Politecnico di Milano
Facoltà di Ingegneria dell'Informazione
Corso di laurea in Ingegneria Informatica

Trimming the bit: optimizing high-level synthesis of floating-point based descriptions by extending value range analysis

Relatore: Prof. Fabrizio Ferrandi

Tesi di laurea di
Michele Fiorito
Matr. 893957

Anno Accademico 2018/2019

Abstract

Field Programmable Gate Arrays are one of the most interesting and powerful pieces of hardware available today to the computer science community: they allow an incredibly fine-grained tuning of the implementation bringing development process from high-level description down to the architectural hardware specification. Such a definite level of description sets up the way for deeper analysis on applications to squeeze performance out of every line and smooth every bit achieving the most out of the silicon.

To easily exploit this highest detailed implementation capability an automated tool is mandatory for developers: while standard compilers have been optimized and refined to abstract software to CPU level language, new High-Level Synthesis tools have been developed to bring on this abstraction a step further to the hardware description language and to extend and increase available optimizations to achieve the best possible result at this new layer.

One of the optimizations already available from a standard compiler is that of value range analysis, meaning a program is analyzed to compute numerical bounds on each one of its variable: this information can be exploited to perform dead code elimination, branch prediction and avoid the insertion of arithmetic operation checks which would slow the execution of the compiled software. The purpose of this thesis work is to bring all of the available capabilities of value range analysis inside a High-Level Synthesis tool and to extend them taking advantage of the more powerful description language available at this new implementation level. The fact that we are actually building hardware arithmetic operators from scratch enables this analysis to be applied in a finer-grained flavor and to better analyze floating-point operations too, allowing the use of shrunk numerical encoding, thus tinier, faster and less power demanding architectures. Furthermore, this work will allow the application of optimizations directly on standard IEEE 754 floating-point values, without the need for a fixed-point representation.

Abstract in italiano

Le schede Field Programmable Gate Array (FPGA) si possono considerare l'hardware più interessante e versatile nel campo dell'informatica moderna: garantiscono agli sviluppatori la possibilità di descrivere non solo le specifiche di alto livello di un'applicazione ma anche di progettare l'architettura nei minimi dettagli. Grazie a queste superiori capacità descrittive è possibile effettuare nuovi tipi di analisi sulle applicazioni in modo da ottimizzare al meglio ogni bit ed ottenere le massime prestazioni possibili.

Per sfruttare al meglio il livello di dettaglio implementativo offerto dalle schede FPGA è sicuramente necesssario automatizzare la progettazione. Come i compilatori odierni sono stati raffinati per tradurre le applicazioni in linguaggio macchina, così i moderni software di sintesi ad alto livello contribuiscono a raffinare questo processo arrivando a produrre una descrizione architetturale delle applicazioni, migliorando di pari passo anche le ottimizzazioni e le analisi effettuate durante tale processo per sfruttare al meglio le maggiori possibilità offerta dall'hardware.

Tra le ottimizzazioni già disponibili nei compilatori odierni è di particolare interesse l'analisi dei valori: tramite questa analisi è possibile ottenere informazioni sui valori assunti dalle variabili durante l'esecuzione del programma e, grazie a questo livello di consapevolezza, eliminare numerosi controlli normalmente effettuati su tali variabili durante le operazioni aritmetiche, effettuare *branch predictions* più accurate e determinare in modo più preciso le parti di codice non raggiungibili, ottenendo così un'applicazione più efficiente e performante. L'obiettivo di questa tesi è di trasportare all'interno di un programma di sintesi ad alto livello questo tipo di analisi, estendendone le capacità, per poter sfruttare appieno il miglior livello descrittivo a disposizione. Nel nuovo ambiente di sviluppo l'architettura delle operazioni aritmetiche viene concepita durante il processo di compilazione, sarà dunque possibile sfruttare questo tipo di analisi per effettuare ottimizzazioni più approfondite. Sarà inoltre possibile ottimizzare gli operatori a virgola mobile, non accessibili ai normali compilatori, adattandone la precisione e le specifiche per ottenere unità aritmetiche su misura a seconda delle necessità. Infine un metodo di modifica diretto della specifica floating-point IEEE 754 sarà sviluppato per semplificare l'adattamento della rappresentazione alle necessità delle applicazioni, senza l'obbligo di ripiegare su soluzioni a virgola fissa.

Ringraziamenti

Vorrei ritagliare questo piccolo spazio tra le pagine della mia tesi per ringraziare tutte le persone che mi hanno supportato e offerto la possibilità di migliorare sia dal punto di vista personale che professionale durante questi anni al Politecnico di Milano.

Innanzitutto ringrazio il professor Ferrandi per avermi affiancato e guidato verso la conclusione del percorso di studi, grazie a lui mi sono appasionato ancor di più al campo dell'ingegneria e della ricerca.

Ringrazio i miei compagni di studi e colleghi, con i quali ho trascorso tutti gli anni al Politecnico, dai primi momenti della triennale a Cremona fino agli ultimi esami della magistrale a Milano. Con loro ho condiviso le difficoltà e i successi accademici, e insieme siamo arrivati alla realizzazione dei nostri obiettivi.

Mando un abbraccio ai miei amici più cari con i quali ho vissuto molte avventure al di fuori degli ambiti accademici. Insieme a loro sono cresciuto e ho condiviso innumerevoli momenti della vita e, anche se i nostri percorsi sono diversi, rimaniamo comunque complici e sempre in contatto per condividere vittorie e sconfitte.

Infine il ringraziamento più grande va sicuramente alla mia famiglia, e in particolare a mia madre, che mi ha sempre sostenuto e spronato ad affrontare nuove sfide e mi ha permesso di raggiungere questi risultati.

Contents

1 Introduction **1**

2 Definitions **5**

2.1 High-Level Synthesis . . . 5

2.1.1 Generic compiler . . . 6

2.1.2 HLS tool . . . 7

2.2 Interval arithmetic . . . 9

2.3 Floating-point arithmetic internals . . . 11

3 State of the art **13**

3.1 Value Range Analysis . . . 13

3.1.1 Static Range Analysis . . . 14

3.1.2 Dynamic Range Analysis . . . 15

3.2 Abstract interpretation algorithm . . . 16

3.2.1 Range Definition . . . 17

3.2.2 Constraints Definition . . . 19

3.2.3 Extended SSA . . . 20

3.2.4 Solving Range Analysis Problem . . . 24

3.3 Bit Value Inference . . . 29

3.4 Conclusions . . . 30

4 Proposed solution **33**

4.1 Range Analysis at HLS Level . . . 33

4.1.1 Design Choices . . . 34

4.2 Value Range Analysis Algorithm . . . 35

4.2.1 Range representation . . . 36

4.2.2 Generating e-SSA . . . 40

4.2.3 Constraints graph definition . . . 41

4.2.4 Floating-point range constraints . . . 45

4.2.5 Resolution algorithm . . . 47

4.2.6 BitValue Inference enhancement 50
4.3 Floating-point encoding customization 52
4.4 Conclusions 53

5 Results 55
5.1 Experimental setup 55
5.2 Benchmark suite 56
5.2.1 Single operators 56
5.2.2 Complete applications 57
5.3 Result evaluation 57
5.4 Conclusions 68

6 Conclusions and future work 69
6.1 Design flow evaluation 69
6.2 Future developments 70

References 74

List of Tables

5.1 CHStone benchmark suite list 57
5.2 Customized floating-point representations test for double precision division operator 58
5.3 Customized floating-point representations test for single precision division operator 59
5.4 CHStone benchmark detail for GCC 4.9. AREA 61
5.5 CHStone benchmark detail for Clang 4. AREA 61
5.6 CHStone benchmark detail for GCC 4.9 frontend. PERFORMANCE 63
5.7 CHStone benchmark detail for Clang 4 frontend. PERFORMANCE 63
5.8 CHStone benchmark detail for GCC 4.9 frontend. PERFORMANCE-MP 64
5.9 CHStone benchmark detail for Clang 4 frontend. PERFORMANCE-MP 64
5.10 CHStone benchmark detail for GCC 4.9 frontend. BALANCED . 66
5.11 CHStone benchmark detail for GCC 4.9 frontend. BALANCED-MP 66
5.12 CHStone benchmark detail for Clang 4 frontend. BALANCED . . 67
5.13 CHStone benchmark detail for Clang 4 frontend. BALANCED-MP 67

List of Figures

2.1 IEEE 754 32bit floating-point representation 11

3.1 (a) Example program. (b) Control Flow Graph in e-SSA form. (c) Constraints extracted from the program. (d) Possible solution to the range analysis problem. 21
3.2 (a) Example program. (b) SSA form. (c) e-SSA form. (d) u-SSA form. 22
3.3 Constraints graph genereted from constraints in Figure 3.1 (c) . . 25
3.4 Resolution phases snapshots of the last SCC of Figure 3.3. (a) After removing control dependence edges. (b) After running the widening phase. (c) After running future resolution phase. (d) After running narrowing phase. 28
3.5 The bit values lattice. The ordering is defined by the "information content". 29
3.6 A C function and the associated data-flow graph. The types inferred by forward(backward) propagation are shown in the left (right) figure. We assume that a char has 8 bits. 30

4.1 (a) Ranges from set $\mathcal{T}$ without *anti-range*. (b) Ranges from set $\mathcal{T}_A$ with *anti-range*. 41
4.2 (a) Sample function code. (b) Corresponding constraints graph. . 42
4.3 (a) Sample function code. (b) Function code in e-SSA form. (c) Constraint graph after build stage. (d) Solved constraints graph. 43
4.4 (a) e-SSA form code. (b) Constraints graph with future ranges and control dependence edges. 44
4.5 (a) e-SSA form code. (b) Constraints graph with symbolic constraints. 45
4.6 (a) Common unpacking sequence. (b) Constraints graph from the unpacking sequence. 46

4.7 (a) Standard lattice. (b) Extended lattice. 49
4.8 50
4.9 (a) Simple cycle code. (b) Range analysis (upper) and BitValue inference (lower) results for variable i 51
4.10 (a) Pragma mask quick reference. (b) Custom floating-point specification example. 52

5.1 CHStone benchmark overview for GCC 4.9 (left side) and Clang 4 (right side) frontends. AREA 60
5.2 CHStone benchmark overviw for GCC 4.9 and Clang 4 frontends. PERFORMANCE setup (above) and PERFORMANCE-MP (below) 62
5.3 CHStone benchmark overviw for GCC 4.9 and Clang 4 frontends. BALANCED setup (above) and BALANCED-MP setup (below) . 65

Chapter 1

Introduction

In modern computer science, even the most powerful CPU is no longer the best choice to perform a sequence of operations, newer and more specific hardware is required in some cases to achieve a higher level of performance: this is the case where FPGAs come into play, enabling developers to fine-tune hardware architecture and push performance to the limit. While standard CPUs do offer an efficient general-purpose solution, FPGAs offer the possibility to build ad-hoc hardware design to achieve the best out of the silicon, thus a deeper knowledge is required to the user to be able to exploit it completely. To overcome the enormous effort required by the architecture design process, new design tools have been implemented and optimized nowadays: as compilers helped in the process of translating from high-level specifications to CPU language, new High-Level Synthesis tools do the same generating optimized and efficient architectures from software specifications. Thanks to the finer-grained approach granted by FPGAs and High-Level Synthesis tools, it is possible to explore newer and deeper analyses and optimizations to improve performances, reduce power consumption and obtain tinier hardware designs.

Among the vast set of available optimizations, this thesis work aims to explore enhancement capabilities offered from value range analysis. The goal of this analysis is to gather information about the set of values associated with each program variable at runtime: to know this information at compile-time

can be extremely useful to carry out numerous optimizations, such as branch prediction, numerical checks and array bounds checks elimination, dead code elimination, security checks and many more. While there are simple implementations targeting common compilers for CPUs, there are none concerning hardware design and HLS tools in the current state of the art. Furthermore none of the current works on value range can deal with standard floating-point representations, leaving a big gap in possible utilization scenarios.

The main purpose of this thesis work is to exploit value range analysis applying it inside a high-level synthesis design flow. This work will produce an efficient implementation starting from the state of the art algorithms and enhancing value range analysis with new features and capabilities: a new abstract interpretation engine will be designed to achieve an optimal trade-off between precision and performance, all available variables and operations will be considered, including both integers and floating-point types, a new floating-point tuning feature will be tested to enable compile-time floating-point representation customization, furthermore an inter-procedural propagation technique will assure global awareness leading to better optimizations. Finally, the considered design flow will be featured by a bit value inference algorithm, which will be upgraded too to support new floating-point designs and to strictly cooperate with value range analysis to achieve the maximum level of optimization. Given these premises a significant reduction in the bitwidth of program variables is expected, resulting in smaller resources utilization and better performance for all types of applications. Concerning floating-point applications only, a new level of customization will be available through the tunable floating-point representation feature introduced, thus an even more pronounced performance and resources gain is expected, not only because smaller functional units will be generated, but even thanks to the inter-procedural approach which will result in globally optimized applications.

From the above a summary of this thesis contributions can be drawn as follows:

- Integration of a static value range analysis algorithm into an existing HLS tool deisgn flow
- Enhancement of the value range algorithm to improve analysis efficacy on integers
- Upgrade of the value range analysis to include floating-point variables as part of the analysed values
- Enhancement of an existing BitValue Inference algorithm to include floating-point types
- Adaptation of the HLS tool design flow to enable value range and BitValue inference cooperation
- Design of a new method for compile-time floating-point representation customization

The thesis document is composed by this introduction, a chapter containing some background definitions and information about concepts and algorithms discussed in the thesis core, an initial review on the current state of the art in the field of value range analysis, High-Level Synthesis tools and current implementations, then the actual implementation proposed is explained discussing design choices and new features introduced, experimental results are described and compared with state of the art and finally conclusions are drawn and possible future implementations are discussed.

Chapter 2

Definitions

2.1 High-Level Synthesis

The process of translating a software specification into a hardware description is called high-level synthesis: this operation begins from a software specification, commonly written using a high-level language such as C/C++, which is then translated into a hardware specification of an RTL design, most likely VHDL or Verilog. The obtained RTL design specification can be subsequently used to program an FPGA or to support an ASIC design.

The high-level synthesis process is composed of many translation and optimization steps which progressively build a hardware design which corresponds exactly to the input software representation: this procedure is pretty complex and difficult to be done by hand and most likely requires an expert RTL designer to produce a decent outcome, but, as common high-level code compilation, it can be automated and optimized as well. After many years of development and improvements, so-called HLS tools can now produce top performance RTL designs from high-level software specifications exploiting an automated flow: thanks to these tools high-level developers can easily take advantage of hardware design to enhance performances without the need to become RTL design experts.

In the following some general information on how standard compilers and HLS tools work are discussed.

2.1.1 Generic compiler

A common compiler outline can be subdivided into three main components: a frontend, which parses the input specification, an intermediate core, where optimizations take place, and a backend, which takes care of the output generation. In more detail, a frontend produces an initial translation from a given input specification language, such as C++, to a generic intermediate representation (IR) defined by the compiler; many frontends can be exposed from a single compiler to handle different input languages which are then translated to the same IR. Thanks to this unique IR the intermediate part of the compiler will always deal with the same type of representation while performing transformations and optimizations: it strictly depends on compiler implementation, but commonly the IR is pretty similar to a generalized assembly language, composed by standard mathematical operations, branch and jump instructions and memory access primitives, only load and store in most cases. The IR represents the internal backbone of a compiler and all operations performed will be based on it: it is commonly free from real assembly languages' constraints such as limited number of registers or calling conventions, furthermore IR instructions are generally expressed in SSA form, which significantly eases most of the analyses performed. When compiler transformations are completed the backend part comes into play to finally translate the obtained IR into specific assembly code and subsequently binary code: as for the frontend, many backends can be exposed by the same compiler, enabling the user to produce executables for different architectures. Thanks to this modular approach, a modern compiler is versatile and can be easily integrated with new components: as an example, if a new programming language needs to be handled it is only necessary to implement a new frontend for the compiler to translate this language to the compiler IR, then all existing optimizations and transformations are still valid and each backend module will be reusable to produce a specific architecture output from the new input language.

While frontends and backends are pretty simple, it is necessary to describe in more detail the compiler internals: after IR is generated there an incredible amount of transformations which can be performed to analyze and improve

the input application, but they cannot be applied randomly. Each optimization is implemented and performed by what is called a *step*: each step can modify the IR, thus invalidating previously executed steps or requiring new ones to be triggered. To manage these inter-step dependencies a dependencies flow is generated and updated during compilation process to correctly perform all required steps; compilation ends when generated steps flow is completed and each step has no more transformations to apply: only at this point IR is ready to be handled by the backend. It may seem a pretty complex procedure, but this guarantees an incredibly efficient interaction between steps which can often improve each other adding information to the IR on subsequent runs; as an example a dead code elimination step can produce a much better result if executed after constants propagation has been completed, thus ensuring this precedence rule will result in a better IR.

2.1.2 HLS tool

A High-Level Synthesis tool is pretty similar to a standard compiler in most of its aspects, same outline of frontend, intermediate and backend is still used, but operations performed have been adapted to the new architecture. Frontend still translates a high-level input specification into a compiler-specific IR which will then be modified from internal compiler steps, but this new IR has to be much richer in information to be efficient once hardware synthesis is triggered. A standard compiler IR is commonly based on a generic assembly language which abstracts common instructions found in all assembly languages related to real CPU architectures; when talking about HLS the *"real architecture"* assembly emulation is not an option as it is up to the HLS tool to define the architecture itself based on the application, but architecture definition itself is constrained by available components offered by the target hardware (the FPGA board), thus from its so-called *components library*. These components libraries expose what components can be implemented into an actual piece of hardware, an FPGA for example, thus they represent the new *"assembly language"* to be emulated by HLS tool IR: exposed component can vary pretty much in complexity, ranging from simple logical gates to complex mathematical operators, furthermore, the

same component can be found for different operands' bitwidth, enabling much more complex optimizations to be carried out during the design process. It is clear now why the IR needs to be much more informative to enable a deeper optimization process and a more accurate design of the final architecture.

Back to the synthesis process, as said above, after the IR has been generated from the input, internal analysis can begin. The greatest power of HLS is the ability to deeply parallelize operations, but to do so a correct and accurate detection of parallelizable operations is a central concern: data dependencies needs to be defined, thus a Control and Data Flow Graph has to be generated. This graph is composed by basic blocks, which are sequences of instructions without conditional statements, and these blocks are connected by edges which represent the application control flow; this graph helps to detect which operations can be executed in parallel thanks to the clear definition of blocks of instructions and flows of execution. Finally, when compiler analysis and transformations have been completed on the IR it is time to generate the actual hardware design; while standard compilers translate IR to the specific assembly language of the target architecture, performing simple register allocation, the backend of an HLS tool needs to perform three strictly entangled operations before generating the output RTL design: **resource allocation**, **scheduling** and **binding**. Resource allocation consists of selecting necessary hardware components from RTL libraries to execute all operations performed by IR instructions; as said before, more components performing the same operation can be present, thus the choice depends on timing, area, and power consumption constraints. Scheduling procedure is necessary to define operations timing: each operation is associated to a clock cycle, or a subsequent number of cycles if necessary, determining actual operations parallelism; furthermore, data forwarding ability between subsequent operations is checked and load and store operations are added and scheduled when direct forwarding is not possible. Next, the binding process takes care of better exploiting available resources and optimize occupied area: first module binding is performed, which associates each operation to a functional unit considering timing and parallelism, then register binding is computed to allocate resources needed from functional units to carry out their operations and store results. It is clear how these three steps are strictly related

and should be performed together to obtain an optimal result, but this would require an excessive computational power, thus these operations are executed sequentially, first resource allocation, then scheduling and finally module and register binding, exploiting particular heuristics to estimate information which are not yet available. The process may seem rough, but it can still produce good results and generate performing architecture designs. Finally the output hardware specification is generated by the HLS tool and the process is completed: obtained hardware design can be then synthesized on an FPGA generating and uploading the actual bitstream on the board through the manufacturer software.

2.2 Interval arithmetic

The main problem to be solved by value range analysis is to find bounds for all program variables: in a program each variable can be an external input, a constant or the result of an operation between one or more variables. Given that each variable is the result of an operation, there is the need to be able to propagate bounds of the operands to the operation's result: this bounds propagation technique is exactly what interval arithmetic defines.

Standard arithmetic deals with single numbers, which are then combined through operations and produce another number as result. Interval arithmetic is an extension of standard arithmetic where the basic block is no longer a single number, but an interval of subsequent numbers enclosed between two bounds, thus operations will now combine multiple intervals to produce a new interval as a result. As it will be explained later, this thesis work will need to deal with integer values only, as computer architectures do not consider the set $\mathbb{R}$ of numbers nor set $\mathbb{Q}$, even if floating-point values are there, they are only generated by an encoding of multiple integer numbers, then the focus is moved on integer interval arithmetic in the following.

First it is necessary to point out what an *interval* is in integer interval arithmetic. An integer interval I represents the set of subsequent integers between two numbers and it is defined as follows:

$$l, u \in \mathbb{Z}, \quad I = \{x \in \mathbb{Z} | l \leq x \leq u\}$$

Interval I is composed by all numbers between l and u and l and u themselves, which are then called respectively lower and upper bound of the interval I. Given the set definition of an integer interval, which is pretty long to write, it is convenient to give a more concise representation for intervals, which is the one that follows:

$$I = [l, u]$$

In this compact definition of interval I, symbols l and u still represent lower and upper bounds respectively, while square brackets represent the fact that given bounds are contained inside the interval. Furthermore to easily represent lower and upper bound of a range I it is possible to write $I_\downarrow$ and $I_\uparrow$ respectively.

Now it is possible to give a definition of what is an operation between two intervals and what is its result: applying an operator to two intervals gives, as a result, another interval which is the union of results obtained applying the same operator from standard arithmetic between each of the numbers from the two operands:

$$\forall x \in I_x,\ \forall y \in I_y \ : \quad x \odot y = r \in I_r = I_x \odot I_y$$

While computing I_r could be done applying operator $\odot$ to every single combination of numbers from I_x and I_y, main advantage of interval arithmetic is that it is defined to avoid this. To compute a valid result it is necessary to manipulate only bounds of intervals of the operands. As an example, the addition operation between two intervals is defined as follows:

$$[l_1,\ u_1] + [l_2,\ u_2] = [l_1 + l_2,\ u_1 + u_2]$$

Thanks to interval operators it is possible to reduce the computational complexity of interval operations and make it constant. It may not be so simple to obtain the same result for more complex operations, but it is always possible to reduce the computational effort and obtain a valid result with the correct computation on interval bounds only, thus in constant-time independently of the size of the intervals considered.

2.3 Floating-point arithmetic internals

While binary integer arithmetic goes directly to the computation of result from given operands, standard IEEE 754 floating-point representation requires many checks and processing before and after the actual result computation takes place. Binary integers' straightforward representation enables as well a simple arithmetic architecture, which can be directly applied to input operands, without the need to decode contained numerical information: this brings great computational performance at the cost of a relatively small range of possible numbers to be represented. When a wider numerical range is required, floating-point representation is best suited to solve the problem, but with some cost in performance due to the added representation complexity.

31	30 … 23	22 … 16 | 15 … 8 | 7 … 0
S	E E E E E E E E	M M M M M M M M M M M M M M M M M M M M M M M
	Exponent	Significand

Figure 2.1: IEEE 754 32bit floating-point representation

Taking as an example a 32bit IEEE 754 floating-point, it can be split in three ranges representing sign (1bit), exponent (8bits) and significand (23bits), as shown in Figure 2.1, which become parameters in the following formula:

$$(-1)^S \times 2^{(E-127)} \times (1 + M/2^{23}) \tag{2.1}$$

Thanks to this particular encoding it is possible to represent a much wider range of numbers, with the addition of some special numbers, such as positive and negative ∞, NaN, and denormal numbers through the particular encoding of exponent and significand bits. Because of this complex representation, each floating-point operation needs to *unpack* its operands and check for special cases described above before eventually performing the actual mathematical operation between the operands. At this point the obtained values for the sign, exponent and significand need to be *packed* back together to form the output floating-point representation. These pre-processing and post-processing operations are called respectively *unpacking* and *packing* and they consist in some bit-shifts and logical operations to respectively extract and merge sign, expo-

nent and significand values from/to the input/output representation.

It is now clear why floating-point numbers require a much bigger effort to be computed than binary integers: because of this complexity most of the actual CPUs have dedicated functional units to perform floating-point operations on standard IEEE 754 32bits, 64bits and even 128bits representations. The worst downside of these dedicated floating-point FUs is that they force the program to use one of the supported representation format, even when less precision or numerical range would be acceptable for the application: this issue is not addressable in standard CPUs, but it is of main interest in FPGAs, which enable to design ad hoc FUs based on the application needs. In FPGA design it is possible to choose the most suitable representation needed for numerical operands, effectively tweaking exponent and significand bitwidths, and design perfectly sized floating-point FUs. It would be possible too to remove unneeded special numbers, as ∞ or NaN, to streamline the logic avoiding related checks and special behaviors.

Chapter 3

State of the art

The following will explain the range analysis problem and currently proposed algorithm and implementations to solve it: since it is not a new problem, many approaches have been proposed through the years, here only the most significant ones are reported. Finally a brief introduction to BitValue inference is present, because of its complementary role about program variables' state definition and proven enhancement brought by its application along with range analysis.

3.1 Value Range Analysis

Value range analysis is the problem of finding lower and upper bounds to numerical values that variables can assume during program execution: this type of analysis has been know for quite a long time, since the '70s, so it has been already implemented and used to statically detect security issues, to avoid integer overflow checks, to remove array bounds checks and to enhance dead code elimination procedures. Knowing what values will be inside a certain variable at compile-time can help to perform much deeper and precise optimizations, but it can be quite difficult to compute and many methods are there to carry out this operation. Two main approaches can be used to achieve a valid result for value range analysis: a static analysis, which only requires the application to be analyzed at compile-time, and dynamic analysis, which requires instead the

application to be examined at runtime to extract value range information which is then used to support a second compilation. Both these approach can be useful and effective, but they have different requirements and different outcomes in terms of computational complexity, result accuracy and final application execution correctness.
Existing approaches for both cases will be discussed in the following, pointing out strengths and downsides of each.

3.1.1 Static Range Analysis

Performing a static analysis means that all information has to be gathered from the application specification coming from the user, thus it is necessary to rely on some sort of abstract interpretation of the code which can produce these information. The most common way to extract value range information from the specification is to read instructions into it as constraints: translating every instruction into a constraint which puts used variables in relation between each other. There can be other ways to generate constraints on program variables, such as considering conditional statements to produce scoped constraints depending on the branch taken after the condition is evaluated, or even more complex procedures, such as exploiting pointer analysis to deduce variable relationships. Independently of methods used to obtain these range constraints on program variables and despite their complexity, what discriminates the most between static abstract interpretation methods is how constraints are merged and solved; there are mainly two possibilities: to initialize a Satisfiability Modulo Theory (SMT) solver and let it do the job or to design an ad hoc algorithm to solve the constraints' system.

While it may seem a pretty straight forward decision to take advantage of an SMT solver, it has to be considered that such a tool is supported by a really computational intensive backbone; as pointed out in [1], where an SMT solver has been configured to solve value range analysis constraints' system, this method is the most accurate when looking at results, but because of the inner complexity it is viable only when program variables' set is relatively small. SMT solver method's performances heavily degrades with the growth of pro-

gram variables number leading to unacceptable computational requirements for large programs, thus it can not be considered as a general solution to the value range analysis problem.

Conversely designing an ad hoc algorithm to handle the abstract interpretation can be an incredibly performing and scalable approach while still achieving reasonably precise results. To define an efficient and accurate algorithm will not be as simple as feeding an SMT solver, but the effort will be worth it, as shown by Campos et al. [4], who customized an abstract interpretation method from Cousot and Cousot [5] to adapt to value range analysis problem and successfully solve it in linear time complexity with respect to the number of program instructions. This excellent ad hoc solution is capable of computing quite accurate results and scales well even with large applications: the main idea of the algorithm is that of representing the problem as a graph where nodes define variables and constraints and edges represent relationships between them; thanks to this intuition it is possible to drive the computation and consistently cut its complexity. While the core idea is quite simple, there are many more aspects related to constraints definition, graph generation and about the actual solution algorithm to be discussed, thus the full algorithm will be exposed in detail in section 3.2.

3.1.2 Dynamic Range Analysis

A simple and extremely effective solution to the value range analysis problem is to dynamically detect value ranges at runtime and use them to optimize the application at compile time. To gather such accurate and specific information about the runtime state of internal variables it is necessary to have the capability to perform this profile operation on a running application or it is possible to emulate instructions in software and collect information from that. In both cases the application is first compiled without any value range based optimization, then it is executed with a given set of inputs while value range information is collected, finally gathered information is exploited during a second compilation process, which produces the optimized version of the application. Following this procedure much more aggressive optimizations are carried out at the cost

of general application correctness: as stated before the information gathering process is performed running the application on a given set of inputs, which is often quite small compared to the full set of possible inputs, consequently the optimized application generated will not be generally correct, but it will be valid only for tested inputs. As just outlined the main downside of a dynamic analysis approach is the loss of correctness, thus this method is only suitable for scenarios where an application receives predictable input values which can be as well used during the analysis procedure to ensure the optimized application will be correct on those inputs. However a much better performance gain can be achieved with dynamic analysis, when applicable, as shown by Gort and Anderson [6], who point out an average 25% improvement over the static analysis implementation. Another similar approach has been proposed by Roy and Banerjee [10]: in this case, a dynamic value range analysis is applied onto a MATLAB application to detect ranges of floating-point variables; this information is then exploited to support a quantization algorithm, which converts the floating-point representation into a fixed-point one, trying to minimize the error. Furthermore, an enhanced version of this last algorithm is discussed in [8], where both floating-point and integer variables are considered by the dynamic value range analysis, but it is still underlined by the authors how resulting applications are correct only for a restricted set of inputs, not far from the input training set used during the dynamic analysis process. In conclusion, it is clear how dynamic value range analysis can be a very effective tool, but only for application domains with predictable inputs.

3.2 Abstract interpretation algorithm

As discussed previously, when dealing with a static approach to the range analysis problem, abstract interpretation is necessary to extract value ranges from the application specification. Moreover to implement an efficient and scalable algorithm to carry out necessary computation an ad hoc design is mandatory. In the following section a range analysis ad hoc approach from Campos et al. [4] will be explained in detail because of its prominent performance in terms of

both accuracy and computational complexity: the proposed algorithm grants a linear time complexity in the number of program instructions and still a quite high grade of precision in resulting ranges. The solution procedure is composed of a consistent pre-processing phase necessary to extract constraints from the application specification and to build a particular graph to represent them, after that the core resolution algorithm begins to compute value ranges exploiting the constraints graph to efficiently drive the computation. The pre-processing phase is composed of two steps: generation of an Extended Static Single Assignment (e-SSA) representation for program instructions and a subsequent instruction translation to constraints, which are then used to build a graph representation. The main algorithm is split into three operations: range widening, future resolution, and range narrowing. All these phases will be explained in detail in the following.

3.2.1 Range Definition

Before starting to navigate through the algorithm steps it is necessary to define some useful symbols and conventions. To define ranges the convention from section 2.2 will be used, thus range $I = [l, u]$ will denote the set of values contained between l and u included. In previously defined integer interval arithmetic a lattice composed by numbers in $\mathbb{Z}$ was used, but because of the actual context, it is necessary to extend the lattice with two more symbols to allow the representation of a generalized maximum and minimum respectively through $+\infty$ and $-\infty$ symbols. These generic extreme bounds are necessary to express unconstrained ranges, which include the whole set of numbers in $\mathbb{Z}$. The new lattice $\mathcal{Z}$ can now be defined as follows:

$$\mathcal{Z} = \mathbb{Z} \cup \{-\infty, +\infty\}$$

Given newly introduced symbols it is therefore required to define their behavior in operations as follows, where $-\infty < x < +\infty$:

$$x + \infty = +\infty \qquad x - \infty = -\infty \tag{3.1}$$

$$x \times \infty = sign(x) \times \infty \qquad 0 \times \infty = 0 \tag{3.2}$$

When both operands are ∞ the operation loses meaning, thus it is not useful to define it in this context.

Being necessary to define new range operators, not considered by previously defined interval arithmetic, it is useful to define ranges within the product lattice $\mathcal{Z}^2$ as follows:

$$\mathcal{Z}^2 = \{\emptyset\} \cup \{[z_1, z_2] \mid z_1, z_2 \in \mathcal{Z},\ z_1 \leq z_2\} \tag{3.3}$$

Given lattice $\mathcal{Z}^2$ it is now possible to define union operator $\cup$ and phi operator ϕ as follows:

$$[l_1, u_1] \ \cup \ [l_2, u_2] = [\, min(l_1, l_2),\ max(u_1, u_2)\,] \tag{3.4}$$

$$\phi(\ [l_1, u_1],\ [l_2, u_2]\ ) = [l_1, u_1] \ \cup \ [l_2, u_2] \tag{3.5}$$

It can be observed that the union operator can significantly reduce accuracy when intervals are not overlapping, in this case numbers contained between the two operands but not included in them are however included in the resulting interval. Finally the intersection operator $\cap$ is defined as follows:

$$[l_1, u_1] \ \cap \ [l_2, u_2] = \begin{cases} [\, max(l_1, l_2),\ min(u_1, u_2)\,] & \text{, if } l_1 \leq l_2 \leq u_1 \text{ or } l_2 \leq l_1 \leq u_2 \\ \emptyset & \text{, otherwise} \end{cases} \tag{3.6}$$

As last tool for range definition, it is necessary to set a convention to declare an uninitialized range, this can be defined by using the bottom symbol $\perp$ for both the upper and lower bounds; furthermore the phi operator ϕ needs to be extended too to allow uninitialized ranges to be considered:

$$R_u = [\perp,\ \perp] \qquad \phi([l, u], [\perp, \perp]) = [l, u]$$

All other operators do not need to consider the uninitialized range, because such a case will never be possible given the resolution algorithm explained later, which will ensure every range is initialized before being used in a non-phi op-

eration.
Given these definitions and conventions about ranges, it is now possible to proceed with adequate tools to actual constraints' definition for the range analysis problem.

3.2.2 Constraints Definition

To solve the range analysis problem it is first necessary to define the actual problem formally. Range analysis is about associating a value range to each program variable, thus it is necessary to define a mapping from variables to ranges. Let program variables be part of set $\mathcal{V}$ and define ranges from product lattice $\mathcal{Z}^2$ previously discussed: it is now possible to represent mapping I and evaluation function e from program variables to ranges as follows:

$$I : \mathcal{V} \mapsto \mathcal{Z}^2 \quad I[V] = [l, u] \qquad e : \mathcal{V} \to \mathcal{Z}^2 \quad e(Y) = [l, u]$$

Given the new set $\mathcal{V}$ of program variables, it is necessary to define its relative arithmetic as follows:

$$\forall Y, X_1, X_2 \in \mathcal{V} \mid Y = X_1 \odot X_2,\ R = I[X_1] \odot I[X_2] \implies e(Y) = R$$

Arithmetic of set $\mathcal{V}$ strictly relays on $\mathcal{Z}^2$ arithmetic defined in the previous section, in fact operations between variables will actually modify their associated range to the resulting one obtained from the required operation applied to intervals associated to operands. The just defined arithmetic on set $\mathcal{V}$ will include all operators from interval arithmetic plus the additional operators defined in the previous section, such as phi operator ϕ, union operator $\cup$, and intersection operator $\cap$, which can be all used as shown above. Furthermore, it is useful to define even operators between $\mathcal{V}$ and $\mathcal{Z}^2$ as follows:

$$\forall Y, X \in \mathcal{V},\ \forall T \in \mathcal{Z}^2 \mid X = Y \odot T, R = I[Y] \odot T \implies e(X) = R$$

Now it is possible to arbitrarily define any type of constraint between program variables in $\mathcal{V}$ and between $\mathcal{V}$ and $\mathcal{Z}^2$, thus it is time to give the formal defi-

nition for range analysis problem as follows: given a set of constraints $\mathcal{C}$ over variables set $\mathcal{V}$ with associated ranges from $\mathcal{Z}^2$, value range analysis is defined as the problem of finding a mapping I such that $\forall\ V \in \mathcal{V}\ ,\ \exists\ e(V) = I[V]$.

As stated by the definition a set $\mathcal{C}$ of constraints has to be given to solve the problem, thus it is necessary to derive them from the input specification, as the current algorithm is based on a static analysis approach. The application specification will then be accessible from its IR produced inside the compiler, as previously discussed, which is in SSA form: this particular form of representation enables to directly map each IR statement to an equivalent constraint for range analysis problem, thus the whole specification can be translated to the needed set $\mathcal{C}$ of constraints. An example of this process is shown in Figure 3.1, where the input specification (Figure 3.1 (a)) is first translated into compiler IR in e-SSA form, an extended SSA form which will be explained just below, from which the constraints set can be directly derived through a one-to-one mapping from IR statements; Figure 3.1 (d) shows a possible solution to the constraints set for completeness.

3.2.3 Extended SSA

As outlined in section 2.1.1, compilers commonly use Single Static Assignment (SSA) as the standard form for their intermediate representations, because of its clarity in highlighting data dependencies. This peculiarity of SSA form is really useful when it comes to constraints definition for range analysis, but it is not sufficient to extract all possible information from program instructions. To better expose data flow split due to conditional branches it is useful to write intermediate representation in Extended SSA form, as defined by Bodik et al. [2]. This particular form requires each variable which is used both inside a branch condition and after it to be renamed in branches after the conditional statement. This means a new variable is added after each branch statement to shadow all usages of branch condition variables: the new variable will be initialized as equal to its associated branch variable and will replace it in all subsequent use statements. Thanks to this variable renaming the data flow split produced by the branch is explained through the renaming instruction into the IR; this allows

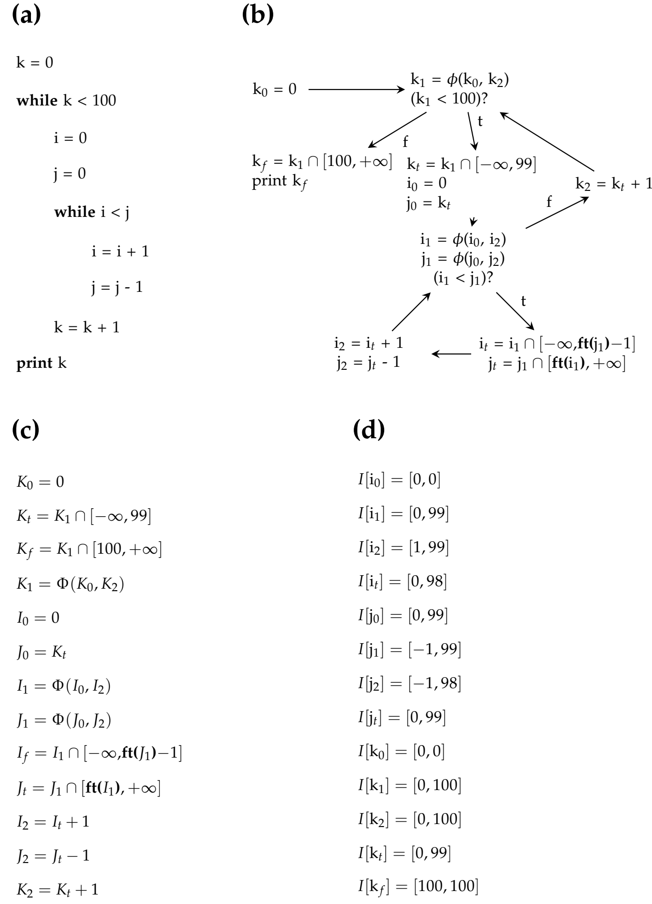

Figure 3.1: (a) Example program. (b) Control Flow Graph in e-SSA form. (c) Constraints extracted from the program. (d) Possible solution to the range analysis problem.

to express this splitting constraint even during constraints generation for range analysis.

An example of e-SSA form is shown in Figure 3.2 (c), where it is clearly visible how branch variable renaming helps range propagation through the two branches: variable v_0 is redefined in each branch after the conditional statement, so that these two new variables can be associated with the two different parts of v_0's split range.

As seen from the example above, live range split when considering a variable and a constant, such as $v_0 > 0$, is pretty straight forward, but when two variables are compared by a conditional statement a further step to enhance ac-

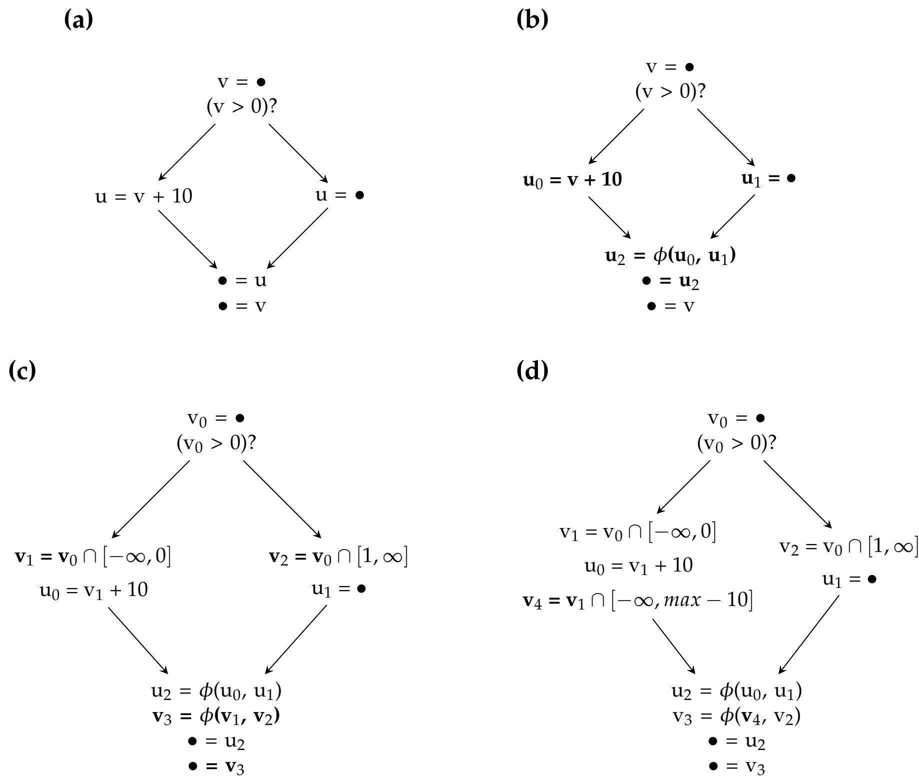

Figure 3.2: (a) Example program. (b) SSA form. (c) e-SSA form. (d) u-SSA form.

curacy is necessary. Starting from the example in Figure 3.2 (c) it is possible to say that for a comparison such as v>c, where c is a constant, the live range split is represented by range $[-\infty, c]$ in the false branch and by $[c+1, +\infty]$ in the true branch. That is not possible when constant c is substituted by variable w: considering now the new comparison v>w it is still possible to define v and w associated ranges as respectively $v \cap [-\infty,$ **ft(w)**$-1]$ and $w \cap [$**ft(v)**$+1, +\infty]$ for false branch and $v \cap [$ **ft(w)**$, +\infty]$ and $w \cap [-\infty,$**ft(v)**$]$ for true branch. Because it is not possible to know the actual value of variables v and w before constraints computation takes place during the resolution phase of the algorithm, the new concept of *futures* has been introduced by Campos et al. [4]: notation **ft(**v**)** seen above is used to represent the value which variable v will have once the analysis has defined it, thus when the algorithm will compute a valid range for variable v the correct value will be replaced to each occurrence of **ft(**v**)** to complete the so-called *future range*. Thanks to *future ranges* it is possible to keep

the relationship created by a conditional statement between two variables and avoid precision loss by effectively propagating results once they are ready; in the following the resolution of *futures* will be explained in detail as part of the resolution algorithm. An actual example of *future ranges* is shown by Figure 3.1, where ranges for variables i and j can be exactly derived by range analysis thanks to this new concept.

A further step in live range split enhancement can be performed to highlight constraints implicitly subsumed by binary integer mathematical operations performed by IR statements. This type of inference is thus applicable only to programs that are not exploiting integer overflow as a feature: in these cases it is possible to deduce additional information about operands after each usage in mathematical operations. To correctly propagate these constraints it is necessary to redefine each operand variable after each use statement, like e-SSA does after branch statements for conditional variables used there: this new representation is called u-SSA form and subsumes e-SSA too. Thanks to u-SSA it is possible to expose constraints defined by mathematical operations on operands, exploiting the fact that overflow would cause an exception at run-time, thus it is not an acceptable case during the correct run of the application. Figure 3.2 (d) shows an example of u-SSA form: if instruction $u_0 = v_1 + 10$ does not result in an overflow, causing the program to terminate, it is correct to say $v_4 = v_1 \cap [-\infty, max - 10]$, where *max* is the maximum value for v_1 integer type (ex. 127 for an 8bit signed integer). It is again clear how renaming of v_1 into v_4 helps to associate the new range constraint to variable v generating a new data flow after the considered instruction.

Exploiting these new representation forms can significantly improve results of range analysis because of the increased number of constraints considered, but the more the constraints the higher the complexity, thus it is important to analyze performance impact of these transformations compared to their effective contribution to the final result precision. Because range analysis is implemented inside a compiler it is assumed that SSA form comes for free, as it is already generated by the compiler which is using it in the standard compilation process, whilst it is not the case for e-SSA form and u-SSA forms which should be specifically generated to support the analysis and would affect compiler ef-

ficiency because of the added statements. Considering a generic program, it is reasonable to say branch instructions are about 15% of total and branch conditions will contain at most two conditional variables (example: a > b, x != 0), thus moving from SSA to e-SSA will increase program size of approximately 30%. Considering now numerical operations in a generic program to be about 50% of the total and saying that most of them are binary operations, it is reasonable to say u-SSA form will increase program size of 130%. Finally it is important to consider new constraints introduced in the range analysis constraints set: about two constraints per branch with e-SSA and about two constraints per numerical operation with u-SSA. In conclusion it has to be taken into consideration that e-SSA will increase program size of about 30%, while u-SSA will increase the size of 130%, as well as constraints set size, which is increased of the same percentage because of the one to one relation between instructions and constraints.

3.2.4 Solving Range Analysis Problem

Once the constraints set $\mathcal{C}$ has been populated with constraints gathered from IR statements it is time for the last pre-processing step before the core resolution algorithm can be applied. Starting from the formal problem definition given in section 3.2.2 it is necessary to build a so-called constraints graph, as defined by Su and Wagner [11] to move from a generic set of constraints $\mathcal{C}$ to an organized structure which can be used to consciously drive the computation through the final solution. The constraints graph is composed of nodes, which are variables from set $\mathcal{V}$ and constraints from set $\mathcal{C}$, connected by edges based on constraints relationships: if a constraint C defines variable V, then edge $\overrightarrow{CV}$ is added to the graph, conversely if variable V is used by constraint C, then edge $\overrightarrow{VC}$ is added. Furthermore this graph representation has been enhanced by the addition of *control dependence* edges: this new type of edge is used to explicitly represent the relation between the future variable and the future range associated to it, thus when variable V is used by future **ft(**V**)** an edge from V to the future range containing **ft(**V**)** will be in the graph. An example of constraints graph built starting from constraints set in Figure 3.1 is shown in Figure 3.3: solid edges

represent standard data dependencies as defined above, while dashed edges represent control dependencies generated by futures.

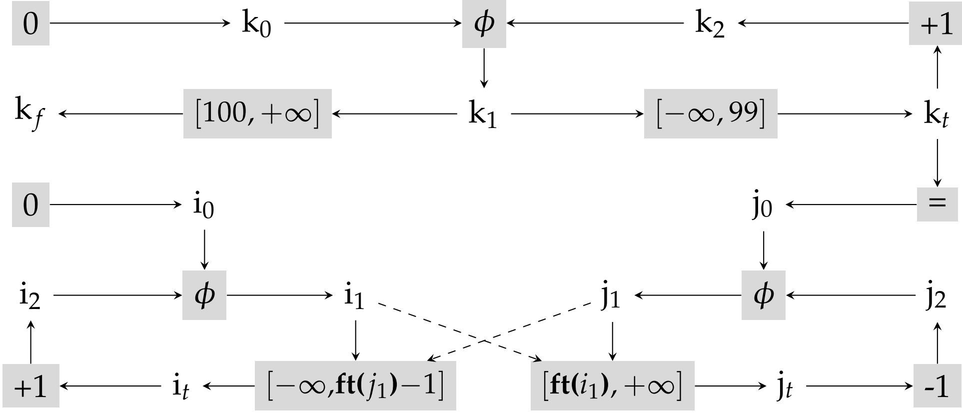

Figure 3.3: Constraints graph genereted from constraints in Figure 3.1 (c)

After constraints graph has been built from constraints set, the last step is necessary before the actual resolution algorithm can be applied to produce the final result. It is now necessary to check for Strongly Connected Components (SCCs) inside the graph, then once they have been detected, considering them as supernodes, a topological ordering of nodes has to be computed: this order will drive the resolution algorithm through the problem as each node will be solved subsequently propagating obtained results forward until the last node in the topology where the computation ends. Given this topologically ordered digraph composed by nodes and SCCs, it is now possible to outline the resolution algorithm which will be applied to each of the nodes to produce results and propagate them through the graph until all nodes have been explored and solved. Because of the topological ordering and the SCCs supernodes, each node needs to be visited only once to correctly compute the final result: the proposed method has been described by Cousot and Cousot [5] and consists of two steps known as widening and narrowing, Campos et al. [4] have introduced an intermediate step between these two to take care of future resolutions; this particular step requires ranges of variables associated to futures to be already solved before they are needed, thus it is necessary that nodes associated with these variables are solved in advance or during the same run of the algorithm: this precedence is ensured by the presence of control dependence edges, which

force related variables to be in the same SCC or strictly before the node with the associated future with respect to the topological ordering.

Then the resolution algorithm is actually split into three steps: widening, futures resolution and narrowing; an example of each algorithm step result is shown in Figure 3.4 for last SCC of the constraints graph of Figure 3.3, starting from the uninitialized situation, where only ranges of entry points i_0 and j_0 have been computed, through the final result obtained after narrowing step.

Widening: this phase is needed to initialize all ranges of variables and determine their growth direction, if there is one. This step is performed by applying the widening operator as defined by Cousot and Cousot [5], which is described below:

$$I[Y] = \begin{cases} e(Y) & \text{, if } I[Y] = [\bot, \bot] \\ [-\infty, +\infty] & \text{, if } e(Y)_\downarrow < I[Y]_\downarrow \text{ and } e(Y)_\uparrow > I[Y]_\uparrow \\ [-\infty, I[Y]_\uparrow] & \text{, if } e(Y)_\downarrow < I[Y]_\downarrow \\ [I[Y]_\downarrow, +\infty] & \text{, if } e(Y)_\uparrow > I[Y]_\uparrow \end{cases}$$

The operator evaluates each variable and updates bounds of its associated range to meet the actual growth direction, as shown by Figure 3.4 (b): after the operator has been applied to the example it is clear that i_1, i_2 and i_t are strictly growing variables, while j_1, j_2, and j_t are strictly decreasing variables and finally i_0 and j_0 are constants, because none of their bounds have been modified by the operator. Standard widening operator shown above only uses $-\infty$ and $+\infty$ as a lattice to change evaluated range bounds, but a more accurate technique called jump-set widening, which consists in an extension of this lattice, will be shown later on. Both standard and jump-set widening operators have constant time complexity, thus widening step itself will have linear time complexity in the number of nodes contained in the analyzed SCC.

Future resolution: after widening operator has defined growth direction, it is possible to propagate range bounds required by futures to complete ranges initialization. It is reasonable to presume that future variable ranges have been already initialized at this point because of the dependence edges introduced in the constraints graph: thanks to them future variables will be part of the same

SCC where dependent variables belong. Formal future replacement rules can be schematized as follows:

$$Y = X \cap [l, \mathbf{ft(}V\mathbf{)} + c] \implies Y = X \cap [l, I[V]_\uparrow + c]$$

$$Y = X \cap [\mathbf{ft(}V\mathbf{)} + c, u] \implies Y = X \cap [I[V]_\downarrow + c, u]$$

As shown in Figure 3.4 (c), lower bound of i_1 is constant, because the variable is strictly increasing, thus it is possible to resolve lower bound future for the conditional range of j_t, similarly upper bound future for conditional range of i_t is resolved with j_1 fixed upper bound, leading to a complete initialization of all ranges.

Narrowing: after growth direction has been determined and all ranges have been initialized, it is now possible to apply constraints derived from conditional tests through the narrowing operator. A standard narrowing operator as defined by Cousot and Cousot [5] is shown below:

$$I[Y] = \begin{cases} [e(Y)_\downarrow, I[Y]_\uparrow] & \text{, if } I[Y]_\downarrow = -\infty \text{ and } e(Y)_\downarrow > -\infty \\ [I[Y]_\downarrow, e(Y)_\uparrow] & \text{, if } I[Y]_\uparrow = +\infty \text{ and } e(Y)_\uparrow < +\infty \\ [e(Y)_\downarrow, I[Y]_\uparrow] & \text{, if } I[Y]_\downarrow > e(Y)_\downarrow \\ [I[Y]_\downarrow, e(Y)_\uparrow] & \text{, if } I[Y]_\uparrow < e(Y)_\uparrow \end{cases}$$

For narrowing as well as for widening step it is possible to increment precision by using a jump-set narrowing operator with an extended lattice without changing time complexity with respect to the number of nodes contained in the analyzed SCC. In Figure 3.4 (d) it is shown how exact ranges for i_t and j_t are computed intersecting respectively i_1 and j_1 with their conditional ranges to obtain $i_t = [0, 98]$ and $j_t = [0, 99]$.

Thanks to this three-steps algorithm it is possible to solve each one of the topologically ordered nodes and SCCs and obtain a result with a linear time complexity with respect to the number of constraints graph nodes. Furthermore it has been shown experimentally by Campos et al. [4] that performing a preliminary abstract interpretation step before starting the widening phase can

lead to more accurate results and even lead to a fixed point solution making it possible to abort any further computation for the current SCC and directly pass to the next. Finally as said above it is possible to use jump-set widening and narrowing operators instead of standard ones to further improve accuracy: jump-set operators perform the same operation of standard ones, but on a lattice with more components, thus each application of the operator will result in a gradual increment or decrement of affected bounds values enabling a finer tuning along available lattice components. In Campos et al. [4] implementation the jump-set operators lattice is generated on each execution of the resolution algorithm including all program constants present into the considered SCC along with $-\infty$ and $+\infty$ from the standard operators lattice; thanks to this specific procedure it is possible to generate a lattice including only meaningful components for considered constraints, avoiding useless additions which would slow down computation for no reason.

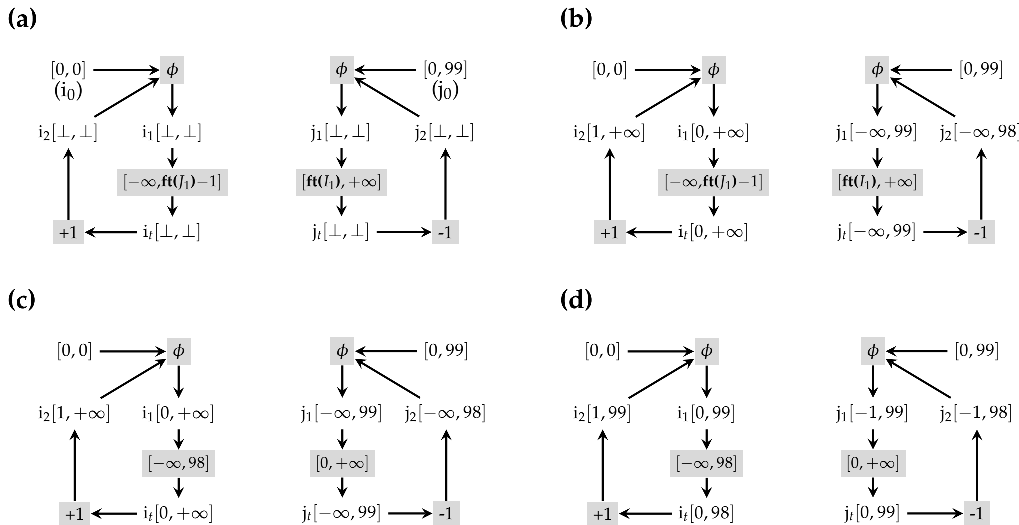

Figure 3.4: Resolution phases snapshots of the last SCC of Figure 3.3. (a) After removing control dependence edges. (b) After running the widening phase. (c) After running future resolution phase. (d) After running narrowing phase.

In conclusion, the ad hoc algorithm proposed by Campos et al. [4] to solve range analysis problem succeeds in providing a scalable and still quite accurate method to find a solution to the value range analysis problem.

3.3 Bit Value Inference

A complementary analysis which is often associated with the implementation of range analysis is the bitvalue inference: the algorithm described by Budiu et al. [3] is designed to detect useful and useless bits in program variables enabling a more efficient use of hardware resources. The algorithm is based on an iterative data-flow analysis approach on a given lattice of four symbols: 1, 0, x (don't care), u (don't know).

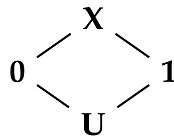

Figure 3.5: The bit values lattice. The ordering is defined by the "information content".

The data-flow input is fed with $\langle u \rangle$ bits and output bits are computed using direct transfer functions according to data-flow operators (forward propagation), then computed output bits are backward propagated through operators' inverse transfer functions back to the input: this forward/backward operation is iterated until no change in bits value is detected. Following this iterative pattern a representation in the 1/0/x/u lattice will be computed for all variables in the data-flow showing exactly which one of the bits is constant, useless, or useful. More specifically forward propagation helps in defining *don't care* bits through the output variables, while backward propagation task is to discover *don't cares* of the inputs starting from previously computed outputs: to correctly perform this process without modifying the results it is necessary that propagation functions are monotone and conservative, as shown by Budiu et al. [3]. An example of forward and backward propagation for a C function is shown by Figure 3.6.

Because of the bitwise approach, this analysis is not aware and can not compute program variables bounds, which instead could be useful too in the effort to reduce computation: range analysis is committed in performing exactly this task, so these two types of analysis match perfectly to produce a complete variable state analysis. Considering an iterative compilation process it is also

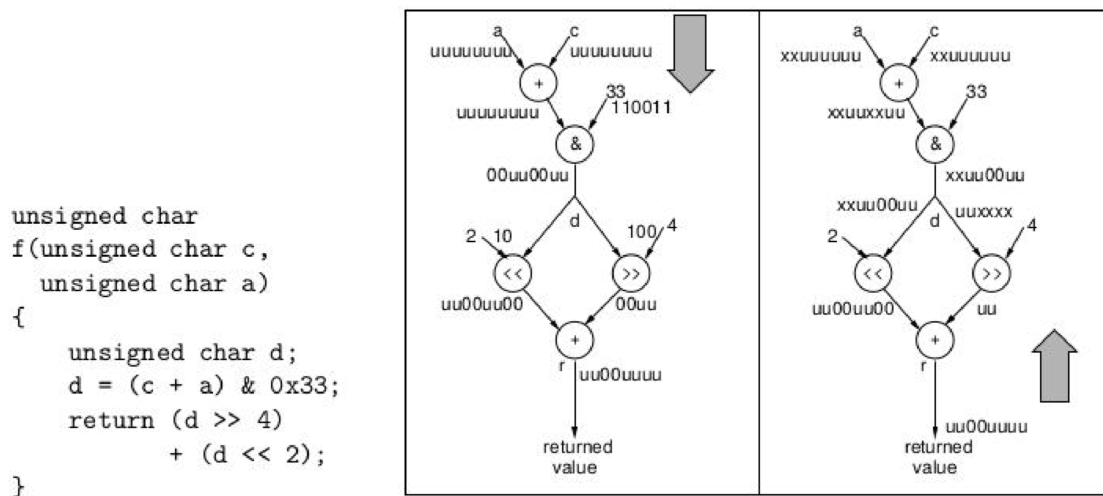

Figure 3.6: A C function and the associated data-flow graph. The types inferred by forward(backward) propagation are shown in the left (right) figure. We assume that a char has 8 bits.

possible that these two analyses cooperate leading to a more accurate result. While BitValue inference is more suitable to detect useless and constant bits, range analysis is better when variables bounds are needed, such as in conditional checks for branches, array bounds, and integer operations.

3.4 Conclusions

As seen in this chapter, many implementations of range analysis have been tested, from simple cases as in [4], where range analysis is implemented from scratch, or in [1], where the analysis is implemented through an SMT solver, to more complex ones where it is implemented and paired with BitValue inference, as in [6, 3]. However current implementations are mostly based on an Intermediate Representation which is not built to represent a reconfigurable architecture, but a standardized and fixed one, thus fine-tuning and internal manipulation of floating-point operations is not possible, because most of the underlying complexity of the hardware is hidden into instructions themselves, which are already fixed in standard CPUs. While some fixed-point tweaking approaches have been tested, such as the one in [1], trying to push the limit

of current methods, they are still black-box approaches, which are not effectively dealing with the underlying hardware directly. As will be explained in the following, the purpose of this thesis work is to apply some of these methods inside an HLS tool to exploit an IR nearer to the hardware and achieve a better and finer grade of manipulation over the output hardware design.

Chapter 4

Proposed solution

Starting from the state of the art algorithms, this thesis work approach will be explained, pointing out new challenges encountered and new features implemented to set up the ultimate algorithm to solve the value range analysis problem. Each aspect of the implementation will be discussed from the main algorithm choice to smaller implementation details, which will grant more accurate results with the least complexity increment possible. Finally, a peek on the new floating-point customization will be there, explaining its capabilities and interesting features.

4.1 Range Analysis at HLS Level

Currently explored range analysis implementations are mainly focused on a high-level intermediate representation directly generated from common compilers: while it already brings great improvements to exploit the analysis at this level, this approach could be pushed even more applying range analysis on a lower-level intermediate representation. Exploiting the HLS process it is possible to have a deeper and more accurate view of the operations performed by a program, so that even improvements can become more precise and specific. The purpose of this thesis work is to explore what enhancements can arise from the application of range analysis at HLS level, with a specific focus on floating-point operations. While high-level representation hides these types of opera-

tions inside single instructions, as explained in section 2.3, HLS design tools are indeed actively involved in the design of floating-point functional units, thus their intermediate representation is actually defining each internal step of these operators: thanks to this internal view it is possible to enhance range analysis and apply deeper optimizations based on input operands value ranges, such as special floating-point symbol removal, constant operators optimization and other customization over standard IEEE 754 representation, such as significand or exponent bitwidth manipulation. In the following it will be explained how value range analysis has been integrated into PandA Bambu HLS tool and coupled with the already implemented BitValue inference to verify potential improvements given by the application of such an approach inside an HLS tool; furthermore a new method to customize and manipulate standard IEEE 754 floating-point representation with user-defined hints will be discussed.

4.1.1 Design Choices

There are two main decisions to take before starting the actual implementation of this thesis work: choose an HLS tool where it is possible to implement a value range analysis algorithm and what type of algorithm to implement. The first choice quickly falls on the home competitor in the field of HLS tools: PandA Bambu is an HLS tool entirely implemented and maintained at Politecnico di Milano by professors and researchers, it already supports and integrates many versions of GCC and LLVM compilers, along with many optimizations to translate OpenMP descriptions and some CNN descriptions to hardware designs, furthermore an implementation of the BitValue inference, as described by Budiu et al. [3], is present in this tool, thus it can be effortlessly entangled with value range analysis to contribute for a better overall optimization. Because of its wide range of input specifications and the solid framework it offers, PandA Bambu is the best place for this thesis work to be integrated into and tested.

As for the choice of a method to implement value range analysis, it has to be considered what type of programs this work will target: given a generic approach is preferable, it has to be considered that this thesis work is mainly focused on enhancement and manipulation of floating-point functional units,

which are quite complex thus it is necessary that the implemented algorithm can handle large numbers of program variables easily. Because of these considerations an SMT solver approach has been excluded: while it offers the best precision it is not scalable, thus it is not viable in this case. Furthermore, a static approach is preferred over a dynamic one for the moment, because it is important for this work to explore what improvements can be achieved without sacrificing program correctness: a dynamic approach could give better performance for sure, but it imposes a loss in generality on the set of inputs of considered applications, which is not an intent of this thesis work. Finally, while it could have been an option to implement a new algorithm from scratch, it is not worth it because of the ad hoc method proposed by Campos et al. [4]: the algorithm they presented already ensures a linear time complexity in the number of program instructions and quite a good accuracy in results, thus it is exactly what requirements for this thesis work ask for. Furthermore, this ad hoc solution will be easily customizable to consider floating-point variables too and manipulate them as necessary to explore the efficiency of modified IEEE 754 floating-point representations. In conclusion, it has been decided to implement a modified version of Campos et al. [4] algorithm inside the PandA Bambu HLS tool to explore and test the capabilities of a value range analysis algorithm including floating-point manipulation and BitValue inference interaction. In the following the actual implementation of such an algorithm will be discussed in detail, highlighting similarities with existing approaches and new features added, and describing how existing BitValue inference algorithm has been entangled in the optimization process; finally, the new method introduced to customize IEEE 754 floating-point encoding through input specification will be presented.

4.2 Value Range Analysis Algorithm

As explained before, an enhanced version of Campos et al. [4] algorithm for value range analysis will be implemented in this thesis work: the main algorithm outline will actually be the same as explained in section 3, but some additions have been implemented trying to optimize the overall computation and to

introduce the floating-point handling features required. Each step of the algorithm will be explained in the following, pointing out the main implementation details and discussing newly introduced features: a formal definition of range and constraint will be given, then pre-processing steps, extended SSA and constraints graph generation, and core resolution algorithm will be discussed.

4.2.1 Range representation

Starting from the range representation defined in section 3.2.1, where ranges belong to set $\mathcal{Z}^2$, it is first necessary to extend this set to include a new value as part of the range. While the mathematical meaning of bounds is still the same, when dealing with binary integers from a program it is necessary to define even their bitwidth, which is necessary to carry out some specific bitwise operations such as bit shift or truncation and to consider even overflow, which can occur in fixed binary integers. Thus it is required to compose a new set $\mathcal{T}$ defined as follows:

$$\mathcal{T} = \{\emptyset\} \cup \{[z_1, b, z_2] \mid [z_1, z_2] \in \mathcal{Z}^2, b \in \mathbb{N}^*\}$$

Given $\mathcal{T}$ it is clear how z_1 and z_2 still represent respectively the lower and upper bounds of the interval, while the new component b represents range bitwidth; again lower and upper bounds of range R can be respectively referred to as $R_\downarrow$ and $R_\uparrow$, while bitwidth will be R_*. As well as for set $\mathcal{Z}^2$ a new arithmetic needs to be defined for set $\mathcal{T}$: that will not be a standard arithmetic anymore, because of the binary integer behavior which needs to be reproduced. As for program variables, arithmetic of set $\mathcal{T}$ only allows operations between ranges with the same bitwidth value, furthermore, a modulo arithmetic is applied on ranges, where the modulo value is given by the bitwidth of the operands. Thus modulo arithmetic of set $\mathcal{T}$ can be defined as follows:

$$\forall R_1, R_2 \in \mathcal{T}, \quad R_1 \odot R_2 = \begin{cases} R_1 \overset{b}{\odot} R_2, & \text{, if } R_{1*} = R_{2*} \\ undefined & \text{, otherwise} \end{cases} \tag{4.1}$$

The $\overset{b}{\odot}$ symbol in 4.1 represent the equivalent modulo operator of $\odot$, where the modulo value is given by b, which will be equal to the bitwidth value of the operands; furthermore resulting range from $\overset{b}{\odot}$ operator will still have b as bitwidth value. Intended modulo arithmetic for the set $\mathcal{T}$ is again non-standard given the fact it has to reproduce exactly binary integers' behavior which has a different modulo depending on the sign of the operand: as an example, an 8bit binary integers allows a value of -128, while it does not allow a value of 128 which would result in an overflow. Because of this particular complexity brought by the custom modulo arithmetic necessary for set $\mathcal{T}$ to perform correctly, this section will just report some simple examples, without giving complete definitions for all arithmetic operators available to avoid unnecessary complexity and still keep the focus on value range analysis algorithm.

The following example will show the difference between the plus operator applied between two ranges from $\mathcal{Z}^2$ and the equivalent custom modulo operator applied on two equivalent 8bit ranges from $\mathcal{T}$:

$$[56,\ 126] + [3,\ 3] \ = \ [59,\ 129] \tag{4.2}$$

$$[56,\ 8,\ 126] + [3,\ 8,\ 3] \ = \ [-\infty,\ 8,\ -127] \cup [59,\ 8,\ +\infty] \tag{4.3}$$

It is quite clear as result of 4.3 is different from that of 4.2, in fact the overflow behavior of 8bit binary integers is not considered by the standard arithmetic, while the same instance of the operation with ranges from $\mathcal{T}$ is indeed generating the expected overflow behavior of an 8bit signed integer variable. Thanks to this example is thus possible to outline a common problem which arises from the application of modulo arithmetic: when an overflow occurs, such as in 4.3, the resulting range is composed by two ranges representing the two sections of the actual result, in 4.3 the left part of the result represents the *"overflowed"* range while the right part the standard range produced by sum operator; because of this *"split"* result it would not be possible to represent it with a single range from set $\mathcal{T}$, thus a new notation has been defined to allow a compact definition of such a case with a single range. While set $\mathcal{T}$, as well as set $\mathcal{Z}^2$, is representing inclusive ranges, where numbers included into an interval goes from the lower

bound to the upper bound, as previously defined, it is convenient to define an extended set $\mathcal{T}_A$ to include the *anti-range* notation as follows:

$$\mathcal{T}_A = \mathcal{T} \cup \{\ )z_1, b, z_2(\ |\ )z_1, b, z_2(= [-\infty, b, z_1 - 1] \cup [z_2 + 1, b, +\infty]\}$$

Arithmetic of set $\mathcal{T}$ needs to be extended as well to consider this new notation too. Thanks to the *anti-range* notation it is possible to represent exclusive ranges as well, where the range enclosed between lower and upper bounds is excluded from the actual represented range: given range $)l, b, u($ the actual set of numbers considered by this range is the full set considered by bitwidth b without the range of values represented by $[l, b, u]$. Thus given new set $\mathcal{T}_A$, it is possible to rewrite 4.3 as follows:

$$[56,\ 8,\ 126] + [3,\ 8,\ 3]\ \ =)-126,\ 8,\ 58($$

Furthermore *anti-range* notation allows a better live range split which can improve range analysis accuracy in some cases, as it will be explained later. As far as it is concerned for ranges of integer variables, set $\mathcal{T}_A$ as it has just been defined, with its related arithmetic, is sufficient to successfully support value range analysis algorithm, thus it is now necessary to define another ranges set which will handle floating-point variables.

It is then necessary to define product set $\mathcal{P} = \mathcal{T}_A^3 \times \mathbb{N}^*$ as follows:

$$\mathcal{P} = \{[S, E, M, b]\ |\ S, E, M \in \mathcal{T}_A,\ b = S_* + E_* + M_*\}$$

A floating-point range from set $\mathcal{P}$ just defined is composed by three ranges from set $\mathcal{T}_A$ which are defined as S, E and M and respectively represent sign, exponent and significand components ranges, as the three parts of the IEEE 754 encoding does; furthermore the last component of the range from set $\mathbb{N}^*$ does represent the overall bithwidth value of the floating-point range. As well components of a given range R can be referred to as R_s for the sign related range, R_e for the exponent range, R_m for the significand range and R_* for the bitwidth value. Conversely from previously defined range sets, in this case, it is not necessary to define an arithmetic over set $\mathcal{P}$ because all floating-point

operations will be performed through a set of integer operations after the actual floating-point value has been decoded, it is thus necessary to define only union operator $\cup$, intersection operator $\cap$ and phi operator ϕ as follows:

$$R_1 \overset{b}{\cup} R_2 = [R_{1s} \cup R_{2s},\ R_{1e} \cup R_{2e},\ R_{1m} \cup R_{2m},\ b] \quad , b\ = R_{1*} = R_{2*}$$

$$R_1 \overset{b}{\cap} R_2 = [R_{1s} \cap R_{2s},\ R_{1e} \cap R_{2e},\ R_{1m} \cap R_{2m},\ b] \quad , b\ = R_{1*} = R_{2*}$$

$$\phi_b(R_1, R_2) = R_1 \overset{b}{\cup} R_2 \quad , b\ = R_{1*} = R_{2*}$$

As per set $\mathcal{T}$ and $\mathcal{T}_A$ any operator is applicable between ranges with the same bitwidth value only and will produce a result with that bitwidth value. Furthermore, it is necessary to define the generic view convert unary operator w in its two versions, one to convert ranges from set $\mathcal{P}$ to set $\mathcal{T}_A$ is defined as $w_T\ :\ \mathcal{P}\ \rightarrow\ \mathcal{T}_A$ while the opposite view convert operation is defined as $w_P : \mathcal{T}_A \rightarrow \mathcal{P}$ and has two versions w_{P32} and w_{P64}, representing the actual IEEE 754 32bit and 64bit encoding respectively; their behavior is described below:

$$w_T(R) = R_S \ll (R_{E*} + R_{M*}) | R_E \ll R_{M*} | R_M,\ \ w_T(R)_* = R_{S*} + R_{E*} + R_{M*} \tag{4.4}$$

$$w_{P32}(R) = [R_{[31,31]}, R_{[23,30]}, R_{[0,22]}, 32] \quad , R_* = 32 \tag{4.5}$$

$$w_{P64}(R) = [R_{[63,63]}, R_{[52,62]}, R_{[0,51]}, 64] \quad , R_* = 64 \tag{4.6}$$

Operators $\ll$ and $|$ in 4.4 represents respectively left shift and bitwise inclusive or operators from $\mathcal{T}_A$ set arithmetic and they are used there to compose the three internal ranges inside the floating-point range from set $\mathcal{P}$ to build a single range of set $\mathcal{T}_A$. Conversly in 4.5 and 4.6 notation $R_{[a,b]}$ represents the range of values of bits from index a to index b of the integer range R and has a bitwidth value of $b - a + 1$; it is clear how w_{P32} and w_{P64} operators allow to convert an integer range into a floating-point range from set $\mathcal{P}$ extracting each component defined by the IEEE 754 encoding. For clarity generic results obtained after the application of w_{P32} and w_{P64} are shown below respectively as range R_{32} and R_{64}:

$$\mathcal{R}_{32} = [\ [l_s,\ 1,\ u_s],\ [l_e,\ 8,\ u_e],\ [l_m,\ 23,\ u_m],\ 32]$$

$$\mathcal{R}_{64} = [\ [l_s,\ 1,\ u_s],\ [l_e,\ 11,\ u_e],\ [l_m,\ 52,\ u_m],\ 64]$$

As previously said, no other operator will be defined for set $\mathcal{P}$ because those just explained are sufficient for the value range analysis algorithm to work as expected, thus it is now possible to proceed to the actual algorithm discussion.

4.2.2 Generating e-SSA

First pre-processing step necessary to optimize range analysis is the generation of e-SSA form from the input program representation: as explained in section 3.2.3, a further step after e-SSA form could be u-SSA form, but it is only applicable in specific cases when overflow is not exploited, furthermore, it implies a much greater effort in terms of implementation and computation time because of the significant increment in the number of constraints. Thus e-SSA form has been chosen as base representation to support value range analysis in constraints inference, because of the lack of limitation on the input program and its good trade-off between added accuracy and complexity.

PandA Bambu is already fed with a standard SSA form of the input program, thus it is possible to avoid a full SSA computation applying only *extended* features to existing SSA representation. As previously seen, e-SSA requires renaming for conditional variables, thus it is necessary to analyze conditional statements only and luckily in PandA Bambu IR conditional branch instructions can be found only at the end of a basic block, further reducing required effort. Thanks to this representation feature, a simplified three-phases algorithm can be applied: first the dominator tree for function's basic blocks is computed from Control Flow Graph (CFG), as in standard SSA algorithms, then each basic block is visited following a depth-first search path and checking for conditional branches in the last statement, while annotating conditional variables, finally annotated variables and their relative uses are topologically ordered with respect to the depth-first search ordering acquired during the tree scan and the resulting stack is used to rename uses when necessary. Given the depth-first search following the structure of the dominator tree, the resulting ordering of statements guarantees that precedence also implies dominance, thus renaming can be performed easily proceeding through the list of annotated

variables. Thanks to this approach only conditional variables and their uses are considered during the scan, without considering the whole IR during the renaming procedure.

Proposed e-SSA form generation procedure explained above performs the IR formatting with the smallest possible overhead, while providing a significant improvement to the overall result facilitating live range splitting after conditional branches. Furthermore, thanks to the newly introduced *anti-range* notation, an even more accurate range constraint can be generated after equality and inequality conditional checks, as shown by Figure 4.1, where renamed variable x_t is now able to be constrained too, as well as variable x_f.

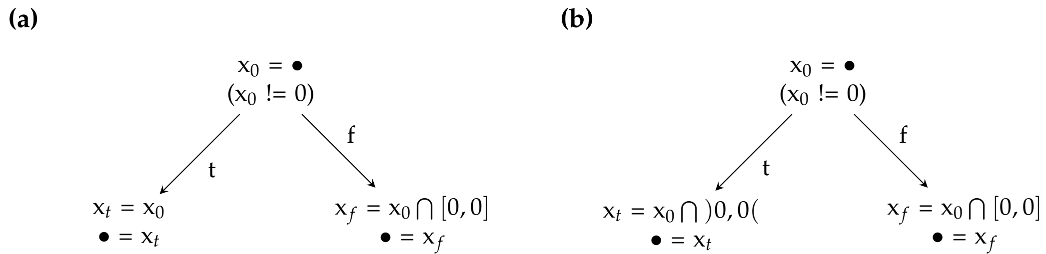

Figure 4.1: (a) Ranges from set $\mathcal{T}$ without *anti-range.* (b) Ranges from set $\mathcal{T}_A$ with *anti-range.*

4.2.3 Constraints graph definition

A second and last pre-processing phase is necessary to transform e-SSA form program instructions into the constraints graph used to solve range analysis. This process is pretty simple: each instruction is decomposed to extract its operands and result variables, which are now added to the constraints graph as nodes with an associated range and connected by an operation node corresponding to the initial operation carried out by the e-SSA instruction. Figure 4.2 shows how a little code snippet could be translated into constraints graph form.

As seen above, statements involving standard numerical and logical operators can be simply translated into their corresponding graph representation: an uninitialized range is associated with each variable, using the convention

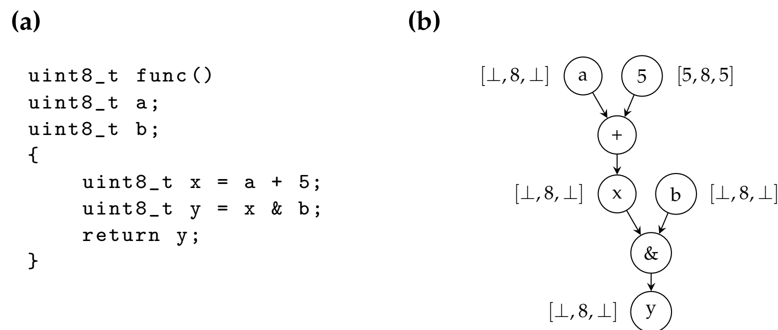

Figure 4.2: (a) Sample function code. (b) Corresponding constraints graph.

explained in section 4.2.1, and to each constant is assigned its constant range counterpart as well. The whole graph is built reading all available statements from each program function, as already pointed out in section 3.2.2, but this procedure is slightly different in some cases to enhance accuracy.

The first peculiar behavior is that enabled by e-SSA form: when a conditional variable is used again after a branch condition a new range inferred from that condition is associated with the variable; this range can be both a common range in case of a conditional check against a constant or a future range in case of two variables being compared. To correctly perform this association between conditional branch range and renamed conditional variable, while statements are scanned, branch conditions are stored and any time a renaming statement is detected stored conditions are checked to look for a valid constraint to associate: when the statement can be associated with an active conditional variable, its inferred range from the branch condition is propagated to the new variable. Thanks to this procedure live range split produced by the e-SSA form is exploited to propagate conditional constraints successfully.

As shown in Figure 4.3, sample function *cond* has a branch condition comparing parameter *a* and a constant value, which means a_0 range can be split immediately during constraint building procedure associating restricted ranges to both a_1 and a_2. While scanning the e-SSA form in Figure 4.3 (b) conditional statement $a_0 > 5$ is stored, then renaming definition of a_1 is read and a_0 is found to have live conditional constraints in a_1 scope, thus these constraints are directly applied to a_1, as shown by the resulting graph in Figure 4.3 (c). Thanks

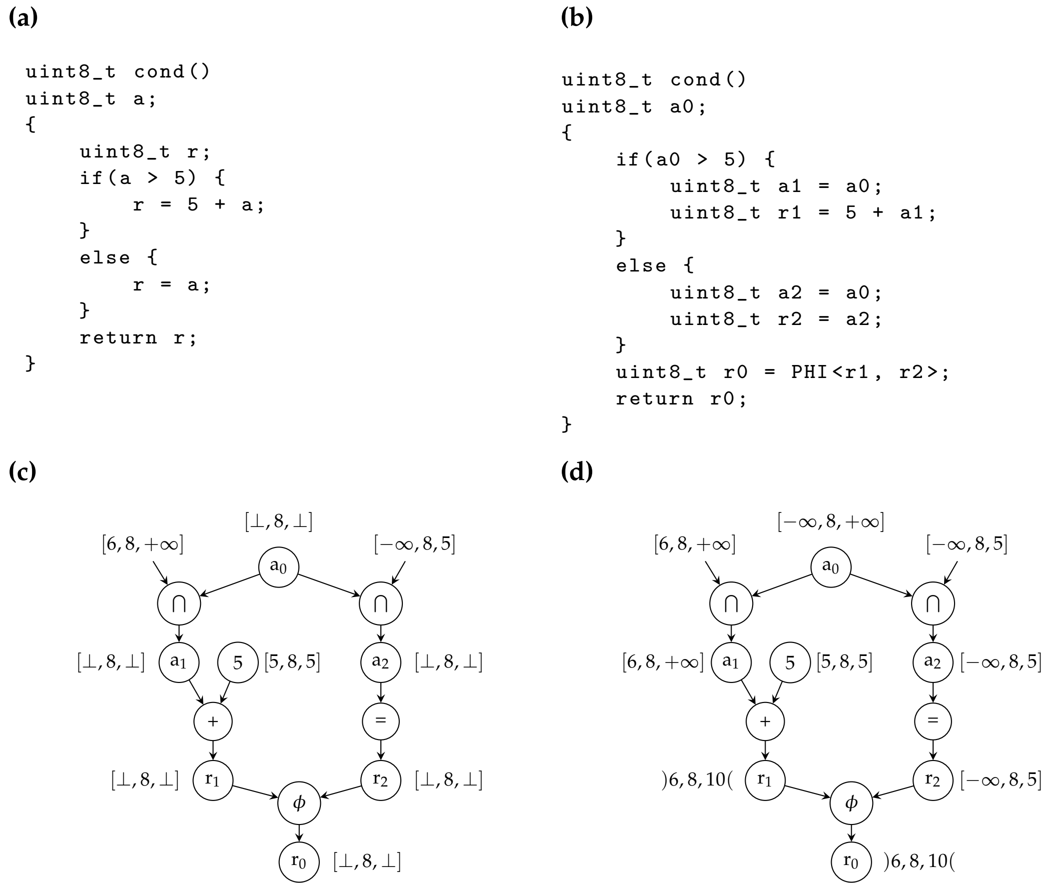

Figure 4.3: (a) Sample function code. (b) Function code in e-SSA form. (c) Constraint graph after build stage. (d) Solved constraints graph.

to this propagation enabled by the e-SSA form it is clearly visible how range of r_0 is affected observing the graph solution presented in Figure 4.3 (d) where ranges show the final results from the analysis; solving the graph without constraints inferred from conditional statement would lead to an unrestricted range solution for r_0.

While conditional statements involving constant values are immediately propagated as standard ranges into the graph, when two actual variables are compared in a statement the process slightly changes: propagation of conditional range constraint derived is delayed attaching a *future* range to the renamed variable. As explained in section 3.2.4, *future* ranges enable range cor-

relation to be expressed inside the constraint graph through *control dependence* edges associated to them.

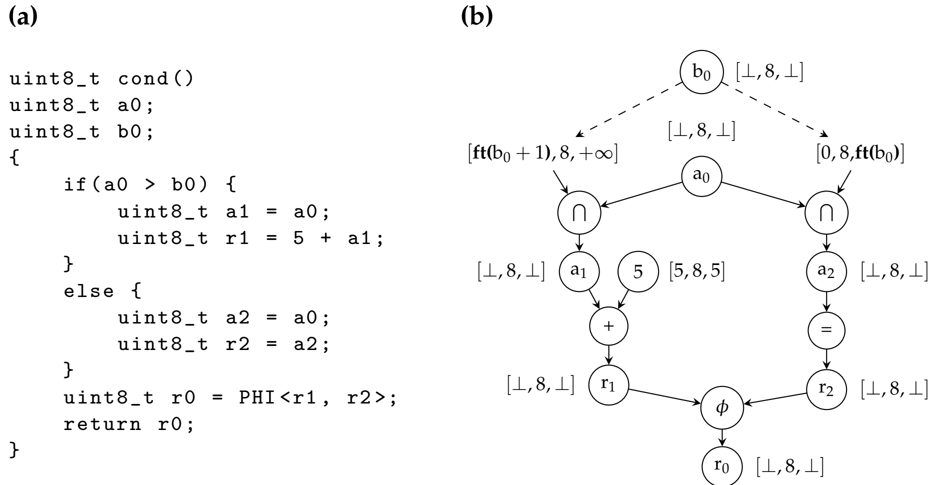

Figure 4.4: (a) e-SSA form code. (b) Constraints graph with future ranges and control dependence edges.

In Figure 4.4 a modified version of the previous example is shown, where the conditional check is now performed between two variables: it is clear how the new conditional statement involving two actual variables is translated into the new constraints graph where *control dependence* edges now appear highlighting the correlation between b_0 and conditional constraints associated with a_1 and a_2. These future ranges will be solved during the second phase of the resolution algorithm from Campos et al. [4] as explained in section 3.2.4.

All previously discussed propagation enhancements are relative to an intra-procedural point of view, thus significant improvements can be achieved by applying an inter-procedural propagation. As in most of the proposed implementations discussed in chapter 3, this implementation provides inter-procedural range propagation too: each function call is stored and used to compute a virtual phi operation to merge ranges from real parameters to compute ranges of formal parameters; the same procedure is applied to return values: each return statement is stored and their ranges are merged by a virtual phi and propagated after each function call to the return variable. In Figure 4.5 an example of inter-procedural propagation involving function *cond* from Figure 4.4 is shown.

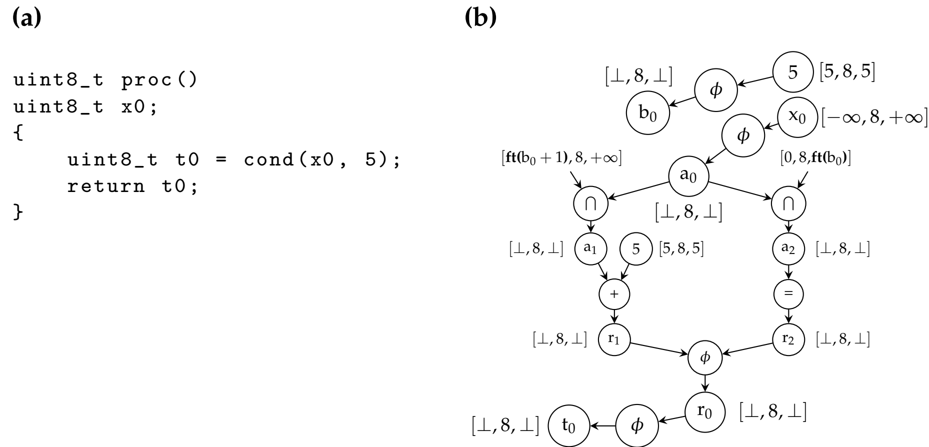

Figure 4.5: (a) e-SSA form code. (b) Constraints graph with symbolic constraints.

When the function call instruction is detected its real arguments and return variable are stored and associated with formal parameters and variable from return statements inside the called function once the constraints graph is built for it. From the constraints graph of Figure 4.5 (b) it is clear how real parameters x_0 and 5 are associated through phi operators to their formal counterparts a_0 and b_0; as well r_0, which is the returned variable from function *cond*, is connected to the actual return variable t_0 from the call statement. Thanks to inter-procedural propagation a much better and complete constraints graph can be built from the IR analysis, thus resulting in a more accurate final result.

4.2.4 Floating-point range constraints

Furthermore, it is important for this thesis work to correctly propagate ranges of floating-point representation components during *packing* and *unpacking* operations, as defined in section 2.3: to store and propagate floating-point range information a new type of range has been introduced, as defined in section 4.2.1: through this new type of range it is possible to correctly propagate range information for floating-point variables through the range analysis and perform better when handling this type of representation. To apply this new type of propagation it is first necessary to detect when it needs to be done, thus it is necessary to detect when *packing* and *unpacking* operations are performed into

the IR. Starting with *packing* operation, it is quite difficult to detect it, because such an operation is not always performed following a standard pattern, thus a generic sequence of IR instructions can not be outlined as a *packing* operation easily, in fact, the actual sequence of operations depends on the particular manipulation which has been performed on the floating-point representation. All considered it would require an unjustified effort to correctly detect this procedure, thus its operations will be just considered as common instructions by the implemented range analysis algorithm. Anyway, it is not completely lost, because exploiting the backpropagation mechanism of the BitValue inference, it will be anyhow possible to propagate information correctly through floating-point variables, as it will be explained later on in section 4.2.6. On the other hand *unpacking* operations are easier to detect: they always begin with a view convert operation followed by some shiftings and bit-masking operations to extract each component of the floating-point representation into a new variable. Then it is possible to trace operations involving the view converted value of the floating-point variable, storing this sequence of operations and detecting when an exact slice of the initial floating-point value is obtained: at this point the corresponding range from the composite floating-point range is associated directly with the new variable skipping intermediate operations, thus avoiding any information loss.

(a)

```
double d;
int64_t v = *((int64_t*)((*double)(&d)));
bool s = v < 0;
int64_t b = v >> 52;
int16_t c = (int16_t)b;
int16_t e =  c & 2047;
int64_t m = v & 4503599627370495;
```

(b)

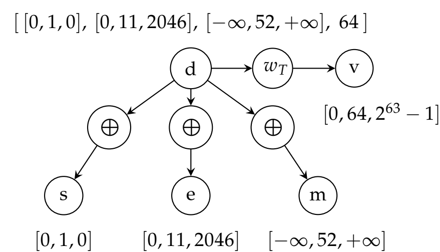

Figure 4.6: (a) Common unpacking sequence. (b) Constraints graph from the unpacking sequence.

As an example, Figure 4.6 shows a common *unpacking* procedure for a 64-bit IEEE 754 floating-point representation: starting from variable d its view converted value is stored in v, then the sign is tested and stored in s through a

conditional operation, exponent bits are extracted through a sequence of shift, cast and bitmask application into variable e, finally significand bits are stored in m. This case shows one of the simplest *unpacking* sequences, there could be many, but for the sake of range analysis it is not important the sequence, but the accuracy deriving from the fact that it is possible to skip these sequences of operations and directly assign the correct range from d to either s, e and m without losing precision on single ranges. Furthermore, the sequence of operations performed is of no importance, because thanks to the tracing procedure explained above, the constraint generation algorithm implemented can detect when any sequence of instructions results in a range containing exactly one of the range which compose the initial floating-point range. Finally the view convert operation is still included in the constraints graph, because this value could be used in other operations apart from the *unpacking* ones.

4.2.5 Resolution algorithm

Once the constraints graph has been generated, the last pre-processing step can be carried out before the actual computation can start. As defined in the algorithm proposed by Campos et al. [4], it is necessary to detect Strongly Connected Components (SCCs) inside the constraints graph, so that they can be solved as supernodes when encountered during the main computation, and to topologically order the resulting graph, now composed of both standard nodes and SCCs. Thanks to this last procedure, as explained in 3.2.4, it will be possible to efficiently drive the computation and obtain final results with a single application of the core algorithm to each node; furthermore the SCC supernodes will ensure the future resolution step can be carried out correctly, without the possibility to find uninitialized variables linked to *control dependence* edges. To perform SCCs search and their topological ordering as a single operation, a modified version of Nuutila's algorithm has been implemented, which finds SCCs and directly stores them in topological order exploiting the depth-first search ordering generated from the algorithm. Finally the core resolution algorithm from Campos et al. [4], as previously defined in section 3.2.4, has been fully implemented: proposed enhancements have been added such as jump-set

widening and narrowing operators and the preliminary abstract interpretation phase to achieve a better initialization of ranges. In the following, each step of the core algorithm will be explained again to point out the main implementation details.

As first step some preliminary abstract interpretation iterations are performed, thus each operation is evaluated propagating results as defined by range arithmetic: thanks to these preliminary iterations it is possible to initialize all ranges and in some cases even reach a fixed point solution, which allows to skip any other computation and immediately proceed with the next node. The addition of this preliminary step has been proven to be effective from experimental evidence by Campos et al. [4]. Now the main algorithm can begin, first the widening phase will continue the growth process started during preliminary abstract interpretation, but this time the actual widening operator will be used to quickly detect growth directions of ranges; after that future resolution is performed to complete the initialization of future ranges replacing futures with actual bounds from related variables; finally when all ranges have been initialized, narrowing phase is carried out to refine ranges and obtain final results.

About the widening phase, it is useful to point out what is the difference between standard and jump-set widening operators: while the standard widening operator, as defined by Cousot and Cousot [5], works on a minimal lattice, as shown in Figure 4.7 (a), thus it will always set unrestricted bounds in case of growth detection, a jump-set widening operator is stepping on a more complex lattice, as the one in Figure 4.7 (b), which has more components, thus offers the possibility of finer adjustments in between the starting value of considered bounds and the unrestricted case, eventually leading to shorter ranges with respect to standard widening operator.

The extended lattice is more accurate, but more complex to traverse too for the widening operator. To generate a functional extended lattice it is necessary to have some information about the specific problem to be solved, a generic approach is not viable, thus, as suggested by Campos et al. [4], it is possible to exploit constant values to build a custom lattice based on the input specification. While gathering all application constant values into a unique lattice could be a simple approach, the performance impact of such a lattice on widen-

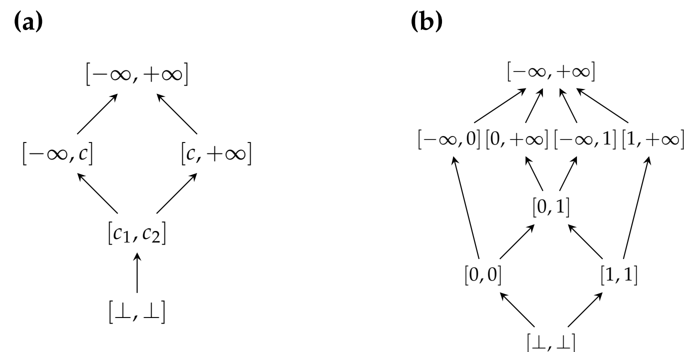

Figure 4.7: (a) Standard lattice. (b) Extended lattice.

ing operator has to be considered too: as shown by Figure 4.7 (a), the standard lattice is pretty simple, but adding just two constant values pushes complexity far away from starting point (Figure 4.7 (b)), thus considering all program constants would not have a good performance and would not be even useful because each part of the program would need only a small part of them in most cases. Because of these aspects, it is possible to exploit previously computed SCCs, in fact, they include strictly related constraints, which can be considered singularly during the resolution process, thus a significant lattice can be computed for each one of these supernodes, considering only its constant values and so reducing actual lattice size. Thanks to this approach, accuracy enhancement granted from the extended lattice is not affected, because all interesting constants for the currently analyzed slice of the graph are still considered, but lattice size is consistently reduced and so is the widening operator complexity. The same approach is used for the narrowing operator too, which exploits the same lattice previously generated for the widening operator. An example of improved accuracy given by this approach is shown in Figure 4.8 where given program is analyzed using both standard lattice from Figure 4.7 (a) and extended lattice from Figure 4.7 (b).

Range bounds computed using standard lattice fail to return a strict interval on variable *tooLong* (Figure 4.8 (c)), while with a properly generated extended lattice tighter results can be obtained, as shown in Figure 4.8 (d).

In conclusion, the complete algorithm can be summarized as follows: extended SSA form is generated from each program function, constraints graph is

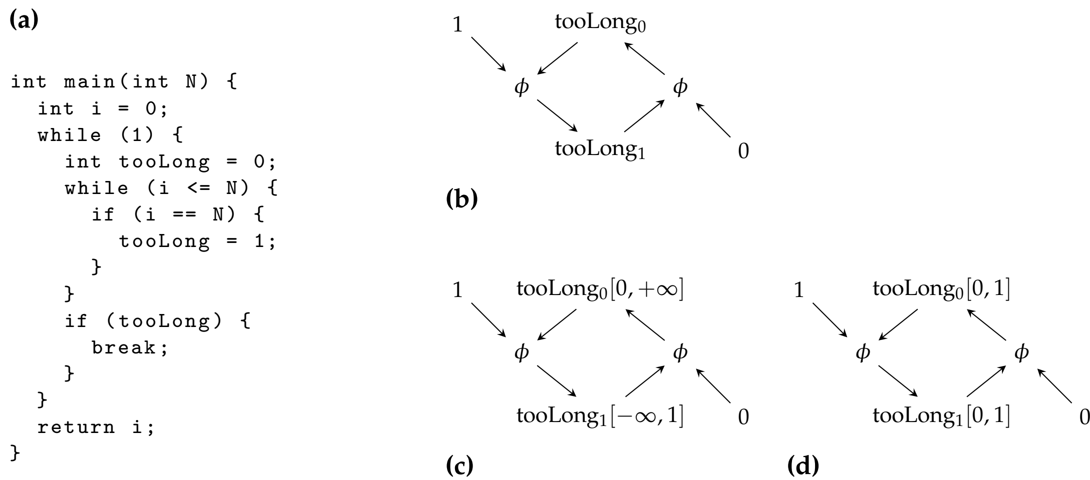

Figure 4.8

built extracting constraints and variables from statements and generating necessary inter-procedural relations from function calls, then the graph is searched for SCCs and the resolution algorithm is applied to nodes and supernodes in topological order. Finally, the resolution algorithm is composed of the following: a preliminary abstract interpretation phase to initialize range bounds, which in a case can immediately lead to a valid fixed point result, then the jump-set widening step, followed by the future resolution, which completes range bounds initialization, and finally a jump-set narrowing phase to finalize results. For completeness even a simple version based on standard widening and narrowing operators as defined by Cousot and Cousot [5] has been implemented.

4.2.6 BitValue Inference enhancement

Along with range analysis, also a BitValue inference algorithm, similar to that described by Budiu et al. [3], has been implemented inside PandA Bambu: while this is not part of this thesis work because it was already up and running inside the HLS tool, it has been teamed with the new range analysis implementation to achieve the best result out of a static analysis approach. While range analysis is more focused on word's meaning to detect its numerical bounds, thus it can perform best in trimming most significant bits in a word, BitValue

inference is better suited to analyze bitwise behaviors which lead to the loss of the big picture, but enables to detect useless bits even in the lower part of the word.

(a)

```
uint8_t i = 0;
while(i < 32) {
    i += 4;
}
```

(b)

$$\mathrm{I}[i] = [0, 32]$$
$$\mathrm{BV}[i] = 000\mathrm{UUUXX}$$

Figure 4.9: (a) Simple cycle code. (b) Range analysis (upper) and BitValue inference (lower) results for variable i

As an example the code snippet in Figure 4.9 can be considered, where variable i is incremented each time the loop is executed: the range analysis approach would correctly report the needed range is $[0, 32]$, without caring about the increment, conversely BitValue inference would immediately point out the two lowest bits of i are useless in this context because they are not changed nor read during the execution. Observing results from the two types of analysis in Figure 4.9 (b) it is clear that with both pieces of information variable i can be fully described, while with only one of them the picture is incomplete.

During range analysis implementation a sharing mechanism has been integrated enabling already available information to be used during analysis initialization steps and new information acquired from results to be merged with existing one to enable an improvement in accuracy on each iteration. In fact during the High-Level Synthesis process both range analysis and BitValue inference are executed and they can both require a further iteration of their counterpart to check for possible improvements: this circular execution environment is the key which brings, through multiple subsequent iterations, to a more accurate and effective final result, which can improve both performance and efficiency of the output design.

4.3 Floating-point encoding customization

While previously described analysis are considering the input program "as-is", acting to improve it as much as possible without affecting its correctness, it is in the interest of this thesis work to explore the possibility to arbitrarily manipulate standard IEEE 754 floating-point encoding customizing exponent and significand bitwidth or removing unnecessary symbols such as *NaN* or ∞. To actually define these custom requirements into the program specification, some pragma annotations have been defined: each function definition can be annotated specifying necessary customization for each function parameter and the return value when present. Thanks to these pragmas it is possible to inject user-customized floating-point representations into the HLS tool which will take care, through range analysis and BitValue inference, to perform all operations needed to remove unnecessary logic from standard floating-point operators and from the whole program through inter-procedural propagation process previously described in section 4.2.3. Figure 4.10 (a) shows available pragma annotations which can be used to customize standard IEEE 754 floating-point representation.

(a)

```
#pragma mask <par_name> sign {0,1}
                        exponent <lower_bound> <upper_bound>
                        significand <bitwidth>
```

(b)

```
#pragma mask a exponent -1023 1023
#pragma mask a significand 35
#pragma mask b exponent -1023 1023
#pragma mask b significand 35
#pragma mask @ exponent -1023 1023
#pragma mask @ significand 38
double custom_func(double a, double b)
{...}
```

Figure 4.10: (a) Pragma mask quick reference. (b) Custom floating-point specification example.

An example of customized double where *NaN* and ∞ symbols have been removed and significand bitwidth has been reduced to 35 is shown in Figure 4.10

(b); furthermore the same customization has been applied to the return value of *custom_func*, specified by the special symbol @: *NaN* and ∞ have been removed as well, while significand bitwidth has been decreased to 38 bits. Through these simple annotations, it is possible to easily tweak float and double representations to match the application needs, removing what is unnecessary and limiting the range of values and precision strictly to what is needed, avoiding unnecessary logic in the final design to achieve better performance, lower power consumption, and a smaller footprint.

4.4 Conclusions

A fully functional value range analysis algorithm has been implemented in this thesis work, successfully reproducing the approach proposed by Campos et al. [4] with new features, such as *anti-range* and floating-point variables handling; furthermore the existing BitValue inference algorithm already present in the chosen HLS tool has been upgraded and linked to this implementation to cooperate and allow a better overall awareness which leads to better optimizations on the output hardware design generated. Finally, a new set of pragma annotations have been defined to allow developers to fine-tune floating-point operations into their applications and let them use only what they need, enabling the HLS tool to remove unnecessary logic further improving performance, area footprint and power consumption of the output design.

Chapter 5

Results

The experimental setup used during all tests will be illustrated in the following, then the new benchmark suite will be outlined and discussed to explain how each test included will be useful to evaluate different aspects of the implemented design flow. Finally, the obtained test results will be described and analyzed.

5.1 Experimental setup

The algorithm discussed in chapter 4 has been entirely implemented into PandA Bambu, the High-Level Synthesis tool under development at Politecnico di Milano. This tool offers the possibility to use many versions of both GCC and LLVM compilers as a frontend for the synthesis process, then a common middleware is fed with the resulting IR and finally the actual RTL design flow takes place. Furthermore, many configurations regarding memory allocation and access policies, optimization policies that favor area footprint or application performance, and other specific tweaks are available in PandA Bambu and will be specified for each testing environment. It is thus important to say that all experiments will be performed starting from a software defined math library to allow the newly implemented design flow to deeply analyze each arithmetic operation gaining a complete overview of the tested application. Starting from scratch should grant a precise and authentic result on what are the capabilities

offered by the implementation.

Therefore a modified version of PandA Bambu v0.9.7-dev will be used in the following with both GCC and LLVM frontends; experiments will take place on a dual Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz machine with 256GB of RAM and generated RTL designs will be synthesized on some FPGAs from Xilinx, Altera and Lattice through their proprietary RTL synthesis software. Any specific configuration parameter will be specified with each test, including frontend compiler version and the FPGA board used for the synthesis.

5.2 Benchmark suite

The purpose of the proposed test suite is to evaluate the capabilities of the implemented design flow, enabling a better understanding of what type of improvements can be achieved thanks to the discussed algorithm. Enhancements are expected both on performances and on area footprint of generated designs, furthermore, a particular focus on the new possibility of floating-point representation customization is necessary to verify the efficacy of this approach. All considered, a complete benchmark suite was put together to cover both full-fledged applications and arithmetic operators alone to better observe results from each aspect of the implemented design flow. While complete applications are useful to check the overall gain in performance and resources consumption, testing arithmetic operators as stand-alone functions gives the advantage of a more specific peek on actual gain brought by custom floating-point representations.

5.2.1 Single operators

The first part of the proposed benchmark suite is composed of a subset of standard floating-point arithmetic operations available in *libm* library: these operations are synthesized as stand-alone functions during tests to specifically evaluate custom floating-point representations. When a single function is compiled it is not possible to exploit the full capabilities offered by value range analysis and the new design flow, but it is easier to appreciate what can be achieved with

floating-point representation tuning. Proposed tests will show how each arithmetic operation is affected when modifying floating-point precision through significand bitwidth manipulation and when removing special symbols such as ∞ and *NaN*.

5.2.2 Complete applications

The second part of the benchmark suite is composed of standard programs to allow a full evaluation of the implemented design flow. When a complete application is synthesized it is possible to exploit the whole set of features and optimizations offered by the new design flow and observe how the inter-procedural approach behaves in real-world scenarios. To perform these experiments the well-known CHStone HLS benchmark suite has been chosen. Each application will be tested with different parameters to gather significant results. The 12 programs from CHStone benchmark suite are selected from various application domains such as arithmetic, media processing, security, and microprocessors; they are relatively large applications, compared to the standard from HLS literature, thus they offer quite a challenging environment to test this thesis work implementation. The complete CHStone benchmark suite is listed by Table 5.1.

Benchmark Name	Description	Source
DFADD	Double-precision floating-point addition	SoftFloat
DFMUL	Double-precision floating-point multiplication	SoftFloat
DFDIV	Double-precision floating-point division	SoftFloat
DFSIN	Sine function for double-precision floating-point numbers	CHStone group, SoftFloat
MIPS	Simplified MIPS processor	CHStone group
ADPCM	Adaptive differential pulse code modulation decoder and encoder	SNU
GSM	Linear predictive coding analysis of global system for mobile communications	MediaBench
JPEG	JPEG image decompression	The Portable Video Research Group, CHStone group
MOTION	Motion vector decoding of the MPEG-2	MediaBench
AES	Advanced encryption standard	AILab
BLOWFISH	Data encryption standard	MiBench
SHA	Secure hash algorithm	MiBench

Table 5.1: CHStone benchmark suite list

5.3 Result evaluation

The first test results are about stand-alone floating-point arithmetic operators; each test consists of the synthesis of a floating-point operator with different

customized representations to mainly evaluate area footprint reduction. The actual synthesis is performed on a Xilinx Zynq xc7z020-1clg484 FPGA board, Bambu is configured to use GCC 7 as fronted, and *-lm* and *–soft-float* parameters are passed to the HLS tool to force the usage of software implemented floating-point operators. Thanks to these measures the proposed design flow will have complete control over operators design during the synthesis process. Results obtained from a test on the floating-point double precision division operator are shown in Table 5.2.

Benchmark Name	Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Frequency	Clock Slack
fdiv64	33	803	267	654	0	0	119.50	1.63
fdiv64-wo_nan	33	655	223	654	0	0	112.93	1.14
fdiv64_45	33	749	250	612	0	0	114.48	1.27
fdiv64_40	33	699	251	582	0	0	117.65	1.50
fdiv64_35	33	644	219	552	0	0	118.55	1.56
fdiv64_30	33	548	207	522	0	0	116.40	1.41
fdiv64_23	33	482	170	480	0	0	118.25	1.54

Table 5.2: Customized floating-point representations test for double precision division operator

Test *fdiv64-wo_nan* from Table 5.2 shows how removing special symbols, such as *NaN* and infinity, from standard double-precision representation can help reducing area footprint. Value range analysis is exploited to specify a smaller range for the exponent, $[-1023, 11, 1023]$ in this case, and propagate it through the operator logic: specifying an exponent range deprived of the 1024 value means all conditional checks about special symbols of the IEEE 754 representations will be removed, thus eliminating the relative computational logic. Furthermore, tests labeled as *fdiv64_X* deal with a customized representation featuring an *X* bits significand, thus a reduced precision representation which can be used when full precision is unnecessary, to lower resources requirements and improve power consumption. A quite consistent area reduction can be obtained by the combination of shorter significand bitwidth and special symbology removal from the standard representation. It is interesting too to observe that the area gain from significand bitwidth reduction is directly proportional to the number of removed bits, thus floating-point precision and resource utilization

are linearly related. Finally, a single-precision test result is reported too in Table 5.3.

Benchmark Name	Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Frequency	Clock Slack
fdiv	17	383	117	312	0	0	121.82	1.79
fdiv-wo_nan	17	325	105	290	0	0	123.49	1.90
fdiv_17	17	313	100	270	0	0	122.09	1.81
fdiv_12	17	266	86	234	0	0	118.57	1.57
fdiv_8	17	228	80	206	0	0	115.59	1.35

Table 5.3: Customized floating-point representations test for single precision division operator

As seen previously for the double-precision case, similar conclusions can be drawn from the proposed benchmark: the relation between significand bitwidth and area reduction is still directly proportional even for the single precision customization case.

The same type of tests have been executed with other compilers as fronted, such as GCC 4.9, Clang 4, and Clang 7, and relative results observed above where confirmed: running with GCC 7 gave the best absolute results in terms of area footprint among the four tested frontends.

For the second part of the benchmark, the CHStone suite was synthesized on a Xilinx Zynq xc7z020-1clg484 FPGA board, while PandA Bambu was tested with many configurations which will be described in the following. As a baseline for the comparison, the same programs were synthesized with value range analysis disabled. Experimental results are presented in two forms for each test configuration:

Box plot: gives a quick overview of how the experiment performed in global terms; each graph features five statistics about the generated designs reported as percentual difference from the baseline. Covered aspects are overall design area (lower is better), an *Area*Time* metric value, which helps to figure out the overall optimization behavior both in terms of area and latency (lower is better), frequency, which represents the expected clock frequency of the generated design (higher is better), registers count (lower is better), and slices count (lower is better).

Table list: a detailed report of each benchmark along with its related

baseline, to better visualize absolute values of generated designs. The table reports ten different aspects for each benchmark: total design latency, number of required clock cycles, some detailed information about area footprint, such as LUTs, slices, registers, DSPs and BRAMs count, design clock frequency and clock slack and finally the HLS time, which reports PandA Bambu design time.

The first configuration proposed will be addressed as BAMBU-AREA and is concerned about producing the smallest possible design from the synthesis. Figure 5.1 shows results for this type of build performed using GCC 4.9 and Clang 4 as frontends for the HLS tool; Table 5.4 and Table 5.5 report detaild results about the test.

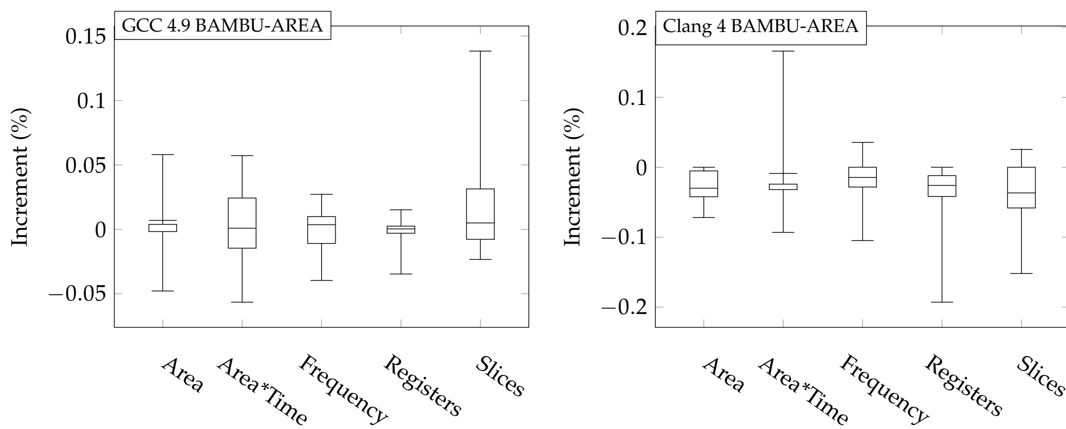

Figure 5.1: CHStone benchmark overview for GCC 4.9 (left side) and Clang 4 (right side) frontends. AREA

It is immediately clear how results are quite sparse, this is due to a particular application dependant behavior of the proposed design flow: it has been observed that effectiveness can be pretty high for some programs, while being insignificant for others. This is related to the fact that it is not always possible to apply effective constraints to program variables without affecting the correctness of results, nor it is always possible to propagate constraints successfully if memory operations are frequent. Anyhow, good results have been observed on affected programs, which presented an area reduction of 4.5% on average at the cost of some latency.

Benchmark Name	Tot. Latency	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Clock Frequency	Clock Slack	HLS Time(s)
adpcm	$2.881 \cdot 10^{2}$	21575	4085	1463	3348	30	3	74.88	1.64	43.93
adpcm-base	$2.908 \cdot 10^{2}$	21575	4290	1452	3468	30	3	74.19	1.52	34.12
aes	$5.180 \cdot 10^{1}$	4032	3264	1222	2534	0	4	77.83	2.15	27.65
aes-base	$5.184 \cdot 10^{1}$	4032	3085	1123	2496	0	4	77.77	2.14	21.65
bf	$2.073 \cdot 10^{3}$	155524	3669	1169	2542	0	11	75.02	1.67	21.70
bf-base	$2.129 \cdot 10^{3}$	155524	3672	1181	2542	0	11	73.04	1.31	15.01
dfadd	$5.336 \cdot 10^{0}$	400	2133	741	1768	0	0	74.97	1.66	36.57
dfadd-base	$5.171 \cdot 10^{0}$	400	2128	709	1757	0	0	77.36	2.07	17.69
dfdiv	$2.382 \cdot 10^{1}$	1990	3216	1172	2985	12	0	83.54	3.03	37.46
dfdiv-base	$2.400 \cdot 10^{1}$	1990	3213	1200	2975	12	0	82.90	2.94	19.46
dfmul	$1.773 \cdot 10^{0}$	124	2039	658	1210	10	0	69.93	0.70	30.55
dfmul-base	$1.785 \cdot 10^{0}$	124	2022	663	1209	10	0	69.47	0.61	14.05
dfsin	$8.145 \cdot 10^{2}$	57033	7564	3062	7132	15	0	70.02	0.72	92.26
dfsin-base	$7.987 \cdot 10^{2}$	57033	7555	2690	7116	15	0	71.41	1.00	23.77
gsm	$5.152 \cdot 10^{1}$	3462	3804	1222	2328	22	1	67.20	0.12	40.77
gsm-base	$5.110 \cdot 10^{1}$	3462	3821	1219	2343	22	1	67.75	0.24	16.08
jpeg	$9.104 \cdot 10^{3}$	635367	10551	3348	5964	5	44	69.79	0.67	190.37
jpeg-base	$9.212 \cdot 10^{3}$	635393	10672	3321	5976	5	44	68.98	0.50	90.56
mips	$2.957 \cdot 10^{1}$	2680	1152	337	397	4	2	90.65	3.97	8.99
mips-base	$2.957 \cdot 10^{1}$	2680	1152	337	397	4	2	90.65	3.97	5.99
mpeg2	$3.288 \cdot 10^{1}$	2221	1822	613	967	0	1	67.54	0.20	8.04
mpeg2-base	$3.158 \cdot 10^{1}$	2221	1799	597	973	0	1	70.33	0.78	5.09
sha_driver	$2.088 \cdot 10^{3}$	160075	2094	727	1811	0	9	76.65	1.95	8.52
sha_driver-base	$2.115 \cdot 10^{3}$	160075	2093	733	1808	0	9	75.70	1.79	6.22

Table 5.4: CHStone benchmark detail for GCC 4.9. AREA

Benchmark Name	Tot. Latency	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Clock Frequency	Clock Slack	HLS Time(s)
adpcm	$2.817 \cdot 10^{2}$	19342	6924	2524	5641	73	5	68.67	0.44	54.06
adpcm-base	$2.992 \cdot 10^{2}$	19842	7186	2976	5814	73	5	66.31	-0.08	47.00
aes	$5.421 \cdot 10^{1}$	4250	2918	1159	2250	0	4	78.39	2.24	36.94
aes-base	$4.853 \cdot 10^{1}$	4250	2979	1204	2298	0	4	87.58	3.58	31.00
bf	$1.717 \cdot 10^{3}$	152494	3486	1158	2515	0	11	88.83	3.74	15.57
bf-base	$1.717 \cdot 10^{3}$	152494	3486	1158	2515	0	11	88.83	3.74	15.01
dfdiv	$2.452 \cdot 10^{1}$	1822	3240	1087	2019	14	0	74.29	1.54	39.82
dfdiv-base	$2.324 \cdot 10^{1}$	1822	3262	1060	2046	14	0	78.41	2.25	28.33
dfmul	$1.291 \cdot 10^{0}$	105	1420	460	737	10	0	81.33	2.70	33.21
dfmul-base	$1.237 \cdot 10^{0}$	105	1530	479	766	10	0	84.90	3.22	34.41
dfsin	$6.788 \cdot 10^{2}$	46684	8650	2642	4307	27	0	68.78	0.46	87.61
dfsin-base	$6.653 \cdot 10^{2}$	46867	8984	2806	4425	27	0	70.44	0.80	50.09
gsm	$5.259 \cdot 10^{1}$	3531	4466	1463	2418	28	1	67.14	0.11	78.07
gsm-base	$5.225 \cdot 10^{1}$	3523	4641	1482	2555	27	1	67.43	0.17	55.38
jpeg	$8.841 \cdot 10^{3}$	624116	13524	4652	8808	6	55	70.60	0.84	160.87
jpeg-base	$9.103 \cdot 10^{3}$	624119	13850	4826	8881	5	55	68.56	0.41	110.96
mips	$2.803 \cdot 10^{1}$	2679	1156	401	544	3	2	95.58	4.54	13.02
mips-base	$2.803 \cdot 10^{1}$	2679	1156	401	544	3	2	95.58	4.54	11.34
mpeg2	$2.433 \cdot 10^{1}$	2244	1634	609	1150	0	1	92.22	4.16	10.45
mpeg2-base	$2.380 \cdot 10^{1}$	2244	1730	694	1274	0	1	94.30	4.40	9.07
sha_driver	$1.873 \cdot 10^{3}$	155701	2147	789	1786	0	9	83.14	2.97	13.04
sha_driver-base	$1.842 \cdot 10^{3}$	155701	2148	771	1832	0	9	84.52	3.17	11.74

Table 5.5: CHStone benchmark detail for Clang 4. AREA

Considering a more aggressive setup for PandA Bambu, called BAMBU-PERFORMANCE, which instead instructs the design flow to privilege performance at the expense of the area, the value range analysis performed better overall, generating smaller designs and positively affecting performance too with respect to the baseline. Some of the programs failed to be optimized also in this case, but no significant degradation was observed. Figure 5.2 show results for GCC 4.9 and Clang 4 frontends; Table 5.6 and Table 5.7 show detailed results.

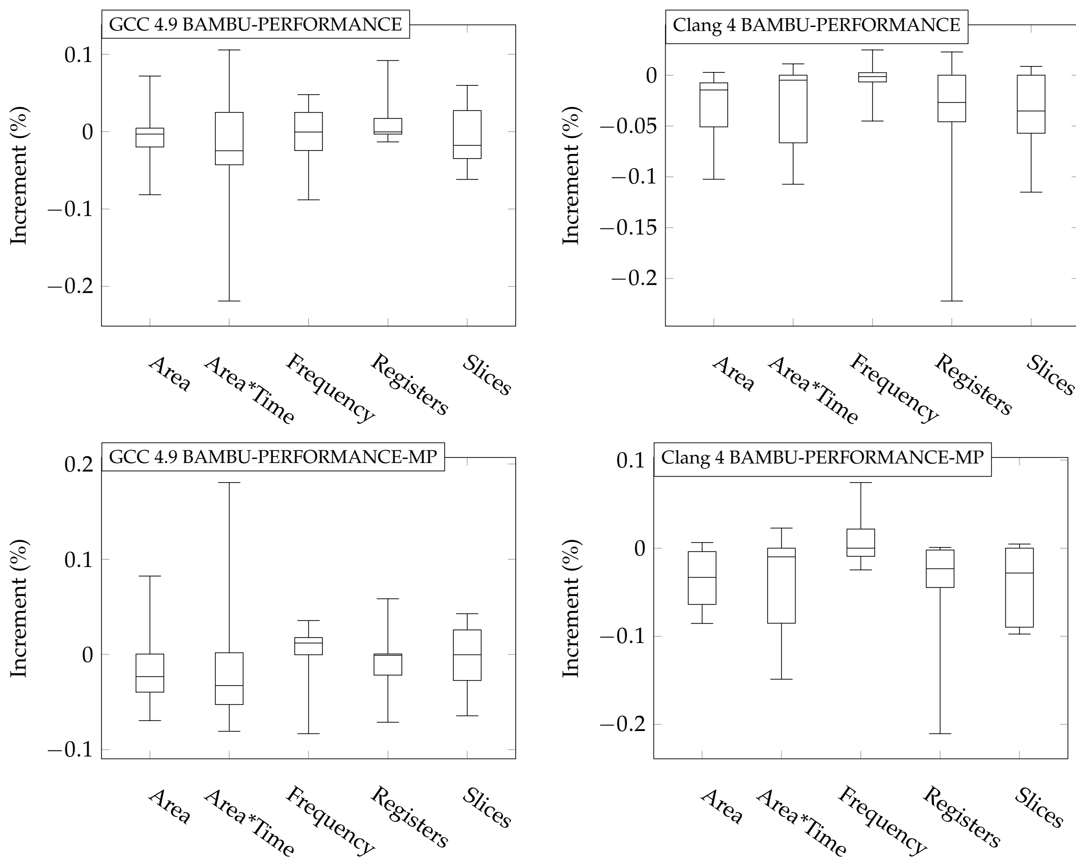

Figure 5.2: CHStone benchmark overviw for GCC 4.9 and Clang 4 frontends. PERFORMANCE setup (above) and PERFORMANCE-MP (below) .

While the BAMBU-PERFORMANCE setup presented above features a single-channel memory access design, also a similar one offering dual-channel memory access has been tested (BABMU-PERFORMANCE-MP). As shown

Benchmark Name	Tot. Latency	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Clock Frequency	Clock Slack	HLS Time(s)
adpcm	$7.633 \cdot 10^{1}$	5034	13339	4180	7677	126	3	65.95	−0.16	124.20
adpcm-base	$7.266 \cdot 10^{1}$	4934	12758	3944	7031	117	3	67.91	0.27	82.06
aes	$4.188 \cdot 10^{1}$	3082	5525	2279	4698	0	4	73.60	1.41	39.82
aes-base	$4.155 \cdot 10^{1}$	3082	5543	2320	4679	0	4	74.17	1.52	33.19
bf	$2.065 \cdot 10^{3}$	150595	3661	1215	2494	0	13	72.92	1.29	21.83
bf-base	$2.024 \cdot 10^{3}$	150595	3671	1243	2494	0	13	74.40	1.56	16.95
dfadd	$3.014 \cdot 10^{0}$	203	1949	608	804	0	0	67.35	0.15	100.27
dfadd-base	$3.013 \cdot 10^{0}$	203	2019	648	791	0	0	67.38	0.16	62.66
dfdiv	$2.440 \cdot 10^{1}$	1785	3045	950	1777	18	0	73.14	1.33	56.92
dfdiv-base	$2.534 \cdot 10^{1}$	1785	2841	919	1680	18	0	70.44	0.80	45.60
dfmul	$1.213 \cdot 10^{0}$	85	1433	443	566	10	0	70.08	0.73	33.60
dfmul-base	$1.534 \cdot 10^{0}$	105	1451	457	571	10	0	68.47	0.39	22.35
dfsin	$7.061 \cdot 10^{2}$	45339	10339	3147	5097	41	0	64.21	−0.57	288.72
dfsin-base	$6.796 \cdot 10^{2}$	45156	11256	3344	5165	41	0	66.44	−0.05	132.00
gsm	$3.557 \cdot 10^{1}$	2328	8318	2767	4918	50	5	65.45	−0.28	527.79
gsm-base	$3.604 \cdot 10^{1}$	2322	8551	2879	4832	55	5	64.43	−0.52	310.76
jpeg	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
jpeg-base	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
mips	$3.431 \cdot 10^{1}$	2670	1178	355	470	4	2	77.82	2.15	13.96
mips-base	$3.521 \cdot 10^{1}$	2670	1177	353	470	4	2	75.84	1.81	10.66
mpeg2	$1.475 \cdot 10^{1}$	1171	1479	494	928	0	2	79.40	2.41	9.19
mpeg2-base	$1.545 \cdot 10^{1}$	1171	1485	484	934	0	2	75.78	1.80	5.46
sha_driver	$1.521 \cdot 10^{3}$	116897	3474	1455	3065	0	9	76.84	1.99	132.06
sha_driver-base	$1.387 \cdot 10^{3}$	116897	3446	1398	3064	0	9	84.27	3.13	108.83

Table 5.6: CHStone benchmark detail for GCC 4.9 frontend. PERFORMANCE

Benchmark Name	Tot. Latency	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Clock Frequency	Clock Slack	HLS Time(s)
adpcm	$2.475 \cdot 10^{2}$	17045	7907	2820	6031	75	5	68.88	0.48	122.08
adpcm-base	$2.611 \cdot 10^{2}$	17545	8041	2884	5959	74	5	67.21	0.12	88.96
aes	$5.337 \cdot 10^{1}$	4183	3185	1200	2238	0	4	78.38	2.24	67.29
aes-base	$5.315 \cdot 10^{1}$	4183	3217	1261	2332	0	4	78.70	2.29	54.80
bf	$1.788 \cdot 10^{3}$	152494	3490	1174	2507	0	11	85.28	3.27	24.75
bf-base	$1.788 \cdot 10^{3}$	152494	3490	1174	2507	0	11	85.28	3.27	23.14
dfadd	$3.313 \cdot 10^{0}$	224	2533	771	854	0	0	67.60	0.21	87.95
dfadd-base	$3.295 \cdot 10^{0}$	224	2663	841	1098	0	0	67.99	0.29	64.70
dfdiv	$2.429 \cdot 10^{1}$	1794	3207	1036	1967	18	0	73.87	1.46	63.16
dfdiv-base	$2.409 \cdot 10^{1}$	1784	3198	1028	1923	18	0	74.07	1.50	43.24
dfmul	$1.216 \cdot 10^{0}$	90	1353	435	660	10	0	74.04	1.49	55.22
dfmul-base	$1.232 \cdot 10^{0}$	91	1466	482	693	10	0	73.89	1.47	54.18
dfsin	$6.623 \cdot 10^{2}$	44843	10322	3219	4416	31	0	67.71	0.23	229.95
dfsin-base	$6.689 \cdot 10^{2}$	45111	10944	3374	4625	31	0	67.44	0.17	114.64
gsm	$5.229 \cdot 10^{1}$	3537	4573	1499	2565	27	1	67.65	0.22	203.94
gsm-base	$4.993 \cdot 10^{1}$	3537	4750	1543	2644	28	1	70.84	0.88	189.41
jpeg	$7.643 \cdot 10^{3}$	545328	16765	5934	10958	11	55	71.35	0.98	272.23
jpeg-base	$7.577 \cdot 10^{3}$	545921	16974	6193	11118	5	55	72.05	1.12	166.38
mips	$3.350 \cdot 10^{1}$	2679	1220	422	564	3	2	79.97	2.50	19.02
mips-base	$3.350 \cdot 10^{1}$	2679	1220	422	564	3	2	79.97	2.50	17.66
mpeg2	$3.192 \cdot 10^{1}$	2205	7144	2680	4857	0	1	69.07	0.52	64.36
mpeg2-base	$3.162 \cdot 10^{1}$	2207	7223	2657	4975	0	1	69.80	0.67	51.97
sha_driver	$1.917 \cdot 10^{3}$	155437	1875	684	1550	0	9	81.07	2.66	18.89
sha_driver-base	$1.928 \cdot 10^{3}$	155441	2089	773	1686	0	9	80.63	2.60	19.17

Table 5.7: CHStone benchmark detail for Clang 4 frontend. PERFORMANCE

from tables in Figure 5.2, an increased overall optimization was obtained on dual-channel memory designs. Peaks observed in BAMBU-PERFORMANCE-MP graphs are due to an anomalous behavior of the adpcm program, which experienced a consistent degradation with respect to the baseline synthesis. Table 5.8 and Table 5.9 show detailed results.

Benchmark Name	Tot. Latency	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Clock Frequency	Clock Slack	HLS Time(s)
adpcm	$6.038 \cdot 10^{1}$	4266	16500	4945	9534	120	6	70.66	0.85	120.22
adpcm-base	$6.063 \cdot 10^{1}$	4216	16146	4923	9007	106	6	69.53	0.62	78.19
aes	$2.185 \cdot 10^{1}$	1595	5625	2114	4212	0	8	73.00	1.30	39.11
aes-base	$2.215 \cdot 10^{1}$	1595	6045	2032	4409	0	8	72.00	1.11	32.79
bf	$1.280 \cdot 10^{3}$	89799	3557	1241	2618	0	20	70.16	0.75	20.83
bf-base	$1.309 \cdot 10^{3}$	89799	3579	1247	2618	0	20	68.62	0.43	16.11
dfadd	$2.819 \cdot 10^{0}$	203	1926	567	783	0	0	72.00	1.11	94.23
dfadd-base	$2.920 \cdot 10^{0}$	203	2019	606	843	0	0	69.53	0.62	57.04
dfdiv	$2.478 \cdot 10^{1}$	1785	3153	1017	1811	18	0	72.02	1.12	55.46
dfdiv-base	$2.432 \cdot 10^{1}$	1785	3225	995	1809	18	0	73.41	1.38	44.58
dfmul	$1.488 \cdot 10^{0}$	105	1489	468	631	10	0	70.54	0.82	32.18
dfmul-base	$1.499 \cdot 10^{0}$	105	1547	480	622	10	0	70.07	0.73	21.38
dfsin	$6.679 \cdot 10^{2}$	45522	10403	3217	5148	41	0	68.16	0.33	283.01
dfsin-base	$6.721 \cdot 10^{2}$	45339	11012	3370	5221	41	0	67.46	0.18	128.05
gsm	$3.104 \cdot 10^{1}$	2048	9421	3033	5267	56	10	65.97	−0.16	517.49
gsm-base	$3.125 \cdot 10^{1}$	2042	9655	3020	5276	59	10	65.33	−0.31	306.94
jpeg	$6.446 \cdot 10^{3}$	450799	21915	6907	12241	7	58	69.94	0.70	222.84
jpeg-base	$6.662 \cdot 10^{3}$	450802	21959	7152	13043	7	58	67.66	0.22	126.40
mips	$3.154 \cdot 10^{1}$	2471	1085	338	406	4	4	78.34	2.23	13.98
mips-base	$3.006 \cdot 10^{1}$	2471	1077	326	406	4	4	82.21	2.84	10.30
mpeg2	$1.634 \cdot 10^{1}$	1162	1842	609	1165	0	4	71.10	0.94	8.97
mpeg2-base	$1.498 \cdot 10^{1}$	1162	1702	584	1170	0	4	77.55	2.11	5.64
sha_driver	$1.071 \cdot 10^{3}$	82480	4158	1579	3370	0	10	77.01	2.01	131.04
sha_driver-base	$1.086 \cdot 10^{3}$	82480	4315	1589	3369	0	10	75.94	1.83	108.27

Table 5.8: CHStone benchmark detail for GCC 4.9 frontend. PERFORMANCE-MP

Benchmark Name	Tot. Latency	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Clock Frequency	Clock Slack	HLS Time(s)
adpcm	$2.408 \cdot 10^{2}$	16333	10762	3994	8848	81	10	67.83	0.26	119.41
adpcm-base	$2.537 \cdot 10^{2}$	16934	10655	3981	8610	69	10	66.75	0.02	87.52
aes	$3.280 \cdot 10^{1}$	2890	4092	1567	3087	0	8	88.10	3.65	67.70
aes-base	$3.294 \cdot 10^{1}$	2881	4054	1558	3161	0	8	87.47	3.57	54.85
bf	$9.522 \cdot 10^{2}$	87056	3607	1330	2874	0	12	91.42	4.06	23.90
bf-base	$9.522 \cdot 10^{2}$	87056	3607	1330	2874	0	12	91.42	4.06	21.77
dfadd	$3.174 \cdot 10^{0}$	224	2442	769	995	0	0	70.58	0.83	86.32
dfadd-base	$3.013 \cdot 10^{0}$	224	2390	737	995	0	0	74.34	1.55	64.97
dfdiv	$2.456 \cdot 10^{1}$	1806	3142	1038	1902	18	0	73.54	1.40	61.52
dfdiv-base	$2.303 \cdot 10^{1}$	1784	3160	1040	1922	18	0	77.45	2.09	42.04
dfmul	$1.186 \cdot 10^{0}$	90	1198	394	613	10	0	75.88	1.82	53.13
dfmul-base	$1.237 \cdot 10^{0}$	91	1458	491	693	10	0	73.54	1.40	51.81
dfsin	$6.632 \cdot 10^{2}$	44843	10452	3201	4436	31	0	67.61	0.21	226.19
dfsin-base	$6.726 \cdot 10^{2}$	45111	10822	3319	4594	31	0	67.07	0.09	112.72
gsm	$4.092 \cdot 10^{1}$	2908	5105	1691	2987	32	3	71.07	0.93	208.92
gsm-base	$4.098 \cdot 10^{1}$	2900	5421	1814	3033	31	3	70.77	0.87	191.51
jpeg	$7.426 \cdot 10^{3}$	500903	19548	6571	10922	12	84	67.45	0.17	643.74
jpeg-base	$7.321 \cdot 10^{3}$	501487	19759	6961	11259	9	84	68.50	0.40	168.90
mips	$3.061 \cdot 10^{1}$	2488	1035	367	521	3	4	81.29	2.70	19.43
mips-base	$3.061 \cdot 10^{1}$	2488	1035	367	521	3	4	81.29	2.70	17.47
mpeg2	$2.792 \cdot 10^{1}$	2205	7223	2737	5274	0	1	78.99	2.34	60.70
mpeg2-base	$2.610 \cdot 10^{1}$	2205	7376	2770	5412	0	1	84.49	3.16	49.59
sha_driver	$1.311 \cdot 10^{3}$	109206	2790	924	2112	0	10	83.27	2.99	18.50
sha_driver-base	$1.279 \cdot 10^{3}$	109210	2902	981	2232	0	10	85.40	3.29	18.13

Table 5.9: CHStone benchmark detail for Clang 4 frontend. PERFORMANCE-MP

Finally, an intermediate setting has been tested, again with both single-channel and dual-channel approaches, to observe the proposed design behavior in balanced scenarios. As in the BAMBU-PERFORMANCE case, an unexpected degradation was detected with the adpcm program, thus peaks observed in the results are due to this particular case; all other programs experienced an

improvement in performance and resource utilization, as shown by presented data. Furthermore, it has been interesting to observe that affected programs from all previously described tests experienced an improvement regarding slack timings. In the following, Figure 5.3 shows box plots for BAMBU-BALANCED and BAMBU-BALANCED-MP configurations for GCC 4.9 and Clang 4; Table 5.10, Table 5.11, Table 5.12, and Table 5.13 report detailed results.

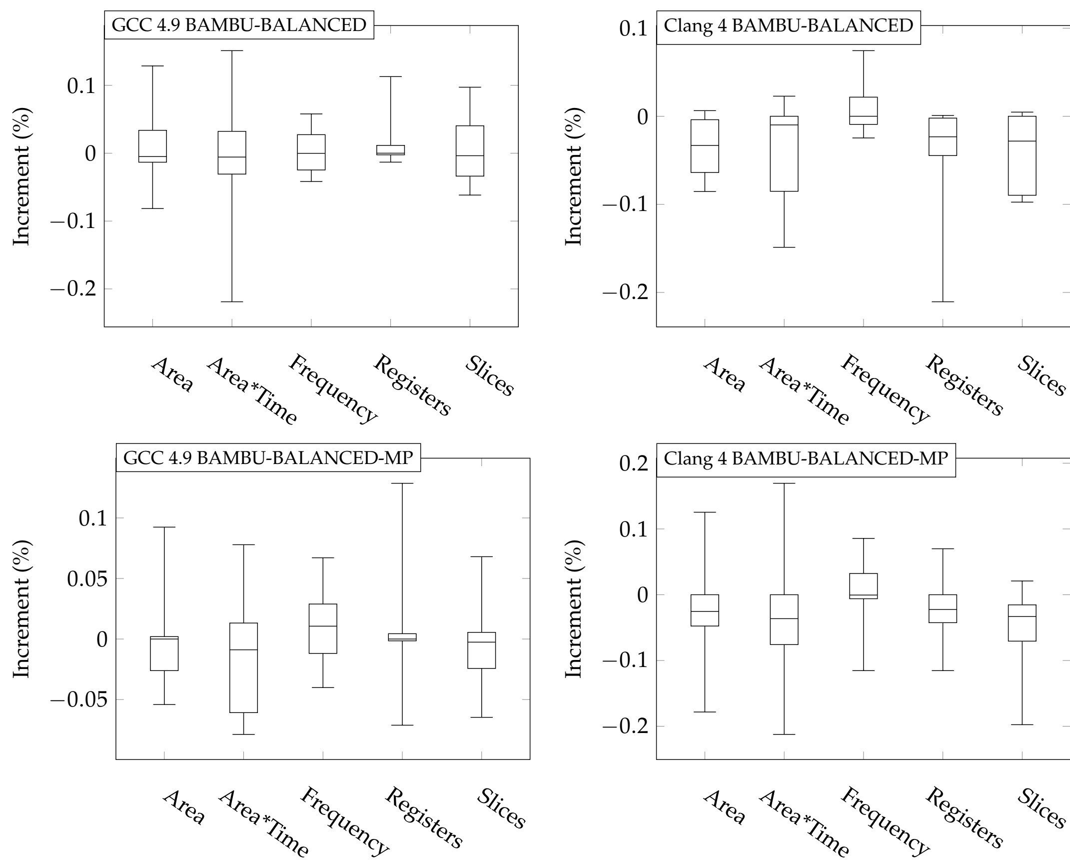

Figure 5.3: CHStone benchmark overviw for GCC 4.9 and Clang 4 frontends. BALANCED setup (above) and BALANCED-MP setup (below)

Benchmark Name	Tot. Latency	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Clock Frequency	Clock Slack	HLS Time(s)
adpcm	$2.378 \cdot 10^{2}$	16535	7443	2393	5234	49	3	69.55	0.62	66.89
adpcm-base	$2.331 \cdot 10^{2}$	16485	6595	2181	4703	41	3	70.72	0.86	47.21
aes	$5.316 \cdot 10^{1}$	4031	3309	1205	2472	0	4	75.82	1.81	35.44
aes-base	$5.548 \cdot 10^{1}$	4031	3135	1135	2464	0	4	72.66	1.24	28.12
bf	$1.990 \cdot 10^{3}$	149934	3621	1176	2498	0	11	75.36	1.73	21.94
bf-base	$2.035 \cdot 10^{3}$	149934	3613	1229	2498	0	11	73.68	1.43	17.12
dfadd	$3.014 \cdot 10^{0}$	203	1949	608	804	0	0	67.35	0.15	100.30
dfadd-base	$3.013 \cdot 10^{0}$	203	2019	648	791	0	0	67.38	0.16	61.85
dfdiv	$2.440 \cdot 10^{1}$	1785	3045	950	1777	18	0	73.14	1.33	56.93
dfdiv-base	$2.534 \cdot 10^{1}$	1785	2841	919	1680	18	0	70.44	0.80	45.33
dfmul	$1.213 \cdot 10^{0}$	85	1433	443	566	10	0	70.08	0.73	33.77
dfmul-base	$1.534 \cdot 10^{0}$	105	1451	457	571	10	0	68.47	0.39	21.74
dfsin	$7.061 \cdot 10^{2}$	45339	10339	3147	5097	41	0	64.21	−0.57	287.21
dfsin-base	$6.796 \cdot 10^{2}$	45156	11256	3344	5165	41	0	66.44	−0.05	130.95
gsm	$3.276 \cdot 10^{1}$	2346	4226	1395	2348	26	1	71.60	1.03	64.80
gsm-base	$3.466 \cdot 10^{1}$	2346	4117	1324	2348	26	1	67.69	0.23	32.79
jpeg	$7.109 \cdot 10^{3}$	504693	15665	5066	9674	7	44	71.00	0.91	215.61
jpeg-base	$6.939 \cdot 10^{3}$	504696	15546	4890	9580	7	44	72.73	1.25	115.58
mips	$3.574 \cdot 10^{1}$	2679	1221	375	494	4	2	74.96	1.66	13.64
mips-base	$3.574 \cdot 10^{1}$	2679	1221	375	494	4	2	74.96	1.66	10.76
mpeg2	$2.988 \cdot 10^{1}$	2200	1661	541	991	0	1	73.64	1.42	9.84
mpeg2-base	$2.907 \cdot 10^{1}$	2200	1633	545	997	0	1	75.69	1.79	5.76
sha_driver	$2.111 \cdot 10^{3}$	155437	2013	719	1768	0	9	73.62	1.42	22.53
sha_driver-base	$2.023 \cdot 10^{3}$	155437	2046	727	1770	0	9	76.82	1.98	12.74

Table 5.10: CHStone benchmark detail for GCC 4.9 frontend. BALANCED

Benchmark Name	Tot. Latency	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Clock Frequency	Clock Slack	HLS Time(s)
adpcm	$2.186 \cdot 10^{2}$	15452	9556	2939	6415	55	10	70.70	0.86	65.65
adpcm-base	$2.215 \cdot 10^{2}$	15252	8748	2752	5684	45	10	68.86	0.48	45.86
aes	$3.285 \cdot 10^{1}$	2635	4193	1434	2762	0	8	80.21	2.53	34.31
aes-base	$3.153 \cdot 10^{1}$	2635	4138	1397	2713	0	8	83.56	3.03	27.28
bf	$1.186 \cdot 10^{3}$	89157	3362	1056	2141	0	18	75.20	1.70	21.54
bf-base	$1.203 \cdot 10^{3}$	89157	3362	1082	2141	0	18	74.12	1.51	16.30
dfadd	$2.819 \cdot 10^{0}$	203	1926	567	783	0	0	72.00	1.11	93.44
dfadd-base	$2.920 \cdot 10^{0}$	203	2019	606	843	0	0	69.53	0.62	58.77
dfdiv	$2.478 \cdot 10^{1}$	1785	3153	1017	1811	18	0	72.02	1.12	55.64
dfdiv-base	$2.432 \cdot 10^{1}$	1785	3225	995	1809	18	0	73.41	1.38	44.63
dfmul	$1.488 \cdot 10^{0}$	105	1489	468	631	10	0	70.54	0.82	32.26
dfmul-base	$1.499 \cdot 10^{0}$	105	1547	480	622	10	0	70.07	0.73	21.10
dfsin	$6.644 \cdot 10^{2}$	45522	10416	3152	5098	41	0	68.51	0.40	284.21
dfsin-base	$6.721 \cdot 10^{2}$	45339	11012	3370	5221	41	0	67.46	0.18	130.61
gsm	$3.339 \cdot 10^{1}$	2303	4750	1515	2708	31	3	68.98	0.50	63.28
gsm-base	$3.278 \cdot 10^{1}$	2303	4736	1519	2710	30	3	70.25	0.77	32.10
jpeg	$6.119 \cdot 10^{3}$	458799	15815	5149	9358	7	58	74.98	1.66	208.24
jpeg-base	$6.530 \cdot 10^{3}$	458802	15788	5162	9351	7	58	70.26	0.77	109.84
mips	$3.163 \cdot 10^{1}$	2484	1084	333	454	4	4	78.54	2.27	13.05
mips-base	$3.163 \cdot 10^{1}$	2484	1084	333	454	4	4	78.54	2.27	10.71
mpeg2	$2.879 \cdot 10^{1}$	2189	2027	650	1257	0	1	76.02	1.85	9.45
mpeg2-base	$2.851 \cdot 10^{1}$	2189	2026	651	1262	0	1	76.78	1.98	6.12
sha_driver	$1.343 \cdot 10^{3}$	113318	2265	769	1794	0	12	84.36	3.15	21.88
sha_driver-base	$1.403 \cdot 10^{3}$	113318	2309	778	1794	0	12	80.79	2.62	12.55

Table 5.11: CHStone benchmark detail for GCC 4.9 frontend. BALANCED-MP

Benchmark Name	Tot. Latency	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Clock Frequency	Clock Slack	HLS Time(s)
adpcm	$2.583 \cdot 10^2$	17445	7832	2756	5715	79	5	67.53	0.19	122.17
adpcm-base	$2.542 \cdot 10^2$	17345	7782	2806	5710	69	5	68.23	0.34	88.28
aes	$4.931 \cdot 10^1$	4183	3133	1168	2238	0	4	84.82	3.21	67.54
aes-base	$5.040 \cdot 10^1$	4183	3337	1289	2335	0	4	83.00	2.95	54.68
bf	$1.717 \cdot 10^3$	152494	3486	1158	2515	0	11	88.83	3.74	24.29
bf-base	$1.717 \cdot 10^3$	152494	3486	1158	2515	0	11	88.83	3.74	22.71
dfadd	$2.562 \cdot 10^0$	188	1854	566	686	0	0	73.37	1.37	74.54
dfadd-base	$2.688 \cdot 10^0$	188	1986	619	869	0	0	69.95	0.70	48.38
dfdiv	$2.467 \cdot 10^1$	1782	3092	999	1915	18	0	72.24	1.16	62.80
dfdiv-base	$2.409 \cdot 10^1$	1784	3198	1028	1923	18	0	74.07	1.50	42.50
dfmul	$1.216 \cdot 10^0$	90	1353	435	660	10	0	74.04	1.49	55.44
dfmul-base	$1.232 \cdot 10^0$	91	1466	482	693	10	0	73.89	1.47	53.97
dfsin	$6.512 \cdot 10^2$	46065	8455	2613	4119	31	0	70.74	0.86	181.25
dfsin-base	$6.677 \cdot 10^2$	46248	8932	2757	4235	31	0	69.26	0.56	93.73
gsm	$5.265 \cdot 10^1$	3523	4503	1496	2536	27	1	66.92	0.06	207.11
gsm-base	$5.172 \cdot 10^1$	3523	4538	1489	2568	28	1	68.12	0.32	193.30
jpeg	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
jpeg-base	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
mips	$3.350 \cdot 10^1$	2679	1220	422	564	3	2	79.97	2.50	19.79
mips-base	$3.350 \cdot 10^1$	2679	1220	422	564	3	2	79.97	2.50	17.75
mpeg2	$3.179 \cdot 10^1$	2201	7128	2657	4859	0	1	69.23	0.56	64.06
mpeg2-base	$3.162 \cdot 10^1$	2207	7223	2657	4975	0	1	69.80	0.67	52.58
sha_driver	$1.808 \cdot 10^3$	155439	1875	686	1541	0	9	85.98	3.37	17.96
sha_driver-base	$1.943 \cdot 10^3$	155441	2050	759	1678	0	9	80.02	2.50	18.32

Table 5.12: CHStone benchmark detail for Clang 4 frontend. BALANCED

Benchmark Name	Tot. Latency	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Clock Frequency	Clock Slack	HLS Time(s)
adpcm	$2.547 \cdot 10^2$	16737	10464	3510	7886	78	14	65.72	-0.22	138.10
adpcm-base	$2.451 \cdot 10^2$	16284	9298	3438	7371	73	14	66.44	-0.05	89.01
aes	$3.207 \cdot 10^1$	2881	4065	1545	3093	0	8	89.84	3.87	68.84
aes-base	$3.481 \cdot 10^1$	2881	4050	1590	3190	0	8	82.75	2.92	53.19
bf	$9.912 \cdot 10^2$	92256	3343	1107	2267	0	14	93.08	4.26	24.00
bf-base	$9.912 \cdot 10^2$	92256	3343	1107	2267	0	14	93.08	4.26	22.46
dfadd	$2.853 \cdot 10^0$	210	1821	557	720	0	0	73.61	1.41	69.87
dfadd-base	$2.837 \cdot 10^0$	210	1915	600	772	0	0	74.02	1.49	47.11
dfdiv	$2.323 \cdot 10^1$	1794	3165	999	1966	18	0	77.24	2.05	64.39
dfdiv-base	$2.489 \cdot 10^1$	1784	3199	1020	1921	18	0	71.67	1.05	42.85
dfmul	$1.186 \cdot 10^0$	90	1198	394	613	10	0	75.88	1.82	54.15
dfmul-base	$1.237 \cdot 10^0$	91	1458	491	693	10	0	73.54	1.40	52.24
dfsin	$6.652 \cdot 10^2$	46065	8324	2571	4142	31	0	69.25	0.56	180.15
dfsin-base	$6.685 \cdot 10^2$	46248	8736	2672	4289	31	0	69.18	0.54	90.48
gsm	$3.978 \cdot 10^1$	2894	4568	1534	2566	31	5	72.75	1.25	204.24
gsm-base	$4.001 \cdot 10^1$	2886	4678	1568	2617	32	5	72.13	1.14	188.04
jpeg	$7.402 \cdot 10^3$	507940	17375	5625	9582	12	84	68.62	0.43	725.52
jpeg-base	$7.346 \cdot 10^3$	507935	17967	6050	9743	11	84	69.14	0.54	131.66
mips	$3.061 \cdot 10^1$	2488	1035	367	521	3	4	81.29	2.70	18.96
mips-base	$3.061 \cdot 10^1$	2488	1035	367	521	3	4	81.29	2.70	17.40
mpeg2	$2.846 \cdot 10^1$	2201	7173	2650	5275	0	1	77.33	2.07	65.96
mpeg2-base	$2.625 \cdot 10^1$	2203	7374	2803	5412	0	1	83.93	3.09	52.36
sha_driver	$1.312 \cdot 10^3$	113318	2106	731	1609	0	12	86.34	3.42	19.14
sha_driver-base	$1.357 \cdot 10^3$	113322	2246	805	1729	0	12	83.53	3.03	18.86

Table 5.13: CHStone benchmark detail for Clang 4 frontend. BALANCED-MP

5.4 Conclusions

The first set of benchmarks revealed the efficacy of the proposed floating-point customization technique, enabling the HLS design flow to remove unnecessary logic when a lower precision level is required from the application or when special symbols offered by IEEE 754 representation are not required. Furthermore, the overall efficacy of the proposed design flow has been proved, as shown by the CHStone benchmark suite. It is important to point out that tests on CHStone programs where performed from a completely non-optimized starting point, thus the design flow proved to be able to efficiently deduce and propagate value range constraints during the design process generating an efficient result compared with already optimized library functions.

Chapter 6

Conclusions and future work

A final summary about this thesis work will be presented, analyzing main goals reached through the proposed HLS design flow implementation, followed by some possibilities which could be interesting to explore with future development work.

6.1 Design flow evaluation

As a first consideration of this thesis work it is worth saying that the possibility to perform a static value range analysis inside an HLS design flow has been successfully proven. The discussed algorithm has been fully implemented and integrated into PandA Bambu HLS tool and the new design flow has proven to be effective after testing on the proposed benchmark suite.

Taking up contributions listed in the introductory chapter, a full-fledged algorithm starting from the state of the art has been implemented inside an existing HLS tool design flow, enabling the value range analysis to gather necessary information from the new Intermediate Representation (IR) considered. Furthermore, the algorithm implemented to solve the value range analysis problem has been proven to have a linear time complexity with respect to the number of instructions of synthesized programs, as pointed out during the algorithm discussion.

Many enhancements to the state of the art proposals have been implemented along with the standard algorithm, to achieve a better efficacy in both integer and floating-point value range analysis. A new range representation has been designed to support a more accurate description of value ranges and to include floating-point representation as a new feature.

The existing BitValue inference algorithm featured in PandA Bambu HLS tool has been successfully upgraded to support floating-point representation analysis and to allow an effective propagation of constraints also through floating-point operations. Moreover, a profitable interaction between value range analysis and BitValue inference has been implemented, enabling cooperation of the two algorithms to achieve a better result through data sharing.
Additionally, thanks to the new capabilities included in the design flow, it has been possible to design a floating-point representation customization interface, which has been proven to be effective as shown by proposed experiments on floating-point arithmetic operators.

In conclusion, the overall efficacy of the proposed design flow has been illustrated through the specifically arranged benchmark suite exposed in chapter 5, proving expected results which certify the capabilities of the proposed implementation.

6.2 Future developments

The proposed implementation could still be enhanced to improve awareness on values stored in variables loaded from memory: current implementation can not track memory interactions of programs, thus it is quite inaccurate when dealing with values loaded from memory. It would be of great impact to perform a pointer analysis to track values associated with pointers by load and store operations, to gain a deeper knowledge of memory structure and allocated values: thanks to this information it would be possible to propagate constraints even through memory operations.

Furthermore, it could be of interest to add support for a dynamic value range analysis, along with the static one, to enable more aggressive optimizations,

when global correctness is not required by the applications. As explained before, a dynamic analysis would allow an emphasized optimization at the cost of correctness outside the input training set used during the evaluation process. This type of analysis can give better results when program inputs are well known and restricted to a small subset of the possible ones.

Finally, it could be very useful for critical applications to implement the possibility to statically check for undesired overflows, unwanted memory location accesses and all value range related undesired behaviors, which could be detected during the abstract interpretation of the program performed by the value range analysis algorithm. Thanks to these types of checks, developers could exploit the analysis to fix numerous issues which normally require much more effort because of the need to design ad-hoc tests.

References

[1] V. Benara, S. Rampalli, Z. Choudhury, S. Purini, and U. Bondhugula. Synthesizing power and area efficient image processing pipelines on fpgas using customized bit-widths. *CoRR*, abs/1803.02660, 2018. URL `http://arxiv.org/abs/1803.02660`.

[2] R. Bodik, R. Gupta, and V. Sarkar. Abcd: Eliminating array bounds checks on demand. *SIGPLAN Not.*, 35(5):321 – 333, May 2000. ISSN 0362 - 1340. doi: 10.1145/358438.349342. URL `https://doi.org/10.1145/358438.349342`.

[3] M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein. Bitvalue inference: Detecting and exploiting narrow bitwidth computations. In A. Bode, T. Ludwig, W. Karl, and R. Wismuller, editors, *Euro-Par 2000 Parallel Processing*, pages 969–979, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44520-3. doi: 10.1007/3-540-44520-X_137.

[4] V. Campos, R. Rodrigues, I. de Assis Costa, and F. Pereira. Speed and precision in range analysis. In Francisco Heron de Carvalho Junior and Luis Soares Barbosa, editors, *Programming Languages*, pages 42–56, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33182-4. doi: 10.1007/978-3-642-33182-4_5.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238 – 252, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373500. doi: 10.1145/512950.512973. URL `https://doi.org/10.1145/512950.512973`.

[6] M. Gort and J. H. Anderson. Range and bitmask analysis for hardware optimization in high-level synthesis. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 773–779, 2013. doi: 10.1109/ASPDAC.2013.6509694.

[7] R. Rodrigues, V. Campos, and F. Pereira. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2013. doi: 10.1109/CGO.2013.6494996.

[8] L. Rosa and V. Bonato. A method to convert floating to fixed-point ekf-slam for embedded robotics. *Journal of the Brazilian Computer Society*, 19, 06 2012. doi: 10.1007/s13173-012-0092-4.

[9] S. Roy and P. Banerjee. An algorithm for converting floating-point computations to fixed-point in matlab based fpga design. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 484–487, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138288. doi: 10.1145/996566.996701. URL https://doi.org/10.1145/996566.996701.

[10] S. Roy and P. Banerjee. An algorithm for trading off quantization error with hardware resources for matlab-based fpga design. *IEEE Transactions on Computers*, 54(7):886–896, 2005.

[11] Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 280–295, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24730-2. doi: 10.1007/978-3-540-24730-2_23.

[12] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012. ISBN 0321842685.