be made a part of the architecture search.

6.2 Final remarks

When considering the above suggestions for improvement, it is humbling to realize the immensity and complexity of the problem of bit-width allocation, and at the same time how it is just one of many parts of the overall design flow of digital integrated systems. Throughout the tenure of the research in this thesis we have learned many lessons. We trust that by sharing them here, they may be of benefit to more people from the research and development communities.

Three years ago this research was born out of our involvement in porting a couple of scientific applications onto hardware accelerators. Through those experiences we realized the importance of custom representations to enable high compute throughput. When the project began, there were no computational approaches to the custom data representation problem for iterative algorithms. Over these three years we have:

- attained a robust understanding of the range aspect of the problem
- attained a robust understanding of dealing with vector calculus for large problem instances
- attained a robust understanding on how to abstract the implementation, thus enabling platform independence
- attained the fundamental understanding about how to formulate the precision aspect of the problem
- gained valuable experience in breaking down iterative algorithms for analysis
- recognized the need to expand on both the scientific and implementation fronts to enable automatic scalability to large scientific applications.

With these lessons learned and the future directions highlighted, we consider this thesis as an important step toward the faster implementation and hence adoption of large-scale custom compute platforms. We hope eventually to see these platforms achieve unprecedented compute throughput with low energy requirements in a small form-factor. We anticipate such platforms to not only enable new systems and technologies to be created, but also to push the frontiers of scientific discovery.

Bibliography

[1] A. Ahmadi and M. Zwolinski. Symbolic noise analysis approach to computational hardware optimization. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 391 –396, June 2008.

[2] G. Arfken. *Mathematical Methods for Physicists, 3rd Edition*. Orlando, FL: Academic Press, 1985.

[3] B. Armstrong and R. Eigenmann. Application of automatic parallelization to modern challenges of scientific computing industries. In *Proceedings of the International Conference on Parallel Processing*, pages 279 –286, September 2008.

[4] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[5] P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, D. Bagchi, S. Pal, N. Tripathi, D. Zaretsky, R. Anderson, and J.R. Uribe. Overview of a compiler for synthesizing MATLAB programs onto FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):312–324, March 2004.

[6] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho. Entering the petaflop era: The architecture and performance of Roadrunner.

In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, November 2008.

[7] P. Belanovic and M. Rupp. Automated floating-point to fixed-point conversion with the Fixify environment. In *Proceedings of the International Workshop on Rapid System Prototyping*, pages 172–178, 2005.

[8] A. Benedetti and P. Perona. Bit-width optimization for configurable DSP's by multi-interval analysis. In *Proceedings of the Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 355 –359 vol.1, 2000.

[9] R. Bergamaschi, L. Benini, K. Flautner, W. Kruijtzer, A. Sangiovanni-Vincentelli, and K. Wakabayashi. The state of ESL design [roundtable]. *IEEE Design Test of Computers*, 25(6):510 –519, November-December 2008.

[10] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[11] R.N. Bracewell. *The Fourier Transform and its Applications, 3rd Edition*. McGraw-Hill, 1999.

[12] R.L. Burden and J.D. Faires. *Numerical Analysis, 7th Edition*. Brooks Cole, 2000.

[13] D. Burger and J.R. Goodman. Billion-transistor architectures: there and back again. *Computer*, 37(3):22 – 28, March 2004.

[14] W. Cammack and M. Paley. Fixpt: a C++ method for development of fixed point digital signal processing algorithms. In *Proceedings of the Hawaii International Conference on System Sciences*, volume 1, pages 87 –95, January 1994.

[15] M.-A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie. An automatic word length determination method. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 5, pages 53 –56 vol. 5, 2001.

[16] M.-A. Cantin, Y. Savaria, and P. Lavoie. A comparison of automatic word length optimization procedures. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 2, pages II–612 – II–615 vol.2, 2002.

[17] J.M.C. Carvajal, R. Schwemmer, J.-C. Garnier, and N. Neufeld. A high-performance storage system for the LHCb experiment. In *Proceedings of the IEEE-NPSS Real Time Conference*, pages 426 –430, May 2009.

[18] G. Chen, G. Li, S. Pei, and B. Wu. GPGPU supported cooperative acceleration in molecular dynamics. In *Proceedings of the International Conference on Computer Supported Cooperative Work in Design*, pages 113–118, April 2009.

[19] J.-Il Choi, H.-S. Jun, and S.-Y. Hwang. Efficient hardware optimisation algorithm for fixed point digital signal processing ASIC design. *Electronics Letters*, 32(11): 992 –994, May 1996.

[20] T.F. Clayton, A.F. Murray, and I. Lindsay. GP-GPU: Bridging the gap between modelling and experimentation. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, pages 453–459, 29 2009-Aug. 1 2009.

[21] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens. A methodology and design environment for DSP ASIC fixed point refinement. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, pages 271–276, 1999.

[22] V. Cocilovo, G. Calabro, A. Cucchiaro, A. Pizzuto, G. Ramogida, and C. Rita. Toroidal field coil thermal analysis for fast tokamak. In *Proceedings of the IEEE/NPSS Symposium on Fusion Engineering*, pages 1 –4, June 2009.

[23] J. Cong, K. Gururaj, B. Liu, C. Liu, Z. Zhang, S. Zhou, and Y. Zou. Evaluation of static analysis techniques for fixed-point precision optimization. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 231–234, April 2009. doi: 10.1109/FCCM.2009.35.

[24] G.A. Constantinides. Perturbation analysis for word-length optimization. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 81 – 90, April 2003.

[25] G.A. Constantinides, P.Y.K. Cheung, and W. Luk. The multiple wordlength paradigm. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 51 – 60, 2001.

[26] G.A. Constantinides, P.Y.K. Cheung, and W. Luk. Optimum wordlength allocation. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 219 – 228, 2002.

[27] G.A. Constantinides, P.Y.K. Cheung, and W. Luk. Wordlength optimization for linear digital signal processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(10):1432 – 1442, October 2003.

[28] G.A. Constantinides, P.Y.K. Cheung, and W. Luk. Optimum and heuristic synthesis of multiple word-length architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(1):39 – 57, January 2005.

[29] B.R. Coutinho, G.L.M. Teodoro, R.S. Oliveira, D.O.G. Neto, and R.A.C. Ferreira. Profiling general purpose GPU applications. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing*, pages 11–18, October 2009.

[30] D.E. Culler and A. Singh, J.P. with Gupta. *Parallel Computer Architecture, A Hardware/Software Approach.* Morgan Kafmann, 1999.

[31] S.M. Easterbrook and T.C. Johns. Engineering the software for understanding climate change. *Computing in Science Engineering*, 11(6):65 –74, November-December 2009.

[32] N. En and N. Srensson. MiniSat page. Online. URL `http://minisat.se/Main.html`.

[33] C.F. Fang, R.A. Rutenbar, and T. Chen. Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 275–282, 2003.

[34] M. Flynn, R. Dimond, O. Mencer, and O. Pell. Finding speedup in parallel processors. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, pages 3 –7, July 2008.

[35] M. Franzle and C. Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):178–198, June 2007.

[36] Free Software Foundation (FSF). GSL-GNU scientific library - GNU project. Online. URL `http://www.gnu.org/software/gsl/`.

[37] A.A. Gaffar, O. Mencer, W. Luk, P.Y.K. Cheung, and N. Shirazi. Floating-point bitwidth analysis via automatic differentiation. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 158 – 165, December 2002.

[38] A.A. Gaffar, O. Mencer, and W. Luk. Unifying Bit-width Optimisation for Fixed-point and Floating-point Designs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 79–88, 2004.

[39] D. Geer. Networks on processors improve on-chip communications. *Computer*, 42 (3):17 –20, March 2009.

[40] M.K. Gobbert. Configuration and performance of a Beowulf cluster for large-scale scientific simulations. *Computing in Science and Engineering*, 7(2):14–26, March-April 2005.

[41] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[42] G.H. Golub and C.F. Van Loan. *Matrix Computations, 3rd Edition.* John Hopkins University Press, 1996.

[43] S. Gopalakrishnan, P. Kalla, and F. Enescu. Optimization of arithmetic datapaths with finite word-length operands. In *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference*, pages 511 –516, January 2007.

[44] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10 –24, March-April 2006.

[45] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji. A comparative study on ASIC, FPGAs, GPUs and general purpose processors in the $O(N^2)$ gravitational n-body simulation. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, pages 447–452, August 2009.

[46] M.S. Hameed and I.A. Manarvi. Using FEM and CFD to locate cracks in compressor blades for non destructive inspections. In *Proceedings of the IEEE Aerospace Conference*, pages 1 –11, March 2009.

[47] K. Han and B.L. Evans. Wordlength optimization with complexity-and-distortion measure and its application to broadband wireless demodulator design. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages V – 37–40 vol.5, May 2004.

[48] K. Han, A.G. Olson, and L. Evans. Automatic floating-point to fixed-point transformations. In *Proceedings of the Asilomar Conference on Signals, Systems and Computers*, pages 79 –83, November 2006.

[49] R.-Y. Han. Fast fourier transform correlation tracking algorithm with background correction. US Patent number: 6970577, November 2005. Lockheed Martin Corporation.

[50] L. Hasan, Z. Al-Ars, and S. Vassiliadis. Hardware acceleration of sequence alignment algorithms - an overview. In *Proceedings of the International Conference on Design Technology of Integrated Systems in Nanoscale Era*, pages 92 –97, September 2007.

[51] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 4th edition*. Morgan Kauffman, San Francisco, 2007.

[52] L.C. Higbie. Vector floating-point data format. *IEEE Transactions on Computers*, C-25(1):25 –32, January 1976.

[53] A. Hosangadi, F. Fallah, and R. Kastner. Factoring and eliminating common subexpressions in polynomial expressions. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 169 – 174, November 2004.

[54] A. Hosangadi, R. Kastner, and F. Fallah. Energy efficient hardware synthesis of polynomial expressions. In *Proceedings of the International Conference on VLSI Design*, pages 653 – 658, January 2005.

[55] C.C. Hurd. Early computers at IBM. *IEEE Annals of the History of Computing*, 3 (2):163 –182, April-June 1981.

[56] IEEE. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–58, 29 2008. doi: 10.1109/IEEESTD.2008.4610935.

[57] T. Ikeda, F. Ino, and K. Hagihara. A code motion technique for accelerating general-purpose computation on the GPU. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1–10, April 2006.

[58] V.I. Istratescu. *Fixed Point Theory: An Introduction.* Springer, 1981.

[59] W. Kahan. Why do we need a floating-point arithmetic standard? Technical report, EECS Department, University of California, Berkeley, 1981. URL `http://www.eecs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf`.

[60] W. Kahan. The improbability of probabilistic error analyses for numerical computations. Technical report, EECS Department, University of California, Berkeley, 1998. URL `http://www.eecs.berkeley.edu/~wkahan/improber.pdf`.

[61] W. Kahan. On the cost of floating-point computation without extra-precise arithmetic. Technical report, EECS Department, University of California, Berkeley, 2004. URL `http://www.eecs.berkeley.edu/~wkahan/Qdrtcs.pdf`.

[62] W. Kahan. How futile are mindless assessments of roundoff in floating-point computation? Technical report, EECS Department, University of California, Berkeley, 2006. URL `http://www.eecs.berkeley.edu/~wkahan/Mindless.pdf`.

[63] H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A fixed-point design and simulation environment. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, pages 429–435, 1998.

[64] C.N. Keltcher, K.J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66 – 76, March-April 2003.

[65] A.B. Kinsman and N. Nicolici. Finite precision bit-width allocation using SAT-modulo theory. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, pages 1106–1111, 2009.

[66] A.B. Kinsman and N. Nicolici. Computational bit-width allocation for operations in vector calculus. In *Proceedings of the IEEE International Conference on Computer Design*, pages 433–438, 2009.

[67] A.B. Kinsman and N. Nicolici. Robust design methods for hardware accelerators for iterative algorithms in scientific computing. In *Proceedings of the IEEE/ACM Design Automation Conference*, June 2010. To appear.

[68] A.B. Kinsman and N. Nicolici. Bit-width allocation for hardware accelerators for scientific computing using SAT-modulo theory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3):405 –413, March 2010.

[69] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10 –23, May-June 2006.

[70] J. Kolodzey. CRAY-1 computer technology. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, 4(2):181 – 186, June 1981.

[71] J. Krikke. Near real-time tsunami computer simulations within reach. *IEEE Computer Graphics and Applications*, 25(5):16 – 21, September-October 2005.

[72] G. Kron. Numerical solution of ordinary and partial differential equations by means of equivalent circuits. *Journal of Applied Physics*, 16(3):172 – 186, March 1945.

[73] K.-Il Kum and W. Sung. Word-length optimization for high-level synthesis of digital signal processing systems. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 569 –578, October 1998.

[74] K.-Il Kum, J. Kang, and W. Sung. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47 (9):840 –848, September 2000.

[75] K.-Il Kum and W. Sung. Combined word-length optimization and high-level synthesis of digital signal processing systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(8):921 –930, August 2001.

[76] J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra. The PlayStation 3 for high-performance scientific computing. *Computing in Science Engineering*, 10(3):84 –87, May-June 2008.

[77] W.B. Ligon, S. III McMillan, G. Monn, K. Schoonover, F. Stivers, and K.D. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 206 –215, April 1998.

[78] K. Lillywhite, D.-J. Lee, S. Antani, D. Zhang, and R. Long. Lessons learned in developing a low-cost high performance medical imaging cluster. In *Proceedings of the IEEE International Symposium on Computer-Based Medical Systems*, pages 1–6, August 2009.

[79] Y. Liu, S. Jiao, W. Wu, and S. De. GPU accelerated fast FEM deformation simulation. In *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems*, pages 606–609, December 2008.

[80] J.A. Lopez, C. Carreras, and O. Nieto-Taladriz. Improved interval-based characterization of fixed-point LTI systems with feedback loops. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(11):1923 –1933, November 2007.

[81] R. Mafi, S. Sirouspour, B. Moody, B. Mahdavikhah, K. Elizeh, A.B. Kinsman, N. Nicolici, M. Fotoohi, and D. Madill. Hardware-based parallel computing for

real-time haptic rendering of deformable objects. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, pages 4187–4187, 2008.

[82] R. Mafi, S. Sirouspour, B. Mahdavikhah, B. Moody, K. Elizeh, A.B. Kinsman, and N. Nicolici. A parallel computing platform for real-time haptic interaction with deformable bodies. *IEEE Transactions on Haptics*, 2009. doi: 10.1109/TOH.2009.50.

[83] A. Mallik, D. Sinha, P. Banerjee, and H. Zhou. Low-power optimization by smart bit-width allocation in a SystemC-based ASIC design environment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(3):447–455, March 2007.

[84] D. Manocha. General-purpose computations using graphics processors. *Computer*, 38(8):85–88, August 2005.

[85] Maplesoft. Math software for engineers, educators and students. Online. URL `http://www.maplesoft.com/`.

[86] The MathWorks. Fixed-point toolbox - MATLAB. Online. URL `http://www.mathworks.com/products/fixed/`.

[87] D.W. Matula and P. Kornerup. Finite precision rational arithmetic: Slash number systems. *IEEE Transactions on Computers*, C-34(1):3–18, January 1985.

[88] D. Menard, R. Rocher, and O. Sentieys. Analytical fixed-point accuracy evaluation in linear time-invariant systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 55(10):3197 –3208, November 2008.

[89] N. Metropolis and S. Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335341, September 1949. doi: 10.2307/2280232.

[90] E. Mollick. Establishing Moore's law. *IEEE Annals of the History of Computing*, 28 (3):62 –75, July-September 2006.

[91] R.E. Moore. *Interval Analysis*. Prentice Hall, 1966.

[92] V. Natoli. HPCwire: Heterogeneous processing: Trite or trend? Online, June 2009. URL http://www.hpcwire.com/topic/processors/Heterogeneous-Processing-Trite-or-Trend-49029591.html?viewAll=y. Stone Ridge Technology.

[93] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee. Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, pages 722 –728, 2001.

[94] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer-Verlag, 1999.

[95] Numerical Algorithms Group. Online. URL http://www.nag.co.uk/.

[96] University of Oldenburg. Hysat download. Online. URL http://hysat.informatik.uni-oldenburg.de/26273.html.

[97] Massachusetts Institute of Technology. Maxima, a computer algebra system. Online. URL http://maxima.sourceforge.net/.

[98] W.G. Osborne, R.C.C. Cheung, J.G.F. Coutinho, W. Luk, and O. Mencer. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 617–620, 2007.

[99] W.G. Osborne, R.C.C. Cheung, J.G.F. Coutinho, W. Luk, and O. Mencer. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 617 –620, August 2007.

[100] W.G. Osborne, J.G.F. Coutinho, R.C.C. Cheung, W. Luk, and O. Mencer. Instrumented multi-stage word-length optimization. In *Proceedings of the International Conference on Field-Programmable Technology*, pages 89 –96, December 2007.

[101] Y. Pang and K. Radecka. Optimizing imprecise fixed-point arithmetic circuits specified by Taylor series through arithmetic transform. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 397–402, 2008.

[102] Y. Pang, K. Radecka, and Z. Zilic. Arithmetic transforms of imprecise datapaths by Taylor series conversion. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems*, pages 696 –699, December 2006.

[103] C.H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.

[104] A. Peymandoust and G. De Micheli. Application of symbolic computer algebra in high-level data-flow synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(9):1154 – 1165, September 2003.

[105] Princeton University. Boolean satisfiability research group at princeton. Online. URL `http://www.princeton.edu/~chaff/zchaff.html`.

[106] Y. Pu and Y. Ha. An automated, efficient and static bit-width optimization methodology towards maximum bit-width-to-error tradeoff with affine arithmetic model. In *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference*, page 6 pp., January 2006.

[107] K. Radecka and Z. Zilic. Specifying and verifying imprecise sequential datapaths by arithmetic transforms. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 128 – 131, November 2002.

[108] K. Radecka and Z. Zilic. Arithmetic transforms for compositions of sequential and imprecise datapaths. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1382 –1391, July 2006.

[109] L. Ros, A. Sabater, and F. Thomas. An ellipsoidal calculus based on propagation and fusion. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 32(4):430 –442, August 2002.

[110] S. Sarkar, S. Dabral, P.K. Tiwari, and R.S. Mitra. Lessons and experiences with high-level synthesis. *IEEE Design Test of Computers*, 26(4):34 –45, July-August 2009.

[111] R.A. Serway, R.J. Beichner, and J.W. Jewett. *Physics for Scientists and Engineers, Volume II, 5th Edition.* Harcourt, 1999.

[112] D.E. Shaw, M.M. Deneroff, R.O. Dror, J.S. Kuskin, R.H. Larson, J.K. Salmon, C. Young, B. Batson, K.J. Bowers, J.C. Chao, M.P. Eastwood, J. Gagliardo, J.P. Grossman, C.R. Ho, D.J. Ierardi, I. Kolossvry, J.L. Klepeis, T. Layman, C. McLeavey, M.A. Moraes, R. Mueller, E.C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S.C. Wang. Anton: A special-purpose machine for molecular dynamics simulation. In *Proceedings of the International Symposium on Computer Architecture*, pages 1–12, 2007.

[113] J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, EECS Department, University of California, Berkeley, August 1994. URL `http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`.

[114] C. Shi and R.W. Brodersen. An automated floating-point to fixed-point conversion methodology. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 529–532, 2003.

[115] S.G. Shiva. Automatic hardware synthesis. *Proceedings of the IEEE*, 71(1):76 – 87, January 1983.

[116] J. Smith and G. De Micheli. Polynomial methods for allocating complex components. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, pages 217 –222, 1999.

[117] Stanford University. Folding@home distributed computing. Online. URL `http://folding.stanford.edu/`.

[118] J. Stewart. *Calculus: Early Transcendentals, 6th Edition.* Brooks Cole, 2007.

[119] J. Stolfi and L.H. de Figueiredo. *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquim Monographs. IMPA/CNPq, Rio de Janeiro, Brazil, 1997.

[120] W. Sung and K.-Il Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *IEEE Transactions on Signal Processing*, 43(12):3087 –3090, December 1995.

[121] M.R. Titchener. Towards real-time measurement of information in a scientific setting. In *Proceedings of the International Symposium on Communication Systems, Networks and Digital Signal Processing*, pages 316 –320, July 2008.

[122] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: Architectures and design methods. *IEE Proceedings - Computers and Digital Techniques*, 152(2):193–207, March 2005.

[123] University of California, Berkeley. SETI@home. Online. URL `http://setiathome.berkeley.edu/`.

[124] J.E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, August 1990.

[125] S.A. Wadekar and A.C. Parker. Accuracy sensitive word-length selection for algorithm optimization. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 54 –61, October 1998.

[126] M. Willems, V. Bursgens, H. Keding, T. Grotker, and H. Meyr. System level fixed-point design based on an interpolative approach. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 293 –298, June 1997.

[127] M. Woh, S. Mahlke, T. Mudge, and C. Chakrabarti. Mobile supercomputers for the next-generation cell phone. *Computer*, 43(1):81 –85, January 2010.

[128] Wolfram. Wolfram Research: Mathematica, technical and scientific software. Online. URL `http://www.wolfram.com/`.

[129] B. Wu, J. Zhu, and F.N. Najm. Dynamic-range estimation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(9):1618 –1636, September 2006.

[130] W.A. Wulf and S.A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):2024, March 1995.

[131] R. Yates. Fixed-point arithmetic: An introduction. Technical report, Digital Signal Labs, July 2009. URL `http://www.digitalsignallabs.com/fp.pdf`.

Index