Simulation and Verification of non-functional properties for Embedded Systems

By

Pablo González de Aledo Marugán

BSc University of Cantabria

MSc University of Cantabria

A thesis submitted to Macquarie University and University of Cantabria

for the degree of

Doctor of Philosophy

Department of Computing

September 2017

Typeset in $\LaTeX\,2_\varepsilon$.

Declaration

As a cotutelle student, I certify that the work in this thesis entitled SIMULATION AND VERIFICATION OF NON-FUNCTIONAL PROPERTIES FOR EMBEDDED SYSTEMS has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree to any other university or institution other than Macquarie University and University of Cantabria. I also certify that the thesis is an original piece of research and it has been written by me. Any help and assistance that I have received in my research work and the preparation of the thesis itself have been appropriately acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

[illegible]

Pablo González de Aledo Marugán

Acknowledgements

There are many people I would like to thank.

First and foremost, my supervisors Pablo Sánchez and Franck Cassez, for their help and for all they have taught me about how to do original research work, how to stay focused and motivated, and for patiently discussing with me the details of our research. At the end of my journey, I'm just sorry I could only learn this much.

To my family (especially Carolina), for their love and support, and for all the time that this thesis has stolen from them.

During my Ph.D., meeting people interested in what I built has been one of the most gratifying experiences, and working with them a marvelous gift. Special thanks to Ralf Huuck, Nils Przigoda and Robert Willie, for their clear thinking and for sharing their experience with me along the way.

Finally, I would like to thank some people that (without knowing it) have motivated me to do my best and have influenced the way I have approached some decisions in my research: Fernando, Hector, LUIS and Daniel.

– Pablo

Abstract

Unlike other generic computing systems, embedded real-time systems (ERTSs) interact with their environment and are subject to specific constraints; so their design usually faces particular challenges that are unique to this discipline. In these systems, the correctness of the design depends not only on the fact that it can compute something, but also on the fact that it can deliver a result before a certain deadline, with low energy consumption, requiring a minimal memory footprint, and in such a way that the thermal behavior of the chip does not degrade its functionality and reliability. These non-functional properties are as important as (or even more important than) the result of the computation per se. By "non-functional" properties, we refer to energy consumption, memory transactions, bus and NoC contention, memory optimization, load balancing, execution time, thermal behavior, etc. These non-functional properties are meant to induce longer battery life, smoother user interaction, better quality in the transmissions or more reliability. This dissertation presents various techniques and tools for tackling the design challenges that are unique to ERTSs. We propose to combine "native simulation", "symbolic execution" and "trace refinement" to analyze the non-functional properties of embedded systems and demonstrate that such combination advances the state of the art in the analysis and verification of embedded real-time systems.

Contents

List of Figures

List of Tables

"enforme de pogresos 1 marso 3"

– Daniel Keyes, Flowers for Algernon

1 Introduction

From the most mundane activities of our daily lives, such as listening to music or cooking, to the most formidable endeavors of humankind, such as space exploration, modern embedded systems play a fundamental role in almost all enterprises of our society. Considering that simple calculators devoted to the narrow task of solving differential equations were of the size of a house not so long ago, this is an impressive achievement, to say the least. Still, we are nowadays surrounded by these devices and barely notice or appreciate it. When we drive, they take great care that the powerful machinery that we are operating works in a harmonic and controlled way; when we fly, they measure hundreds of variables per second and orchestrate the complicated process of moving a plane from one place to another; they monitor and in some cases control our heartbeats when we are sick, and keep us in touch with our beloved families and friends when we are away. When they do not fail, embedded systems just make us forget about their existence so we can use our time for more important tasks.

However, the process of designing those systems, as well as their operations, is far from trivial. To correctly integrate the nearly four billion transistors that modern platforms are composed of, the designers need to be proficient with different hardware platforms, compilers, debuggers... as well as the so-called CAD tools to integrate them all together and deal with the vast design space. This thesis aims to provide tools, methodologies, and techniques to help in this labor. In particular, we will concentrate on the design of real-time embedded systems with a special focus on their timing behavior.

Unlike other generic computing systems, real-time computing systems (sometimes called "reactive systems") interact with their environment and need to provide results at a pace that is regulated by the environment that they have to control. Therefore, their correctness depends not only on the fact that they can perform some computation but also on the fact that they can deliver a result before a specified deadline. For example, the anti-blocking system controller in modern cars has to analyze the environment in which it operates (the vehicle speed, the terrain in which it is moving, the steering of the wheels, etc.) and compute the frequency at which the brake is activated in a fraction of a second. This short time frame is as important as (or even more important than) the result of the computation per se, and if the computation exceeds this deadline, the system integrity is at stake.

Therefore, the design of embedded systems usually faces particular problems very different to the ones that arise when creating programs that compute a result, and then terminate. On the one hand, the functionality itself challenges the design of embedded systems when they execute complex algorithms. For example, the encoding of video requires the computation of non-trivial digital signal processing tasks, such as filtering and signal transformations. Non-functional requirements also constrain embedded systems. Some of these constraints are the cost (determined by the selection of the microprocessor, memory, and I/O devices), and (sometimes more importantly) time and energy. These constraints vary between different application cases, but the effect of missing a temporal deadline may range from a subtle annoyance to the loss of human lives.

Real-time requirements, however, are not always so dramatic to harm people. Even when many presentations about verification start with pictures about rockets in flames or catastrophes caused by software miss-operations, the number of those events that can be directly associated with design mistakes are rather low (arguably, thanks to designers that do those presentations). In many other cases, missing a deadline is not catastrophic, and therefore, non-functional requirements are classified in hard and soft requirements. Applications such as the anti-lock braking system or the air-bag are said to be under hard real-time constraints because missing the deadline can cause enormous losses or even loss of life. These systems are said to be safety-critical and usually interact directly with the physical equipment that they control. On the other side of the spectrum, encoding a video stream must be reasonably fast and power-efficient, but it is said to require soft real-time constraints. Even though the system is required to operate at a certain rate by the specification, missing the deadline results in a degradation of the quality of the service but this quality loss is usually acceptable. Soft real-time applications often require a probabilistic guarantee that the execution time is within certain bounds. As a consequence, simulation is commonly used in the first stages of the design flow to *understand* the behavior of the system under certain assumptions about the initial state, and verification is used in the last stages to *prove* that the expected behavior is correct given *any* initial state. There is, therefore, a subtle balance to keep to make the most of the tools used during the different phases of the design of embedded real-time systems. In the early stages of the design, fast tools are required that do not require many transformations or annotations of the source code, in order to be able to simulate the different configurations, and that can quickly disregard those that are unfeasible. In the last stages of the design, the tools are required to be formal and accurate, in order to ensure certain properties of the final product. It is also useful to be able to use some of the information collected during the early phases of the design to accelerate the (otherwise costly) computations that are required for the verification of the last stages of the design. In this dissertation, Chapter 3 presents a simulation framework that is quick and requires minimal annotation and manual transformations of the source code, Chapter 4 presents some verification techniques that can re-use the verification results, from the early stages of the design flow, in the later phases.

In this complicated scenario, some of the questions that might arise during the development of these systems, are as follows:

- Which platform to choose? Nowadays there are a large variety of platforms to choose from, in order to implement functionality. In some of them, we can also make a choice between different amounts of memory or different CPUs. The simple option of choosing the most powerful platform is deemed ineffective if we have to take into consideration the cost. Many platforms might meet our requirements, but many are too expensive to consider. To be able to quickly test many hardware configurations and choose the most cost-efficient solution, we require the possibility of obtaining quick estimates about the performance of the code in different hardware and configurations. Chapter 3 of this dissertation presents several techniques that enable this rapid iteration over different configurations.

- How do the various parameters of the platform affect the execution time? If the selected platform does not meet the requirements, it might seem obvious that a solution can be to increase the CPU frequency. Unfortunately, reality is not that simple; the CPU clock rate might have little effect on the execution time since the program may be limited by the speed or size of the memory, a bottleneck in a bus, congestion in the Network On Chip (NoC), a poor parallelization scheme or the input-output rate. Even in the case that the computation is limited by the frequency of the CPU, solely increasing it might increment the temperature of the chip, dramatically boosting the power consumption and reducing the reliability of the system in the long term. If the chip is equipped with dynamic voltage and frequency scaling (DVFS) capabilities, the increased temperature might make the chip to switch to a less performant mode. In this dissertation, Chapter 3 presents a holistic simulation framework that can explore the interactions between the performance of the chip and the temperature, which is useful for detecting this kind of problem.

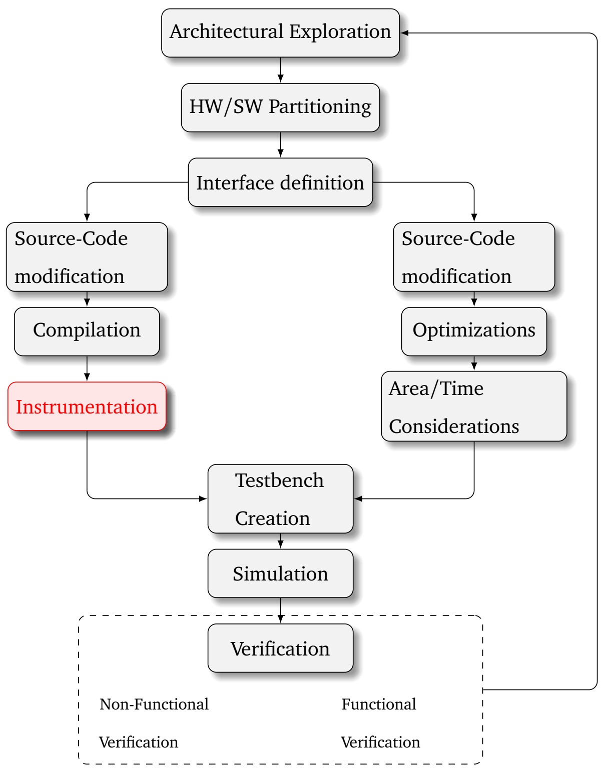
- Will it work? In conclusion, will the complex system still work when operated in an environment in which it may not have been tested before and during a period that might span several orders of magnitude the testing time? Reliability is of particular importance in safety-critical systems and especially when dealing with non-functional validation. These errors may be so harmful that fixing them can require restarting the design from the very beginning and, in some cases, even reconsidering the requirements. Traditional design cycles have focused on functional correctness and relegated the non-functional analysis to the last phases of the design. This methodology (comically called the “fix-it-later” approach by Connie U. Smith [SW02]) is no longer valid for modern embedded systems. If we wait until the system is running to analyze the non-functional properties, when we are able to do so, it will be too late. The number of errors may be overwhelming and impossible to fix. In this dissertation, Chapter 4 presents a tool that can detect functional errors based on a specification given in UML/OCL language.

The computational complexity of obtaining safe bounds for the non-functional properties of a system is related with the certainty with which we provide our answer. Computing the exact value of the worst-case execution time (for example) is an NP-Hard problem that is unattainable even for the most simple cases and simply impossible for real-case scenarios. The approach that this thesis will follow will be to bind these properties in between two values; a lower bound will be provided by the simulation of feasible traces, and an upper bound will be provided by eliminating infeasible traces in an abstract domain. Even when the idea of combining formal techniques with non-functional estimation is not a novel one, the particular details of the implemented ideas and techniques discussed in this thesis are. In particular, this dissertation advances the theory and practice of the analysis of non-functional properties with an emphasis on the combination of native simulation, symbolic execution, and static analysis. The particular combination of these techniques provides a methodology with the following advantages over other state-of-the-art methods.

- The first part of this thesis (Chapter 3) describes several advances in a native simulation technique that enables to obtain estimates about non-functional requirements in the early stages of the design flow. These estimates only require a rough model of the platform under development and, unlike other state-of-the-art techniques, do not need a precise modeling of the instruction set, memory mapping, or functional models of the peripherals. This particular focus on the early stages of the design flow is motivated by the need to choose a valid platform when the application has not been optimized yet for any particular target. We also describe a technique and implementation details about how to obtain performance estimations in designs where only a rough partitioning between hardware and software elements is available. Being able to quickly estimate the effect of accelerating some sections of the code in a hardware accelerator before the tedious work of implementing such parallelization is a significant time-saver and a key element in the effective design flow for heterogeneous embedded devices.

- The second part of this dissertation (Chapter 4) addresses a well-known problem of simulation when applied to ensuring certain properties in the design of embedded systems. As the input of the program and the hardware is unknown, even when a simulation is certainly useful to analyze and understand some properties of the system, the observed behavior offers just a limited view of all the possible evolutions. This partial view is limited to the executions observed in the test-benches. To be able to extend the set of observed behaviors automatically, we describe a *symbolic execution* framework for LLVM programs that generates traces that can be used as candidates to find falsification conditions for properties of programs. We avoid exploring useless traces combining the symbolic execution technique with static analysis. We also extend these techniques developed for the verification of functional properties to the domain of worst-case execution time. In contrast to other state-of-the-art techniques, the described method provides safe and tight bounds, is entirely automated, and does not require a manual mapping between the source code and the final executable.

- In the conclusion part, we wrap up with observations about the techniques and considerations described in previous chapters concerning the benefits that their combination brings to the analysis and design of embedded real-time systems. The requirements of the design toolset change considerably during the different phases that are involved in the creation of an embedded system. Therefore, it is important to analyze how our design tools can adapt to these changing requirements, how we can integrate the results from the early phases of the design into following steps, and how we can join the analysis produced at different abstraction levels to draw conclusions about the non-functional characteristics of the final result.



2 Instrumentation

Most of the techniques described in this thesis can be classified as "Native Simulation". In this approach, the source code is compiled and run in the developer host computer, but during its run, some extra functionality enables to obtain performance measurements about how the code would behave when running on the target platform. To implement these techniques, the code needs to be "instrumented" with extra functionality (function calls in our case) to perform the estimation. These functions run jointly with the executable, usually without modifying the original behavior, but computing some extra information that may be useful for the developer or the designer. This information is called "metadata" by Nicholas Nethercote in his dissertation [Net04]. Nicholas describes in his thesis how this metadata is crucial for dynamic analysis and establishes some terminology that we will borrow in this chapter.

In our approach the code is statically instrumented; we add the instrumentation before running the program, in a phase that transforms the intermediate representation generated by the compiler before producing the final executable. This instrumentation includes (but is not limited to) function calls. An example of this kind of instrumentation is adding a function call to `ADD_OPERATION("r1", "r2", "r3")` after the addition of two registers. These 'hooks' are added automatically by an optimization pass. During the compilation process, the executable is linked with libraries that implement the newly added operations. Libraries perform different analysis that we will describe in subsequent chapters of this dissertation; in general, they update a mapping between variables and their metadata. We say that every real operation performed by the code is "shadowed" by an equivalent operation provided by the analysis framework. In Chapter 3, this metadata is related to the parameters that affect the power consumption and the execution time in single, many-core and hardware accelerated systems, such as cache hits/misses, bus and NoC transactions or the temperature in various elements of the hardware. In Chapter 4, we associate the metadata with a logical formula that we use to express some properties that hold during the execution of the program.

The different techniques described in this dissertation are different instances of "shadowing" functions. As all of them require some common functionality; that functionality will be described in this chapter and referred later when needed.

2.1 Instrumentation

The first step is to instrument the code using "instrumentation functions". This instrumentation could be performed by hand, but the process would be arduous for medium or big sized programs, so we will automate this step by providing optimization passes that will do the instrumentation for us.

[Net04] describes two types of instrumentation depending on the type of code that is instrumented:

- In "source code" instrumentation, the program is instrumented at the level of source code. This category includes analysis performed over the textual representation of a program. Those analyses, therefore, have high-level information about the code at their disposal such as the hierarchical decomposition of datatypes or the control-flow-graph. In Chapter 3, we present a simulation tool that can be used to simulate the execution time of a program when running on a platform by inspecting the source code at a high level of abstraction (C source code).

- In “binary analysis”, the program is analyzed at the level of machine instructions. As the counterpoint of “source code” analysis, those categories lack information about the high-level description of the source code, but have detailed information about the final form that it takes when running on the platform. Compilers usually perform many optimization steps between the initial textual representation and the final binary form. Examples of those transformations include converting recursive functions into iterative ones, combining loops or inlining functions. As the consequence of these transformations, the final representation of the code can be substantially different in its original and binary forms (an example of this can be seen in Figure 2.1). Therefore, estimation tools based on “binary analysis” tend to be more accurate in their analysis of the program, but the lack of high-level information complicates the implementation and limits the availability of those tools in the early stages of the design. In Chapter 2, we present a tool that can reconstruct high-level information (the control-flow-graph) from the binary representation of a program, easing the implementation of various analysis such as the derivation of bounds of the worst-case execution time, as we show in Chapter 4.

In addition to the two representations described before, this dissertation emphasizes the importance of considering intermediate levels of abstraction in the analysis of programs. In [GADSSE10] we perform tests about the accuracy and speed of different techniques situated at various points of the spectrum. The results obtained in this analysis motivated the implementation of an instrumentation and analysis framework that lies between the two “levels” mentioned before: In “Intermediate Representation” analysis, we perform the instrumentation over a representation of the program that is an intermediate step between the source code and the binary executable. These intermediate representations have proved to be valuable in the development of compilers allowing much more code reuse than its monolithic counterparts. In Section 2.1.1, we present an instrumentation

```c
int main(int argc, const char *argv[]) {
  int a = 5;
  int i, ret1=0, ret2=0;

  for (i = 0; i < argc; i++) {
    ret1 += a;
  }

  for (i = 0; i < argc; i++) {
    ret2 += a;
  }

  return ret1+ret2;
}
```

```
main:
        cmp     r0, #0
        movle   r0, #0
        addgt   r0, r0, r0, lsl #2
        mov     r0, r0, asl #1
        bx      lr
```

FIGURE 2.1: Example of an optimization in which the compiler removes various loops. As we can see, in the presence of optimizations, it can be very difficult to relate the high and the low-level representations of the code.

strategy that operates over the intermediate representation of the program generated by the LLVM compiler.

2.1.1 LLVM Instrumentation

For the techniques described in the following chapters of this dissertation, we have used LLVM-IR as the intermediate language to implement our instrumentation functions.

The following sub-sections will describe which elements of the intermediate representation are instrumented and how. For the ease of understanding, some of the run-time functionality that goes along with the instrumentation will be described as well, especially when it is common to various run-time libraries described in the following chapters. We will refer to this section when needed.

Simple example Figure 2.2 (left) presents a C-Code and the equivalent LLVM representation with some operations we want to instrument. The purpose of the instrumentation presented in the present section is to add the instrumentation function calls that appear in Figure 2.2 (right) in an automated way. The semantics (the action that each function

```
int main() {
   int a,b;
   return a+b;
}
```

```
define i32 @main() {
entry:
  %ret = alloca i32
  %r = alloca i32
  %a = alloca i32
  %b = alloca i32
  %r1 = load i32* %a
  %r2 = load i32* %b
  %r3 = add nsw i32 %r1,%r2
  store i32 %r3,i32* %r
  %r4 = load i32* %r
  store i32 %r4,i32* %ret
  br label %return

return:
  %ret1 = load i32* %ret
  ret i32 %ret1
}
```

```
define i32 @main() nounwind {
entry:
  call void beginSim()
  call void beginFn("main")
  call void beginBb("entry")
  ...
  %a = alloca i32
  call void allocaInstr("a","i32")
  %b = alloca i32
  call void allocaInstr("b","i32")
  %r1 = load i32* %a
  call void loadInstr("r1","a")
  ...
  %r3 = add nsw i32 %r1,%r2
  call void binaryOp("r3","r1","r2","+")
  ...
  call void endBb("entry")
  br label %return

return:
  call void beginBb("return")
  %ret1 = load i32* %ret
  call void loadInstr("ret1","ret")
  call void returnInstr("ret1")
  call void endBb("entry")
  call void endSim()
  ret i32 %ret1
}
```

FIGURE 2.2: Example of instrumentation. Note that some simplifications have been introduced in the annotated code for the lack of space (text between quotes in the instrumented code needs to be declared as a global variable and needs to be passed to the instrumentation functions as a pointer).

performs when called) depend on the analysis that is performed and will be explained in later chapters of this dissertation. Note that we might not need all the instrumentation for every analysis, so we might be interested in enabling or disabling the instrumentation of certain elements selectively.

To systematically annotate the variety of instructions from the LLVM intermediate representation, we analyze the instrumentation of the following elements:

- Operations: Instrumentation added to the program when two or more registers are operated to produce a result.
- Control-Flow: Instrumentation added to the instructions of the LLVM-IR responsible for controlling the execution flow of the program (branches and function calls).
- Structural instrumentation: Instrumentation added to handle specific events in the execution of the program that are not associated with any particular instruction (such as starting the simulation or entering and exiting a function)
- Memory handling: Instrumentation added to obtain information about the memory utilization, such as pointer arithmetic, load/store instructions or operations with the stack and/or heap.

Operations: The most simple unit of computation is the update of a register with the value of computing an operation between two other registers. Operations are instrumented in the framework with a function call that receives the name of the three registers and the name of the operation. Some of the comparisons that are involved in dealing with the control flow of a program can also be seen as binary operations, such as comparing two registers previous to a jump instruction and storing the value of the comparison (as a boolean value) in a third register. The supported operations are $\leq$, $<$, $\geq$, $<$ (signed and unsigned), $\neq$, $=$, % (signed and unsigned), left and right shift, and, or, xor, $+$, $-$, $*$ and $/$. During runtime, the particular analysis that we implement on top of the framework is responsible for updating the metadata associated with the destination register considering the input registers as the input to the computation. An example of this instrumentation can be seen in Figure 2.2.

Control-Flow: Basic blocks in LLVM end with a branch instruction (either conditional or non-conditional) that transfer the execution to another basic block, possibly conditioned to a branch condition. Some of the analyses described in the following chapters require from the run-time library not just to be passively observing the program behavior but to be able to control the execution flow. Branch instructions are elements where this behavior can happen. If the analysis requires it, the branch instruction can ask to the run-time library for a successor and branch to that one instead of the one that the execution would dictate. As we will see, this is particularly useful for verification. Besides this case, however, the analysis libraries described in this dissertation mostly “observe” the execution, and only in verification they can interact with the execution flow. We present an example of instrumenting a conditional branch instruction in Figure 2.3.

Structural instrumentation: While not related to particular instructions, the instrumentation also introduces instructions at the beginning and end of functions, basic blocks, and programs. Some of the analysis presented later on can use these functions to initialize their internal structures or to cache different information at different abstraction levels (i.e. information about the execution of a function can be cached at the end_function instrumentation and reused in the begin_function if the function is called again). Instrumentation functions at the beginning and end of a program are also used to initialize global variables. We present an example of this instrumentation in Figure 2.2.

Memory handling: Related to memory handling, we instrument the instructions in the LLVM-IR that are responsible for allocating space (alloca), accessing it (getElementPointer, load and store instructions), and the operating-system functions to implement the heap (malloc, free, realloc...).

The instrumentation of these instructions is responsible for creating the equivalent variables that will hold the metadata associated with each register in the original program. For that, they need the name and the type of the variable. Regarding the name of the newly created variable, it is important to be able to link the variable back to the original one present in the code. This link can be complicated not only by the fact that there may be different variables with the same name in the program (in various statically defined scopes), but also that the same variable can be treated as different separated entities during the execution. In the example presented in Figure 2.4 it is important to understand during execution that the “b” variable in the first example may be overwritten several times with new information and there is no need to use a newly allocated variable to hold

```
int main() {
   char a;
   if(a)
     return 1;
   else
     return 0;
}
```

```
define i32 @main() {
entry:
  ...
  %r1 = load i8* %a
  %r2 = icmp ne i8 %r1, 0
  br i1 %r2, label %bb,
    label %bb1

bb:
  store i32 1, i32* %r
  br label %bb2

bb1:
  store i32 0, i32* %r
  br label %bb2

bb2:
  ...
  br label %return

return:
  %retval3 = load i32* %
    retval
  ret i32 %retval3
}
```

```
define i32 @main() nounwind {
entry:
  ...
  %r2 = icmp ne i8 %r1, 0
  call void cmpInstr("r2","r1","0", "ne")
  %0 = call i1 brInstrCond("r2")
  br i1 %0, label %bb, label %bb1

bb:
  ...
  call void brInstrIncond()
  br label %bb2

bb1:
   ...
  call void brInstrIncond()
  br label %bb2

bb2:
  ...
  call void brInstrIncond()
  br label %return

return:
  call void returnInstr("retval3")
  ret i32 %retval3
}
```

FIGURE 2.3: Instrumentation of a conditional branch instruction. Note that by changing the value returned by the function “brInstrCond”, the run-time library can decide which branch of the if statement is followed.

the associated data because only the last one is relevant. However, in the second example, a new variable needs to be created every time that the execution calls function fib (Figure 2.4), even when the lexical scope of the variables is the same.

```
int sum(n){
  ...
  for(a=0; a < n-1; a++)
    b = b + a
  return b;
}
```

```
int fib(n){
    ...
    a = fib(n-1)
    b = fib(n-2)
    return a+b
}
```

FIGURE 2.4: Example showing the importance of maintaining the dynamic scope when creating new annotated variables

```
int fib(int n){
  ...
  if(n == 1 || n == 2){
    n = 1;
    return n;
  } else {
    a = fib(n-1);
    b = fib(n-2);
    return a+b;
  }
}
```

```
fib(3)
  n = 3
  a = fib( 2 )
    n = 2
    n = 1
    ret 1
  b = fib( n - 2 )
    = fib( 1 - 2 )
    = fib( -1 )
```

FIGURE 2.5: Example of an (erroneous) recursive computation due to an invalid naming of instrumented variables

To account for the different instantiations of variables, the run-time library keeps an update of the dynamic scope that the program is executing and uses this as a binding to name new variables. We implement the dynamic scope as a stack of function names that is updated in every call and return. To see the importance of this disambiguation of variables, suppose an instrumentation library that just propagates the concrete value of every variable (the metadata mimics the actual value associated with every register). Figure 2.5 shows an example of a recursive function, and the (erroneous) computation tree that would result if the instrumentation does not consider different "names" for the various instances of the variable "n".

Once the memory has been allocated, a pointer is used to access to it, and the pointer operations dictate which particular memory position will be accessed. The getElement-Pointer operation in LLVM is responsible for handling pointer arithmetic in the intermediate representation language. Many high-level constructions such as classes, arrays, structures, global variables, references, iterators, function pointers, class inheritance and polymorphism use or directly map to the getElementPointer instruction, and its correct implementation is particularly important in code that requires a detailed representation of the memory instead of an abstraction based on mathematical arrays.

The first argument of the getElementPointer instruction is an aggregated datatype; considered as the base element from which the offset is computed. The second argument is a vector of pointers. The remaining arguments are the indexes that are used to access the successive elements of the array. The amount that each index "advances" the offset relative to the first argument is dependant of the datatype it is associated with.

The annotation pass will, therefore, introduce a function call in the program with enough information to compute the value of the output register depending on the datatypes that are used in the getElementPointer instruction (information available statically) as well as the indexes used in the offsets (information available during run-time).

Because getElementPointer is used to access irregular data structures, we can not compute the final address as a linear combination of indexes in the general case. Instead, the annotator introduces a tree in the instrumented code that can be used to retrieve the offset from the base element based on the indexes values. This tree can be obtained by recursively analyzing the composite datatypes until we reach the primitive datatypes. We present an example of this tree in Figure 2.6.

During runtime, the same tree is traversed based on the values of the indexes to compute the final offset. This offset is later passed to a Load or Store instruction to alter the memory.

Models for the memory: Despite having an instruction called 'alloca', LLVM does not implement dynamic memory management. All the allocated registers are stored on the stack, and therefore, its size must be known at compile time. If a developer allocates or deallocates memory in a dynamic way during execution, the standard library functions 'malloc' and 'free' will be called instead. These functions also need to be intercepted to explicitly model the behavior of the heap and be able to provide the counterpart variables

```
struct RT {
  char A;
  int B[2][2];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};

struct ST s[2];
s[1].Z.B[1][1];
```

```
struct.RT = type { i8, [2 x [2 x i32]], i8 }
struct.ST = type { i32, double, %struct.RT }
...
r1 = getelementptr [2 x %struct.ST]* %s, i64 0, i64 1
r2 = getelementptr %struct.ST* %r1, i32 0, i32 2
r3 = getelementptr %struct.RT* %r2, i32 0, i32 1
r4 = getelementptr [2 x [2 x i32]]* %r3, i64 0, i64 1
r5 = getelementptr [2 x i32]* %r4, i64 0, i64 1
...
```

```
call void getElementPtrInst("r1","s" ,(0,1),tree1)
call void getElementPtrInst("r2","r1",(0,2),tree2)
call void getElementPtrInst("r3","r2",(0,1),tree3)
call void getElementPtrInst("r4","r3",(0,1),tree4)
call void getElementPtrInst("r5","r4",(0,1),tree5)
```

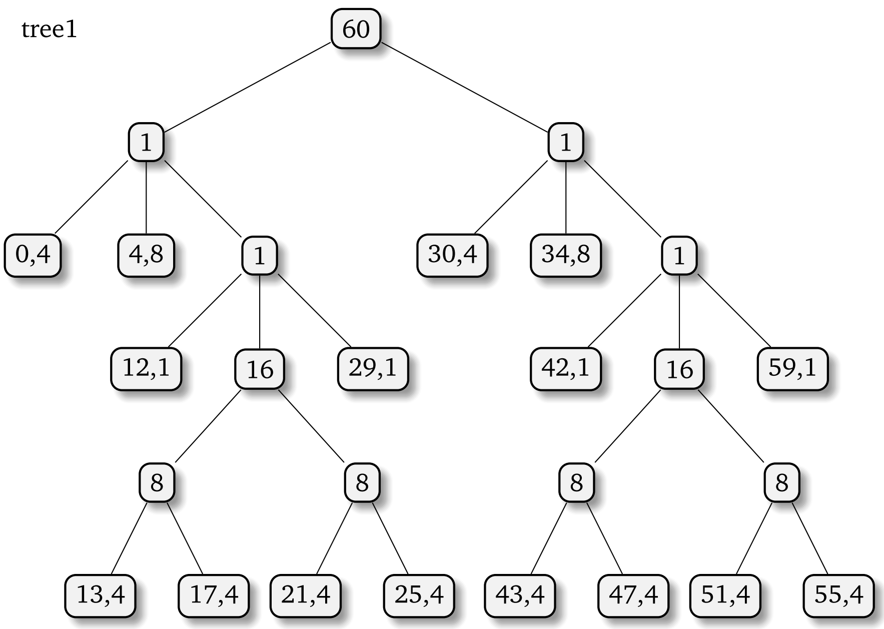


FIGURE 2.6: Instrumentation for array access

with the metadata information. We will delegate the task of updating the variables mapped to the heap to the run-time library. To this end, the run-time library contains code that is common to all analysis passes, which is responsible for handling the heap. A pool of free variables is created at the beginning of the execution and is kept updated during the run of the program. ‘malloc’ and ‘free’ are implemented as operations over these variables, with different levels of detail depending on the analysis that we are performing, as will be described in following sections.

2.1.2 Implementation

We have implemented the instrumentation strategy presented as an LLVM optimization pass. Like other optimization passes, the input of the pass is an LLVM-IR module, that is then traversed and instrumented according to the specification described in this section. The output is an instrumented code also represented in the LLVM-IR intermediate representation. The source code of the instrumentation pass is part of the different tools presented in this dissertation, and the most recent version can be downloaded at `https://github.com/pablo-aledo/forest`

2.2 Binary Analysis

Even when the usage of higher levels of abstraction in the analysis of programs might be beneficial as it represents a trade-off between accuracy and speed, certain analysis can only be performed on binary code. As we will see in Chapter 4, this is the case for example in the computation of worst-case execution time, where even over-estimated semantics of intermediate representation instructions might not be enough if we require a sound analysis of the whole system. For those cases in which the binary form is the only representation of the program that contains all the necessary information, we provide the necessary elements to analyze ARM instruction set as well. This instruction set differs considerably to the LLVM intermediate representation that we have been using until this point, so the remaining of this chapter is meant to provide a summary of the differences to explain what we need to be able to analyze binary code.

2.2.1 Scope

Whereas the ARM Reference Manual includes many extensions of the instruction set and many different architectural decisions, in this work we will focus on a subset of the instructions that are most relevant for timing analysis of embedded system code.

For example, ARM defines the Jazelle extensions that enable native execution of Java bytecode or the Debug extension that allows engineers to analyze the contents of memory or registers using special hardware attached to the chip. We will not consider those extensions in this dissertation.

ARM also uses the concept of coprocessors to support additional instructions such as floating point operations and NEON instructions. Each coprocessor can define new opcodes and registers. As the semantics of these opcodes and registers are delegated to the coprocessor designer, we do not consider them in this dissertation either.

2.2.2 Difficulties in the analysis of binary code

Arguably, the differences between LLVM intermediate representation code and ARM assembly language can be classified into two groups:

- Supporting the ARM instruction set requires supporting the semantics of the new instructions, which are substantially different from the LLVM counterparts. LLVM intermediate representation has been designed with the idea that every instruction is responsible for doing 'one single thing'. This is no longer true in the case of ARM assembly language, in which –for performance reasons–, instructions can include conditional execution, barrel-shifting, pre/post increment/decrement of registers involved in the instruction or usage and modification of implicit registers (register that are not explicitly mentioned in the instruction but that are modified as an effect of the execution). For example, the ARM assembly instruction `POP {R1, R2}` is equivalent to the following LLVM instructions: `STORE R1, SP; ADD SP, SP, 4; STORE R2, SP; ADD SP, SP, 4; ADD PC, PC, 4`.

- Implementation of control-flow and data-flow computation: In LLVM IR the control-flow is independent of data computation. This means that registers are operated with each other via arithmetic operations, and in the cases in which the control-flow of the program depends on the result of some previous computations, the dependencies are clearly identified in previous or next instructions via comparison and branch instructions. While we can directly map LLVM code to an equivalent Control-Flow-Graph, this is no longer true in the case of the ARM assembly.

This last difference is commonly referred in the literature as binary bytecode being “unstructured code”. This term expresses the lack of several attributes of the binary representation of a program:

- The order in which functions are called is kept when callee functions return to the caller. For example, if function `fn1` calls `fn2` and `fn2` calls `fn3`, the order of calling/returning functions is `call fn1, call fn2, call fn3, return to fn2, return to fn1`. In unstructured code, however, the order might be (for example) `call fn1, call fn2, call fn3, return to fn1 .`
- The arguments of function calls are clearly identifiable in the function call.
- Code is structured in basic-blocks; sets of instructions that always executes together. The successor or successors of a basic block are easily identifiable and do not depend on the content of program variables. This property is violated when advanced features such as function pointers are used, but these features are not prevalent in high-level languages.

When the high-level language is transformed into machine code via a compiler, for performance reasons, these constraints are relaxed. These transformations hinder the application of software analysis techniques over the binary source code. In particular

- Functions adhere to a calling convention, but neither the order in which we call functions nor the parameters that we pass to the function are clearly identifiable.
- Indirect jumps are prevalent in binary code, and the destination addresses cannot always be obtained directly by static analysis. Even though function pointers are rather scarce in high-level languages, indirect jumps to addresses that are defined by registers are ubiquitous in binary code. The information about the destination address of an instruction can not be resolved by statically analyzing the code, using local reasoning or pattern-matching.

The differences in the semantics of instructions are expected, and in the following analysis we assume there is a *local* transformation between instructions in ARM assembly language and our internal representation for analysis. For example, we can associate the instruction `PUSH r1` with the instructions `store r1, sp`, `sp = sp - 4` and `pc = pc + 4`. We refer to this transformation as *local* because it only depends on the particular instruction that we are analyzing, and *does not* depend on the context that the program is when we execute the instruction.

However, some of the analysis techniques will require the control-flow-graph of a program as an input. For example, one of the first steps in the estimation of the worst-case execution time (as in Chapter 4) is being able to reconstruct the control-flow-graph from a binary program. This step is of fundamental importance since it is only in the final binary form where the precise information about the instructions executed by the microcontroller resides. While LLVM provides this information by design, because ARM assembly language is unstructured, the reconstruction of the control-flow-graph in binary programs requires further analysis that we will present in this section.

2.2.3 Control-Flow-Graph reconstruction from binary code

This section introduces a technique that can reconstruct the control-flow-graph from a binary executable. This CFG will constitute a model of the program regarding WCET computation. In later chapters (Chapter 4), we will derive how this model can be integrated with a model of the hardware and be used to obtain safe estimations of the worst-case execution time.

2.2.4 Algorithm for reconstructing the Control-Flow-Graph

In this section, we introduce an example that shows the difficulties of control-flow-graph reconstruction in binary code. We will deduce what is needed to solve this example and generalize our approach afterward.

Figure 2.7 (left) shows the program of which we want to compute the CFG. It comprises two function calls and a non-bounded loop. Figure 2.7 (right) shows the generated assembly language by the `arm-linux-gnueabi-gcc` compiler and Figure 2.8 shows the final CFG.

```c
int fun2(){
  return 3;
}
int fun(int n){
  int a,b;
  for(a = 0; a < n; a++){
    b++;
  }
  return fun2();
}
int c_entry(){
  int n;
  fun2();
  return fun(n);
}

```

```
00010014 <fun2>:
   10014: mov r3, #3
   10018: mov r0, r3
   1001c: bx  lr
00010020 <fun>:
   10020: push  {lr}
   10024: sub sp, sp, #20
   10028: str r0, [sp, #4]
   1002c: mov r3, #0
   10030: str r3, [sp, #12]
   10034: b 10050 <fun+0x30>
   10038: ldr r3, [sp, #8]
   1003c: add r3, r3, #1
   10040: str r3, [sp, #8]
   10044: ldr r3, [sp, #12]
   10048: add r3, r3, #1
   1004c: str r3, [sp, #12]
   10050: ldr r2, [sp, #12]
   10054: ldr r3, [sp, #4]
   10058: cmp r2, r3
   1005c: blt 10038 <fun+0x18>
   10060: bl  10014 <fun2>
   10064: mov r3, r0
   10068: mov r0, r3
   1006c: add sp, sp, #20
   10070: pop {pc}
00010074 <c_entry>:
   10074: push  {lr}
   10078: sub sp, sp, #12
   1007c: bl  10014 <fun2>
   10080: ldr r0, [sp, #4]
   10084: bl  10020 <fun>
   10088: mov r3, r0
   1008c: mov r0, r3
   10090: add sp, sp, #12
   10094: pop {pc}

```

FIGURE 2.7: Illustrating example

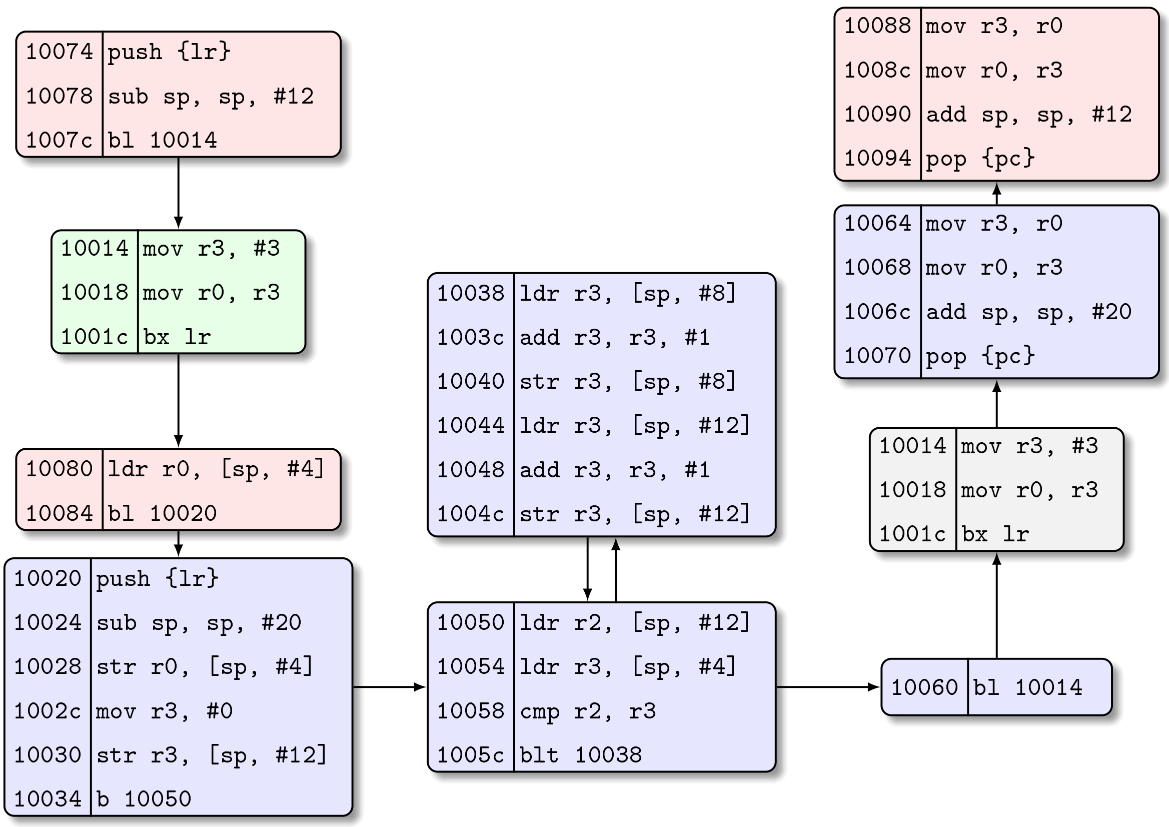


FIGURE 2.8: CFG of program in Figure 2.7

The proposed example can be used to illustrate some of the difficulties in the reconstruction of the control-flow-graph from the assembly language:

- Usage of branch instructions to register values is prevalent in the ARM assembly language, so in lines 4, 26 or 36 (in the assembly code), it is not immediate what is the successor node in the control-flow-graph.
- Several instructions have different branching values depending on the context; line 4 (for example) can have as the return address the value 10080 or 10064 depending on the caller function that calls fn2().
- The presence of (irregular) loops complicates the reconstruction of the control-flow-graph from execution traces. In this example, the loop in lines 6-8 can be executed an indefinite number of times (the number of times that this loop is iterated depends

on the uninitialized variable 'n' –line 12 of the C source code), so the execution of the program might take a long time to reach line 9. In the presence of infinite loops we might be unable to observe all the executions of the program. That complicates the reconstruction of the CFG by execution of various traces.

Based in the example presented, we can see what information we need to extract the control-flow-Graph from the binary code:

- Although some of the instructions in the CFG have an immediate successor, some others require executing the code to obtain the successor.
- We need to disambiguate nodes based not only on their addresses but also depending on the context in which the instruction is executed.

To implement the CFG reconstruction from the binary code, we present an algorithm that alternates between three phases (CFG unfold, analysis and simulation). The algorithm starts by adding the entry point of the program as the initial node in the control-flow-graph. Then, the graph is iteratively unfolded until a successor cannot be computed statically just by inspecting the binary code and some run-time information is required. An example of this situation is that we find a `bx lr` instruction –`bx lr` continues the execution flow in the instruction pointed by the value of `lr`, that is unknown at compile time–. Then, we analyze the partial control-flow-graph that we have obtained until that point to obtain the set of instructions that are relevant for computing the value of the successor (analysis phase). Following the previous example, we compute which instructions may affect the value of `lr` in the instruction `bx lr`. The last phase executes those instructions and computes the successor node(s) for the given instruction. Applying the three phases until there are not nodes left to expand, we can reconstruct the full CFG.

To exemplify the execution of the algorithm we start with the program of Figure 2.7. We start by adding the singleton address and instruction `10074 push {lr}` and initiate the *unfold* phase. Instructions `10078, 1007c, 10014, 10018` all have an immediate successor that can be computed statically (without the need of executing the code). Therefore, the algorithm adds them to the CFG creating the (partial) graph that we

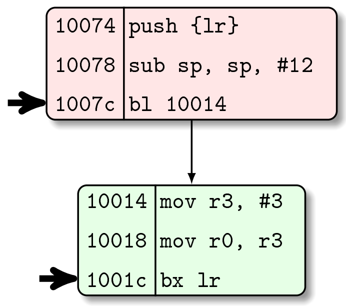


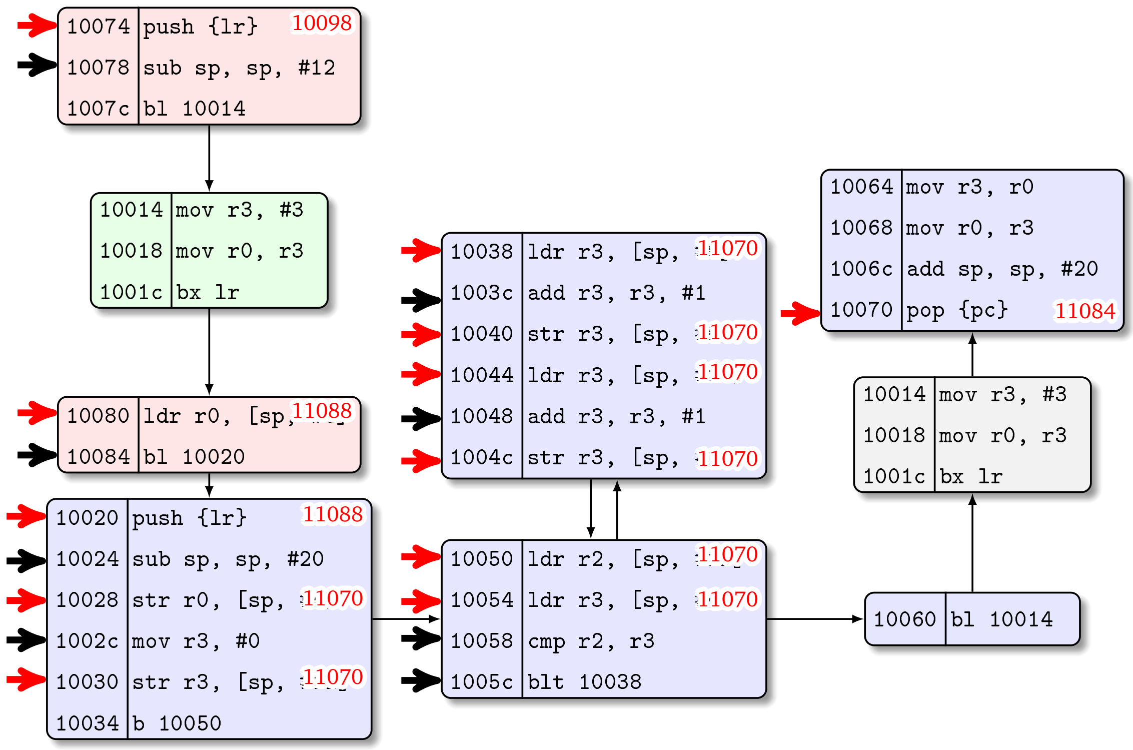
FIGURE 2.9: First iteration of the *expand* phase of the CFG reconstruction.

present in Figure 2.9. Note that during the addition of nodes, we take into consideration function calls and returns and add nodes to the graph with information about which dynamic scope the program is in when we execute the instruction. We represent this information in the figure using different colors.

When the algorithm reaches instruction `1001c bx lr`, because the value of `lr` is unknown at compile time, we need to execute the partial CFG and retrieve its value to know what is the successor of the instruction. If we execute more instructions that what are essentially required to obtain the value of `lr`, however, the execution may follow a different path and do not reach the instruction `1001c`. It is important, therefore, to analyze the program and execute only those instructions that can affect the value of `lr`. To compute this set of instructions we use a *slicing* technique based on Weiser algorithm [Wei84]. Weiser's algorithm propagates the dependencies of each instruction in the CFG based on the *uses* and *defs* of its instructions. These are attributes of each instruction defining what registers are used and modified by each instruction. To have into consideration the fact that some instructions affect the memory as well, we extend the attributes *use* and *def* as explained later.

In the case of instruction `1001c bx lr`, the required instructions to compute its successor are marked in Figure 2.9 with an arrow. The *execution* phase then executes those instructions and retrieves the value of `lr` at the end of the execution. The next node in the CFG can be computed, and the algorithm can iterate again.

We can iterate the same procedure until we reach node `10070 pop {pc}` – this instruction gets the value of the memory pointed by `sp` and continues the execution from

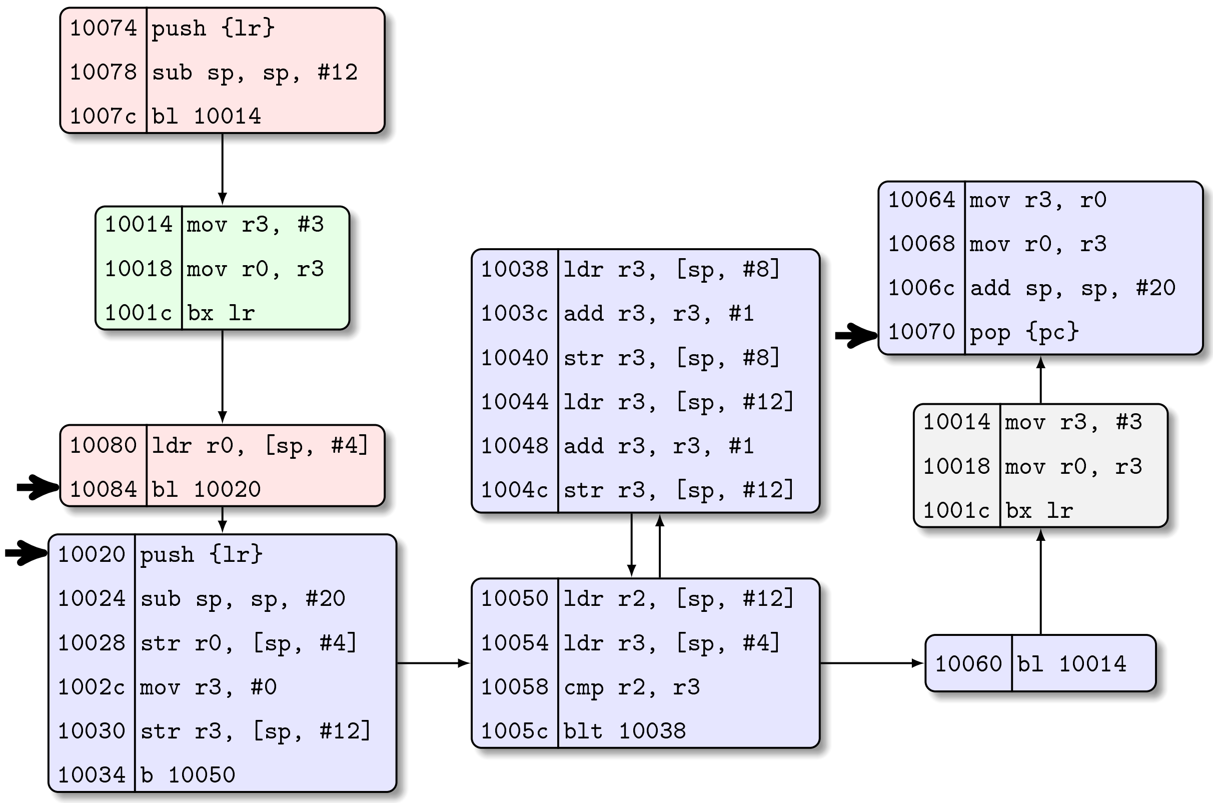


```
10074 push {lr}        10098
10078 sub sp, sp, #12
1007c bl 10014

10014 mov r3, #3
10018 mov r0, r3
1001c bx lr

10080 ldr r0, [sp,11088
10084 bl 10020

10020 push {lr}        11088
10024 sub sp, sp, #20
10028 str r0, [sp,  11070
1002c mov r3, #0
10030 str r3, [sp,  11070
10034 b 10050

10038 ldr r3, [sp,  11070
1003c add r3, r3, #1
10040 str r3, [sp,  11070
10044 ldr r3, [sp,  11070
10048 add r3, r3, #1
1004c str r3, [sp,  11070

10050 ldr r2, [sp,  11070
10054 ldr r3, [sp,  11070
10058 cmp r2, r3
1005c blt 10038

10060 bl 10014

10014 mov r3, #3
10018 mov r0, r3
1001c bx lr

10064 mov r3, r0
10068 mov r0, r3
1006c add sp, sp, #20
10070 pop {pc}         11084
```

FIGURE 2.10: First phase of the *analysis*. Coarse slicing and sp computation.

there–. Again, the successor of this node can not be computed statically. Also, in this case, we have the additional problem that the value that we require is stored in the stack, and therefore, the analysis needs to be more precise to get the particular value that we are interested in. To have into consideration the effect of the stack, we split the slicing phase of the algorithm into two parts: First, we compute the slice considering the stack as a single entity, and propagating it through the *use* and *def* of instructions as if it were a register. This first iteration of this slicing is presented in Figure 2.10.

For the instructions that either use or modify the stack and are contained in this first iteration of the slice (marked in red in Figure 2.10), we compute the value of the register sp when the execution flow reaches those instructions (represented in Figure 2.10 at the right of each instruction). Again, this implies slicing and executing, but in this case, we do not need to treat the stack in a special way. The output of this second phase of *slicing* is a decorated CFG in which some nodes have an attribute with the value of sp in them. We



```
10074 push {lr}
10078 sub sp, sp, #12
1007c bl 10014

10014 mov r3, #3
10018 mov r0, r3
1001c bx lr

10080 ldr r0, [sp, #4]
10084 bl 10020

10020 push {lr}
10024 sub sp, sp, #20
10028 str r0, [sp, #4]
1002c mov r3, #0
10030 str r3, [sp, #12]
10034 b 10050

10038 ldr r3, [sp, #8]
1003c add r3, r3, #1
10040 str r3, [sp, #8]
10044 ldr r3, [sp, #12]
10048 add r3, r3, #1
1004c str r3, [sp, #12]

10050 ldr r2, [sp, #12]
10054 ldr r3, [sp, #4]
10058 cmp r2, r3
1005c blt 10038

10060 bl 10014

10014 mov r3, #3
10018 mov r0, r3
1001c bx lr

10064 mov r3, r0
10068 mov r0, r3
1006c add sp, sp, #20
10070 pop {pc}
```

FIGURE 2.11: Second phase of the *analysis*. Accurate final slice.

can use this information to compute a more *precise* computation of *use* and *def* in which (instead of considering the stack as a single entity), we can decompose it in its internal memory positions and propagate this more detailed information about the dependencies. This step enables us to compute a more tight slice that is presented in Figure 2.11. Finally, applying the execute phase again, we can obtain the successor and the algorithm can continue until the full CFG is reconstructed (Figure 2.8).

Algorithm 1 summarizes the previously described steps and presents a technique that can obtain the control-flow-graph of a program from its binary representation.

The following functions parameterize the algorithm:

slice: given an instruction i and the set of registers that are required to compute the successor (req(i)), *slices* the program and returns an over-approximation of the set

Algorithm 1 Algorithm for CFG reconstruction from binary code

```
while CanExpand(G) do
    G ← G ∪ unfold(analyze(G,frontier))
end while
procedure ANALYZE($G$, $f$)            ▷ returns a successor for every node in the frontier f
    ∀ n ∈ $f$
        i ← instructionAt(n)
        s ← slice(annotate_sp(G,i), i, req(i))
        ⟨ c, v ⟩ ← execute(P, s, c, i)
end procedure
```

of instructions that can affect the value of req(i) in the program. We have developed a Scala library implementing the propagation of dataflow computations and Weiser's algorithm to compute slices in binary code.

execute: given a program P, a set of instructions s, the current scope of the program c and the current instruction i, executes the program until the instruction i is reached in the scope c. It returns the new scope and the value of the registers that are required to compute the branching address of the instruction i.

2.2.5 Implementation

We have implemented a proof of concept of the ideas explained before that can be downloaded at `https://github.com/pablo-aledo/binanalyzer`. We present the general architecture of the developed prototype in Figure 2.12. `Objdump` is used to perform an initial disassembly of the executable. We have implemented a grammar for the generated assembly language with the parser generator sbt-rats [SCB16] and the language library Kiama [SKV13]. The implementation uses Scala as the main implementation language. The implementation of the `execute` function have been performed by integrating the Scala framework with `GDB` and the `qemu` emulator.

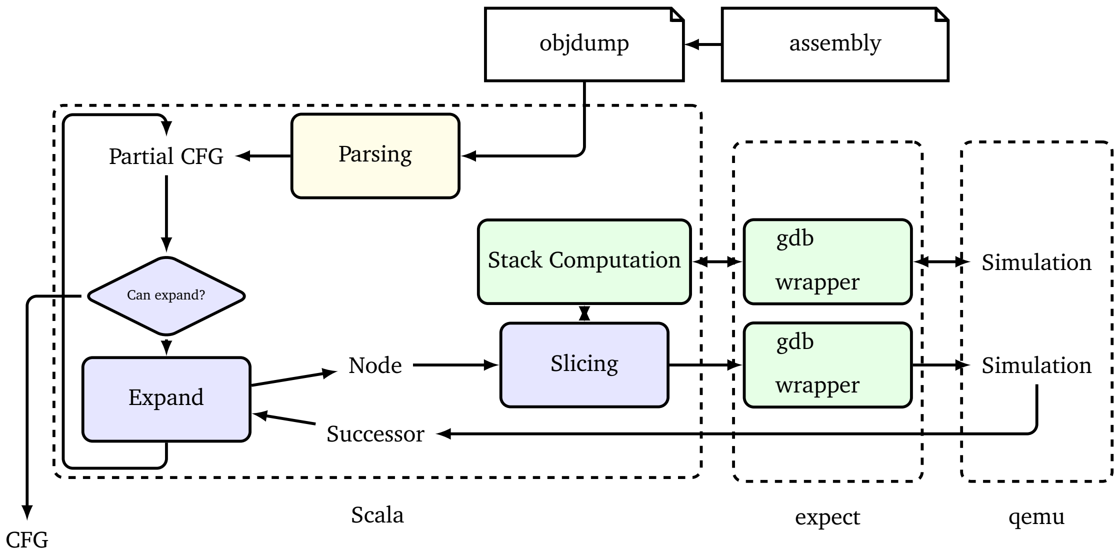


FIGURE 2.12: Architecture of the implemented prototype, languages and interfaces

2.2.6 Results

We have applied the presented algorithm to a set of test-cases from the Mälardalen benchmark suite [GBEL10]; a set of binary programs specifically targeted to benchmark worst-case execution time tools (see Chapter 4). A short explanation of the computation performed in each program is provided in the following list.

- insertsort : Performs the insertion sort algorithm in a linear array.
- fdct : Forward Discrete Cosine Transform.
- matmult : Multiplies matrices to tests a compiler speed in handling multidimensional arrays and simple arithmetic.
- janne_complex : Test interdependencies between two nested loops.
- bs : Performs binary search in a linear array.
- cover : Tests the coverage of if statements.
- fibcall : Computes the Fibonacci series.
- crc : Tests a CRC (Cyclic Redundancy Check) operation.

- jfdctint : JPEG integer implementation of the forward Discrete Cosine Transform.
- duff : Forces the compiler to emit unstructured loops, which is usually problematic for WCET tools to handle.
- lcdnum : Demonstrate effect of flow facts for straight loops.
- ns : Test interdependencies between nested loops.

Table 2.1 shows some statistics about the computed CFGs of the mentioned programs. As we can see, the algorithm is capable of computing the control-flow-graph in a large subset of the mentioned benchmarks even with different optimization levels.

Figures 2.13 and 2.14 show some of the medium-sized graphs. Note that even for relatively small examples, the resulting control-flow-graph can be notably intricate. The generation and verification of correctness of this extraction is notably complicated to be performed by hand, and also due to the huge amount of transformations that the code suffers from the high-level representation to the assembly one – LLVM/Clang compilers, for example, are composed of around 2.5 millions lines-of-code – the automated matching of high and low level constructs is impossible to be performed automatically.

This is why, as we will see in future chapters, the reconstruction of the control-flow-graph is of key importance to obtain tight and safe bounds for the worst-case execution time. There are tools in the state of the art that rely on the debug information to translate formal results from the C source code to the binary analysis (such as loop bounds). Due to the vast amount of transformations that the compiler can introduce between these two representations of the code, the mapping can be difficult to automate or –even worse– incorrect, so the underlying binary analysis can produce incorrect results for the worst-case-execution-time of the program.

TABLE 2.1: Mälardalen reconstruction results

	Optimization level -O0				Optimization level -O2			
name	#instr.	asm size	#nodes	#edges	#instr.	asm size	#nodes	#edges
insertsort	84	2636	81	82	47	2620	44	46
fdct	680	6408	665	666	281	7116	268	269
matmult	175	4376	230	236	142	5488	86	93
janne_complex	67	522	65	69	50	2872	28	32
bs	63	3000	60	62	44	3556	17	18
cover	1047	8584	95	97	469	6480	58	63
fibcall	48	2520	46	47	27	2840	5	4
crc	286	4908	510	527	120	5612	181	198
jfdctint	549	5592	534	536	244	6900	229	232
duff	145	3424	63	63	95	4168	27	28
lcdnum	100	3032	48	49	36	2980	23	24
ns	126	8088	122	126	50	8040	4	3

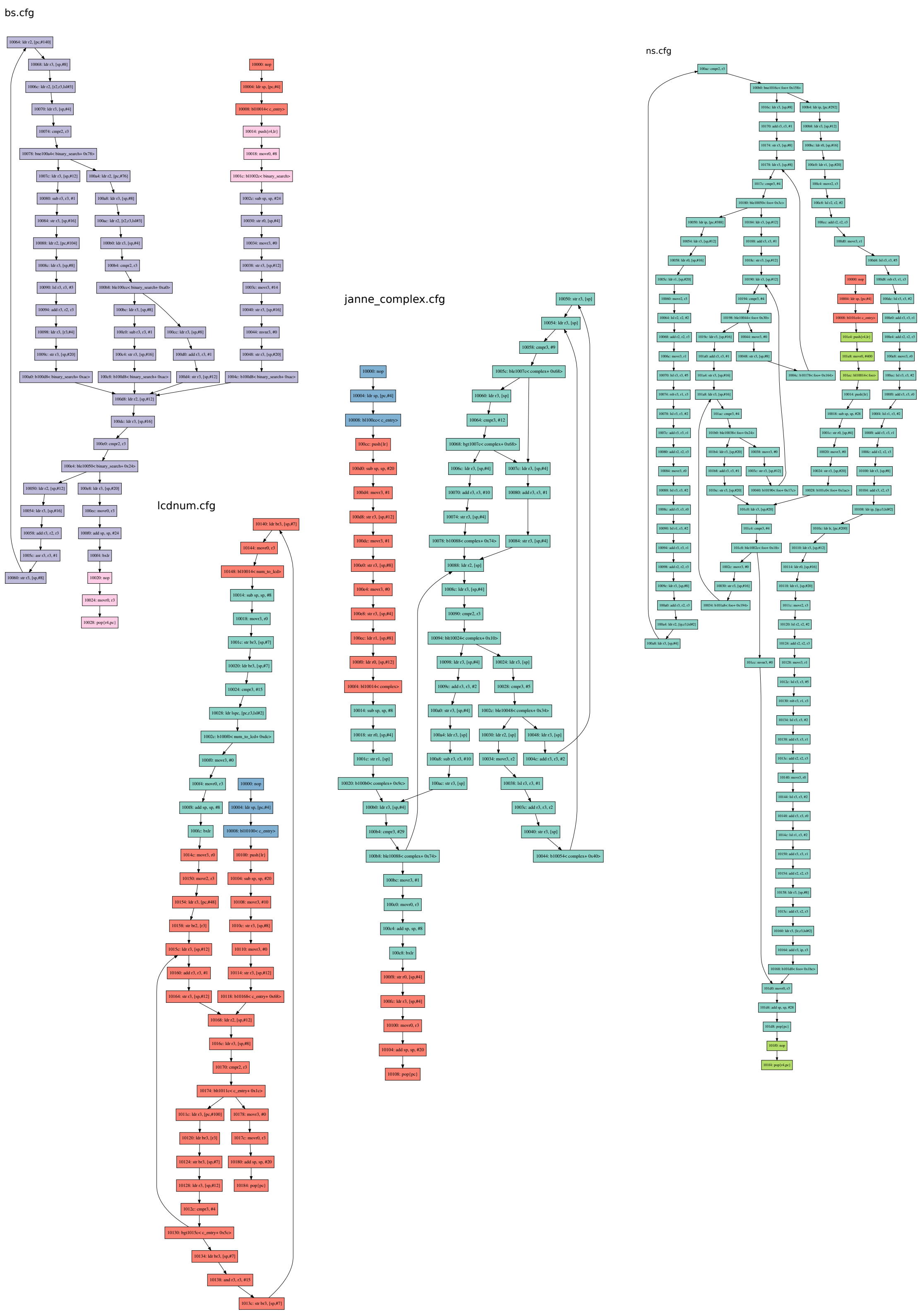


FIGURE 2.13: CFG of some medium-sized implementations.

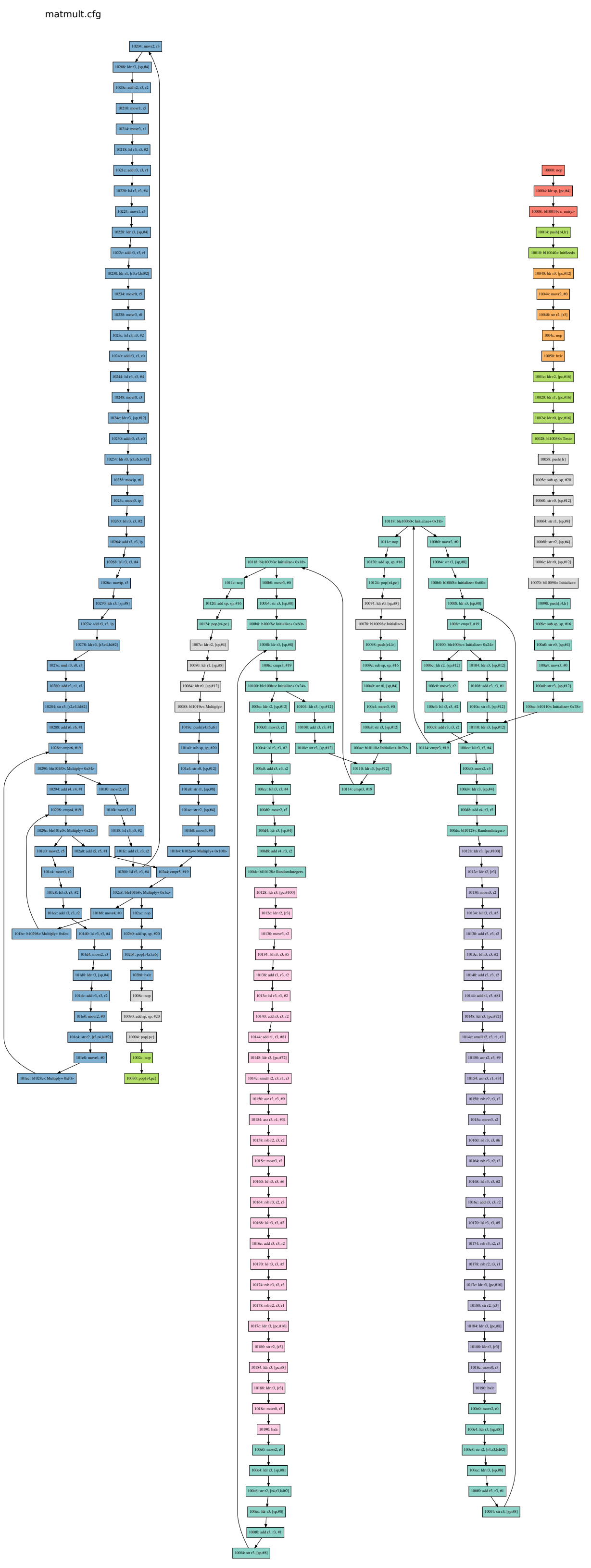
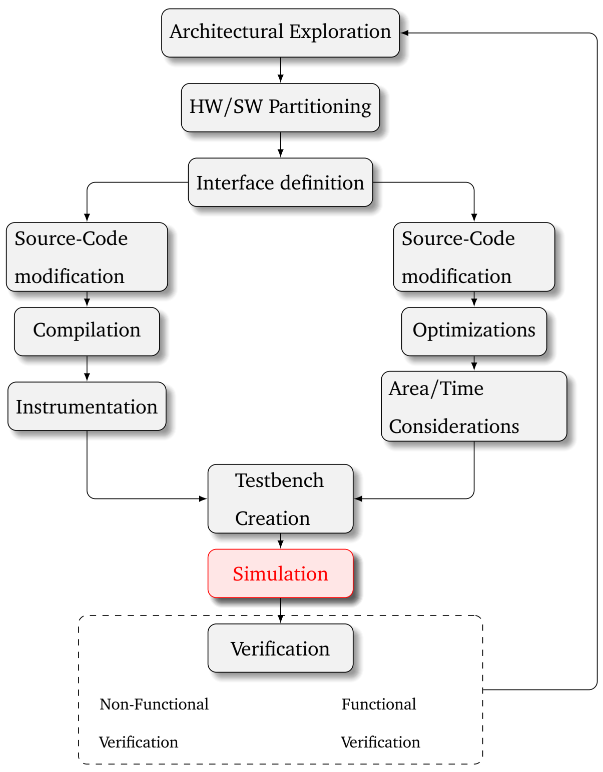


FIGURE 2.14: CFG of some medium-sized implementations.



3 Simulation

3.1 Introduction and state-of-the-art

In this chapter, we will describe techniques to estimate the non-functional properties of a program by running it in a simulated environment.

The main purpose of simulation is to provide insightful estimations of non-functional properties of a program when it executes on a platform, such as the number of cache misses, the energy consumed by the embedded system or the thermal behavior. A simulator needs to be quick and reasonably accurate to be useful, and it should require the minimal intervention of the user when modifying both the program (as a consequence of multiple optimization iterations) and the hardware platform (as a result of unfulfilled requirements).

Both modern programming languages and hardware architectures are complex to simulate, and a detailed description of all the elements that compose them would overwhelm the developer and burden the simulation. Those difficulties have motivated the development of a simulation environment that we describe in this chapter.

To make the analysis of the program unobtrusive for the developer we rely on the automatic annotation techniques that we have described in Chapter 2 [1]. To make the analysis of the hardware architecture quick and accurate enough, we provide reusable high-level models that can be adapted to a variety of platforms.

In this chapter, we start by introducing several techniques (not only simulation) to understand the interactions between hardware parameters and non-functional properties better. We then discuss what the main limitations of those techniques are and our approach for reducing those limitations. Our technique divides the simulation domain in "simulation of single CPU's", "simulation of many-core" and "simulation of hardware accelerators". For each one of these sub-domains, we introduce the state-of-the-art and present what parameters of the domain are most critical to be simulated, to keep the simulation quick and reasonably accurate. Then we introduce our models, results when applying the techniques to small examples and results when applying the techniques to optimize real-life industrial or research projects. These examples are meant to showcase the usability of the presented techniques when applied to big codebases and how the understanding of the interactions between the previously selected parameters and the non-functional properties can lead to significant improvements in the non-functional properties of the program (mostly concerning execution-time). These case studies are presented as practical examples of the utilization of the developed tools but are not essential for understanding the techniques described in the dissertation.

There exist several techniques to study the non-functional properties of programs that execute over a platform. We classify them as:

[1] Note that even when we present the annotation of simple examples to illustrate the concepts during the exposition of this thesis, all the annotation steps performed during the evaluation in each section are performed completely automated by using the tools and techniques described in Chapter 2.

Virtual Platforms

They consist of building a software model of the platform and use it to run the program under analysis. The main benefit of these techniques is that they can be used before building the real platform. At this point in the design, the design space is still very big, and therefore the main objective of these tools is to be able to reduce it. A common technique used in these cases is to combine a design-space exploration tool with a model for the non-functional requirements. For this approach to be feasible, it is important that the simulation can be performed quickly (several orders of magnitude faster than an accurate simulation), and the relative effect of the change of different parameters in the properties of the system is more important than the absolute one.

We can classify “virtual platform techniques” into sub-categories; in “Register-Transfer-Level” we describe the platform at the level of individual hardware registers, in “Transaction-Level-Modeling” we split the platform into modules, and we only focus on the interactions between those modules. In “Instruction-Set-Simulation”, we only concentrate on the individual instructions of the Instruction-Set-Architecture of the microprocessor, and we associate a cost (regarding energy, time or power) to each instruction.

Estimation based on hardware counters

Today, some microprocessors have embedded programmable event counters to measure performance. Several techniques have been developed to estimate non-functional properties of a program (such as execution time, cache misses or bus performance) of a program based on the value of those counters when we execute the program on a platform where they are available.

These techniques are usually of very high accuracy [LJ03], and the fact that the counters are hardware-based makes them very unobtrusive. The explanatory power of these techniques is also high because the purpose of every counter is clearly defined and targeted to detect specific bottlenecks in the architecture.

The main disadvantage of the techniques is that, in many specific microprocessors, these counters are not available, are limited or are not standard. Moreover, in many cases, the platform is not yet constructed, so these techniques are not available in the early stages of the design flow. Therefore, these techniques are more focused on improving and optimizing software for an already established platform rather than selecting the best combination of hardware and software parameters for a particular application. Some examples of these techniques are described in [LSP08][LJ03].

Black-Box macro-modeling

These methods aim to partition the system into several sub-components (either hardware or software), manually characterize and model the different sub-components, and formalize the framework to combine the estimations of these components to derive the overall properties. An example of black-box macro-modeling technique can be to observe that the execution time of a program that writes to a Unix file descriptor can be approximated by $10 \times size(GB)$ sec. High-level system description [NM01][DDSL08] is an instance of black-box macro model where the entire platform is considered as a single black-box. For this technique, a set of hardware and software parameters are defined, and the whole platform is characterized based on them. These parameters are chosen based on experience, although some techniques to automate characterization have been analyzed [TR01][MRR07]. This technique provides very low execution time but also low accuracy. The approach is strictly constrained to particular applications, and it is not easily generalizable (some systems are accurately modeled with this method while others are not, without an apriori way of knowing to which group our system belongs). [TR01] [MRR07] [BBF16] describe some examples of these techniques.

3.2 Limitations of the state-of-the-art techniques

As recalled in the introduction, embedded systems require simulators that can simulate entire platforms, including memories, buses, peripherals, HW accelerators... Additionally, to meet the requirements of embedded system design, multi-cores and many-cores are now mainstream. System modeling using simulation and virtual platforms has become an effective way of handling the demands of complexity, reliability and development time of these systems, so we will focus on these techniques and try to extend its applicability rather than the other ones presented in the state-of-the-art.

In the previously mentioned techniques, we can see a gap in the early phases of the design process regarding the non-functional properties of the system; currently, there is a lack of tools that can be used effectively in the initial phases of the design to guide the design space exploration efficiently. Among other problems, we think the following aspects are of fundamental importance, and we will focus on these elements in this chapter of the thesis:

Limitations in speed The state-of-the-art techniques are very accurate but slow. Some of the examples presented in this thesis and independent studies show that the RTL models of a platform are at least 20 times slower than more abstracted models such as TML or ISS simulations [CSC+09]. To put these numbers into perspective, that means (for example), that simulating a video codec with an RTL virtual platform would imply 2 hours per frame at RTL level, compared to 6 min at TLM or close to real-time (33 ms) in native-simulation.

Limitations in usability The available techniques require several transformations in the design and some work in the tools that implement those techniques until the input fits the tool at hand. Sometimes this work requires already some decisions about the platform; the same decisions that the framework is meant to solve. For example, ISS simulators require a binary version of the executable, already compiled for a particular platform. Developing the ISS simulator, porting the operating system, implementing the drivers, deciding and implementing the hardware elements of the design and optimizing and porting the application for the particular selected platform are some of the steps that need to be performed before a single result about the availability of the platform for the task at hand can be obtained. All the effort of developing those components can be futile if the obtained results are negative.

In the case of many and multi-core platforms, development is difficult because of conflicting requirements: (1) to reduce the cost of development and design, higher level programming constructs such as OpenMP and MPI are required. (2) Improving the

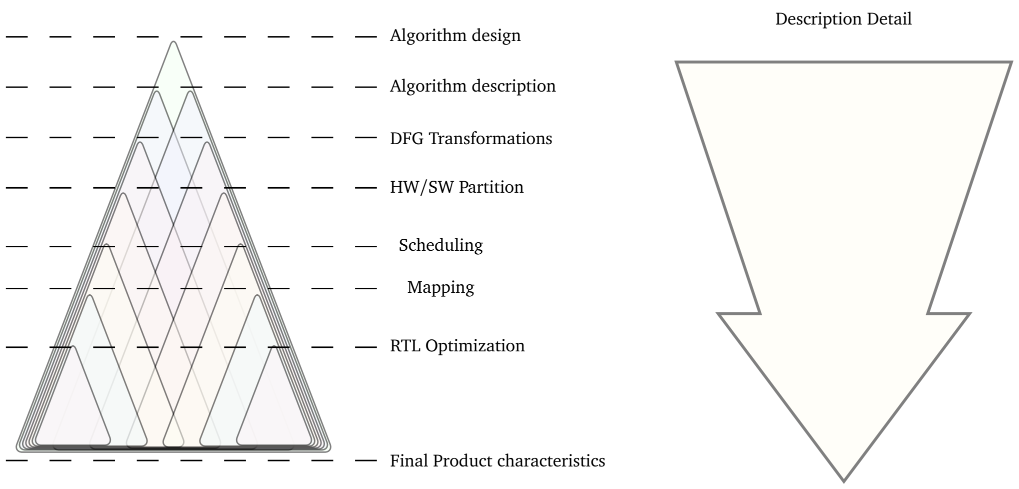


FIGURE 3.1: How design decisions affect the final characteristics of a system

performance of the application over a many-core platform often requires a fine match between the hardware parameters (number of processors, memory sizes, bus speed or Network-On-Chip routers) and software variables. We can only explore this match by low-level programming.

3.3 Overview of our approach

As Figure 3.1 shows, the possibility of improving a system is bigger in the early stages of the design because of the higher number of factors that can be considered and modified. The main aim of the estimation tool that we will present in this chapter is then to provide performance results as early as possible in the design process. The techniques described in the next sections have been developed after the following observations:

The execution time and power consumption of a CPU are highly correlated with the number of instructions executed [LJ03][LSP08]. This is true to a certain extent even in modern architectures with branch prediction, out-of-order execution, and complex pipelines. However, the assumption breaks in the presence of the following elements

- Caches [PDV11]: The size, speed, and associativity of both data and instruction caches greatly affect the rate at which the program can process data and execute, so more level of detail is needed to keep a reasonable accuracy in the simulation.

- Parallelism: Both at a per-core level and at hardware-level, the presence of parallelism in the program or the platform drastically affect the non-functional properties of the system. Because of that, new techniques to deal with parallelism at the operating system level (how to handle threads in multi and many-core platform simulations) and at a hardware level (how to integrate the CPU with hardware accelerators) have to be implemented to keep accurate results in the simulation.

- Dynamic Frequency and Voltage Scaling [SLD+03],[PPB07]: Attributes that have been previously considered to be very detailed to simulate with a coarse-level of abstraction can indeed be considered at that level with acceptable results. An example of that is the modeling of the thermal behavior of modern chips (that have been traditionally considered at RTL level). As we will demonstrate in Section 3.7, the main element that prevented the usage of a higher level of abstraction was the presence of DVFS adaptive elements in some platforms. A high-level model of these elements and its interface with the software through the operating system libraries enable the modeling of the thermal behavior of modern SOC platforms and provide enough explanatory power to be useful in the first stages of the design flow.

To tackle those difficulties, the main state-of-the-art approach is to lower down the description level of the whole system to more accurate levels of detail. However, that solution burdens the simulation and forces the majority of the system to be simulated at a level of abstraction that is very accurate for the purpose that it serves. If we can model those components in a more detailed way but still consider the CPU at a higher level of abstractions, a fast and accurate virtual platform can be created.

3.4 Simulating the CPU part of the system

3.4.1 Introduction

In this section, we will focus on simulating systems with a single CPU and a memory divided into several levels of caching. Even a simple architecture like this can be analyzed at different levels of detail, and choosing the most appropriate representation of the system can lead to noticeable differences in terms of simulation speed and accuracy.

This section starts by presenting several techniques to simulate this kind of platforms. Then we present our approach. Our technique supports different abstraction levels, and we compare them regarding accuracy and execution time.

The main contributions are:

- We compare a technique based on C-source-level time estimation with two techniques based on assembly language regarding accuracy and execution time.
- We present a technique to cluster assembly instructions and speed up the simulation that presents some advantages over the state-of-the-art alternatives.
- We characterize these clusters of assembly instructions at different abstraction levels and compare the results obtained in terms of time estimation accuracy.

3.4.2 State of the art

In the following, we provide an introduction to the state-of-the-art techniques to implement virtual platforms. We classify them into the following groups:

- Register-Transfer Level (RTL) modeling [GS02][BTM00]: Every hardware register in the platform has a model in the simulation, and the behavior is described as the propagation of data through these registers. This method is very accurate but also very slow. Microprocessor, memories, buses, peripherals and any hardware component of the platform can be modeled without losing any generality

- Transaction-Level Modeling (TLM): TLM simulators abstract the hardware behavior into several components and focuses on timed events on the communications between them. Focusing only on the communication part of the transactions and abstracting away the internal timing details of each model it is possible to increase the simulation speed with the added benefit of reducing the development time.

- Instruction-level Modeling [FAM08][LLSV99]: This level of abstraction suppresses the hardware-specific details of the platform and only focuses on microprocessor instructions. Instruction Set Simulators (ISSs) characterize the time spent on each instruction, so a description of the entire instruction set is needed. [KHH03] proposes to characterize pairs of instructions to consider the effects of the internal pipeline but the execution time of this solution is prohibitive for simulation in early design phases.

- Native simulation [GGP08][BGF+10][PDV11]: These approaches execute the application source-code in the host computer (the PC that the developer uses to develop the implementation). Some additional code is included in the application to extract information during execution. This information is used to estimate several parameters such as target execution time and power consumption. This approach has several advantages: as the speed of the host computer is high, these methods are quick. They are also convenient because the developer can use them from the first steps of the design process, and in the same environment used to develop the application.

3.4.3 Our approach

We consider four different techniques to perform software execution estimations in this section. They are based on source-code analysis, instruction-level modeling and black-box macro-modeling. Our method describes the hardware parameters in SystemC (a C++ library and specification language to describe heterogeneous hardware-software systems), so we can simulate the hardware peripherals of the platform such as Ethernet MAC controllers, memories, and buses. Our approach can be used with a general compiler (gcc), so a fast exploration of architectures can be performed without the need to modify a compiler for each type of target architecture.

Next, we will introduce the four techniques we compare in this section. We refer to them as “ISS reference”, “Native-Specific”, “Native-Average” and “C-Level”.

- “ISS reference” is based on a state-of-the-art Instruction Set Simulator with cycle accuracy and is only used as a baseline for reference. Skyeye [KWC+04] is an ISS simulator based on an assembly approach. Each assembly instruction entails an execution time that is dependent on the instruction opcode, the pipeline state and the value of the registers. Skyeye considers Data and Instruction caches. We use this approach only as a gold model and compare other approaches against it.

- “Native-Specific” is based on native simulation. To optimize the simulation time we base our analysis in basic blocks (sets of instructions that are always executed together) instead of individual instructions. We characterize every basic block at assembly level and annotate the code with an execution time per basic-block. This cost depends on instruction opcodes, but we do not consider the value of the operands nor the internal state of the MPSoC (as in “ISS Reference”).

- “Native-Average” is similar to “ISS reference” but we define an average execution time per assembly instruction, assigning the same value for all the instructions. In this case, the execution time of a basic-block is proportional to the number of assembly instructions.

- “C-Level” is a technique based on C-Source Code analysis. In this case, we characterize source code at C-level, using the operator technique explained in Section 3.4.3 [CPVM07][GADSSE10]. The considered code is a high-level view of the functionality of the code and does not take into consideration optimizations performed by the compiler. Also, we do not have details for internal pipeline status, the exact number of instructions per element in the code, cache sizes and policies, or register values.

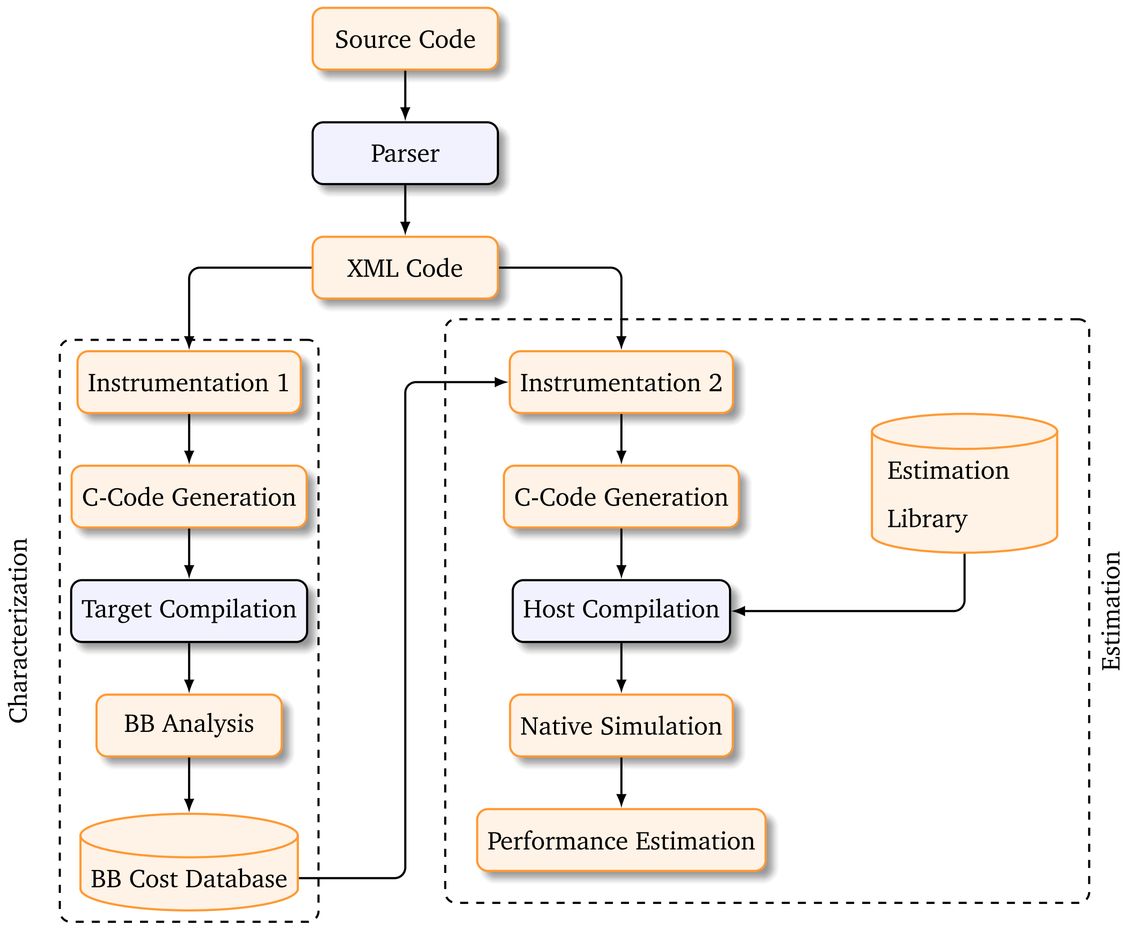


FIGURE 3.2: Time-Estimation methodology. All the steps of this methodology have been automated as described in Chapter 2 and [GADSSE10].

Basic-Block methodologies ("Native-Specific" and "Native-Average")

This section focuses on techniques that operate over the assembly language. First, we present a general overview of the method, a process to extract basic blocks that operates directly on the C-Source code and a way to characterize them that is simple and accurate enough for early design exploration. Figure 3.2 shows the main steps of the process.

```
if(a[0] == 0){
  a[0] = 1
}
```

```
asm("b uc_mark_2__am3")
if(a[0] == 0){
  asm("b uc_mark_3__rm")
  a[0] = 1
  asm("b uc_mark_4__")
}
asm("b uc_mark_5__ai1")
```

FIGURE 3.3: Example of code instrumentation

The application C/C++ source code is analyzed by a parser to obtain a language-independent representation that contains the distinctive elements of the high-level program. In contrast to other basic-block-based approaches for performance and time estimation [TR01], our instrumentation is done directly in the C-source code. The instrumentation process has two steps [CPVM07]:

During the first step (annotation), we insert some assembly directives in the code (Figure 3.3). The goal is to identify the basic blocks even with compiler optimizations. We use the GNU/gcc facilities to mix assembly instructions with source code with the 'asm' directive [2]. The C annotated code is then compiled with a target compiler, and the resultant code is analyzed to detect the introduced marks. The output of this process is a database that characterizes each basic block, storing a list with its instructions on the target platform (the one in which the code will be finally executed).

During the second step (native simulation, right-side of Figure 3.2), the application code is annotated with a function per basic block that provides performance estimation. These functions use two additional parameters that are generated during the execution. These parameters compute the number of cache misses in a basic block:

- ICmiss : number of instruction cache misses
- DCmiss: number of data cache misses

The description of the techniques that generate cache information is out of the scope

[2]To prevent the compiler optimizations from removing these marks they must be declared "volatile"

of this dissertation and the interested reader is referred to [CPVM10]. With this instrumentation and these parameters, the execution time of a block can be estimated as:

$$T_B = C_B + \overline{T_{Imisses}} \cdot \text{ICmiss} + \overline{T_{Dmisses}} \cdot \text{DCmiss}$$

The cost associated with each basic block regarding its instructions (C_B) differentiates the methods "Native-Specific" and "Native-Average". The former uses a cost that is dependent on the mnemonic of the ASM instructions of the basic blocks ($C_B = \sum C_i$). The latter uses an average cost that is the same for all the instructions ($C_B = N \times C$), and therefore the cost of each basic block is proportional to the number of instructions that it contains.

It is important to notice that the execution time of each instruction is also affected by other inter-instruction effects that can occur in real programs. Examples of these alterations are prefetch buffer and write buffer stalls, other pipeline stalls and cache misses. Base cost per instruction does not take into consideration the impact of these effects. Also, extra cost depends on the input data sets of the system and this information is not available at compilation time. Therefore, assigning an average cost in this step implies a loss of accuracy.

Operator-based modeling ("C-Level")

This method performs the estimation based on a high-level language (C in our example). The input is a software code and a rough characterization of the platform concerning hardware and software. Here we use a characterization based on elements of the C/C++ code, assigning a constant cost per element of the code. Time estimation is performed assigning a cost to each C++ operator, and overloading the application software operators to keep track of the total time. We consider N the number of C statements corresponding to some functionality and T the time needed to execute that functionality on the target processor. We can express T as:

$$T = \sum_{n=1}^{N} T_n$$

Where T_n is the time spent executing the functionality of the nth statement of the source code.

We can decompose the functionality of each statement of the high-level C language in a set of machine-level instructions and rewrite T_n as the sum of all them.

$$T_n = \sum_{m=0}^{M_n} T_{mn} \qquad \Rightarrow \sum_{n=1}^{N} \sum_{m=0}^{M_n} T_{mn}$$

Where T_{mn} is the time that is spent executing machine-level instruction “m” when it is associated with C-statement “n”. This time, T_{mn}, depends on of the base costs and extra costs as explained before. To model the basic block at a high level of abstraction we consider only the base cost, that we call here “mean time per instruction” $\overline{T}$. We can rewrite the execution time as:

$$T = \sum_{n=1}^{N} I_n \cdot \overline{T}$$

In which we have obtained the approximate execution time of a task composed of N elements of the C-code, each of them being decomposed by the compiler in assembly instructions. The number of instructions that implement these high-level elements of the code is obtained by characterizing small programs that contain these elements in isolation. Once the elements of the code have been characterized, they can be directly reused for all designs developed on the same HW platform.

3.4.4 Results

To compare the estimation methodologies, we have executed several test benches on an ARM920T-based platform. The average execution time per instruction has been obtained from reference [SOF00]. Average time per cache-miss has been measured on a real hardware platform. We use four test benches: a bubble-sort, a recursive implementation of the Hanoi game, a recursive factorial and a solution to the queens problem. Table 3.1,

TABLE 3.1: Speed-Up and Error in the methods considered

	ISS Reference		"Native-Specific"		"Native-Average"		"C-Level"	
	Simulation Time (ms)	Estimated Time (ns)	Simulation Time (ms)	Estimated Time (ns)	Simulation Time (ms)	Estimated Time (ns)	Simulation Time (ms)	Estimated Time (ns)
Bubble	938	4163	6	4152	7	4122	5	3945
Factorial	206	1961	6	1954	6	1993	3	1558
Queens	489	16909	3	16787	3	16505	1	16479
Hanoi	209	3517	6	3491	6	3305	3	3535
	ISS Reference		"Native-Specific"		"Native-Average"		"C-Level"	
	Speed-up	Error	Speed-up	Error	Speed-up	Error	Speed-up	Error
Bubble	1	0	156	0.26	134	0.98	187	5.24
Factorial	1	0	34	0.36	34	1.63	68	20.55
Queens	1	0	163	0.72	163	2.39	489	2.54
Hanoi	1	0	34	0.74	34	6.03	69	0.51

and Graphs 3.4 present the results. Table 3.1 shows an estimation of the execution times in the target platform (estimated time). The table also shows the simulation time on the host platform (2.61 GHz Dual-core PC). We consider "ISS-Reference" (a cycle-accurate ISS Simulator) as the baseline to compare the performance and accuracy. "Native-Specific", "Native-Average" and "C-Level" are the native simulation techniques that we present in this section. The first conclusion that can be extracted from the data is that native simulation is 2-3 orders of magnitude faster than ISS approaches, even when the error is less than 1%. This information is included in Table 3.1 in which speed-up and accuracy errors are presented. Comparing the results of different approaches, we observe that "Native-Specific" method provides a speed-up of 2-3 orders of magnitude with an accuracy error less than 1%. "C-Level" provides a speed-up of two times compared to "Native-Specific" but with an accuracy penalty of at most 20%.

"Native-Specific" and "Native-Average" have similar speed-ups but the accuracy error of "Native-Average" (2.39%) is higher than "Native-Specific" (0.74%). The speed-up is the same for these methods as the characterization of the basic blocks is performed in the same way in both cases. The reason to use an average value per instruction (method "Native-Average") instead of a full characterization of all the opcodes ("Native-Specific")

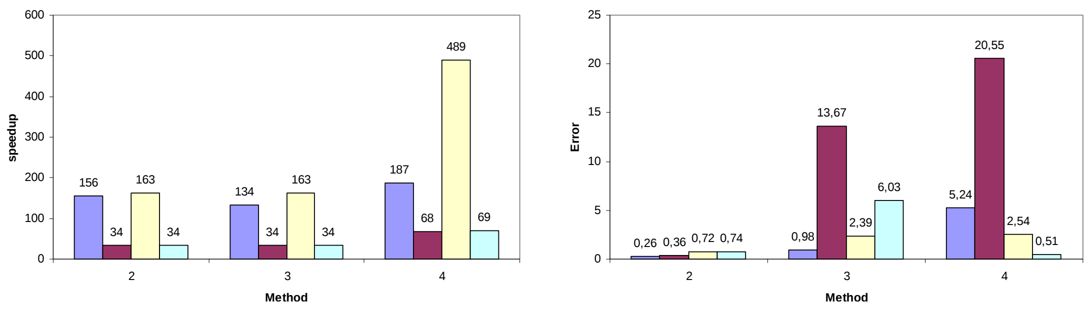


FIGURE 3.4: Speed-up and error of the different methods considered

–even considering that it produces worse results with the same speed-up– is that the average time per instruction is usually given in the processor datasheet, while an entire ISA characterization is difficult to obtain. Operator cost ("C-Level") gives significant speed-ups, but with more error, especially in the case of the factorial test case where the recursive implementations of the factorial function cause errors due to instructions that handle the stack in the function prolog and epilog. This technique can be used for the first steps of platform design, where precise values are not required but fast simulation is a must. We base the operator methodology on the idea that the compiler converts each element of the C-code into a template which is then filled with specific details. However, real compilers (especially in the presence of optimizations), perform some operations at assembly level after this template-filling to increase performance. This explains the errors of this method.

3.4.5 Optimization example

Introduction

In this section, we present an example in which we apply the instrumentation previously described to a real-life program to detect optimization opportunities and reduce the execution time. The example demonstrates that having a precise understanding of the interactions between the source code, the microprocessor, and the memory hierarchy, we can halve the execution time of the application under test. The solution presented in this section won the 3rd place in the 11th Memocode Software design contest [MSSS14].

The contest consisted on optimizing the execution time of a software emulator for the 8080 architecture when running over a RaspberryPi platform. The RaspberryPi platform is a modern Single Board Computer (SBC) designed with embedded system constraints, but also generic enough to run multi-purpose tasks. It has attracted much attention because of its adaptability, size, and price.

At the time of writing this work, there existed no simulator based in RTL, ISS or TLM that was able to consider the interactions between CPU, memory and source code in the RaspberryPi platform. In this section, we present a practical example of how our profiling techniques can be used to obtain valuable information of the program running on the RaspberryPi platform, and how this information can be used to derive useful optimizations. The contributions of this work can be summarized as follows:

- It shows a practical example of how we optimize a program to run on an embedded platform beyond the optimizations that a compiler can perform.
- It considers both the caches (instructions and data) and assembly instructions in the profiling.
- It identifies the main causes of performance loss in programs executed in embedded systems, and how to improve program performance through manual optimizations.

The section is organized as follows: the original architecture of the application to optimize is presented in section "Initial code architecture". The high-level parameters of our models targeting the RaspberryPi architecture are described in section "Hardware Platform". The optimizations that we apply to the code are presented in section "Optimizations". The relative effect of each optimization is presented in "Results".

Initial code architecture

The emulator starts at function main. This function is responsible for the setup of the SDL interface that will show the screen and the initialization of the emulator. For each instruction in the ROM file, function "Emulate8080Op" is called (Figure 3.5). This function simulates an instruction and returns the number of cycles the instruction needs to execute.

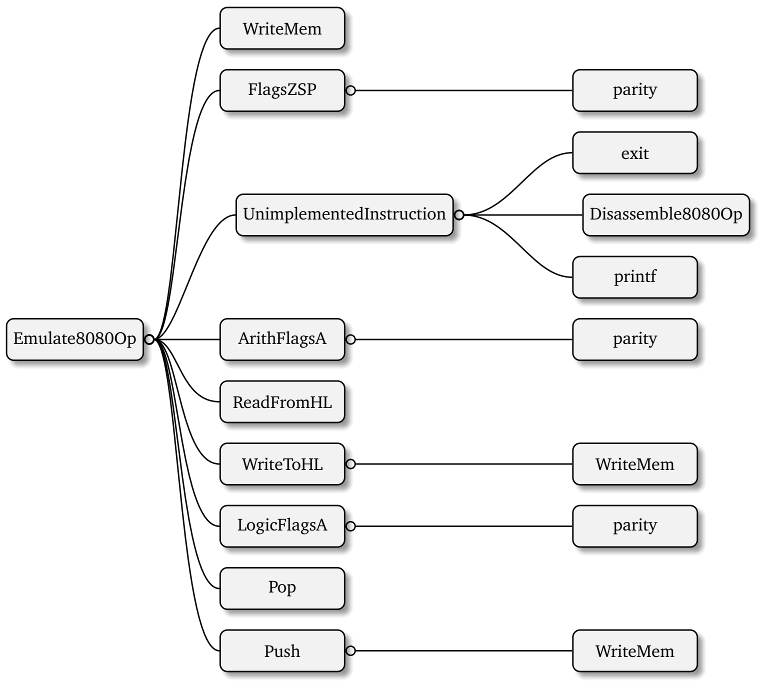


FIGURE 3.5: Function call graph of Emulate8080Op

This function is no more than a big switch statement over the different opcodes in the 8080 Instruction Set Architecture. Depending on the current opcode, the state of the emulator is updated, and in certain cases, the memory is written.

Hardware platform

The RaspberryPi platform is powered by an ARM1176 processor [arm]. This processor implements the ARM v6 architecture and provides a performance of 700MHz in low-cost designs. The architecture is schematized in Figure 3.6. There are two levels of caching; L1 and L2. According to ARM1176JZF-S Datasheet [arm], caches can be configured in the range of 4 to 64 KB and are 4-way associative. For the particular case of Raspberry Pi, L1 is split into 32KB of data and 32KB of instructions. Caches can provide two words per cycle for requesting sources, and line-length is eight words. This data is shown in Table 3.2, and is the main input of our tool for performance analysis.

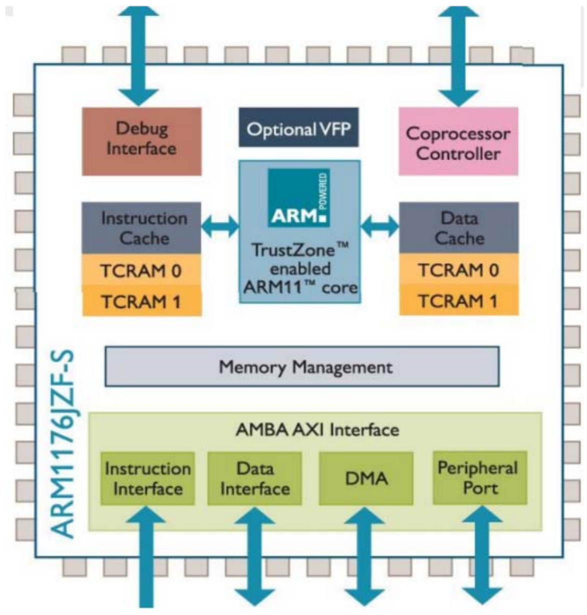


FIGURE 3.6: RaspberryPi System-On-Chip architecture, obtained from ARM1176JZF-S technical reference manual [arm]

TABLE 3.2: Relevant parameters of destination platform

Parameter	Value
Frequency	700 MHz
L1 sizes	32KB Instructions + 32KB Data
L2 size	128 KB
Cache associativity	4
Cache line size	8 words

Optimizations

We have classified optimizations performed in the code in two sections: optimizations targeted to reduce the number of executed instructions and optimizations to improve cache utilization.

Next, we explain the optimizations performed in the code and how they affect performance.

Memory optimizations Dead code elimination has a substantial effect on performance. It reduces code size, and therefore instructions cache hit rate increases. This makes accessing high levels of memory in the hierarchy much less frequent and therefore, reduces the time needed to feed the processor with instructions. In the emulator code, dead code can be reduced by removing unused functions and switch cases. Some of these eliminations cannot be automatically performed by the compiler, as they depend on user-specific knowledge.

The same effect can be observed in data caches. Removing unused variables or unused fields in structures increases data cache efficiency.

Clustering together all the transformations applied to a variable decreases the chance of this variable being replaced in low levels of cache, just to be loaded again later. In the code under test, this is clearly the case for variable "cycles", which is used before and after calling Emulate 8080Op. As this is a large function, "cycles" has certainly been replaced in L1 memory when it is accessed the second time. Clustering together the update of cycles with prior computations over this variable has a substantial effect on performance.

Inlining In some occasions, inlining can be beneficial for performance because it removes the overhead of function calling. However, if the function is called from multiple places, it can increase code size and therefore perform worse on instruction cache. Although compilers do a good job on detecting best candidates for function inlining, the lack of runtime information prevents the compiler to inline specific functions. We used the instrumentation technique presented in Section 3.4 to simulate this information and decide the best candidates for inlining.

Pointer arithmetic and structure flattening Accessing arrays and structure fields are sources of covert operations, as they both involve calculation of target address before pointer dereference. The instrumentation of the code with the technique presented in this section reveals many of such dereferences. Flattening the fields of this structure has a noticeable effect on performance.

Register keyword To match the architecture better, variables that are frequently used can be preceded by the C keyword `register`, which makes them use registers in the target architecture and not move them to memory whenever possible. We used the

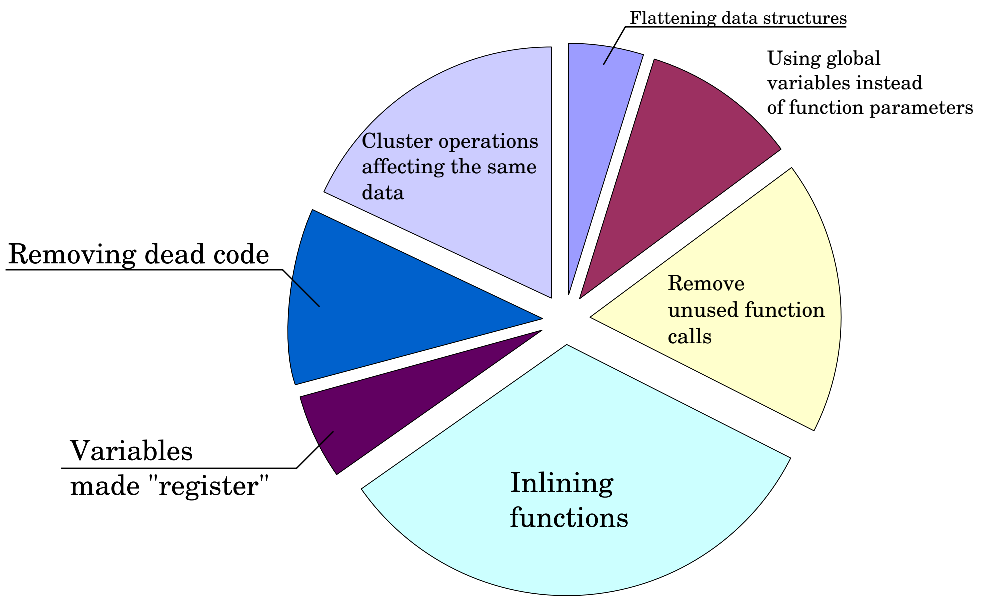


FIGURE 3.7: Relative effect of the techniques discussed in 3.4.5.

instrumentation presented in this section to detect the variables that would benefit the most from this optimization.

Results

Figure 3.7 captures the relative effect of different optimizations conducted in the code. Seeing Figure 3.7, we can sort optimization in order of importance as “inlining”, “clustering of operations over same data”, “removed unused calls and functions”, “make parameters of functions as global variables”, “removing dead code”, “making most frequently used variables as registers”, “flattening structures” and “removing dead code”. The fact that second, third and fifth most important optimization do not affect the number of executed instructions shows the importance of data and instruction caches in overall execution time, and therefore the importance of profilers that considers them to obtain valuable information for optimizations.

3.4.6 Conclusions

This section presents three techniques for software execution time estimation that work at C-Source Code level and assembly level. These techniques are based on native simulation; a set of timing annotations are introduced in the code to provide performance estimations during simulation. We compare these techniques with a cycle-accurate ISS regarding execution time and estimation error.

We show that software execution time can be characterized by associating a constant time cost to each element of the high-level description code. These costs are derived from the number of machine instructions that are needed to execute the corresponding statement. Once the costs have been obtained, they can be reused for all designs developed for the same hardware platform. This technique is useful in early design steps where fast simulation is more important than accurate results.

To achieve more accuracy, we present a methodology that extracts basic blocks from C-Source code, and we describe two ways to characterize these basic blocks. The former needs a complete characterization of the platform regarding a base cost of each opcode in the ISA. The latter considers an average value for every instruction. The results show that considering just the number of assembly instructions and number of cache misses in a block provides enough accuracy and speed-up. Comparing these results with the ISS approach we see that very accurate results (error less than 1%) can be obtained, with a speed-up of 2-3 orders of magnitude.

We have also demonstrated the usage of the techniques to instrument and optimize the source code of an emulator when executed over an embedded platform (RaspberryPi). This application proves that the selected parameters of the architecture are relevant for obtaining insightful results for performance optimization and at the same time easy to be adapted to new hardware architectures. As we have seen, caches may be a bottleneck for applications running on embedded platforms, so profiling and optimizations have to consider them in order to obtain insightful results for optimizing both the source code and the hardware platform.

3.5 Simulation of the many-core part of the design

3.5.1 Introduction

In this section, we extend our simulation framework to be able to simulate shared-memory many-core architectures. This extension is particularly challenging because in many-core architectures it is important to simulate not only each microprocessor in isolation but also the concurrent accesses to the shared memory. This requires consideration in the software analysis (a parallelization API has to be analyzed to detect the creation and destruction of threads) and in the hardware modeling (we require new models to support commonly used communication infrastructures of many-core platforms). In our solution we extend the annotation introduced in Chapter 2 to support OpenMP (a common API for parallelization of C/C++/Fortran code) and develop hardware models for the commonly used communication infrastructures of many-core platforms (Buses and Networks On Chip (NoCs)). As in previous sections, we will put particular emphasis on being able to simulate the platform in the early stages of the design process.

The section is organized as follows: in “Estimation technique” we introduce the extensions performed to our virtual platform to simulate many-core architectures. In “Optimization example” we present several optimization techniques motivated by the parameters of our simulation framework and demonstrate their effectiveness by presenting how we used them to improve the execution time of a biological simulation by a factor of 320 times, and win a first prize in the “Intel Modern Code Developer Challenge” competition. We finish the section by presenting our conclusions.

3.5.2 Our approach

Simulation challenges

There are two main architectures to communicate processing elements in a multi-core System-on-Chip (MPSoC): Bus-Based communication and Network-On-Chip communication). Architectures with “cluster-level granularity” [PP11][YZA+09] use the two: a Network-On-Chip for inter-cluster communications because of its better scalability and

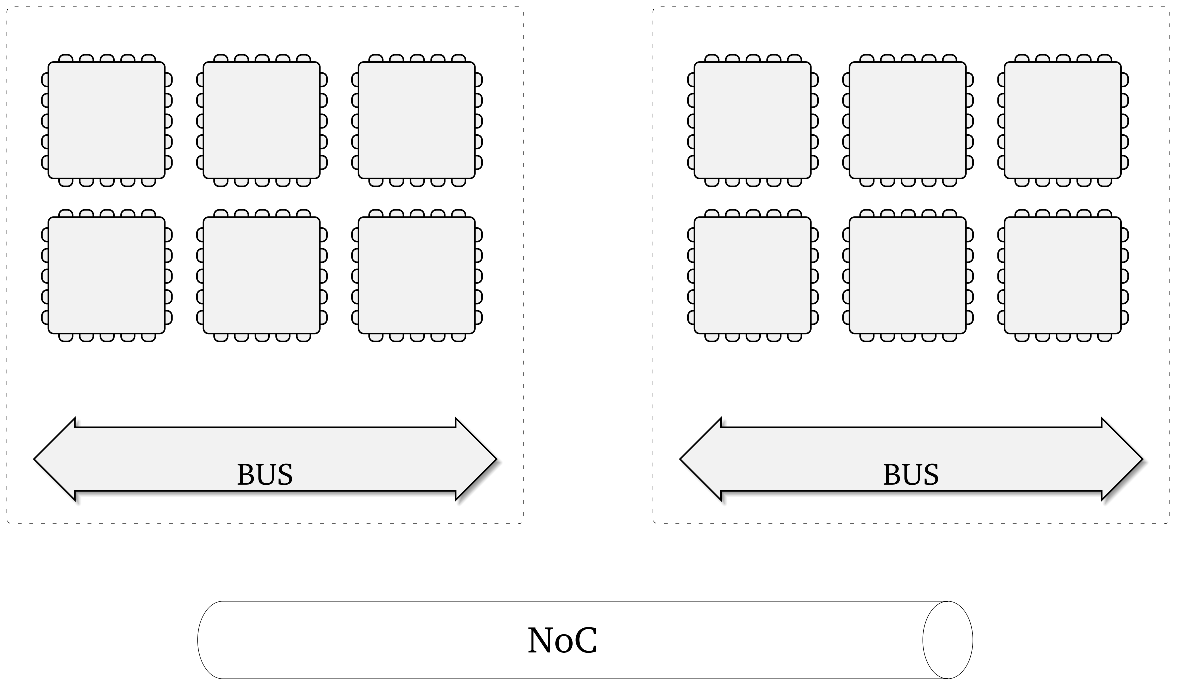


FIGURE 3.8: Double Hierarchy of communications in a modern many-core architecture

power consumption and a bus for intra-cluster communication because of its smaller size and better performance with less communicating elements (see Figure 3.8).

This "double" hierarchy of communications introduces new challenges in the design of many-core systems because performance largely depends on communication patterns between cores and memory hierarchy. Architectures with strong coherence between each local memory and the global memory (ccNuma architectures) are no longer efficient when many cores are communicating. Therefore, caches are usually substituted by scratchpad memories, and the responsibility for keeping them in sync is delegated to the programmer instead of being ensured by the hardware. The modifications that the programmer needs to perform in the code to optimize performance in these platforms require a profound knowledge of both the architecture (latency, bandwidth, the number of memory banks, etc.) and the application (relevant variables and memory access patterns) which are not usually manageable for a single person. In agreement with Moore's law, the density of integration grows 60% each year, while memory bandwidth only grows 10%. The

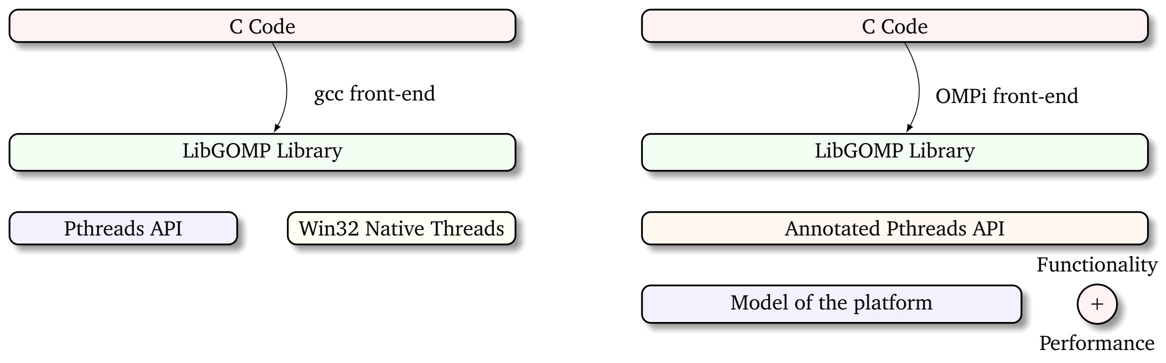


FIGURE 3.9: APIs involved in OpenMP parallelization and estimation

difference between those exponential curves is also exponential, so a new factor in system performance analysis is becoming prevalent. This effect is called “Memory Wall” [BDVS15], [CWLW15], [VB14]; a degradation of performance caused by memory limitations that is very dependent on the mapping between tasks and processing elements as well as the utilization of memory hierarchies. Besides the memory-wall problem, the increased number of cores that are present in current platforms exponentially increases the design-space exploration. Therefore, early performance estimation techniques are necessary to adapt the design process to these platforms. The main trend in the field is the use of instruction set simulators (ISS) [GUA11][GGD+02], but the problem of this approach is the high simulation time that severely increases the design time.

Parallelization language and concurrency modeling

To create parallel tasks and spread them over the different cores available in the platform, we rely on the OpenMP parallelization API. As summarized in Figure 3.9, the parallelization of OpenMP annotated code lies over three different APIs. GCC’s front-end converts the OpenMP ”pragma” directives into a set of calls to functions defined by the “LibGOMP” library. This library provides a thin layer of abstraction that enables OpenMP to be portable over different native thread libraries (i.e. Windows or POSIX APIs).

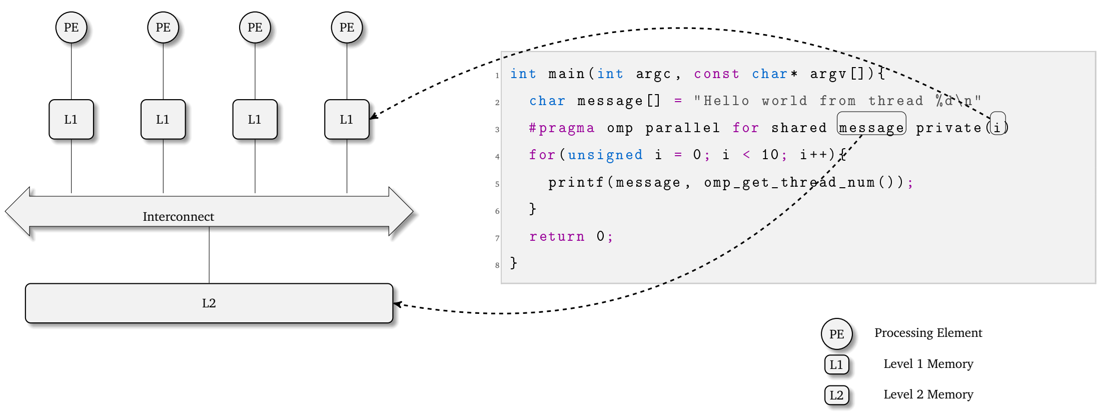
To model the effect of thread parallelism with the run-time library, we modified the LibGOMP API to instrument the creation and destruction of threads and the mutexes operation. Thus, when the program calls a “LibGOMP” library function, there is another call to the same function of our run-time library. These functions keep track of the execution time needed for each task enabling to obtain execution time estimations. Note that even when the functions mapped to each thread model the execution time of tasks when executed on the target platform, the code can be still run on the host computer, and the native simulation can use the host operating system scheduler for synchronization.

The code can then be parallelized using OpenMP directives. Using the OMPi compiler [DLT03] we can convert OpenMP directives to POSIX calls and map the tasks generated in this process to particular processors in the platform.

As mentioned previously, one of the most important aspects to consider when creating applications for many-core platforms is the efficient use of the hierarchical memory levels. Since this is an important aspect to consider, some virtual platforms provide models of memories to be embedded in the platform. However, there is not yet a standardized API to manage DMA transfers, or to assign specific data localization for variables so our technique relies on "shared" and "private" pragmas of OpenMP to assign a location in the hierarchical memory for every variable so this access time can be estimated. This mapping provides a generic and practical way to simulate the code in different platforms and facilitates the native-simulation approach, as these pragmas do not interfere with the normal functionality of the programs. This idea of using "shared" and "private" functions for variable localization in memory and estimation is shown in Figure 3.10. To accomplish this, a parser has been created to extract information about the OpenMP pragma directives. Also, the underlying memory model has been modified to consider different costs for different memory transactions. The technique has been evaluated and tested with models for the P2012 many-core platform [PP11] and the results are further discussed in the publications [GGBSEC13][GGBS12][GGS12].

Hardware models

To model a generic many-core platform, our simulator requires particular details of its internal architecture. In the current version of the tool, it is possible to model the following elements of a many-core architecture:



```
int main(int argc, const char* argv[]){
  char message[] = "Hello world from thread %d\n"
  #pragma omp parallel for shared message private(i)
  for(unsigned i = 0; i < 10; i++){
    printf(message, omp_get_thread_num());
  }
  return 0;
}
```

FIGURE 3.10: Localization of variables based on OpenMP pragmas

- Processor: We characterize the particular model of a processor by defining the number of cycles that each instruction of the ISA needs to execute. The estimation described in 3.4 (method "Native-Average") is used to simulate each of the individual cores in the system.

- Bus interconnection: Cores are associated in "clusters", that share a bus to communicate them with the local L2 memory and with the network interface. This bus is modeled using a TLM generic bus model, with a transfer rate that can be easily modified in the configuration files.

- Memories: Currently, the memory model hierarchy parameterizes memory response delay and memory size. The default access latencies are 1 cycle for in-processor memory, 10 cycles for L1 memory and 100 cycles for L2 memory, but these numbers can be easily modified to fit other platforms. A translation buffer (TLB) model provides physical addresses for memory positions in each core and also logs the requests to a trace file so that memory access patterns can be analyzed. The techniques to model instructions and data caches are described in [PDV11].

- Network-On-Chip (NoC): We provide models of the NoC with two simulation approaches; a high-level TLM model where the transactions are modeled as blocks of data that flow from one node to another through a dedicated collision-free path and a low-level cycle-accurate simulation in which all the micro-architectural details of the router crossbar and switches are considered. A unique network interface is provided for the two approaches so a designer can use the former (if fast simulations are the main goal) or the latter (if simulation accuracy is the main goal).

- Communication patterns: each transfer to memory is annotated in a trace in the form

$$time, thread\_id, Read/Write, memory\_start, size$$

This trace is post-processed when the simulation finishes so communication patterns can be obtained from it. To accomplish this, the trace is read in a time-wise order, and a virtual map of the memory is created. This map presents a picture of the memory content at a time t. When a thread thread$_w$ writes in a memory position a data of size size$_w$, memory positions from memory_start$_w$ and memory_start$_w$+size$_w$ are assigned to this thread. If another thread reads any of these positions in a time $t' > t$, a communication of n bytes is inferred from thread thread$_w$ to thread thread$_r$ of size $[\text{memory\_start}_w, \text{memory\_end}_w] \cap [\text{memory\_start}_r, \text{memory\_end}_r]$. This amount of data is then added to the communication matrix for cell (thread_id$_r$, thread_id$_w$), and can be used later to group together threads that have intense intercommunication.

3.5.3 Optimization example

Introduction

Current research in the field of computational biology often involves simulations using high-performance computer clusters. It is crucial for the code of such simulations to be efficient and correctly reflect the model specifications. In this section, we present an optimization strategy for simulations of biological dynamics demonstrated by our winning entry of the "Intel Modern Code Developer Challenge" competition.

The competition highlights some of the important aspects to tackle when optimizing code for many-core architectures that we have covered in the introduction. Although various programming interfaces such as OpenMP ease the transition from sequential to multi-threaded parallel code, fully-automated parallelization is difficult to archive, and there is still a huge variation in performance due to different programming styles and functionally equivalent versions of the same code. Just understanding this variation is challenging because (both in the competition and also in real-life scenarios), we usually only have access to the net effect of the changes in the execution time. We will see how the understanding of the intermediate variables that indirectly affect execution time (cache misses, memory transferences or load balancing) can effectively guide the optimizations and how this information (combined with a fast simulator of the architecture) can lead to significant optimizations in execution time. Quick and unintrusive profiling is particularly important for applications such as the one presented as example in which the sequential implementation is performed by a different person (or team) than the optimization and the computational platform. Understanding how the three pieces interact together is of paramount importance to obtain the maximum benefit of their combination.

This section presents the subset of the optimizations that are related to the parameters selected in Section 3.5.2. The full description of the original software architecture and explanation of the performed optimizations is being reviewed for publication.

Initial software architecture

The original code to be optimized allows the simulation of millions of neural progenitor cells that interact with each other biochemically in 3D space. In particular, it involves some fundamental processes during the formation of the brain; cell proliferation, secretion and detection of diffusible substances as well as their concentration gradients, and cell migration. Understanding how these mechanisms of brain tissue development play out, by taking into account genetic factors in a spatially and temporally dependent way, is crucial for the identification of the causes and potential treatments for neurodevelopmental diseases, such as epilepsy, autism, and schizophrenia [Ins10][Hag13][HCW14]. Overall, the performance and correctness of such optimized code are paramount for the explanatory power and scientific practicality of computer simulations of biological dynamics.

The original architecture of the code is shown in Figure 3.11. The code can be divided into two phases. Initially, a single precursor cell is placed in the middle of a 3D space.

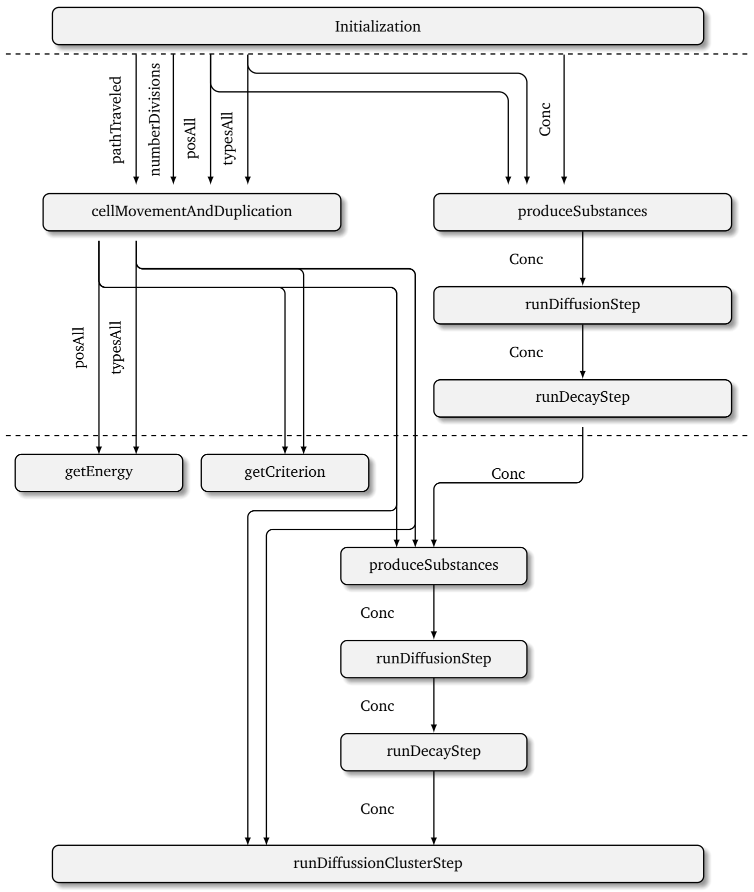


FIGURE 3.11: Data flow of the initial implementation.

In the first phase of the simulation, the cells move randomly and divide until the final number of cells is reached. After each cell division, the daughter cells are assigned to one of two possible cell types, hence giving rise to cell differentiation. This simulation is performed with the functions *produceSubstances* and *cellMovementAndDuplication*. In the function *produceSubstances*, one of the two substances is produced depending on the cell type. In the functions *runDiffussionStep* and *runDecayStep*, the diffusion and decay of the cellularly secreted substances are simulated.

In the second phase, the program simulates the self-organized formation of cell clusters, based on the movement of cells along gradients of extracellular substances, which is a well-known ability of eukaryotic cells [Jin13]. This phase simulates the movement and the secretion of substances by cells. Cells are attracted by and move along gradients of the substance secreted by cells of the same type, but move away from gradients of the opposite substance type, which finally leads to cluster formation.

At the end of the simulation, measurements for the overall energy and a criterion indicating if the clustering has taken place or not are computed. This is done using the functions *getEnergy* and *getCriterion*.

Hardware architecture

The performance of the application was measured on an Intel Xeon Phi 7120P coprocessor. This device, based on the Many Integrated Core (MIC) architecture, comprises 61 cores clocked at 1.238 GHz. Cores have symmetric access to a total of 16 GB of shared memory. This memory is cached with 512 KiB of Level 2 cache and 32 KiB of L1 data cache per core. More details about the Intel Xeon Phi platform can be obtained in [VK].

The code was compiled with Intel C++ compiler version 16.0.0.109. The software stack for Intel Xeon Phi coprocessors was MPSS 3.5.2. The operating system on the host system, CentOS 6.2, used the Linux kernel version 2.6.32-220. The parallelization API was OpenMP.

Code Optimization

In the following sections, we explain the optimizations conducted to increase code performance over the described platform.

Optimizations are grouped based on the main goal to achieve; in “sequential optimization” we focus on single cores and try to reduce the associated computational load. In “parallel optimizations”, the main goal is to increase the parallelism of the code (i.e. the number of operations that cores can execute at the same time). Finally, in “memory optimizations” cache locality is exploited to provide high bandwidth and low latency access to the data.

Sequential optimization

In this section, we describe general optimization techniques that do not rely on parallel architectures or memory layouts.

Loop transformations The benefit of loop transformations is twofold; on the one hand, due to limitations in inter-procedural and dependency analysis, compilers do not usually do a good job at detecting parallelization opportunities when multiple loops are involved. On the other hand, loop transformations increase temporal and spatial memory locality. Some examples of loop transformations are:

- Loop coalescing: The transformation consists in identifying loops that operate over the same input domain but produces results in different output variables. That enables joining them, and expose more code to parallel computation.
- Loop splitting: On the other hand, if we can identify loops that operate over different variables that do not share any dependency, it is useful to split them and include `pragma` directives to enable task-level parallelism.

Function inlining To implement function calling, compilers introduce prolog and epilog instructions at the beginning and end of each function. These instructions can be detrimental to performance in functions that are called very frequently and whose body is relatively small, and it can be more beneficial to inline those functions. On the other hand, this makes the code larger, and more cache misses will occur in the instruction cache. Therefore, measurements of the effect of those optimizations both in the number of instructions and in the instructions cache need to be measured to select which functions to inline.

Optimizations targeted to reduce the number of executed operations This set of optimizations aims to decrease the number of instructions executed. Common examples are factorizing common sub-terms that the program uses several times. The compiler is usually unable to detect these optimization opportunities due to inter-procedural analysis. A good example of such limitations can be found in the pair of functions *getCriterion* and *getEnergy*, whose computations are very similar, but since the commonalities are among two different functions, the code cannot be reused. Extracting the common code out of the two functions and reusing it reduces the number of executed instructions and increase the performance of the instructions cache.

Optimizations to remove expensive operations Due to complex datapaths and the internals of modern architectures, different operations can take a different number of cycles to finish, so a good optimization technique is to transform expensive operations by mathematically equivalent ones that are much cheaper to compute, especially if those operations take place in loops so that they are executed multiple times. Common examples of this transformation are focused on multiplications and divisions since those are usually the most expensive ones in modern pipelines. In our simulator, we output metrics about the number of instructions executed of each type that are useful for detecting these optimization opportunities.

Parallel optimizations

In this section we describe optimizations focused on exploiting the high degree of parallelism available in the platform.

High-level task parallelism Tasks dependencies of the computation described in the introduction can be seen in Figure 3.11. As shown, several computations can already be performed in parallel in the initial implementation of the source. High-level parallelism has been implemented for the functions that do not have dependencies among them. However, the optimizations archived by a simple parallelization like this are sub-optimal. To increase the parallelism in the whole program, we need some architectural transformations.

Double buffering has been used in the array *Conc*. Double buffering is a common optimization technique when implementing stencils, and more details can be seen in references [MRR12][ZZWY16]. Using double buffering reduces the number of instructions executed by the processing elements but impacts negatively in the memory performance, so it does not always lead to improvements in the execution time. Simulation techniques that can analyze trade-offs between the number of executed instructions and memory utilization help in detecting optimization opportunities like these.

Vectorization Besides ‘task-level parallelism’, using vector instructions inside functions can increase the performance of the code substantially. In our optimizations, we use Intel intrinsics to introduce vector instructions in the code.

Parallel reduction Several loops in the original code are used sequentially in conjunction with a global scalar available to all iterations of the loop to compute sums of different terms or to count the number of times an event happens. The way in which the loops are implemented introduces data dependencies caused by reads and writes to the global object, and therefore a parallelization is impossible. To remove these dependencies, we introduce an array or a matrix that stores the result of each iteration individually. This enables executing most of the work in parallel since each thread writes to its private variable and therefore there are no collisions. A final sequential step is then applied to compute the serial part, in which no parallelization is possible. This technique called "parallel reduction" has been used in the functions *cellMovementAndDuplication* to count the number of divisions of each cell, and in *getEnergy* and *getCriterion*.

Memory optimizations

As recalled in the introduction, the memory layout usually constrains algorithms more than the computational speed. Even when we have optimized the volume and cost of operations in parallel and the work given to each core, memory bandwidth can also limit the performance of code in many-core platforms.

Memory allocation with first-touch For allocating data in many-core architectures, Intel compiler provides libraries that implement the function `mm_malloc`. This function implements a lazy initialization policy that defers the allocation of the memory until a core tries to use it, giving the system a "hint" to which memory the array should be allocated on and producing the beneficial effect of locating the memory close to the core that uses the data most frequently.

Optimizations to improve temporal and spatial data locality To maximize the utilization of cores and avoid hitting the "memory wall", it is important to maximize cache utilization. To this end, two techniques have been used in the implementation of the

TABLE 3.3: Simulation parameters for ‘small’ and ‘huge’ testcases.

Small testcase			Huge testcase		
speed	=	0.01	speed	=	0.01
T	=	500	T	=	500
L	=	80	L	=	820
D	=	0.3	D	=	0.3
mu	=	0.1	mu	=	0.1
divThreshold	=	16	divThreshold	=	25
pathThreshold	=	2.0	pathThreshold	=	2.0
spatialScale	=	5.0	spatialScale	=	5.0

most demanding functions: loop tiling and cache-oblivious algorithms. In the former technique, the input data is partitioned in “tiles”, and we transform loops that operate over that data in such a way that they scan the input first with a stride that is equal to the tile size, and then a second loop operates “inside” the tile. This transformation increases locality and keeps more accesses in the low levels of the memory hierarchy. However, the improvement of this technique is highly dependent on the matching between the cache sizes, the size of the “tiles”, and the sizes of the array, so several simulations are needed to select the optimal parameters.

Results

In Figure 3.12, the optimizations performed in the code are classified regarding the “optimization target”, as explained before. We present the relative effect of each optimization in this graph.

In this figure, two examples are shown. We call the former “small test case” and the latter “huge test case”. These names summarize different simulation parameters as presented in Table 3.3. The purpose of the small set of parameters is to be able to quickly test the effect of optimizations in a manageable test case, while the purpose of the “huge” one is to perform the actual high-performance simulation.

Seeing Figure 3.12, we can sort the most important optimizations as “Double-Buffering”, “Parallelization”, and “Tiled and cache-oblivious” implementation. In the case of the huge test case, some of these transformations could not be measured, since the execution at

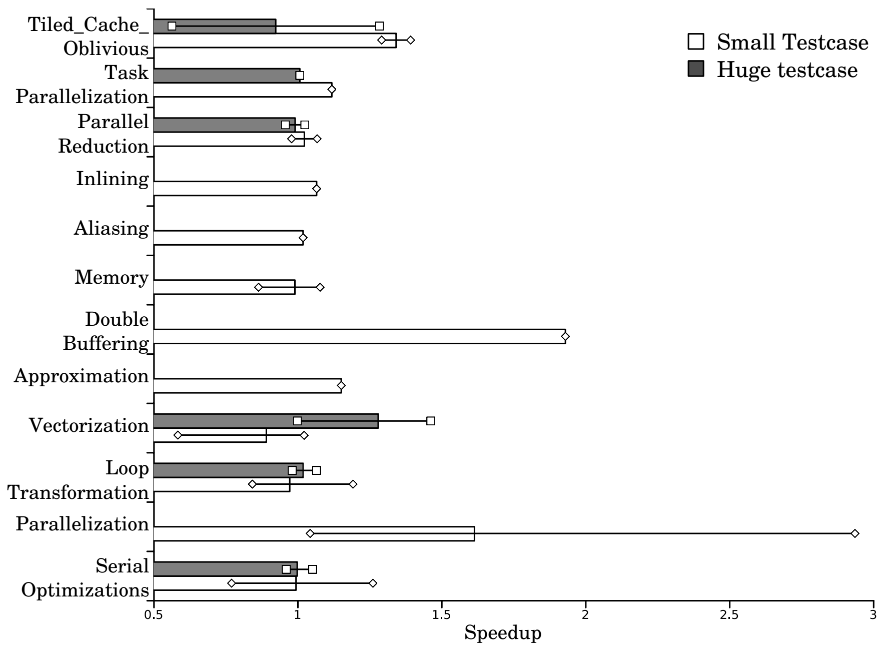


FIGURE 3.12: Effect of different parallelization techniques in the small and huge testcases. Note that first versions of the 'huge' testcase cannot be simulated due to excessive execution time and lack of resources. For each technique, the average improvement is shown, measured as the average of the pairwise division of execution times before and after each step. The error bars also show the maximum and minimum value of that division over the multiple times in which the technique is used. Small and Huge parameters are presented in Table 3.3.

the initial stages was too time-consuming. Because of that, only the first implementation of the sequential code have been measured with the "huge" test case – the complete simulation took 45 hours in the Xeon E5. The optimized version only requires 8 minutes under the platform described in 3.5.3. It is interesting to see that on average, the effect of some techniques is less than 1 (i.e. increments the execution time) in the small test case, but beneficial it in the huge one. This emphasizes the importance of understanding the relationships between the input data, the program, and the hardware architecture.

In the relative order of the optimizations, we can see the importance of modeling the congestion of the communication infrastructure of many-core chips when estimating the performance of embedded code in high-performance computers. Compilers can not neglect the importance of optimizing memory access patterns either. Although compilers do nowadays a great job in implementing automatically such optimizations, we have seen that in the majority of cases, because of the lack of context, they are not able to automatically infer the conditions that make these optimizations possible, and human assistance is usually needed. Finally, we emphasize that some optimizations require "losing" performance temporarily to be able to improve later on. Having an accurate intuition on how hardware and software interact will prevent us from "giving up" too soon in the optimization process and improve quickly later on. This is the case for example in loop transformations that are required to expose more code to parallel execution. The usage of a simulation framework that gives the developer information about the internal parameters of the platform can help in these manual tasks.

3.5.4 Conclusions

This section presents a virtual platform that enables a reliable design-space exploration of shared memory many-core platforms. We consider two major elements in the proposed tool to facilitate the design process. One is the use of OpenMP as a programming paradigm since it is oriented to a shared memory environment. The other is the use of parameterized hardware models for the communication elements of the architecture, that can be used to simulate the platform before a real hardware prototype is available (or when the platform is available, but the inspection of the internal parameters is not). In summary, this section proposes a tool based on native simulation for measuring OpenMP partitioned applications on embedded systems targeting many-core platforms and considering memory and Network-On-Chip contention in the simulation.

We demonstrate the advantages of our simulation framework with an example of simulation of biological dynamics, that is optimized to increase performance when running with many-core parallel processors and provide insights about what are the most important elements to analyze when simulating the effect of code changes in many-core platforms. Although this demonstration is in the context of biological systems, the principles apply to a wide range of scenarios. Our code simulates varying numbers of agents in 3D space, which interact and behave in many different ways; namely by proliferation, migration, secretion of diffusible substances, internal dynamics, and detection of external chemical gradients. Because of this huge diversity of interactions, the techniques presented can be extrapolated to many different contexts.

3.6 Simulation of hardware accelerators

3.6.1 Introduction

In this section, we introduce our approach to simulate the hardware elements of the platform under test. This section is structured as follows; in “Design process” we present a typical codesign process for heterogeneous platforms (platforms that contain hardware and software components) as well as the main limitations that are present in current state-of-the-art techniques for implementing such a design process. In “our approach”, we extend the native execution technique developed in previous sections to the analysis of hardware accelerators. In “results” we evaluate our technique with several test cases from the CHStone benchmarks (a set of programs to assess the effectiveness of synthesis tools). In “Optimization example” we use our technique and simulator to improve the execution time of a computer-vision application in which we reconstruct the volumetric Convex-Hull of a figure based on the images of several cameras. The usage of the presented technique helps in detecting optimization opportunities that lead to a result that is three times faster than other state-of-the-art techniques for the same quality conditions. Finally, in “Conclusions” we close the section with a summary of the presented improvements.

Design Process

Embedded systems are nowadays composed of the hardware part of the system and the software part. Generally speaking, software is easier to develop and update, so most modern systems tend to integrate a microprocessor that executes most of the functionality with several hardware modules (generally called accelerators) that are targeted to the most critical functionality (performance and energy-wise). Given the restrictions and the algorithm to implement, the process of deciding what part of the functionality is implemented in software and which one is synthesized as hardware is called “partitioning”, and the correct execution of this process is essential for the proper development of the system. Figure 3.13 shows a common codesign process that we divided into the following steps:

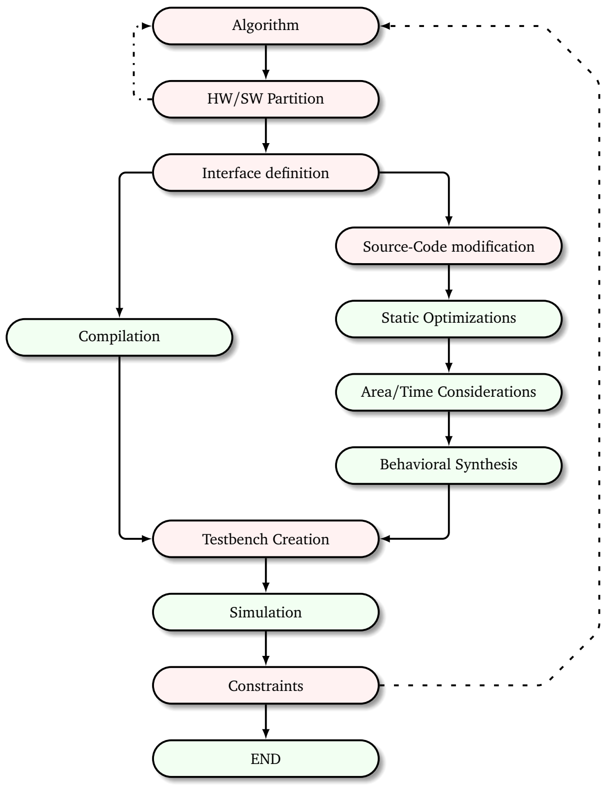


FIGURE 3.13: Typical codesign process. Green boxes represent steps that are generally well-automated. Red boxes represent steps that require manual intervention. Solid lines represent dependencies, dashed lines represent codesign loops with current methodologies (- - -) and our aim (- · - ·).

- Development of the algorithm: The first step in the creation of a heterogeneous embedded system is the development of a software model that implements the desired functionality. This implementation is sometimes called "Executable Specification", because its purpose is to specify how the system behaves even though some functionality may be later implemented as a hardware module. In those cases, the developers of the "executable specification" can be a different team separated from the implements of the final heterogeneous system.

- Partitioning: Once the algorithm has been tested and verified, developers need to specify which elements of the system are implemented in hardware and which are implemented in software. These decisions require deep knowledge of the algorithm, the functionality and the main parameters describing the platform.

- Compiling the software part of the system: Once the hardware and software elements have been selected, the software part of the system is compiled for the target processor. To communicate the software and the hardware parts of the system software developers need to implement several layers of "Hardware-Dependent-Software".

- Implementation of the hardware part: The part of the executable specification that will be implemented in hardware needs to be profoundly transformed (from C, C++ or SystemC to VHDL or Verilog). This transformation is a complex process that is driven by the need of obtaining a heavily parallel implementation to reduce the execution time but also constrained by the area it uses in the chip. The development of this hardware part of the system has usually been performed by hand. Given the key importance of this transformation and the difficulties it entails, new tools that can automate such transformation have appeared in the last years.

- Verification: The verification of the resulting system is divided into two parts; on the one hand, designers must verify that the transformations performed in the algorithm did not affect its functionality. On the other hand, designers must ensure that the system meets the non-functional requirements.

As emphasized for single and multi-core designs, redesigns in the case of unmet non-functional requirements are a serious concern in the development of embedded systems. Regarding heterogeneous systems, if the redesign implies reconsidering the partition between the hardware and the software parts of the system, the consequences are even worse because reimplementing the hardware part of the design requires a substantial effort of code transformation and rewriting. In fact, a failure in the definition of such a partition can imply more than 90% of the costs of the product [HR05]. Therefore, it is important to be able to obtain performance estimates in the early steps of the codesign process. To do it, there exist two main approaches:

Synthesis-then-simulation Given an algorithm described in C/C++, there exist tools, both academic [GDGN03] and under industrial usage [cat], that can convert the sequential implementation into functionally equivalent hardware. An approach to solve the problem then might be to obtain a hardware implementation with such tools and simulate the obtained description with an RTL simulator that enables to obtain estimations of latency and/or power consumption. Even when it looks counter-intuitive to implement a full RTL description of the system to estimate if such implementation is worth the effort to implement, the automated nature of this approach makes it very attractive, especially because of the accuracy of the RTL simulation. Using High-Level-Synthesis tools, the user can generate various implementations and chose the most appropriate one. The main disadvantages of this approach can be summarized as:

- The target platform must be precisely described and characterized before the start of the process. Behavioral synthesizers and co-simulation environments need a very detailed description of the target platform, which is not available in the first stages of the design process.

- Synthesis tools do not support the full grammar of high-level languages. This is the main limitation of the Synthesis + Simulation approach and arises from the fact that the high-level description of the algorithm has been designed to be run on software, in which infinite resources are assumed. Thus, dynamically assigned memory, recursion, pointers, unbounded or irregular loops, etc. are not supported, and the designer has to adapt the algorithm to cope with these limitations manually. Therefore, High-Level Synthesis tools require the original code to be profoundly adapted. Other limitations of the current tools using these methodologies include specific data types, synthesis directives or coding styles.

- The results are obtained by a laborious process that includes HW/SW partition definition, interface creation, behavioral synthesis and simulation, testbench creation and checking. The result is a final implementation that might be totally ineffective because it does not fulfill the constraints.

Static Source analysis In these approaches, a specific parser analyzes the high-level implementation of the algorithm and generates an intermediate format that describes the same functionality in a language-agnostic way. Some examples of intermediate representations in the state of the art are Trimaran IR [CGWm04], VHDL AST, LLVM [LA04] or GIMPLE [Mer03]. From this intermediate representation, a control and data flow graphs are derived. These graphs represent all the operations performed between variables as well as their dependencies, and by analyzing them along with a model of the target platform, performance bounds can be obtained. These approaches analyze the model without introducing any vector to the system, so they do not provide results about a concrete implementation, but several bounds instead; either a lower bound [NR00], an upper bound [GDWL12] or both [AH08].

The main disadvantages of this approach are:

- In algorithms where the complexity is high, the execution time of the estimation algorithm could make it impossible to use.
- Runtime dependencies might not be taken into account (for example, array element dependencies).
- The number of possible execution paths could be very high. This leads to abstractions in the analysis, such as supposing that loops execute until the maximum number of iterations that the variables that control the loops enable. Those abstractions are justified by the fact that static analysis always give 'sound' results, and therefore no assumptions about particular executions can be made. However, they are more often than not too pessimistic, what leads to inaccurate estimations that are not useful if a designer is interested in knowing what would be the minimum execution time rather than the maximum.
- Some language features (such as "while" or "for" loops) are not totally supported.

3.6.2 Our approach

To solve the limitations of the state-of-the-art techniques presented in the previous section, we propose a profiling technique based on dynamic scheduling. In our approach, the embedded software is compiled with the toolchain of the host system but instrumented to reflect its execution in the specific target platform. This technique has several advantages over the analyzed state-of-the-art in the sense that the process is easier than the synthesis + simulation one. However, several difficulties arise when it is used to perform hardware component estimations. Two problems remain unsolved at this point of the discussion:

- The program is run on a sequential platform, while hardware is parallel by nature. We will need to add a hardware planning system that assigns a time to each hardware operation based on its dependencies and not based on the order in which it is executed sequentially.

- The program is run in a system that is different to the final platform. Therefore, we need to model the target architecture of the destination platform instead of relying on the parameters of the platform where the code is being run. An example of this problem is that we cannot use the memory accesses of the program when running on the host platform for memory profiling as the target platform will usually have different memory sizes.

Model of the target architecture

The standard architecture of the platform that we want to simulate is shown in Figure 3.14. The architecture is composed of

- A CPU performs the coarse functionality. To simulate the CPU part of the system we use the techniques described in Section 3.4.

- External memory resources (i.e. DRAM main memory) are used to transfer data from the CPU to the custom hardware.

- The specific logic of the ASIC performs the critical operations and has its own memory to accelerate the access to critical data.

The model of the platform is intended to be as minimal as possible, so a new technology can be easily fitted into the tool even if its internal architecture details are not completely defined. The main high-level parameters of this platform are the speed of the buses and

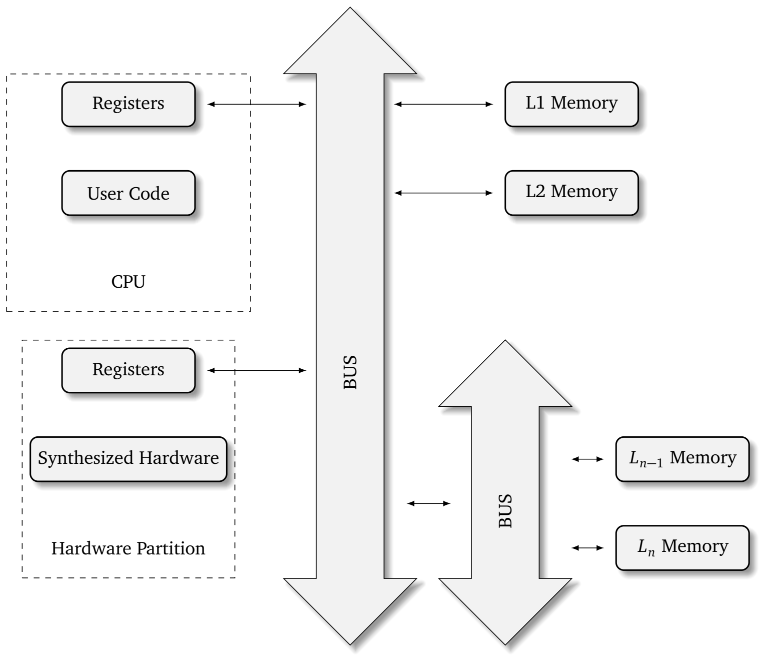
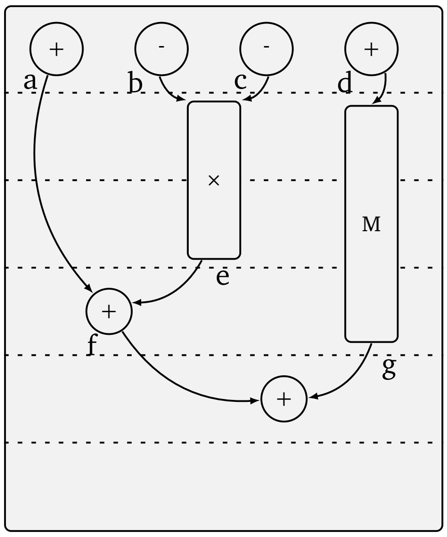


FIGURE 3.14: Architecture of the HW/SW Platform

the memory, the number of operational elements in the FPGA, the time required to execute each operation and the structure of the memory hierarchy.

Overall Performance Estimation Methodology

In our approach, the performance computation is done on-line, as the program runs on the developer computer, so data dependencies are calculated at run time. This approach enables us to know –at any point of the code– the exact path that the algorithm has followed to reach that point and all the links that constrain each instruction with the ones on which it depends. This is possible even if this relationship is not static (i.e. when the code uses a pointer to a variable). With this information, we can schedule every operation to be executed in the soonest time that is allowed by its dependencies. If a register results from a binary operation between other two register (such as in `r1 = r2 + r3` for example), we can associate a time for the register `r1` to be the latest time associated to the register `r2` and `r3` plus the time required to perform the binary operation. When this propagation of times is performed for every variable computed during the execution of the code, the time associated with the last modified variable in the execution corresponds



```
%a = add %1, %2
%b = sub %3, %4
%c = sub %5, %6
%d = getelementptr(d,5)
%e = mul %b, %c
%f = add %a, %e
%g = load %d
%r = add %f, %g
```

FIGURE 3.15: ASAP scheduling for eight simple operations, assuming that additions/substractions take 1 cycle, multiplications take 2 cycles and memory accesses take 3 cycles.

to the execution time of the algorithm in a platform in which several individual operations can be performed in parallel. The result of this scheduling of operations is called ASAP (As Soon As Possible) scheduling. We present an example of ASAP scheduling for eight simple operations in Figure 3.15.

When we apply the ASAP scheduling to a set of operations, the time associated with the last operation (or the height of the corresponding graph) is the minimum time required to execute the computation if the platform had unlimited resources. This is, however, an under-estimation of the optimum execution time in a real platform because in most cases the assumption of unlimited resources does not hold. In the example presented in Figure 3.15, it can be the case that the platform can not compute the four additions/subtractions in the first row at the same time because the platform might not have sufficient computing elements to perform those operations simultaneously. The presented technique considers the resource limitations as well as the memory delays according to the following constraints, which are based on known methodologies commonly used by synthesis tools to reduce execution time:

- There are no limits between basic blocks in the code. This assumes an operation enclosed in a particular basic block can be moved out of it and execute in the same time-slot as operations of another basic block.

- Pre-fetching is allowed: We assume we can predict which operations will be executed in the future and ask for their operands before executing the operation.

- Constant propagation. When a variable is assigned a constant value, the time assigned to this transaction is zero, as the value does not need to be taken from registers. If, after that, and without any change, another variable takes the value from this newly created variable, we can assume that the second variable takes its value from the constant too, so we can also consider zero time for this transaction. This assignment of the constants is of particular importance in the platform, as it performs automatic loop-unrolling in for-loops where the indexes start and are incremented with constant values.

- Speculative execution. With this rule, we assume that, whenever we perform a conditional operation, we can always predict the correct way the algorithm will continue, so all the data needed in this branch can be pre-loaded.

- Last Recently Used (LRU) policy for memories. When an element has to be removed from a hierarchical level in the memory to provide room for newer data, LRU policy expels the element that will be used last.

The algorithms to update the time associated with each variable in the presence of these constraints are part of the Masters Thesis of the same author. The interested reader is referred to [dAMGBE11][dA11].

3.6.3 Results

We have tested the presented technique with a subset of the CHStone [HTHT09] benchmarks. These benchmarks include a set of C programs to evaluate the effectiveness of different synthesis tools. The goal of a synthesis tool is to transform this C source code into a functionally equivalent representation of the same functionality in a Hardware-Description-Language (HDL).

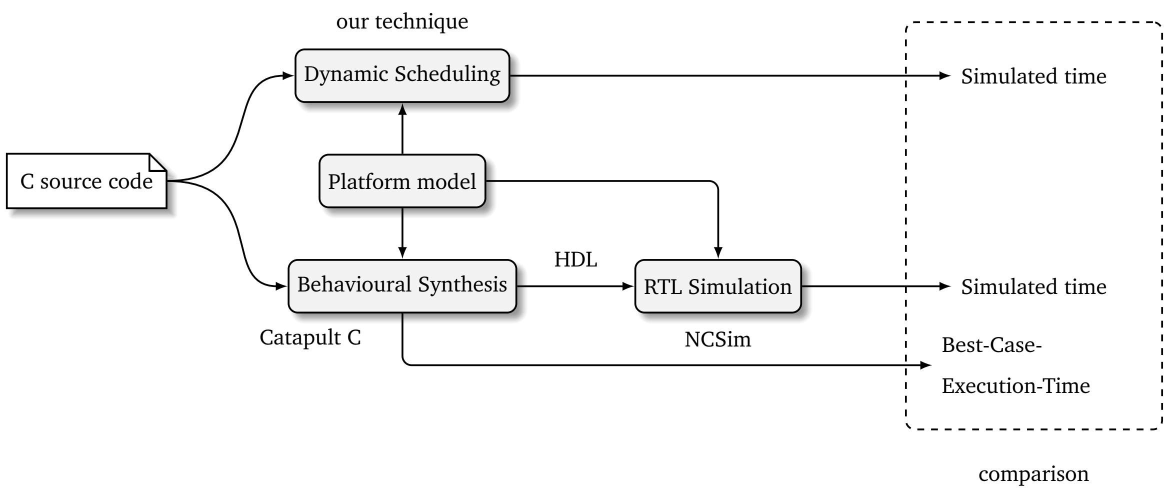


FIGURE 3.16: Experimental setup

In our analysis, we will compare the results of our simulation methodology and two commercial tools. For the behavioral synthesis and static performance estimation, we use CatapultC [cat]. For the RTL simulation, we use NCSim [ncs]. It is worth mentioning that these two tools are among the most advanced commercially available tools in their fields. We present a diagram of the performed experiments in Figure 3.16.

A precise evaluation of our approach with these tools is complicated for the following reasons:

- The benchmarks are meant to pose complicated problems for synthesis tools and therefore, even the most advanced tools at the time of this writing are unable to synthesize all the test benches.
- Even when the tool generates a valid hardware representation, the limitations of the synthesis tool can make the generated solution non-optimal.
- The estimation results of the Static Analysis Approach regarding the execution time is indeed a bound of the best-case execution time and not the result of a concrete simulation. Therefore the difference between this bound and a concrete execution can not be considered an "estimation error" but a measurement of how much over- or under- estimated this bound is with respect to a feasible execution.

Example	Synthesis + Simulation	Static Analysis	Dynamic Analysis	Speed-Up Over S+S	Speed-Up Over S.A.
aes	13002(0%, 9min 25sec)	5529(8min 10sec)	8990(30%, 0.004sec)	141250	122500
dfdiv	3806(0%, 3min 2sec)	2004(1min 47sec)	2352(38%, 0.003sec)	60666	35666
dfmul	1302(0%, 3min 38sec)	642(1min 48sec)	1601(22%, 0.001sec)	218000	108000
gsm	17178(0%, 12min 58sec)	9114(11min 54sec)	10538(38%, 0.006sec)	129667	119000
mips	5722(0%,3min 11sec)	107(2min 16sec)	4777(16%, 0.002sec)	95500	68000

FIGURE 3.17: Comparative results of the different techniques presented in this section

We want to emphasize that we do not consider these techniques as antagonists but complementary. The differences will become more clear in Chapter 4.

CHStone benchmarks include 12 test cases. We compare the results of five examples because the rest of them are not directly synthesizable by the state-of-the-art tool that we are using for comparison, either for the lack of computational resources or because the programs contain elements that are not supported. This can be seen as a representative example of the limitations of synthesis tools. The compared test-benches, including a brief description of the functionality, are:

- aes: Encrypting and decrypting functions.
- dfdiv: Division of two elements represented in floating point arithmetic with double precision.
- dfmul: Multiplication of two elements represented in floating point arithmetic with double precision.
- gsm: Linear predicting encoder for voice transmission in GSM network.
- mips: Simplified MIPS microprocessor.

We present the results of the comparison in Figure 3.17. In each cell, we show the Simulated time (the time that the techniques produce regarding how long it would take to execute the example in the hardware architecture), the error (the variation of this value with respect to the Synthesis + Simulation approach, that is considered to be the most accurate one and therefore is considered as the reference) and the time needed to obtain this result. Note that in the Static Analysis case, the difference with respect to the Synthesis+Simulation approach is not considered an error because Static Analysis provides a lower bound of all the possible execution times. The experiments have been performed over an Intel(R) Core(TM) i7-2600 CPU at 3.40GHz, with 16 GB of RAM. The last two columns show the speed-up of our technique with respect to the Synthesis+Simulation approach and the Static-Analysis approach.

3.6.4 Optimization example

Introduction

This section presents an example of the application of the technique described previously to the interactive reconstruction of 3D volumetric information in real time. The aim of the algorithm is to generate the Convex-Hull of a set of silhouette images. [Lau94] defines the Convex-Hull as the maximal silhouette-equivalent of an object on a set of viewing regions. From an implementation perspective, the most restrictive operations of the state-of-the-art algorithms are processing speed and memory. In this section, the methodology presented in the previous chapter is used to analyze a standard "visual-hull" reconstruction algorithm and propose improvements aimed to reduce the resource consumption. These improvements outperform the reconstruction time of the latest publications (our implementation is three times faster than other techniques present in the state of the art [LB08] for the same implementation conditions), allowing real-time and high-definition implementations.

The section is structured as follows; in "classical algorithm" we present the standard "visual-hull" reconstruction algorithm and present the main limitations of a naive implementation for real-time computation. In "projection matrix transformation" we summarize our techniques to transform the initial projection matrix computation so that the number of operations and their cost is reduced. In "recursive computation" we summarize further improvements to exchange expensive computations by simpler ones while maintaining the

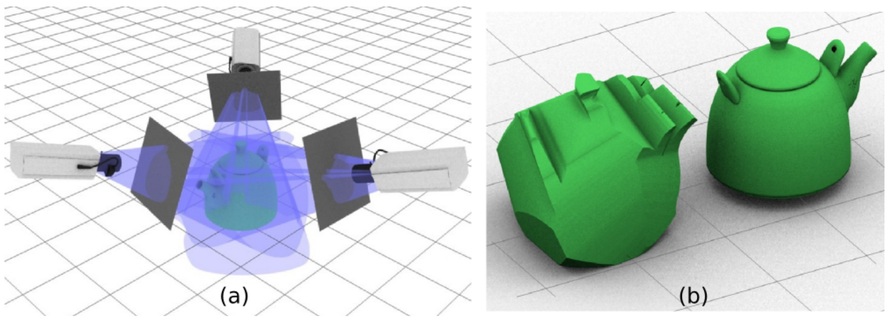


FIGURE 3.18: (a) Reconstruction scan seen from three different points of view. (b) The visual Hull of a teapot computed from three points of view (left) and the object (right).

functional equivalence between the two implementations. In “memory optimizations” we present an optimization to exploit dual port memories present in modern FPGA platforms to reduce the execution time further. Finally, in “FPGA implementation and results” we show the results of our implementation and compare it against previous state-of-the-art techniques. This section summarizes our paper [PAS12]. We refer to the interested reader to that paper for the detail description of our contribution.

Classical algorithm

This section introduces the classic algorithm for voxel-based Visual Hull [Lau94].

First, several cameras capture the scene and extract silhouettes of the objects of interest with Foreground Segmentation algorithms [HCZD04][MBRG00][MBM01]. These algorithms replace the image by a binary mask (silhouette) that indicates which pixels are part of the object image and which ones are part of the foreground. Then, voxel-based Shape-from-Silhouette techniques (SfS) projects each voxel onto all cameras to test if the silhouette contains the projection.

The core of the Visual-Hull algorithm lies in the mapping from the 3D voxel space to the camera space, where each voxel is checked to be inside or outside the silhouette of the object projected into this camera. Projective transform is the projection of the voxel space coordinates $(X_{voxel}, Y_{voxel}, Z_{voxel})$ to the sensor camera coordinates (X_{camera}, Y_{camera}). This projection can be obtained with a simple ray-tracing process as in Figure 3.19, and can be

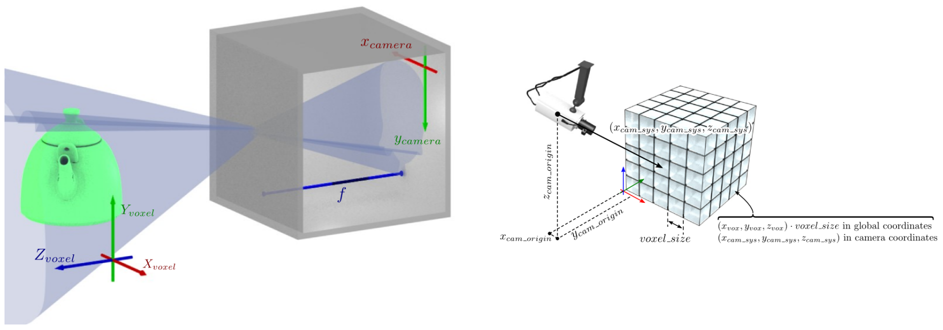

x_{camera}
y_{camera}
f
Y_{voxel}
Z_{voxel}
X_{voxel}
$(x_{cam\_sys}, y_{cam\_sys}, z_{cam\_sys})$
z_{cam_origin}
x_{cam_origin}
y_{cam_origin}
$voxel\_size$
$(x_{vox}, y_{vox}, z_{vox}) \cdot voxel\_size$ in global coordinates
$(x_{cam\_sys}, y_{cam\_sys}, z_{cam\_sys})$ in camera coordinates

FIGURE 3.19: Classic ray-tracing equations for voxel projection. Camera transformations.

expressed as a linear transformation as shown in equation 3.1.

$$\begin{bmatrix} x_{camera} \\ y_{camera} \\ w \end{bmatrix} = \text{In} \cdot \begin{bmatrix} X_{voxel} \\ Y_{voxel} \\ Z_{voxel} \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_{voxel} \\ Y_{voxel} \\ Z_{voxel} \end{bmatrix} \tag{3.1}$$

Taking into consideration the position and orientation of each camera, in [PAS12] we express the projection as:

$$x_{\text{pix}} = \frac{P_0 \cdot x_{\text{cam\_sys}} + P_1 \cdot y_{\text{cam\_sys}} \cdot P_2 \cdot z_{\text{cam\_sys}} + P_3}{P_8 \cdot x_{\text{cam\_sys}} + P_9 \cdot y_{\text{cam\_sys}} + P_{10} \cdot z_{\text{cam\_sys}} + P_{11}} \tag{3.2}$$

$$y_{\text{pix}} = \frac{P_4 \cdot x_{\text{cam\_sys}} + P_5 \cdot y_{\text{cam\_sys}} \cdot P_6 \cdot z_{\text{cam\_sys}} + P_7}{P_8 \cdot x_{\text{cam\_sys}} + P_9 \cdot y_{\text{cam\_sys}} + P_{10} \cdot z_{\text{cam\_sys}} + P_{11}} \tag{3.3}$$

These calculations (3.2 and 3.3) require a significant number of operations per pixel: 10 additions, 14 multiplications and 2 divisions for each voxel. For example, the algorithm requires 1 billion additions, 1.4 billion multiplications, and 201 million divisions to regenerate a 256^3 voxel volume for six cameras. This large number of operations and their complexity (i.e. n-bit multiplications) are two critical limitations for real-time implementations. They also limit efficient hardware implementations (e.g. FPGAs). In this work, we will present methods to reduce such a large number of operations.

Projection matrix transformation

Using the constant propagation techniques presented in the previous section, we detect several terms of the projection that can be pre-computed before the algorithm starts. Note that these optimization opportunities are hidden behind several matrix multiplications and involve various functions. Moreover, the mixture of pointer dereferences for multi-dimensional arrays makes the constant propagation analysis out-of-reach for typical static compiler optimizations. However, running the program enables to detect such optimization opportunities even in cases where the constant propagation involves several elements of an array or a matrix, which is especially true in digital signal processing systems where multiple array elements are updated with similar computations. These optimizations are explained in more detail in [PAS12], and consist in matrix transformations that transform equations 3.2 and 3.3 into

$$x_{pix} = \frac{(P_0 x_{vox} + P_1 y_{vox} + P_2 z_{vox}) + P'_3}{(P_8 x_{vox} + P_9 y_{vox} + P_{10} z_{vox}) + P'_{11}} \quad y_{pix} = \frac{(P_4 x_{vox} + P_5 y_{vox} + P_6 z_{vox}) + P'_7}{(P_8 x_{vox} + P_9 y_{vox} + P_{10} z_{vox}) + P'_{11}} \tag{3.4}$$

These equations include constant parameters (so they can be pre-computed before SfS algorithm starts) and operate with integer numbers instead of floating point numbers, which simplifies the implementation of the algorithm in hardware platforms.

Recursive computation

The second proposed technique is to perform calculations in a recursive way. This improvement uses previous results of voxel projections to calculate next projections. This improvement has a substantial impact on hardware implementations because the cost and execution time of hardware multiplication is higher than hardware addition. For instance, latency time (in an XC5VLX33O-1FF1760 FPGA) for a 64-bit adder is 4.41 ns and for a 64-bit multiplier is 13.674ns (3 times more). Additionally, this transformation reduces the amount of memory used. The delay it takes to execute different operations with different datatypes is usually a well-known parameter in hardware architectures focused on digital signal processing and is a parameter of the simulation framework, so the effect of these optimizations can be measured even before a VHDL implementation is performed, and without costly RTL simulations.

TABLE 3.4: Timing and ocupation results

	cp (ns)	Slice registers	Slice LUTS	Memory	DSP 48
SFS (8 cameras)	7.978	63.647 (30%)	33.641 (16%)	76 (26%)	16 (8%)
SFS (18 cameras)	7.996	137.902 (65%)	71.487 (34%)	171 (59.4%)	30 (15%)

Memory Optimizations

Even when a minimum critical path is reached, the processing time is still proportional to the resolution in every axis and happens to be limited by the memory accesses. To reduce the total processing time we introduce another improvement oriented to FPGA-based implementations. Exploiting the double port topology of the FPGA memories, we divide the projection modules for each camera into two different modules. This approach is also used in the projection test module. Each of these modules will calculate the projection in either odd or even voxels along one axis. As these modules can access the memory concurrently, the total processing time results in equation 3.5.

Next section shows the area and total processing time when using a commercial FPGA-based (Virtex 5 VD(330)) platform.

FPGA Implementation and results

In this section, we present the synthesis results of the proposed SfS architecture in a hardware platform with a single FPGA. The results presented in Table 3.4 were obtained after Place and Route using Synopsys Synplify 9.2. The image resolution used was 640x480 pixels.

Several facts should be highlighted out. Firstly, the DSP482 consumption is very low due to the application of the improvements presented in section 3.6.4. When no algorithmic improvement was applied, the DSP48 consumption for eight cameras was as high as 78.6%. Secondly, the area utilization is practically consumed by the projection modules. The projection test module consists only of simple AND gates. As a conclusion, the area increases proportionally to the number of cameras.

The execution time of the algorithm with the techniques mentioned above can be

TABLE 3.5: Processing time of our proposal compared with two recent implementations.

Resolution	Architecture	Time to process a frame (ms)
$300 \times 300 \times 200$	[LBN08]	179.98
$300 \times 300 \times 200$	our proposal	72
$128 \times 128 \times 128$	[OKS+08]	33.33
$128 \times 128 \times 128$	our proposal	8.4

approximated as in equation 3.5. Since processing time has to be less than 34 milliseconds to reach 30 frames per second (Real-time requirement), the maximum voxel resolution we can reach is 256x256x128. The time of operations in the critical path (cp) has been estimated with the techniques presented in Section 3.6 and validated with Synopsys Synplify 9.2.

$$\text{processing\_time} = \frac{256 \cdot 256 \cdot 128 \cdot 8 \cdot 10^{-6}}{2} = 33.55ms. \tag{3.5}$$

Because some papers focus on high resolution and others focus on real time, it is not easy to compare. For the sake of the reader, we show the processing speed results of our design in the same resolution conditions as [LB08] and [OKST08] (Table 3.5).

3.6.5 Conclusions

In this section we presented our technique for estimating the execution time of a synthesized hardware accelerator directly from the C source code. The technique does only require a rough partition of the HW/SW design, so it is amenable for being used in the early stages of the design process. We evaluate the speedup and the accuracy of the technique with standard test-benches and its usability with a computer-vision application.

3.7 Simulation of the thermal behavior

3.7.1 Introduction

In this section, we propose a complete framework based on native simulation for early estimation of power consumption and thermal analysis in Multi-Processing Systems-on-Chip. These analyses are becoming increasingly important for Multi-Processing Systems-on-Chip design due to its impact on system reliability, power consumption, and cost. Indeed, working above the operating temperature range of the chip may drastically reduce its lifespan and the system reliability, which is crucial in most applications of embedded systems. Moreover, sudden changes or large differences in the temperature of the system can cause material stress, age the components and increase the chances of failure. This aspect is of paramount relevance in current embedded Multi-Processor-System-on-Chips (MPSoCs) where Dynamic Power Management (DPM) and Dynamic Voltage-Frequency Scaling (DVFS) policies are applied, that selectively tune the power supply of processing cores or peripheral components, resulting in both temporal and spatial temperature differences. Temperature also has an impact on power; high temperatures cause a notable increase in leakage currents, which constitutes up to 25% of the total energy consumption in CMOS (Complementary Metal Oxide Semiconductor) technology. Thus, different floorplans of the same SoC (System-on-Chip) can lead to designs with three times larger leakage power due to the differences in component temperatures [KKAD06].

This section is structured as follows; in “State-of-the-art” we present other techniques for thermal analysis and their limitations. In “MPSOC thermal estimation” we present our technique, consisting in integrating the simulations described in Section 3.4 and 3.5 with a thermal model of the System-on-Chip and iteratively compute the software behavior and the thermal behavior. In “Experimental Results” we evaluate the proposed technique by evaluating the accuracy and speedup of the proposed native-execution approach against a more traditional ISS approach. In “Conclusion” we finalize the section emphasizing the benefits of this approach for initial estimation of thermal behavior in MPSoC architectures. The fundamental concepts of this section, as well as the details of the thermal model, are taken from a paper that I co-authored with Daniel Calvo, Luís Díaz, Héctor Posadas, Pablo

Sánchez, Eugenio Villar, Andrea Acquaviva and Enrico Macii [CGD+11].

3.7.2 State of the art and related work

Thermal and power-aware design has become an active research area due to the need of reducing energy consumption and improving reliability in embedded low-power applications.

[KKAD06] proposes a leakage-aware exploration (LAX) framework for IP-based SoC design. LAX explores the relationship between leakage power and different floorplans for the system, allowing early identification of current leakage problems. [KKAD06] performs the exploration at transistor level, so requires a high degree of detail to describe the floorplan, not being appropriate for the first stages of the design process.

In [SSS04], a thermal/power model for super-scalar architectures called HotSpot is presented. It predicts the temperature differences in the processor and also takes into account the increased leakage power and reduced performance. The results prove the importance of hot spots in high-performance systems.

MPARM is a multi-processor cycle-accurate architectural simulator. Its purpose is the analysis of design tradeoffs in the usage of different processors, interconnects, memory hierarchies and other devices. It works with several ISS processor models such as SWARM, CoWare Cores, SimIt-ARM or PowerPC 750 and enables power consumption estimation [BBB+05]. [PPB07] uses MPARM to obtain power consumption from the different components of the system, providing information to a high-level thermal model. Although this approach provides high accuracy, it is based on a cycle-accurate simulator, which makes it too slow to perform efficient design exploration.

[BCTB10] presents a complete virtual platform for exploring power, thermal and reliability management control strategies in high-performance multiprocessor architectures. It can obtain power and thermal estimations of an entire platform. It integrates a framework to perform co-simulation of multicore SoCs to try different control strategies. However, the technique is based on Simics [MCE+02]; a commercial instruction set simulator. As stated before, ISSs are accurate but not appropriate for efficient exploration of design alternatives due to the long simulation times.

Analyzing the previous study, we conclude that ISS techniques dominate the state-of-the-art for thermal simulation. As demonstrated before, despite being very accurate, ISS simulators are not appropriate for practical exploration of many different design alternatives due to the long simulation times. Indeed, it is possible to conclude that fast, sufficiently accurate, power and thermal analysis based on native simulation can provide a powerful technology for supporting architectural design space exploration and optimization. Native simulation is particularly appropriate as it can support efficient modeling of DVFS architectures. Nevertheless, as far as we know, native simulation has never been applied for high-level thermal analysis.

3.7.3 MPSOC thermal estimation

To simulate the thermal behavior, we have integrated a high-level thermal model with the native simulation framework described previously. Next section summarizes the details about the thermal model to facilitate the understanding of the whole work. The thermal model does not constitute a contribution of this dissertation and full details are described in our publication [CGD+11].

High-Level MPSoC Thermal Model

The thermal model of our approach mimics the flow of heat from the bottom of the chip die (where the heat is produced by the switching of internal gates) to the surface and the surrounding environment (where it is dissipated by convection). As described in [SLD+03], the thermal behavior of a die can be modeled as an electrical circuit, and several analogies can be made between how the heat flows inside the chip and how electricity would behave in an equivalent RC circuit. This analogy introduces the idea of modeling the heat dissipation inside the chip as solving the currents and potentials of a circuit composed of resistances and capacitors in an equivalent electric circuit.

To solve the differential equations that govern the electrical circuit response we use classical iterative procedures. Those algorithms require discretizing the space inside the

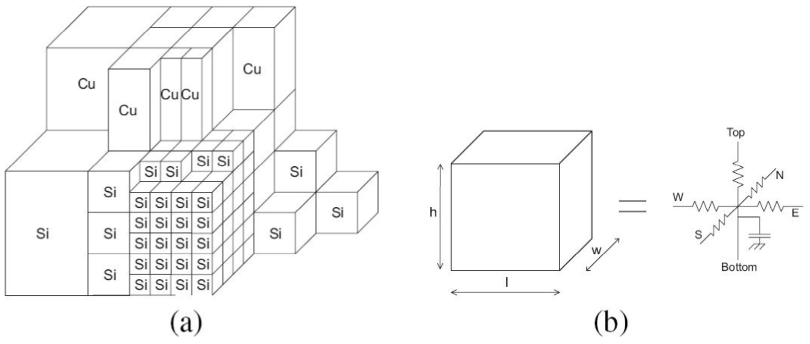


FIGURE 3.20: (a) Chip divided into cells. (b) Equivalent RC circuit.

die as well as the time. The die and heat spreader are divided into cubic cells of several sizes (as seen in Figure 3.20). This enables to vary the level of detail of the simulation by placing the smallest cells at the most important points of the simulation space and inserting larger ones where the simulation does not need a great amount of detail.

High-Level MPSoC Floorplan

A high-level floorplan specifies the size and position of the cells. The only information needed to make up the plan is the position of each component and its approximated size. Thus, in the first stages of the design process, it is possible to explore the choices that may lead to implementations with hot spot problems. The floorplan consists of two text files.

- Floorplan Architecture Model File: We model the chip layout as characters describing the various components in the silicon layer. Each character shows the position of an element on the floorplan. In our model, we assume a planar 2D layout (Figure 3.21). The components are categorized in the figure as Processor (p), Instruction Cache (i), Data Cache (d), Main memory (m) and No component (x).

- Cell Characteristics Definition File: It is used to set the number of layers used for the floorplan, different cell types and sizes, thickness and the number of layers.

We present an example of a high-level floorplan in Figure 3.26. We can create these files representing the MPSoC Floorplan by hand or with a graphical-user-interface.

As Figure 3.22 shows, the thermal model is fed using the power consumptions that are extracted from the component activities using the high-level models of Section 3.7.3.

At the same time, an XML file has been defined so that thermal parameters can be configured easily.

Modeling dynamic voltage and frequency scaling

In our framework, we have modeled Dynamic Voltage and Frequency Scaling techniques which are widely used in embedded systems. Although there are several types of DVFS controllers on the market, in this work we have selected the ARM Intelligent Energy Manager (IEM) Controller [iem]. This IEM is a good example of DVFS controller for embedded systems. An important point in the ARM solution is the use of a pair of tables to define the mapping between a performance index and the platform specific hardware parameters: voltage and frequency. This interface is necessary because voltage scaling is highly technology-specific (for example, the number of voltage steps that can be supported is restricted) and the clock generator is system and technology dependent as here might be only a limited number of divider ratios available. The tables define the possible values of voltage and frequency as well as their relation. To emulate this behavior, we implement the same API that is used in the ARM Intelligent Energy Manager (IEM) Controller that modifies the cost of each instruction in terms of execution time and energy consumption for each instruction as described in Section 3.4.3 during the run-time simulation.

```
iiiippppmmmmmm
iiiippppmmmmmm
iiiippppmmmmmm
iiiippppmmmmmm
ddddxxxxmmmmmm
ddddxxxxmmmmmm
ddddxxxxxxxxxx
ddddxxxxxxxxxx
```

FIGURE 3.21: Example of floorplan architecture model file.

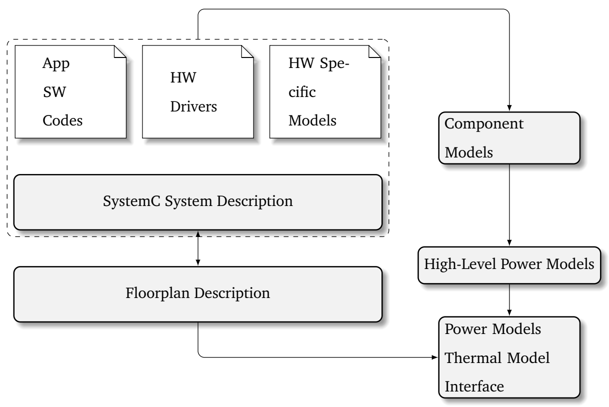


FIGURE 3.22: Simulation environment to estimate MPSoC temperatures

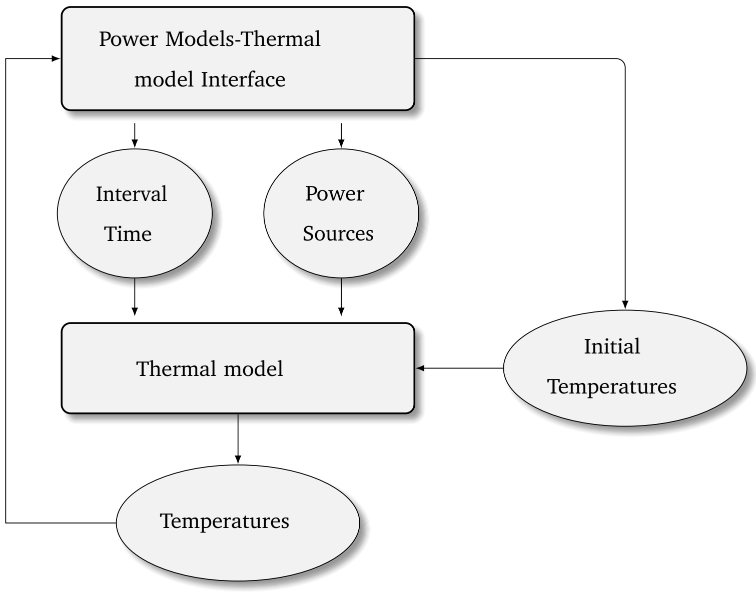


FIGURE 3.23: Thermal Simulation Flow

3.7.4 Experimental results

Accuracy and speed comparison

To verify the quality of our simulation framework and to demonstrate its advantage in simulation time, we present a comparison with a cycle-accurate ISS approach. We use both the ISS and the Native Simulation environments to create power consumption traces of the different components and feed the same thermal model in both cases. To ensure the accuracy of the thermal model, it has been calibrated with a 3D-finite element package [PPB07].

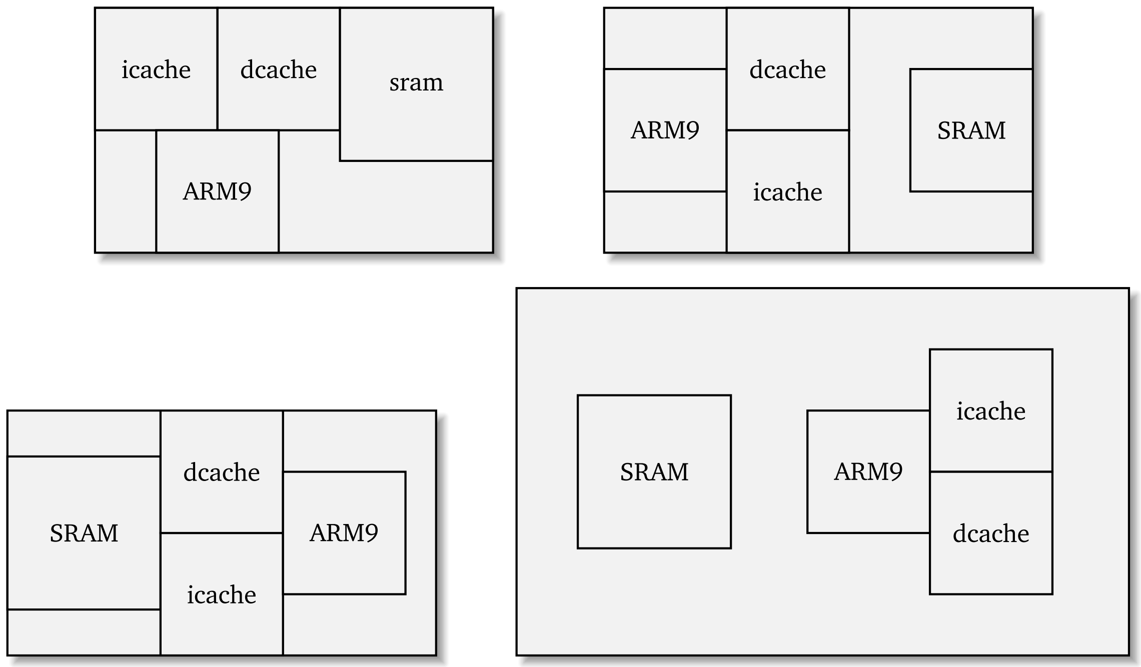


FIGURE 3.24: Floorplans used in accuracy comparison

The comparison experiments simulate a target platform with an ARM920T, including 16KB instruction and data caches and an 8-KB internal SDRAM. The dimensions of the memories, caches, processors, the basic power consumption values and the parameters to customize the thermal model have been provided by an industrial partner.Four different floorplans (Figure 3.24) have been used, studying the error in the dynamic temperature estimation for the different components that are part of the system. The software benchmarks are:

- Matrix Multiplication: This software application performs matrix multiplications. It is an application that computes data intensively, consuming a significant amount of energy and actively heating the system. This example has been used in other works related to thermal estimation to study the estimation accuracy [PPB07].
- GSM Encoder: It consists on an Enhanced Full Rate GSM Speech Coder (GSM-EFR). Working at 12.2kbit/s the EFR provides wire-like quality in any noise free and background noise conditions.

In Table 3.6 the estimation error of our approach is presented, the obtained timing

TABLE 3.6: Thermal estimation accuracy

Application	Maximum	Component			
Software	error (%)	i-cache	d-cache	processor	memory
GSM Encoder	Floorplan 1	2.9651 %	2.7089 %	2.7219 %	2.4752 %
	Floorplan 2	2.6468 %	2.4771 %	2.1581 %	2.5915 %
	Floorplan 3	2.8746 %	2.6842 %	2.8039 %	2.5093 %
	Floorplan 4	1.9097 %	1.6990 %	1.7376 %	1.4392 %
Matrix Multiplication	Floorplan 1	2.0479 %	0.4141 %	0.4749 %	0.3435 %
	Floorplan 2	2.3723 %	2.7182 %	2.2723 %	2.4096 %
	Floorplan 3	1.8821 %	0.3692 %	1.1598 %	0.3493 %
	Floorplan 4	1.6966 %	2.0540 %	1.6645 %	1.5232 %

results are shown in Table 3.7.

The results show that our simulation framework achieves to estimate the temperature of the different components with a great accuracy (less than 3% error). Figure 3.25 shows the estimated processor temperature obtained for the matrix multiplication benchmark with the Floorplan 2. We can notice that our model tracks the thermal transient and responds with the same dynamics with less than 3% error.

Example of Modeling Thermal Management Techniques

To illustrate the capabilities of the system to simulate Dynamic-Voltage-and-Frequency-Scaling, we are going to use two different software applications: an H264 coder and a GSM coder. The H264 or AVC (Advanced Video Coding) is a standard for video compression. It is a block-oriented motion-compensation-based codec standard developed by the ITU-T Video Coding Experts Group (VCEG) together with the ISO/IEC Moving Picture Experts Group (MPEG). The GSM system is composed of the coder and the decoder. Each part contains several tasks that execute concurrently.

The floorplan is presented in Figure 3.26. A thermal sensor monitors the first processor

TABLE 3.7: Timing comparisons

Application Software	Simulation time (sec)	ISS Simulation + Thermal Model	Native Simulation + Thermal Model
GSM Encoder	Floorplan 1	1435.507 sec	0.819sec (1752 x)
	Floorplan 2	1463.287 sec	0.931 sec (1571 x)
	Floorplan 3	1440.026 sec	0.768 sec (1875 x)
	Floorplan 4	1468.079 sec	1.147 sec (1279 x)
Matrix Multiplication	Floorplan 1	1403.688 sec	0.346 sec (4056 x)
	Floorplan 2	1432.865 sec	0.387 sec (3702 x)
	Floorplan 3	1408.207 sec	0.360 sec (3911 x)
	Floorplan 4	1434.206 sec	0.740 sec (2779 x)

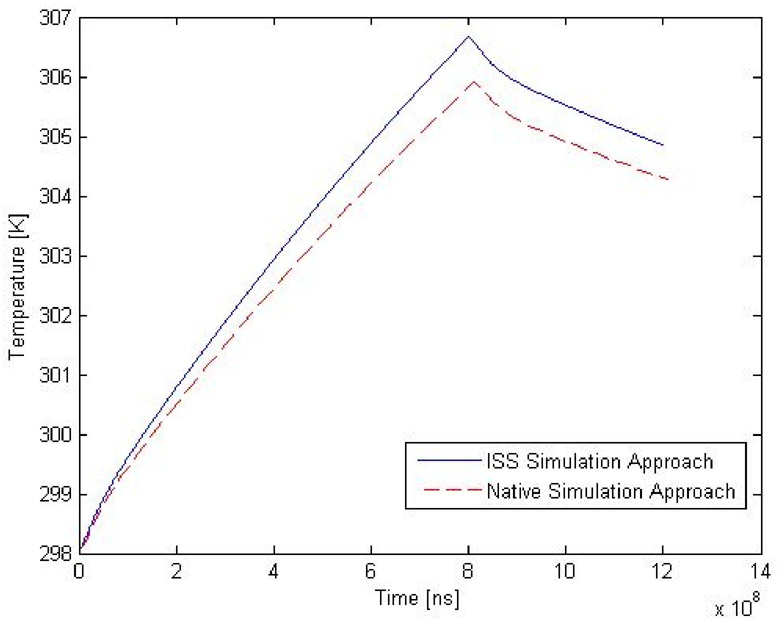


FIGURE 3.25: Transient thermal estimation comparison

temperature and applies the thermal management policy when it reaches 325 K. In Figure 3.27, we show the results for different thermal management policies.

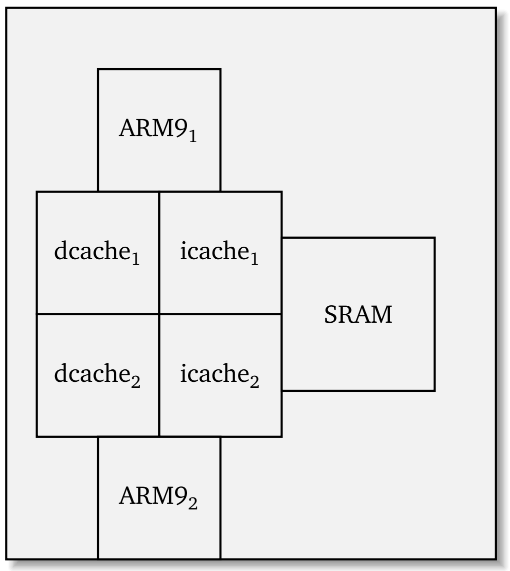


FIGURE 3.26: Floorplan used in the example

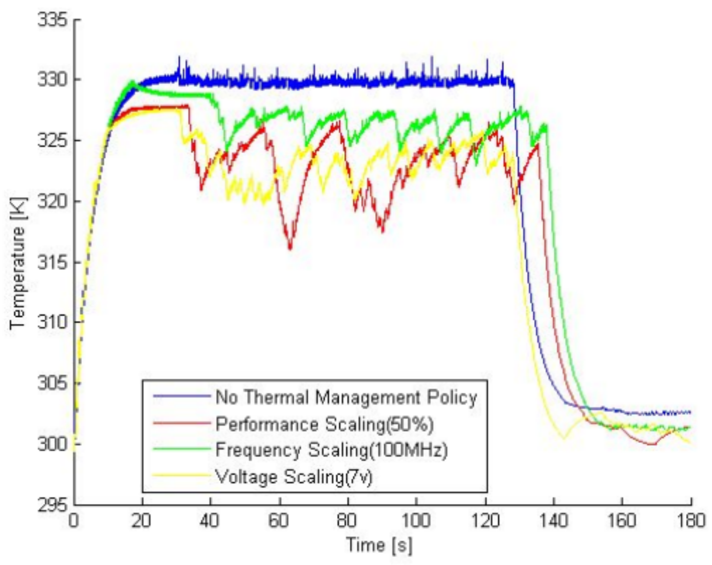


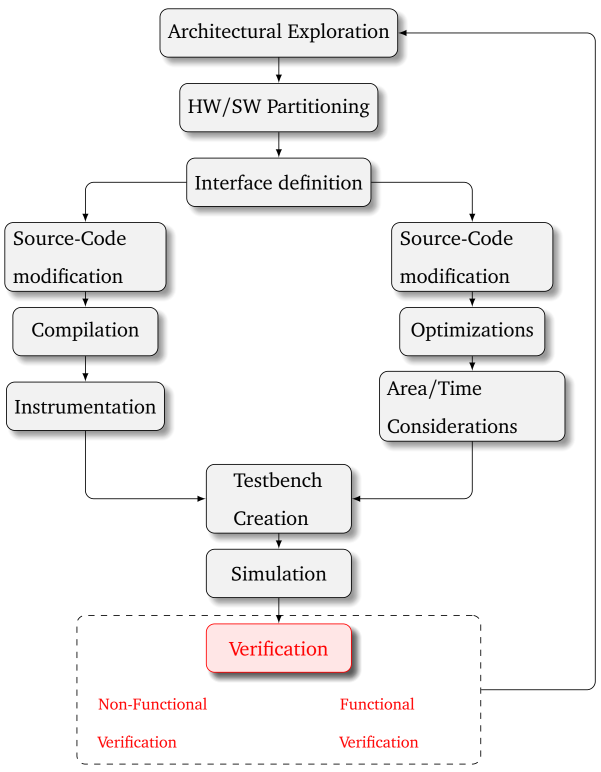
FIGURE 3.27: Transient temperature estimation with thermal management policies applied.

3.7.5 Conclusions

In this section, we have presented a complete MPSoC native simulation framework for power and thermal-aware design that enables to take early decisions about whether these design alternatives could lead to implementations with hot spot problems.

The presented results and examples demonstrate that our framework can be used to estimate temperature accurately in an early co-simulation infrastructure. By using ISSs, it is possible to perform a more detailed power and thermal analysis since the activity of internal processor components can be monitored. It is important to emphasize that both methodologies are complementary, in the sense that native simulation enables to estimate the effect of design decisions in the early stages of the design. ISS approaches can lead to more detailed results at later stages when higher accuracy is needed.

More importantly than introducing a trade-off between speed and accuracy, the proposed framework enables the detection of power and thermal hot spots in the first stages of the design process, allowing the exploration of different alternatives to address these problems both in the hardware as well as in the software domains.



4 Verification

4.1 Introduction

In previous chapters, we have described some techniques for analyzing the non-functional properties of programs with simulation. In these techniques, we provide an initial state of the hardware, a concrete input vector to the program, and estimate how would the program behave when starting from those particular initial conditions.

One significant shortcoming of this approach is that the conclusions obtained through the observation of such a program during a concrete execution do not generalize to other 'unseen' scenarios. When studying the non-functional properties of a system, we are usually interested in analyzing extreme cases (those in which the execution time, the stack size or the power consumption is large, for example). Due to the large set of possible behaviors of a program, these cases are not always covered by the test vectors that we

```
int gcd(int a, int b){
  if( a > 100 || b > 100 ) return -1;
  if( a < 0   || b < 0   ) return -1;
  if ( a==0 ) return b;
  return gcd ( b%a, a );
}
```

```
int main(){
  gcd(1,2);
  gcd(1,4);
  gcd(3,8);
  gcd(1,3);
}
```

FIGURE 4.1: Example showing the difficulties of finding the worst-case execution time of a program

simulate.

A simple example is presented in Figure 4.1. This program computes the greatest common denominator of two numbers that are smaller than 100. In Figure 4.1(left), the C code is presented and in Figure 4.1(right), the function is tested with five inputs. We can use the techniques and tools presented in previous sections to estimate what would be the execution time of this function for the given inputs. In this particular case, none of them corresponds to the longest execution time (the longest trace corresponds to `gcd(89,55)`).

Obtaining these extreme cases is complicated for at least two reasons:

- The program can exhibit many different behaviors and simulating it for all the possible input values is an unmanageable task, due to time constraints. In Chapter 3, we have presented techniques to accelerate the simulation of the program for a concrete execution, but we have not deal with the fact that there can be a large set of possible input values to the program. For the simple example of Figure 4.1, we would need to test the program for all the possible inputs (from the set of 10.000 different inputs in our program, the longest trace is only exhibited for the case $(89, 55)$).

- The hardware can also exhibit many different behaviors, due to initial conditions of the hardware elements. This is particularly the case for data and instruction caches. Depending on the fact that some of the instructions of the program may or may not be in the cache when we start executing the program, execution can take more or less time to finish. For any of the different inputs that we have previously tested, we would need to simulate it from all the possible initial configurations of the cache to withdraw conclusions about all the possible behaviors. This circumstance also highlights the fact that the longest execution regarding execution time does not always correspond to the longest trace.

In this chapter, we will focus on these problems in two stages. First, we concentrate on the "software" aspect of the explained difficulties. In this part of the chapter, we discuss techniques to group several executions into equivalent classes, so we can reason about properties that the program exhibit for a group of possible executions rather than a single concrete execution. These techniques are based on Symbolic-Execution and Static-Analysis. Because of our exclusive focus on the software in this part (not considering the hardware), we demonstrate the techniques with *functional* properties. We refer to a functional property as a requirement of the system related to the expected output of the program, assuming that it eventually terminates and disregarding execution time, power consumption... Informally, a functional property can be "on all possible executions of a program, not any pointer can be dereferenced if it points to the memory location 0x0". When a program violates a functional property, we informally refer to this circumstance as a "bug" in the program. In the first part of the chapter, we will focus on techniques to search for bugs (i.e. violations in the functional specification) and proving correctness (i.e. prove that there does not exist any violation of the specification). We then introduce models of the hardware and apply the techniques used for functional verification to non-functional verification. We restrict our scope to the problem of "worst-case-execution-time" (i.e. finding the longest execution time of a program given any initial condition both in the program and in the hardware state).

This chapter is structured as follows; in "state of the art", we present different techniques to analyze programs and reason about their behavior. In "symbolic state" we will use the annotation described in Chapter 2 to construct a dynamic analysis tool to find bugs in a program. In "combination with static analysis", we will combine our tool with a "static analysis" approach to improve both the capabilities of the "static analyzer" to find bugs and also the capabilities of the "symbolic execution" approach to prove the correctness of the implementation. In “the worst-case execution time problem” we introduce the problem of finding upper bounds of the execution time and how the techniques previously deployed for functional verification can be extended to non-functional verification by providing a –timed– model of the hardware platform. In “hardware models” we present formal models of a hardware platform and combine them with software models to derive the worst-case-execution-time of a program when executed over the modeled platform. Finally, in “Combination with UML/OCL”, we extend our framework to cope with specifications described in the graphical representation given by UML and the specification language OCL.

4.2 State of the art

In this section, we summarize several solutions proposed in the state-of-the-art both for the verification of functional and non-functional properties and classify them as based on static or dynamic analysis. While in dynamic analysis approaches the code is executed (either on the real platform or in a simulated environment), static analysis approaches “observe” the code and construct a model of its behavior without executing it. The different nature of these two approaches implies different design decisions:

- In Dynamic Analysis, it is assumed that not all the behaviors can be simulated, so these techniques try to find good generalizations to cover different behaviors with only a limited number of executions.
- In Static Analysis, the code is not executed, so the analysis needs to abstract the behavior of the program and assume information that could therefore only be obtained during run-time.

Static and dynamic analysis can be seen as two different approximation methods to cover the program semantics as shown in Figure 4.2. While (sound) static analysis over-approximates the program behavior and as such allows false positives, dynamic analysis under-approximates program behavior and can result in false negatives.

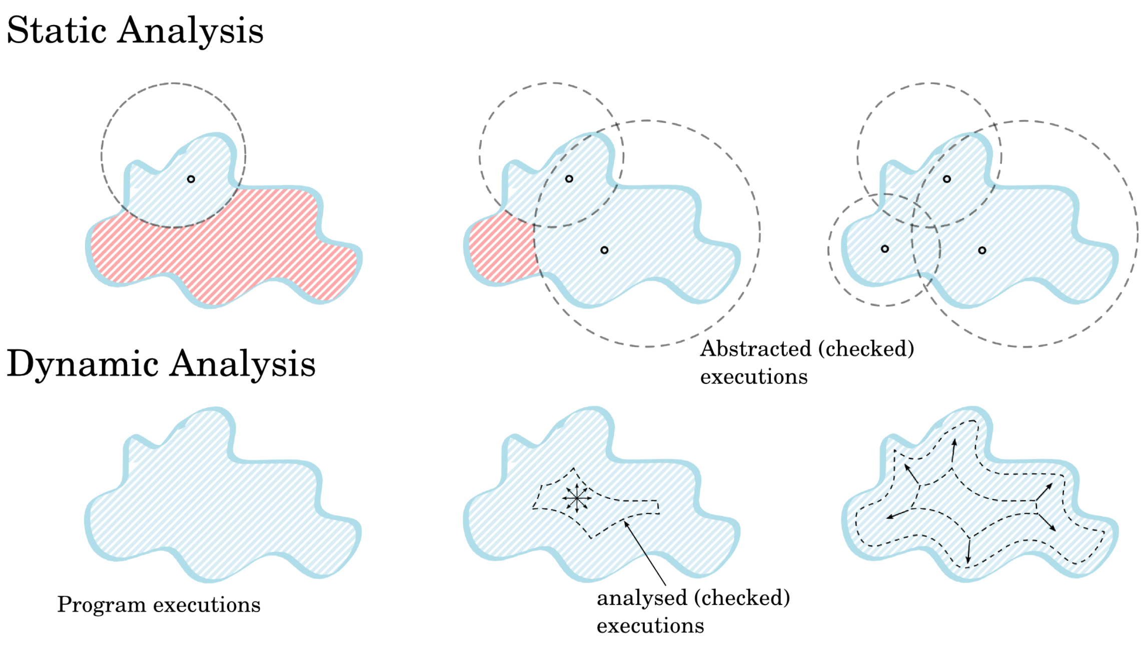

Static Analysis
Abstracted (checked) executions
Dynamic Analysis
Program executions
analysed (checked) executions

FIGURE 4.2: Top Row: static analysis coverage; bottom row: symbolic execution coverage.

4.2.1 Dynamic analysis

Regarding functional checking, software testing is extensively used to evaluate functional properties of a system against requirements to ensure coverage of both the requirements and the actual code. When testing, the code is executed with concrete input values, and the observations of the behavior of the program are limited to the inputs with which the program is tested. Multiple techniques try to alleviate this limitation. We will focus on some representative techniques for programs written in C/C++ or SystemC.

In Bounded-Model-Checking (BMC) the different behaviors of a program are explored up to a depth k (we explore feasible executions that include less than k instructions, branch decisions, basic-blocks or any distance metric of an execution trace). In this approach, the analysis can miss "bugs" that need more than k steps in the code to manifest. CBMC [CKL04b] is a well-known Bounded Model Checker for C and C++ programs that can check a variety of common functional properties in code such as buffer overflows, exceptions, assertions, pointer safety... Klee [CDE08a] is another bounded model checker focused on LLVM intermediate representation.

In those approaches, it is well understood that only a fraction of the actual semantic behavior can be realistically tested (bottom row of Figure 4.2). As shown in industry case studies [QR11][CGK11], BMC techniques are prone to scalability limitations and current tools are not well suited yet for deep embedded applications. As such their adoption in safety-critical industries has so far been limited.

In terms of non-functional properties, SWEdish Execution Time Analysis tool (SWEET) [Lis14] is an academic tool focused on the analysis of best and worst-case execution times using a variety of different methods that include static and dynamic analysis. For the analysis of the source code, the tool is integrated with a specific research compiler. The compiler is engineered in such a way that the relevant optimizations for computing the worst-case execution time are applied before producing the code that is analyzed. This close interaction with the compiler simplifies following steps, since high-level and low-level information can be conveniently mixed, but limits the applicability of the approach to platforms where the specific compiler is available. To deal with the high number of possible software behaviors, the approach uses a combination of symbolic execution and abstract interpretation that enables to compute loop bounds and infeasible paths. On the other hand, for dealing with the high number of possible hardware behaviors, a points-to analysis (an analysis performed over pointers to analyze what regions in memory they can access) is performed for the instructions that may access the memory in order to compute relevant information about the cache accesses. To analyze the pipeline effects, the tool relies on simulation models for individual traces. These models are derived from cycle-accurate simulators and (as any method based on simulation), cannot formally prove assertions about the execution time.

A research prototype developed in Chalmers University [Lun02] focuses on obtaining upper bounds for binary code of high-performance microprocessors such as PowerPC, featuring multilevel caches and deep pipelines. The source code is analyzed at machine level with a combination of symbolic execution and simulation. Being based on simulation, however, the approach has limitations to deal with big state spaces. Although authors prove that very accurate estimations of the execution time can be computed given an initial state space and sufficiently accurate models of the hardware architecture, the worst-case initial state is the key element to compute the WCET. To prove properties for any possible input, the approach uses a technique based on Bounded Model Checking. Therefore, several paths in the simulation might not be explored, what either complicates the production of a proof that ensures the safety of the bound, or delegates some of the responsibility of obtaining this proof to the programmer, who has to provide some facts about the code in the form of annotations. The exponentially large number of paths also makes the analysis very computationally intensive.

4.2.2 Static analysis

Static Analysis is widely used in safety-critical industries to verify the correctness of the implementation of some functionality according to a specification. Static analysis approximates the behavior of source code and detects common coding violations such as the ones defined by MISRA [1], but also possible software bugs that lead to runtime errors such as null pointer dereferences, memory leaks, and buffer overruns. Many commercial tools are based on earlier academic work and are routinely used in industry [BBC+10], [Gra], [Huu15], [YLB+08]. However, while static analysis is scalable to large code bases, it approximates program behavior and as such is prone to false positives (false alarms) and/or false negatives (missed bugs).

This is because static analyzers do not execute the code, so they require assumptions about the content of variables that would otherwise only be available during execution. An example of this is the analysis of infeasible traces (execution paths that are contained in the control-flow-graph of the program but can not be executed due to incompatible constraints when the full semantics of the instructions are considered). Several techniques have appeared recently to mitigate these limitations. Abstract Interpretation [CC77] builds an abstraction of the domain of the program and defines how instructions operate over this abstracted domain, instead of considering only concrete inputs. Counter-Example-Guided-Abstract-Refinement (CEGAR) [CGJ+00] also operates over an abstract domain to prove or disprove functional properties of programs. In this approach, the analyzer tries to

[1]MISRA C is a set of recommendations for the development of embedded software mainly focused on the C language for automotive, aerospatial, telecommunication and medical systems.

```
int main( argc , argv*[] ){
  if(argc > 10)
    return 10*argc + 2
  else
    return 0
}
```

FIGURE 4.3: Example to illustrate computation of the WCET

find a trace that exhibits a “bug”. If a trace is found and is feasible, the analysis concludes providing a concrete witness leading to the undesired behavior. If the trace is not feasible, the abstract domain is “refined”, so a more concrete execution can be simulated in the next iteration. Trace-Abstraction-Refinement [HHP09a] is a practical instance of CEGAR algorithm that has been applied to the verification of functional properties over the LLVM intermediate language [CSR+17].

Regarding the usage of static analysis for reasoning over non-functional properties, Shaw. et. Al [Sha89] proposed a way to obtain bounds for the WCET from the high-level language constructs by analysing the C source code in its primary form (before having been transformed by the compiler), and constructing new bounds for the execution time in a compositional manner, by iteratively abstracting the elements of the tree representation of the program (AST). For example, considering the program of Figure 4.3, the analysis would start by estimating bounds of atomic operators such as $+$ or $*$. Then, new bounds of more complex constructs such as the if statement are created by composing the execution time of their components. Similar approaches are used in Heptane [CP00][CP01].

The problem of computing the worst-case-execution-time has also been analyzed by correspondence with finding the longest path in a graph. The input program can be considered to be abstracted by its control-flow-graph; the nodes representing instructions and the edges representing time bounds that are exhibited by following those edges. Following this analogy, the computation of the CFG can be expressed as finding the maximally long path in the CFG, which can be solved by integer linear programming [Chv83]. ILP has been successfully used to model very simple processors [LM95][LMW95], but the complexity of new architectures disallow its use due to its poor scalability [FHW04].

aiT [FH04] is a commercial tool by AbsInt focused on safety-critical software and therefore aiming to obtain upper bounds for the execution time by static analysis. Its input is the executable binary code of the program, but the high-level C/C++ language may contain annotations to remove infeasible traces and improve the accuracy of the estimates. Internally, the tool uses Value Analysis and Integer Linear Programming to determine safe bounds for the WCET. This Value Analysis processing step is later reused in the analysis of pointer dereferences to feed a model of a cache. To highlight the differences between the aiT approach and our approach, the former uses "abstract interpretation" to implement value analysis, and a bottom-up analysis to reconstruct the CFG. These techniques have limitations that require manual annotation of the source code, and its absence produces highly over-estimated bounds for the WCET.

BoundT [bou] is another commercial tool based on static analysis that can estimate upper bounds for execution time and stack utilization. For the elimination of infeasible traces, the tool relies on user annotations. For the computation of stack and time bounds, "abstract interpretation" is also used. Several ad-hoc heuristics are used to analyze counter-based loops. In this approach, for standard regular loops variables inside the loop body are classified as invariant or variant, keeping loop-invariant variables untouched and assigning unknown values to loop-variant variables. Several other common structures that are known to be generated by standard compilers are identified by pattern-matching. The benefit of this approach is that can be computed in a single pass over the program, without the need of expensive computations, but requires manual annotations to deal with non-rectangular or more complicated loops. Besides pattern-matching, there is no systematic way to identify those constructions, and therefore either non-safe or greatly over-estimated results (obtained for example by assuming rectangular loops in all cases) are produced as a result. These annotations include the structure (nesting) of loops, function calls and some annotations of variable assignments and reads. Regarding the limitations of the hardware model, caches are not modeled and the methodology is limited to processors without timing anomalies.

4.3 Limitations of the state-of-the-art techniques

Having analyzed the main techniques for verifying functional and non-functional properties, we can see that there are several reasons why the previously described methods are of limited use nowadays:

Addressing the effect of the compiler Some of the proposed methods work with the AST representation of the program, or a representation in between the AST and the final executable [Gus00], [EESG03], [GE05], [HSR98], [HAM99]. This is commonly accepted when analyzing functional properties because the compiler is supposed to keep the functionality of the code unaltered. However, compilers may perform several transformations in the code that drastically change the non-functional properties. This complicates these analyses when applied to verifying non-functional properties; if the analysis tool does not take into consideration the effect of the compiler, the results of the analysis (i.e. the worst-case execution time) will be greatly oversimplified or incorrect. On the other hand, the usage of advanced techniques by modern compilers makes the matching of high and low-level constructs difficult or impossible.

Addressing tightness The proposed techniques provide a very coarse over-approximation of the non-functional properties. The main cause of this over-approximation is the lack of analysis of 'infeasible traces'. To exemplify this, Figure 4.4 presents a slightly modified version of the previous example to introduce a more challenging program: In the example of Figure 4.4, a compositional approach of obtaining the worst-case execution time would consider that the number of iterations of the loop at line 4 is at most 200, but would overlook the fact that any trace that exhibits this behaviour is not feasible because it would have taken the other branch of the 'if' statement in line 3.

Considering the hardware elements In the same way than a trace must have a way to collect information about the context in which every operation executes that help us to decide if such a trace is feasible or not in the program, the context also plays a role in the

```
int main( argc, argv*[] ){
  int a = 0;
  if(argc < 100)
    for(n = 0; n < 200 && n < argc; n++)
      a++
  else
    return 0
}
```

FIGURE 4.4: Modified version to illustrate the importance of refinement

hardware model of the system. The execution time of an instruction may be different for different inputs, and more importantly, at different contexts of the internal elements of the processor. Multiplications, for example, are known to last for between 3 and 5 cycles in an ARM architecture depending on what are we multiplying. More important than that, fetching the instruction may take between 3 and 300 clock cycles depending on which level of the memory hierarchy the instruction is in the caches.

4.4 Symbolic representation of the program state

In this section, we focus on constructing an under-approximation of the behaviors of a program that is useful for reasoning about properties of the original code for a group of inputs rather than individual ones. The purpose of this representation is the same one as testing; try to find bugs in the implementation. As an example, we present the code of Figure 4.5 (top left). In this code, the program returns the minimum of two integers. Let's assume that our specification of the function states that the minimum of two numbers has to be smaller than or equal to the first and smaller than or equal to the second. We can test the correctness of the functionality by providing concrete input vectors to the function as in Figure 4.5 (down). The "testing" approach to ensure that the functionality is free of bugs, however, is not sound in this case because the function `min_wrong` does also pass the tests but is, however, incorrect regarding the specification. To prove that the program behaves according to the specification, we can use a model-checking approach

```
int min(int x, int y){
  if( x < y ) return x;
  else        return y;
}
```

```
int min_wrong(int x, int y){
  if( x == 0 && y != 0 ) return x;
  if( x != 0 || ( x == 0 && y == 0 ) ){
  if(x < y) return x;
  else      return y;
  }
}
```

```
int main(){
  assert(min(1,2) <= 1 and min(1,2) <= 2);
  assert(min(5,2) <= 5 and min(5,2) <= 2);
  assert(min_wrong(1,2) <= 1 and min_wrong(1,2) <= 2);
  assert(min_wrong(5,2) <= 5 and min_wrong(5,2) <= 2);
}
```

FIGURE 4.5: Implementation and testing of function min.

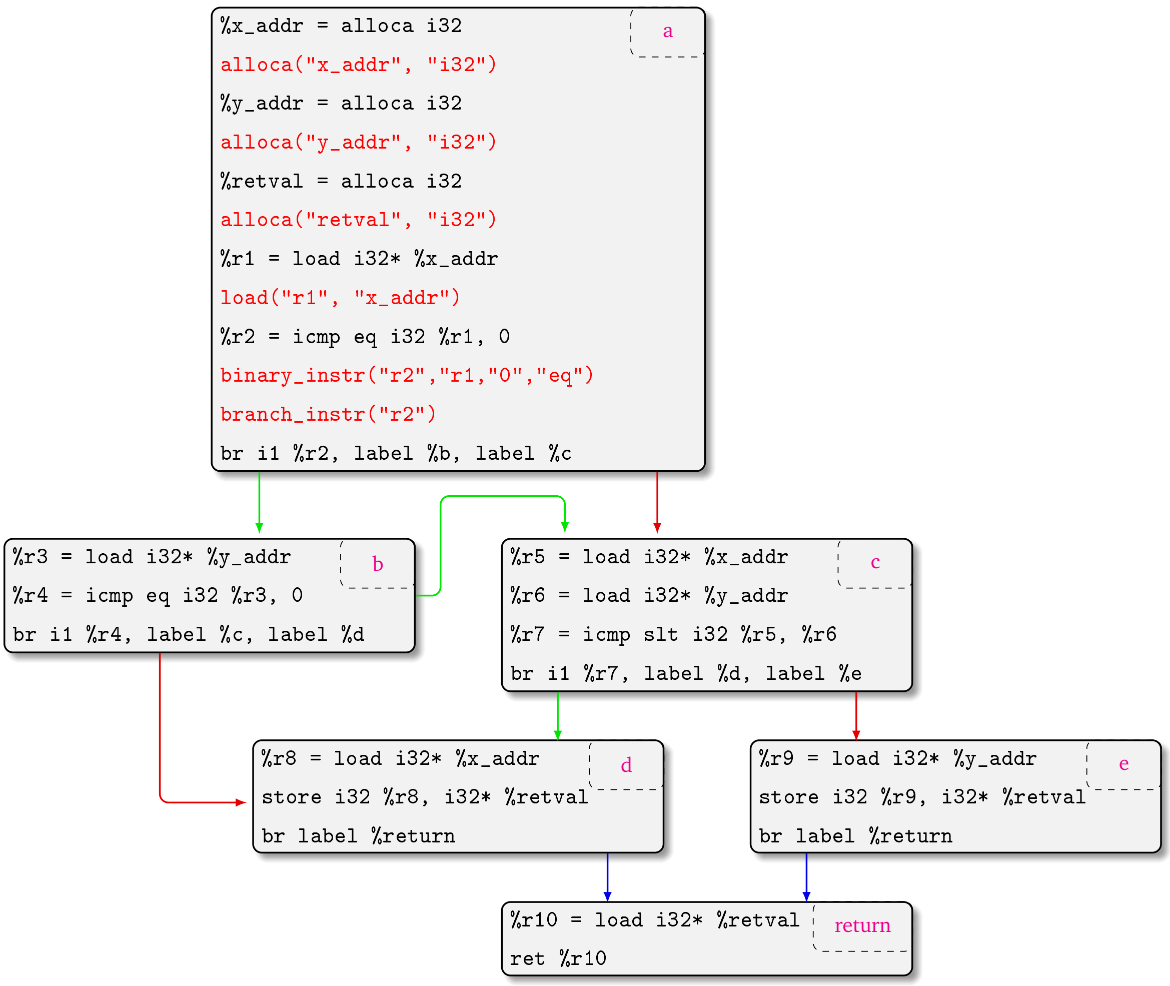
in which we build a model of the program, and we check some properties of it. In a first instance, the model can be a (partial) formula of the program behavior derived from concrete executions. This formula of the two implementations can be the following:

$$min(x,y) = \begin{cases} 1 & \text{if } x = 1 \land y = 2 \\ 4 & \text{if } x = 5 \land y = 4 \end{cases} \qquad min\_wrong(x,y) = \begin{cases} 1 & \text{if } x = 1 \land y = 2 \\ 4 & \text{if } x = 5 \land y = 4 \\ 0 & \text{if } x = 0 \land y = -2 \end{cases}$$

This formula is said to be an under-approximation of the semantics of the functions because it defines the output for fewer cases than the program. We can still use the formula to find bugs in our code, and in this case, it is sufficient to prove that at least one of the functions does not behave according to the specification.

$$min(x,y) = \begin{cases} x & \text{if } x < y \\ y & \text{if } y < x \end{cases} \tag{4.1}$$

In equation 4.1 we present another under-approximation of the semantics of the `min` function. We can intuitively see that this representation captures more behaviors of the code, and it is more likely to catch errors in the implementation. This is because it



```
%x_addr = alloca i32
alloca("x_addr", "i32")
%y_addr = alloca i32
alloca("y_addr", "i32")
%retval = alloca i32
alloca("retval", "i32")
%r1 = load i32* %x_addr
load("r1", "x_addr")
%r2 = icmp eq i32 %r1, 0
binary_instr("r2","r1","0","eq")
branch_instr("r2")
br i1 %r2, label %b, label %c
```

```
%r3 = load i32* %y_addr
%r4 = icmp eq i32 %r3, 0
br i1 %r4, label %c, label %d
```

```
%r5 = load i32* %x_addr
%r6 = load i32* %y_addr
%r7 = icmp slt i32 %r5, %r6
br i1 %r7, label %d, label %e
```

```
%r8 = load i32* %x_addr
store i32 %r8, i32* %retval
br label %return
```

```
%r9 = load i32* %y_addr
store i32 %r9, i32* %retval
br label %return
```

```
%r10 = load i32* %retval
ret %r10
```

Figure 4.6: Annotated Control-Flow-Graph of the function min_wrong. Note that we only show the instrumentation of the first basic block for readability reasons.

groups different executions that behave in a similar way. We will focus now on describing how we can use the annotation introduced in Chapter 2 to create a tool that can extract under-approximation formulas similar to 4.1 from C code. In section 4.5 we combine this under-approximation of the program with an over-approximation. As we will see, while under-approximations of a program are useful for searching for bugs in the code, over-approximations are useful for proving the absence of them.

In the following, we consider a C function as its Control-Flow-Graph. An example of an (instrumented) LLVM control-flow-graph for the `min_wrong` function is presented in Figure 4.6. In this graph, we call a trace (or a walk) of length k an alternating sequence of nodes and edges $v_0, e_0, v_1, e_1 \ldots v_{k-1}, e_{k-1}$ representing an execution of the program. As we have distinct labels in all the edges that depart each node, we will express traces as alternating sequences of nodes and labels in the following. We will cluster different inputs to the function based on the trace the program follows when executed with that input; as in similar approaches such as [KT14], [GKS05], [SMA05]. Here, the semantic coverage criteria are increased by iteratively expanding a set of feasible traces. We generate feasible traces by unfolding the Control-Flow-Graph up to a certain depth. Note that –as we are interested in generating an under-approximation of the function– we want to consider only feasible traces (i.e. we do not want to include the trace (a, t, b, t, c, f, e, t) because that trace can never occur in the code as it would imply that $x = 0$, $y = 0$ and $x < y$). We call a trace feasible if there exists an input that causes the program to follow that trace. To include only feasible traces, we need to provide a way to detect if a trace is feasible or not. We do this using the instrumentation that we describe in Chapter 2 to construct a logical formula that encodes the feasibility of a trace. In this case, the *metadata* is a map that links every variable with a Satisfiability-Modulo-Theory (SMT) formula that represents the relation between the inputs of the program/function and that variable. SMT is a standardized language to encode mathematical and logical formulas; using this format we can interact with a so-called SMT-Solver, that will help us later to decide the feasibility of the trace. The *semantics* of every instruction describes how to propagate this information when executing a trace. Obtaining feasible traces of this graph can be implemented with standard graph traversal algorithms in which we iteratively select a trace in the frontier, execute it and expand it if the expansion is feasible. We exemplify this expansion with the trace (a,t,b,t). This trace can (in principle) be expanded by the traces (a,t,b,t,c,t) and (a,t,b,t,c,f). Before expanding the traces, however, we need to check if the expanded traces would be feasible. We analyze trace (a,t,b,t,c,t) first. This trace will be feasible if there exists an input that makes true all the decisions that result in a “true” edge and false all the ones that result in a “false” edge. In the case of LLVM, the decisions are taken based on the content of a register. To obtain the logical statements that make those decisions true or false in relation to the inputs of the program we execute the trace and update two structures that will be used afterward to derive a formulation of the under-approximated formula:

TABLE 4.1: Execution of the trace (a,t,b,t,c,t,d)

Executed instrumentation	metadata
`alloca("x_addr", "i32")`	
`alloca("y_addr", "i32")`	
`alloca("retval", "i32")`	
`load("r1", "x_addr")`	$\sigma := \{r1 \mapsto x\}$
`binary_instr("r2","r1","0","eq")`	$\sigma := \{r1 \mapsto x, r2 \mapsto (x == 0)\}$
`branch_instr("r2")`	$\sigma := \{r1 \mapsto x, r2 \mapsto (x == 0)\}$ $\phi := (x == 0)$
`load_instr("r3","y_addr")`	$\sigma := \{r1 \mapsto x, r2 \mapsto (x == 0), r3 \mapsto y\}$ $\phi := (x == 0)$
`binary_instr("r4","r3","0","eq")`	$\sigma := \{r1 \mapsto x, r2 \mapsto (x == 0), r3 \mapsto y, r4 \mapsto (y == 0)\}$ $\phi := (x == 0)$
`branch_instr("r4")`	$\sigma := \{r1 \mapsto x, r2 \mapsto (x == 0), r3 \mapsto y, r4 \mapsto (y == 0)\}$ $\phi := (x == 0) \land (y == 0)$
`load_instr("r5","x_addr")`	$\sigma := \{\{r1, r5\} \mapsto x, r2 \mapsto (x == 0), r3 \mapsto y, r4 \mapsto (y == 0)\}$ $\phi := (x == 0) \land (y == 0)$
`load_instr("r6","y_addr")`	$\sigma := \{\{r1, r5\} \mapsto x, r2 \mapsto (x == 0), \{r3, r6\} \mapsto y, r4 \mapsto (y == 0)\}$ $\phi := (x == 0) \land (y == 0)$
`binary_instr("r7","r5","r6","slt")`	$\sigma := \{\{r1, r5\} \mapsto x, r2 \mapsto (x == 0), \{r3, r6\} \mapsto y, r4 \mapsto (y == 0), r7 \mapsto (x < y)\}$ $\phi := (x == 0) \land (y == 0)$
`branch_instr("r7")`	$\sigma := \{\{r1, r5\} \mapsto x, r2 \mapsto (x == 0), \{r3, r6\} \mapsto y, r4 \mapsto (y == 0), r7 \mapsto (x < y)\}$ $\phi := (x == 0) \land (y == 0) \land (x < y)$

- ϕ: A stack that stores the conditions that need to met in the trace to follow that particular path.
- σ: A mapping from registers to SMT expressions that stores the symbolic expression of every variable in the trace in relation to the inputs of the program.

We present in Table 4.1 the propagation of the two expressions for the different individual instrumentation function called during the execution of the trace. When we reach the end, we can use our encoding of the trace to decide if the trace is feasible or not by checking the satisfiability of the equation stored in ϕ.

TABLE 4.2: All feasible traces in graph 4.6

(a,t,b,f,d,t)	$\sigma := \{\{r1, r8, r10\} \mapsto x, r2 \mapsto (x == 0), r3 \mapsto y, r4 \mapsto (y == 0)\}$ $\phi := x == 0 \wedge y \neq 0$
(a,t,b,t,c,f,e,t)	$\sigma := \{\{r1, r5\} \mapsto x, r2 \mapsto (x == 0), \{r3, r6, r9, r10\} \mapsto y, r4 \mapsto y == 0, r7 \mapsto (x < y)\}$ $\phi := x == 0 \wedge y == 0 \wedge x \geq y$
(a,f,c,t,d,t)	$\sigma := \{\{r1, r5, r8, r10\} \mapsto x, r2 \mapsto (x == 0), r6 \mapsto y, r7 \mapsto (x < y)\}$ $\phi := x \neq 0 \wedge x < y$
(a,f,c,f,e,t)	$\sigma := \{\{r1, r5\} \mapsto x, r2 \mapsto (x == 0), \{r6, r9, r10\} \mapsto y, r7 \mapsto (x < y)\}$ $\phi := x \neq 0 \wedge y \geq y$

$$(a, t, b, t, c, t, d) \text{ is feasible } \iff (x == 0) \wedge (y == 0) \wedge (x < y) \text{ is SAT}$$

A state-of-the-art SMT-Solver can easily find that this expression is not satisfiable (there does not exist any input that makes the formula true), while there is one for the trace (a,t,b,t,c,f).

Iterating this process, we exhaust all the feasible traces ending up with the set of traces presented in Table 4.2.

Once this set is computed, we can construct the formula we are interested in by combining the information from several traces as presented in the SMT file shown in Figure 4.7.

Note that even when we do not demonstrate them in this example, our implementation supports more features of the C/C++ language, that we implement by providing semantics that describe how to update ϕ and σ when the instrumentation functions are called. The semantics of LLVM instructions are described in `http://llvm.org/docs/LangRef.html`. In particular, we support function calls, loops, recursion, pointer operations, structs, memory stack and heap operations, function pointers, ternary operator ... Also note that even when the formula of Figure 4.7 has been constructed manually for this example, the process is automated as described in Section 4.7 and [GdAPW+16].

```
(set-option :produce-models true)
(set-logic AUFNIRA)
(declare-fun x  () Int)
(declare-fun y  () Int)
(declare-fun r10  () Int)

(define-fun min_wrong () Bool
  (or
    (and (= r10 x) (= x 0) (not (= y 0)))
    (and (= r10 y) (= x 0) (= y 0) (>= x y))
    (and (= r10 x) (not (= x 0)) (< x y))
    (and (= r10 y) (not (= x 0)) (>= x y))
  )
)

(define-fun min () Bool
  (or
    (and (= r10 x) (< x y))
    (and (= r10 y) (not (< x y)))
  )
)

(define-fun min_spec () Bool
  (and (<= r10 x) (<= r10 y) )
)

(assert (and min (not min_spec)) )
(check-sat)

(assert (and min_wrong (not min_spec)) )
(check-sat)

(get-value (x y r10))
```

FIGURE 4.7: SMT description of functions min, min-wrong and min_specification

Verification of the specification

Once we have created a formula such as in Figure 4.7, we can check for properties of the code. In Figure 4.8, we use the formula derived from `min` to prove that the implementation of the function complies with the specification while `min_wrong` does not. Note that

```
> z3 min.smt2
unsat
      # min is correct according to the specification
sat
x = 0
y = -1
     # min_wrong is incorrect according to the specification
```

FIGURE 4.8: Model-checking min and min_wrong against the specification

when we detect a bug in the specification, we also provide a *model*; a witness input vector that (when given to the function), makes the program to exhibit the bug. This witness is useful for debugging. This symbolic representation of a function will be used in section 4.7 to check that the implementation of a system described in C/C++/SystemC behaves as expected by combining it with a representation of the specification in UML/OCL.

Even though in our example we have used Dynamic-Symbolic-Execution to construct a formula that we can use later on to check that an implementation behaves according to a specification, we can also check certain properties on-line while the code is being symbolically executed. This is useful for properties that are expressed as conditions over variables while the code is being run rather than relations between the input-output variables of a function. For those properties we use the instrumentation described in Chapter 2 to include "monitors" or "observers" in the code. Those are represented as Finite State Machines (FSM) that synchronize with the code in certain locations, and a violation of the specification is defined as an execution trace that makes the observer FSM to reach an error location [BCF+12b].

In Figure 4.9 (right), we present a monitor that checks that a pointer is not used after the memory that it points to is freed. The generation of monitors can be automated avoiding manual code annotations [BCF+12b].

This on-line checking for reachability properties will be used in combination with a Static-Analysis approach in next section to search for "bugs" during run-time in several examples taken from the SV-Comp benchmark suite [Bey15] as well as the test suite from the commercial tool Goanna.

Optimizations Note that there exist several optimizations to the naive process described before. In the following we describe some of these optimizations:

- A trace can be encoded with different levels of detail when checking for satisfiability. For detecting certain bugs, a representation as intervals can be appropriate. Some other representations we support are linear equations, where we represent each variable as a linear formula dependent on input variables, and polynomials, where we represent variables as a polynomial equation. Note that for encoding a strict under-approximation of the functionality we want to account for overflows and precisely capture sign-extension or bitwise operations, so we use a bit-level representation for every variable.

- When a condition depends on constants (or variables whose value directly depends on constants, there is no need to call an SMT solver to decide if both branches of the condition can be explored because only one of them can.

- We use parallelization to explore different traces. This means for every decision point, (i.e., every branching) we spawn separate processes for the true and false branches so we can parallelize the SMT solver computation as well as to follow different search strategies for separate paths independently.

4.5 Combination with Static Analysis

4.5.1 Introduction

Even when the concrete state of the program has been abstracted by a mathematical formula (enabling reasoning about different paths instead of actual values), the number of paths in a real-world program is still too big to be used in practical terms. Therefore, the program is still under-approximated, since not all the paths can necessarily be explored, neither in theory nor in practice. This is caused by (non-regular) loops and recursion in programs.

The problem of finding bugs or proving correctness in a program is therefore typically subjected to both dynamic testing as well as static program analysis. However, while testing is expensive to scale, static analysis is prone to false positives and/or false negatives. This means, there are spurious warnings not relating to actual defects and instances of software bugs that are part of checked defect classes but missed. Both false positives, as well as false negatives, are a serious concern.

To maximize the set of explored states in the under-approximation, different heuristics have been added to these frameworks [BS08][WMMR05]. These heuristics do not solve, however, the fundamental problem of a potentially exponentially growing number of execution paths. Hence, our goal in this section is to use static analysis for defining more constrained bug candidates and provide a guidance of the symbolic execution framework in the search strategy.

In this section, we present a combination of static program analysis and symbolic dynamic execution to minimize false positives and false negatives, while at the same time maintaining scalability. The core idea is to use static program analysis to broadly zoom in on a potential software defect and treat that as a bug candidate. Next, we make this bug candidate a precise target for symbolic analysis. This has a number of advantages:

1. Scalability is maintained by a broad static analysis pass that zooms in on bug candidates.

2. Fine grained symbolic execution has a concrete target as opposed to an unguided crawl, which allows for additional symbolic execution heuristics.

3. Performance can be tuned, by relaxing static analysis constraints and removing false negatives at the expenses of potential false positives, which in turn can potentially be ruled out by symbolic execution. Conversely, only true positives can be reported where symbolic execution provides a concrete witness execution.

4.5.2 Static Analysis as implemented in Goanna

Static Analysis comprises a number of techniques including data flow analysis, abstract interpretation, and software model checking [NNH99][DKW08]. The approach we use in this work is based on model checking and trace refinement as originated in the Goanna tool [FHS10]. The core ideas are based on the observation that data flow analysis problems can be expressed in modal μ-calculus [FHS10]. This has been developed further by Fehnker et al. in [FHJ07] and later expanded in [JHFK12]. The main idea is to abstractly represent a program (or a single function) by its control flow graph (CFG) annotated with labels representing propositions of interest. Example propositions are whether memory is allocated or freed in a particular location, whether a pointer variable is assigned null or whether it is dereferenced. In this way, the possibly infinite state space of a program is reduced to the finite set of locations and their propositions. The annotated CFG consisting of the transition system and the (atomic) propositions can then be transformed into the input language of a model checker. To illustrate the approach, we use a contrived function example shown in Figure 4.9. It works as follows: First, a pointer variable p is initialized, and we allocate memory accordingly. Then, in a loop, a second pointer variable q is assigned the address saved in p. After hundred-thousand assignments, p is freed, and the loop is left. To automatically check for a use-after-free, i.e., whether the memory allocated for p is still accessed after it is freed, we define atomic propositions for allocating memory (define p), freeing memory (free p) and accessing memory (assign p), and we label the CFG accordingly.

Trace Refinement Loop Model checking the above property for the model depicted in Figure 4.9 will find a violation and return a counter example. The following path denoted by the sequence of locations is such a counter example: $l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_3, l_4, l_5$.

However, if we match up the counter-example in the abstraction with the concrete program, we see that this path cannot possibly be executed, as the condition $i == 0$ cannot be true in the first loop iteration and, therefore, l_5 to l_6 cannot be taken. This means, the counter example is spurious and should be discarded. We might get a different

```
void example() {
l0: int i, *q;
l1: int* p = malloc(sizeof(int));
   for(l2: i = 100000; l3: i >= 0; l7: i--) {
     l4: q = p;
     l5: if(i == 0)
      l6: free(p);
   }
}
```

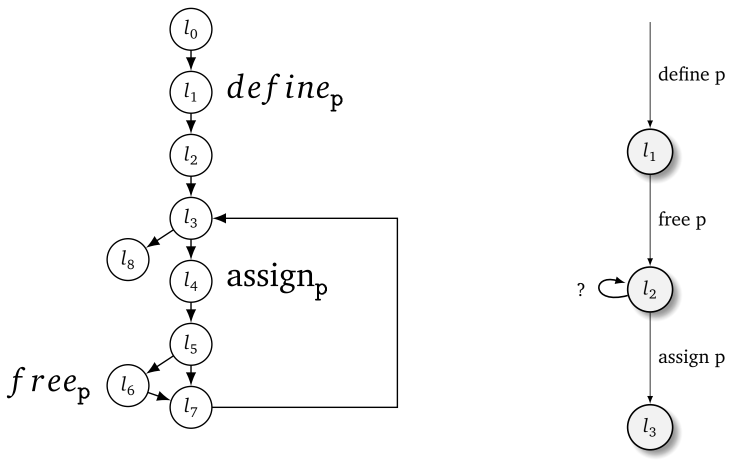


FIGURE 4.9: Original program, automatically annotated CFG and automaton that checks for "use-after-free" errors.

counterexample in the last loop iteration $\ldots, l_5, l_6, l_7, l_3, l_4, l_5$. But again, such a counter-example would be spurious, because once the condition $i == 0$ holds, the loop condition prevents any further iteration. To detect the validity of a counter-example we subjected the path to a fine-grained simulation using an SMT solver. In essence, we perform a backward simulation of the path computing the weakest precondition. If the precondition for the initial state of the path is unsatisfiable, the path is infeasible and the counter example spurious. We use an efficient SMT encoding and a refinement loop by creating observer automata to eliminate sets of infeasible traces successively. For the example in Figure 4.9 the approach is able to create two observer automata from minimal unsatisfiable cores of a single path leading to the elimination of all paths of the same nature, i.e., avoiding an unrolling of the loop. This approach is similar to interpolation-based solutions, and more details can be found in Junker et al. [JHFK12].

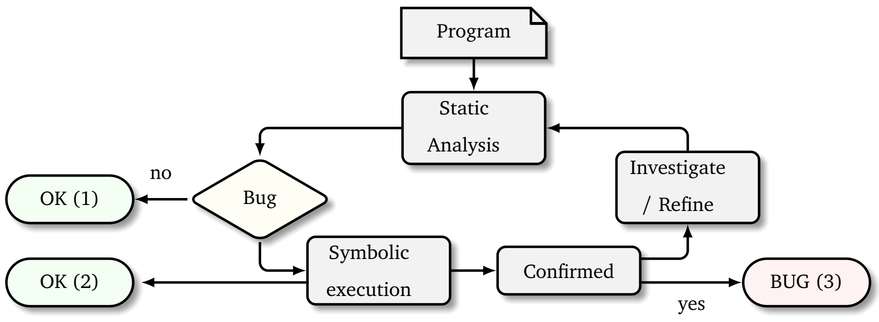


FIGURE 4.10: Architecture of the combined approach

False Positives and Tuning Even in this formal verification based framework of static program analysis, there are possibilities for false positives (wrongly warned bugs) and false negatives (missed bugs). This is caused by the abstraction and encoding into the model checker, which is necessarily sound. For instance, certain semantic constructs such as function pointers are typically not modeled, and their behavior is pessimistically assumed. Also, the false positive elimination itself might time-out, and a judgment call whether to report a potential bug or not is made. Industrial static analysis tools regularly carry out the aforementioned trade-offs. In this work, we scale back the potential false negatives and counter the increasing false positives with symbolic execution.

4.5.3 Our approach to combine Static Analysis and Dynamic Execution

We illustrate our approach in Figure 4.10: We start off with Static Analysis. If there is no vulnerability found the process stops. Otherwise, we submit the bug candidate to the symbolic execution engine. If the symbolic execution engine can confirm the issue, it generates a concrete trace and an input vector. Otherwise, the bug candidate is neither confirmed nor ruled out automatically and needs to be subjected to a manual investigation.

We have implemented the approach mentioned above making use of two approaches: 1) Static analysis based on model checking and SMT-based trace refinement as used in Goanna [BCF+12a], and 2) symbolic execution based on multi-theory SMT solving described in Section 4.1 [2].

At the current stage of development the integrated approach first runs a static analysis pass to determine bug candidates and for each potential bug, creates location information as well as a possible counter-example trace that is then passed on to the dynamic execution phase. The combination of the two approaches requires the following modifications to the static analyzer as well as the introduction of heuristics to guide the search in the symbolic program representation:

- The static analyzer needs to provide a trace indicating what constraints are violated that reach an error location.
- Once a candidate location to reach with Symbolic Execution has been found, we slice the program relative to that location to reduce the search space.
- The Symbolic execution needs to be augmented with heuristics to calculate a distance measure from the last visited node in the program to the reachability target. This distance is computed statically over the control flow graph. The symbolic execution engine can then use that distance to sort the set of candidate paths during the guided search. To do so, we use the standard A* graph traversal algorithm.
- Finally, we use time-outs on each branch of the symbolic execution if we are unable to reach a particular target.

4.5.4 Experiments

We firstly demonstrate our idea by some examples from our internal test suite. The first example program is shown in Figure 4.11 (left). An array of 10 elements ranging from 0 to 9 is initialized in a loop. However, in the last loop iteration, the counter is increased to one beyond the array size and the subsequent access to that array would result in an out of bounds violation. This error can be detected by the static analysis engine alone as the following command shows:

[2]We use Z3 as the underlying SMT solver.

```
1 int main(...) {
2    int a[10];
3    int i;
4    for(i=0; i<10; i++)
5       a[i] = 10;
6    a[i] = 0;
7    return 0;
8 }
```

```
1 void main (...) {
2    char password buffer[10] ;
3    int access = 0;
4    strcpy( passwordbuffer, argv[1] ) ;
5    if(!strcmp(passwordbuffer, "passwd"))
6       access = 1;
7    printf( "Access %d" , access );
8 }
```

FIGURE 4.11: Two examples of buffer overflow. First case can be detected with Static-Analysis and Symbolic-Execution. Second case can be detected with Symbolic Execution.

```
goanna overflow.c
GOANNA - analyzing file overflow.c
line 5: warning: Array 'a' subscript 10 is out of bounds [0,9].
```

For that example, the static analyzer can determine the array bounds as well as the number of loop iterations that are executed and, therefore, can derive the buffer overrun. However, in certain scenarios when the complexity of reasoning is increased by (for instance) copying memory around or reasoning about strings the analysis might lose precision. The Static Analysis phase does not warn in the latter cases. An example is shown in Figure 4.11(right). In the example, the buffer overflow introduces a real vulnerability, as it can be used to write in the memory occupied by the variable access, and grant the access to the application with an incorrect password. This occurs when the size of the string passed as the first parameter to the program is larger than 10 characters. In that case, the strcpy function writes in a space that was not allocated to store the variable password_buffer, but for access. Once access is overwritten with a different value than the initial 0, the access to the application is granted. To be able to detect these kinds of errors we tune the static analysis engine of the combined approach to always emit an error when it is not certain that a bug is absent. This means it will generate a vulnerability candidate for the example in Figure 4.11(right). Moreover, using our symbolic execution engine on the target location of the static analysis candidate we get a concrete confirmation of that bug.

4.5.5 SV-COMP Benchmark Results.

For the evaluation of our integrated solution, we use the well-known SV-COMP benchmark [Bey15], in particular, the loop category. SV-COMP is a set of competition benchmarks used in the automated verification community to highlight complex verification problems and to test the strength of individual tools. The loop category comprises of 117 files. We show the results of our integrated approach in Table 4.3. This table is broken down by the different analysis phases as well as the final verdict, where SA denotes static analysis, SE Symbolic Execution and C Combined. A tick means proven to be correct, a cross that a bug has been confirmed, and a warning triangle means for static analysis that it flags a potential issue and for symbolic execution that it times out. The files names shaded in gray are those containing a bug. We have broken the table in five groups, which are separated by double horizontal lines.

1. In the first set of examples, the static analysis engine can conclude that the program is correct. This is because our static analysis phase over-approximates the possible behavior and the program does not contain any approximation breaking constructs such as function pointers.

2. In the second group, the static analysis engine produces some potential bug candidates that are passed to the symbolic analysis pass. However, the symbolic analysis engine was able to cover all the possible branches in the program faithfully and conclude that all of them are bug-free.

3. In the third group, the full potential of the combined approach is shown. In these cases, Static Analysis concludes that there is a potential bug in the code and provides a set of candidate locations that exhibit the undesired behavior. This set of locations is used as target locations for the symbolic execution heuristics. In each case, we were able to find the bug and provide a witness that demonstrates this behavior.

4. In the next two groups, the relaxation of the rules in the static analysis tool makes the static analysis to produce error candidates in programs that however are correct

TABLE 4.3: Results of each engine and the integrated solution.
SA = static analysis, SE = symbolic execution, C = combined, gray = bug

Filename	SA	SE	C	Filename	SA	SE	C	Filename	SA	SE	C
nested6_true-u…	✓	⚠	✓	simple_false-u…	⚠	×	×	functions_true…	⚠	⚠	⚠
nested9_true-u…	✓	⚠	✓	terminator_01_…	⚠	×	×	simple_true-un…	⚠	⚠	⚠
heapsort_true-…	✓	⚠	✓	underapprox_fa…	⚠	×	×	simple_true-un…	⚠	⚠	⚠
apache-escape-…	✓	⚠	✓	sum01_bug02_su…	⚠	×	×	simple_true-un…	⚠	⚠	⚠
apache-get-tag…	✓	⚠	✓	while_infinite…	⚠	×	×	SpamAssassin-l…	⚠	⚠	⚠
count_by_k_tru…	✓	⚠	✓	for_bounded_lo…	⚠	×	×	sum03_true-unr…	⚠	⚠	⚠
diamond_true-u…	✓	⚠	✓	count_up_down_…	⚠	×	×	trex03_true-un…	⚠	⚠	⚠
gj2007_true-un…	✓	⚠	✓	sum01_bug02_fa…	⚠	×	×	count_up_down_…	⚠	⚠	⚠
gr2006_true-un…	✓	⚠	✓	sum01_false-un…	⚠	×	×	ddlm2013_true-…	⚠	⚠	⚠
seq_true-unrea…	✓	⚠	✓	sum04_false-un…	⚠	×	×	jm2006_true-un…	⚠	⚠	⚠
down_true-unre…	✓	⚠	✓	terminator_02_…	⚠	×	×	jm2006_variant…	⚠	⚠	⚠
phases_true-un…	✓	⚠	✓	trex02_false-u…	⚠	×	×	overflow_true-…	⚠	⚠	⚠
up_true-unreac…	✓	⚠	✓	sum03_false-un…	⚠	×	×	half_true-unre…	⚠	⚠	⚠
bhmr2007_true-…	✓	⚠	✓	trex03_false-u…	⚠	×	×	nest-if3_true-…	⚠	⚠	⚠
hhk2008_true-u…	✓	⚠	✓	terminator_03_…	⚠	×	×	MADWiFi-encode…	⚠	⚠	⚠
half_2_true-un…	✓	⚠	✓	trex01_false-u…	⚠	×	×	trex04_true-un…	⚠	⚠	⚠
string_concat-…	✓	⚠	✓	simple_false-u…	⚠	×	×	trex01_true-un…	⚠	⚠	⚠
eureka_01_true…	✓	✓	✓	functions_fals…	⚠	×	×	sum01_true-unr…	⚠	⚠	⚠
n.c40_true-unr…	✓	✓	✓	simple_false-u…	⚠	×	×	string_true-un…	⚠	⚠	⚠
lu.cmp_true-un…	⚠	✓	✓	overflow_false…	⚠	×	×	vogal_true-unr…	⚠	⚠	⚠
veris.c_sendma…	⚠	✓	✓	phases_false-u…	⚠	×	×	afnp2014_true-…	⚠	⚠	⚠
eureka_05_true…	⚠	✓	✓	eureka_01_fals…	⚠	×	×	array_true-unr…	⚠	⚠	⚠
cggmp2005_true…	⚠	✓	✓	id_trans_false…	⚠	×	×	array_true-unr…	⚠	⚠	⚠
diamond_true-u…	⚠	✓	✓	string_false-u…	⚠	×	×	array_true-unr…	⚠	⚠	⚠
underapprox_tr…	⚠	✓	✓	vogal_false-un…	⚠	×	×	array_true-unr…	⚠	⚠	⚠
large_const_tr…	⚠	✓	✓	NetBSD_loop_tr…	⚠	⚠	⚠	cggmp2005b_tru…	⚠	⚠	⚠
nec40_true-unr…	⚠	✓	✓	sendmail-close…	⚠	⚠	⚠	const_true-unr…	⚠	⚠	⚠
sum04_true-unr…	⚠	✓	✓	simple_true-un…	⚠	⚠	⚠	count_by_1_tru…	⚠	⚠	⚠
terminator_02_…	⚠	✓	✓	terminator_03_…	⚠	⚠	⚠	count_by_1_var…	⚠	⚠	⚠
array_false-un…	⚠	×	×	trex02_true-un…	⚠	⚠	⚠	count_by_2_tru…	⚠	⚠	⚠
array_false-un…	⚠	×	×	css2003_true-u…	⚠	⚠	⚠	count_by_nonde…	⚠	⚠	⚠
const_false-un…	⚠	×	×	n.c11_true-unr…	⚠	⚠	⚠	gauss_sum_true…	⚠	⚠	⚠
diamond_false-…	⚠	×	×	while_infinite…	⚠	⚠	⚠	gj2007b_true-u…	⚠	⚠	⚠
diamond_false-…	⚠	×	×	while_infinite…	⚠	⚠	⚠	gsv2008_true-u…	⚠	⚠	⚠
ludcmp_false-u…	⚠	×	×	while_infinite…	⚠	⚠	⚠	id_build_true-…	⚠	⚠	⚠
multivar_false…	⚠	×	×	cggmp2005_vari…	⚠	⚠	⚠	multivar_true-…	⚠	⚠	⚠
nec11_false-un…	⚠	×	×	for_infinite_l…	⚠	⚠	⚠	nested_true-un…	⚠	⚠	⚠
phases_false-u…	⚠	×	×	for_infinite_l…	⚠	⚠	⚠	nec20_false-un…	⚠	⚠	⚠
simple_false-u…	⚠	×	×	fragtest_simpl…	⚠	⚠	⚠	verisec_NetBSD…	⚠	⚠	⚠

under the fully-accurate semantics of the operations of the program. The set of feasible paths, however, is too big to be fully exercised by symbolic execution, so under the requirements of a sound analysis, the algorithm has to output an inconclusive output. We observe, however, that the fact of having a concrete goal to reach helps a lot in the symbolic execution framework so most of these cases (41 over 43) are actually correct. Considering the two remaining cases as correct would break the soundness of the approach but would leave us with an error rate of only 2/117.

In summary, the combined approach has a detection rate (number of detected errors over files with an error) of 98%. The true negative rate of the combined approach (number of files "proven" as correct when they are correct) is 35%, which is approximately 50% above the rate obtained by only using a static analysis approach.

4.5.6 Conclusions

In this section we have first introduced a symbolic representation of the program state based the annotations introduced in Chapter 2. The proposed symbolic representation of the program captures the semantics of the software operations very accurately and therefore do not introduce false-positives that would appear if the program was over-approximated. The precision in the analysis, however imposes several limitations, especially in terms of scalability. To counter this problems, the symbolic representation have first been combined with an over-approximation of the program based in the techniques described by the tool Goanna [JHFK12]. Also, in cases in which a formal model of the system is available in the form of an UML/OCL specification, we have presented a combined technique that divides the verification approach in two steps, and also helps in reducing the complexity of the analysis.

Regarding the integration of Symbolic Execution and Static Analysis, it is worth noting some observations: Firstly, our solution is quite capable of detecting bugs. From a set of complicated bugs taken from the standard SVComp testbench, all bugs have been identified by our approach and all apart from two have been confirmed with symbolic execution inputs and traces. Secondly, the combined approach gives a slightly better coverage to demonstrate the absence of bugs compared to single static analysis approach.

In our case, if we declared a program bug free when both phases cannot come to a combined negative conclusion, we would correctly identify all benchmark cases apart from two, keeping the overall error rate at around 1%. This is better than the rate exhibited by more mature state-of-the-art tools in this set of programs.

4.6 Verification of non-functional properties

4.6.1 Introduction

In this section, we describe the problem of obtaining worst-case execution time of a program when it runs over a hardware platform. We emphasize that i) we are interested in obtaining an over-approximation of the worst-case execution time, and therefore we can not base our solution in a simulation approach and ii) as well as a model of the software, we need a formal model of the hardware.

4.6.2 The WCET problem

Given a binary program P, some input data d and the hardware H, the *execution time* of P for the input d on H, denoted $\mathsf{Xtime}(P, d, H)$, is measured as the number of processor cycles between the beginning and end of P's computation for d (we assume P always terminates.)

The *worst-case execution time (WCET)* of program P on hardware H, denoted $\mathsf{WCET}(P, H)$, is the supremum of the $\mathsf{Xtime}(P, d, H)$ for d ranging over the input data domain $\mathscr{D}$:

$$\mathsf{WCET}(P, H) = \sup_{d \in \mathscr{D}} \mathsf{Xtime}(P, d, H). \tag{4.2}$$

The WCET problem asks the following:

“Given P and H, compute $\mathsf{WCET}(P, H)$”.

Previous chapters should serve as an introduction of the difficulty of computing this value with a certifiable proof. During last years there has been a tremendous progress in architectural changes to improve the platform performance. Even though this has improved the average execution times, it has also complicated the formal analysis of those architectures, making the problem of computing the WCET an undecidable problem. Because of this complexity, the WCET problem is usually re-stated as:

“Given P and H, compute a *tight upper bound* of $\mathsf{WCET}(P, H)$”.

Tightness can be measured (see [CB13a]) by comparing actual WCET to the ones computed using a particular method. In the sequel, we use $\mathsf{WCET}(P, H)$ to denote an upper bound of the WCET for a given program.

The main concepts that are important when analyzing the time distribution of programs over an arbitrary input are depicted in Figure 4.12. In the previous chapter, we have focused on simulation. Simulation enables us to obtain feasible executions of a program automatically and therefore, can be used to derive “wcet_under”. Heuristics can guide the Symbolic Execution to select long traces and therefore approximate the exact value of the WCET as “wcet_under”. In this section, we focus on obtaining an upper bound of the WCET (denoted “wcet_over” in Figure 4.12). Note that the estimation of the WCET is expected to be tight (precise), or the implementation of the system would incur in costs derived from a very pessimistic approximation. The combination of software loops with over-approximate models of the hardware can lead to very pessimistic results. A clear example of this is produced in instruction caches. It is common in a processor with caches that the first iteration of each loop to be around 100 times slower than subsequent ones due to the filling of the cache with the instructions of the body. A naive approach that does not consider the interactions between hardware and software and assumes all the iterations to take the same amount of time would lead to wcet estimations to be $100 \times$ number_of_iterations bigger than the real value, which can be disastrous for large loops.

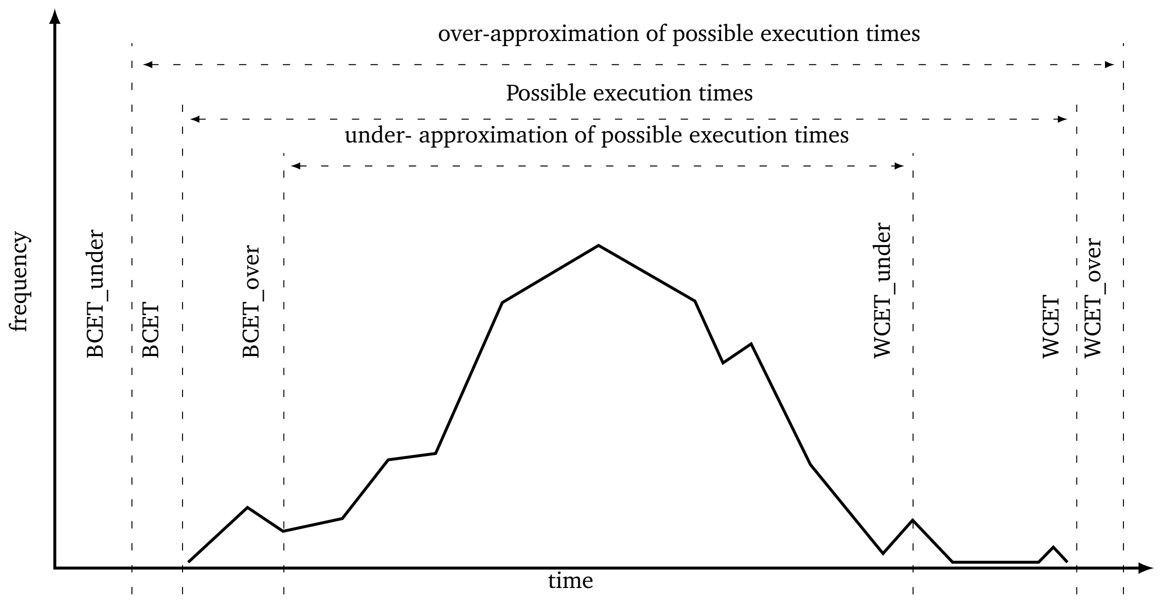

over-approximation of possible execution times
Possible execution times
under- approximation of possible execution times
frequency
BCET_under
BCET
BCET_over
WCET_under
WCET
WCET_over
time

FIGURE 4.12: Main concepts in timing analysis

4.6.3 Proposed solution

We observe that (i) most of the over-approximation of the worst-case execution time comes from the fact that infeasible traces are not refined and (ii) the analysis of hardware models is complicated because hardware platforms are composed of components that execute concurrently and synchronize on timing constraints. Therefore, we propose two contributions in this section to tackle the problems of WCET:

- The use of formal models like timed automata and state-of-the-art real-time model-checkers like Uppaal [LPY97] to model the hardware elements of the platform.
- The use of trace refinement both in the software and hardware domain to remove traces that are infeasible because of infeasible program conditions or because of infeasible conditions in the hardware model.

We have implemented a prototype of these ideas that we describe in this section. In this prototype we use the techniques described in Chapter 2 to extract the control-flow-graph of a binary program and embed it in a model of the hardware platform. The control-flow-graph of the program represents an over-approximation of all the feasible execution traces of the software (a trace in the CFG leading to the end location may be infeasible). We represent it as an (untimed) automata and synchronize it with a model of the hardware, that is described as a network of timed automata. Computing the WCET is then reduced to computing the longest path (time-wise) in a network of timed automata (a set of automata that synchronize on common labels). This prototype has been used to experiment about trace refinement in the hardware domain and is starting to be used to refine the control-flow-graph representation of the software. This technique has demonstrated the following advantages compared to the previous methods described in the state of the art:

Addressing the effect of the compiler The approach does not attempt to match loops or other elements of the high-level language with constructs of the final executable. The input of the technique is the raw binary code, and it does not require annotations in the program to indicate loops bounds or loops invariants like other state-of-the-art techniques [Lun02][FH04][bou]. Also, because the approach operates directly with the binary program, the proposed method works for a variety of programming languages.

Addressing the complexity of the hardware platform Network of Timed Automata have precise semantics described in [AD94], [BLL+96], [LPY97]. Therefore our models are formal behavioral models for the hardware. Despite being formal, the models of the hardware are described as the composition (synchronization) of very simple components that can be validated separately.

This generality in the hardware models enables us to model features that cannot be modeled with other approaches based on abstract interpretation and Integer Linear Programming (ILP) such as intervals in the execution time of instructions or changes of the speed of the CPU during program execution.

The hardware models do not pre-suppose an initial state. As we have seen, making the assumption of a concrete initial state is not sound when providing an upper bound on the execution time. Our methodology is resilient to timing anomalies [LS99],[RWT06],[CHO12]. This is a challenging problem faced by other methods based on abstract interpretation.

Compact representation of the program state space The approach automatically abstracts all the computations that are not relevant to the execution time. Only the values that have an influence in the execution time are considered. For example, for the computation of a bit-wise operation between two registers, the fact that this instruction is in one or another level in the memory hierarchy might make a difference in the execution time, but not the precise value that the instruction computes (unless it is used later as the upper limit in a loop, for example). This compact representation of the program state space leads to less memory utilization in the computation of the WCET (computation of the WCET is often limited by memory constraints).

4.6.4 Hardware model

Introduction

This section describes our models for the hardware and software elements of the model. The formal definition of the hardware model has been summarized from references [Cas11],[CB13b],[BC11] and a paper I co-authored with Franck Cassez [CdAM15].

The hardware is composed of an instruction cache and the main memory to store the program, and a four-stage pipeline (CPU) to execute the instructions.

The main memory component is a table of words of a given width (32-bit or 64-bit words). $\mathscr{M}$ is the (finite) set of main memory cells and we denote $\mathscr{D}$ the memory domain (e.g. 32-bit or 64-bit words). A memory state is thus a map from $\mathscr{M}$ to $\mathscr{D}$.

This model is derived from a generic Von Neumann architecture; it considers an arithmetic-logic unit to perform operations between registers; a control unit containing the program counter and processor registers; a memory to store the data and the instructions of the program and input-output mechanisms. Most modern computing platforms are based in this architecture; from which they mostly diverge in the addition of caches to accelerate memory access. Therefore, the consideration of a different architecture than the one presented in this section implies rethinking each hardware model in isolation, but the overall methodology and results are still applicable. A careful consideration of caches can significantly speed-up the verification, so we devote Section 4.6.7 to explain such considerations.

A *state* of the hardware is fully determined by the contents of the registers, the content of the memory and the content of the pipelines and caches. The hardware has a designated register, the *program counter* that points to the next instruction to process. An example of such an architecture, the AMR920T, is given in Figure 4.13. The orange blocks are the blocks we need to model to compute the execution time of program runs.

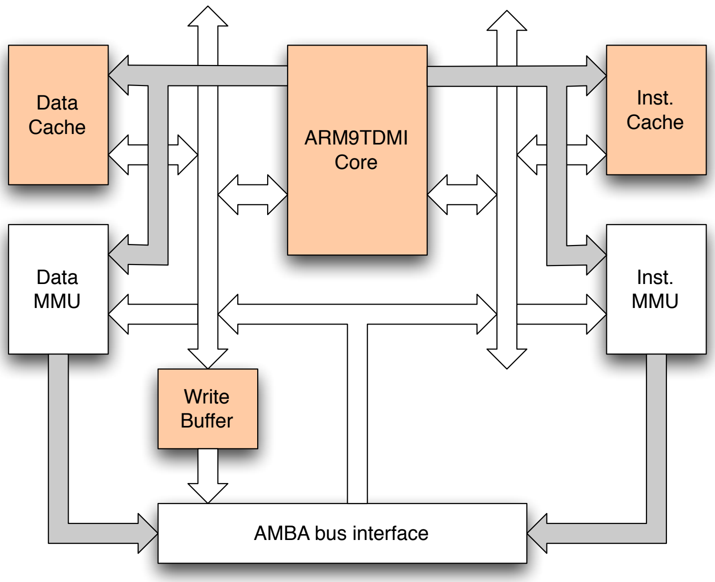


FIGURE 4.13: Simplified ARM920T architecture

Given a program P, we let $\mathscr{L}_H(P)$ be the set of *valid* executions of P on H. To define this set we need to take into account the semantics of each instruction, and the values of the registers of H and the memory state.

A program run (or execution) is completely defined by a finite sequence of (positive) integers given by the successive values of the program counter (there is no input data). The execution time of a run is defined by the time it takes for the hardware to execute the sequence of instructions in the run. To execute a run, the code of instructions has to be fed to the CPU (pipeline) to be executed. To execute the instruction at $PC = i$ the following steps occur:

1. the CPU requests the code of the instruction i.

2. If the code of i is in the cache, this is a cache Hit, and the code of i is transferred from the cache to the CPU, otherwise, a cache Miss occurs. The code of i is fetched from main memory, stored in the cache and then transferred to the CPU.

3. the CPU executes the code of i and is ready to process the next instruction.

Execution time of a run. The execution time of a run depends on the following factors:

- the time it takes for the instructions to flow into the pipeline stages. This is usually non-trivial because the stages run in parallel and because of pipeline stalls.
- the time it takes to fetch instructions and data from the caches and main memory. These memory transactions are usually performed in different pipeline stages and can be concurrent (e.g., an instruction in the *fetch* stage can be fetched from the instruction cache while another instruction in the *memory* stage performs some transactions with the data cache.)

In order to determine how long it takes for a run ρ to execute on the hardware H, it is sufficient to know:

- the processing time of each instruction in the different pipeline stages,
- the registers read from/written to by each instruction (to determine pipeline stalls),
- the status of the memory transactions for the instructions in ρ: cache *hits* and *misses*.

Given a run ρ, we can build an *annotated run* $\tilde{\rho}$ that contains the information required to fully determine the execution time of ρ on H. This extended run may capture the processing time of the instruction in each pipeline stage, the registers read from/written and the cache hits and misses.

As a sequence contains enough information to compute the execution time of a program run $\rho \in \mathscr{L}_H(P)$ we can define an abstract model of the hardware as a timed automaton *transducer*, *Aut*(H), that maps each $\tilde{\rho}$ to a positive natural number $Aut(H)(\rho)$, which is the execution time of ρ on H.

Hence the WCET of a program P on the hardware H is defined by:

$$\mathsf{WCET}(P, H) = \max_{\rho \in \mathcal{L}^a_H(P)} Aut(H)(\rho). \tag{4.3}$$

We can over-approximate the set of runs of a program by assuming that each time we make a choice, the outcome is either true or false and both cases should be taken into account to compute the WCET. As the set of runs constructed this way over-approximates the set of program runs, we ensure that the value of the WCET we compute (equation (4.3)) is an upper bound of the actual WCET (this assumes that the hardware model $Aut(H)$ correctly models the timing behavior of the hardware).

Pipeline model

The ARM920T uses a 5-stage *execution pipeline* (Figure 4.14), the purpose of which is to execute concurrently the different stages (Fetch, Decode, Execute, Memory, Writeback) needed to perform an instruction. An instruction is fetched in F, decoding and operand register accesses occur in D, execution in E and load/store instructions do their memory accesses in M. The results are written back to registers in W. The (normal) flow of instructions in the pipeline is shown in Figure 4.14. This optimal flow may be slowed down when pipeline *stalls* occur (e.g., due to register dependencies).

A formal model of the pipeline of the ARM920T can be specified by a network of 5 timed automata (see Figure 4.15) each of them modeling a single stage of the execution pipeline.

Each stage automaton has a unique identifier `me` (an integer). The values of this identifier for the templates (F, D, E, M, W) are respectively (0,1,2,3,4). This encodes the fact that the stages F, D, E, M, W are ordered. For instance, the F-Stage template automaton is idle until the Program pushes a node via the `pushTo[0]?` transition. It updates the local *state* of this stage 0 (`locState[0]=node`) where `node` is a (meta) variable used to retrieve the value sent by the Program Automaton that issues the `pushTo[0]!` command. The F stage template automaton then synchronises with the instruction cache (see Figure 4.15) to simulate th time it takes to fetch the instruction from the instruction cache.

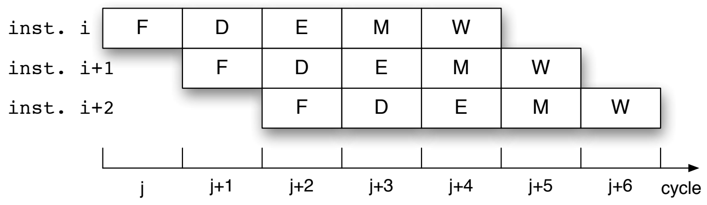


FIGURE 4.14: Pipeline of the ARM920T: Instruction is fetched in F. Instruction decode and operand register accesses are done in D. Execution is done in E. Load/store instructions do their memory accesses in M. Results are written back to registers in W.

Cache model

The model for the instruction cache is given by the timed automaton `FullCache` (Figure 4.16). When a cache read request is received, (`ICacheReadStart ?`) location `Check` is reached. Depending on whether the set of the instruction (`cacheLine`) is in the cache (which is when find(`cacheLine`) returns a positive value) two different behaviors can be observed: a `Hit` occurs, and the delay is set to `HitTime` (e.g., 2 cycles). Otherwise, a `Miss` occurs and the delay is set to `MissTime` (e.g., 20 cycles). In both cases, the cache is updated access(`cacheLine`). The UPPAAL specification of the functions find() and access() are given in [CdAM15].

4.6.5 Tool chain

We have implemented a prototype of the integration of UPPAAL, the control-flow-graph of a program as described in Chapter 2 and the formal models of the hardware as described in 4.6.4 (WUppaal).

The toolchain, visualized in Figure 4.17, is composed of five components:

- a `pre-analysis` module for constructing an annotated program that can be used to generate the program traces with the techniques described in Chapter 2,
- `qemu` [Bel05] to emulate the chosen hardware,
- `gdb` [SPS+02] for inspecting `qemu`,
- `libgdb2uppaal` to implement the execution of traces in uppaal,

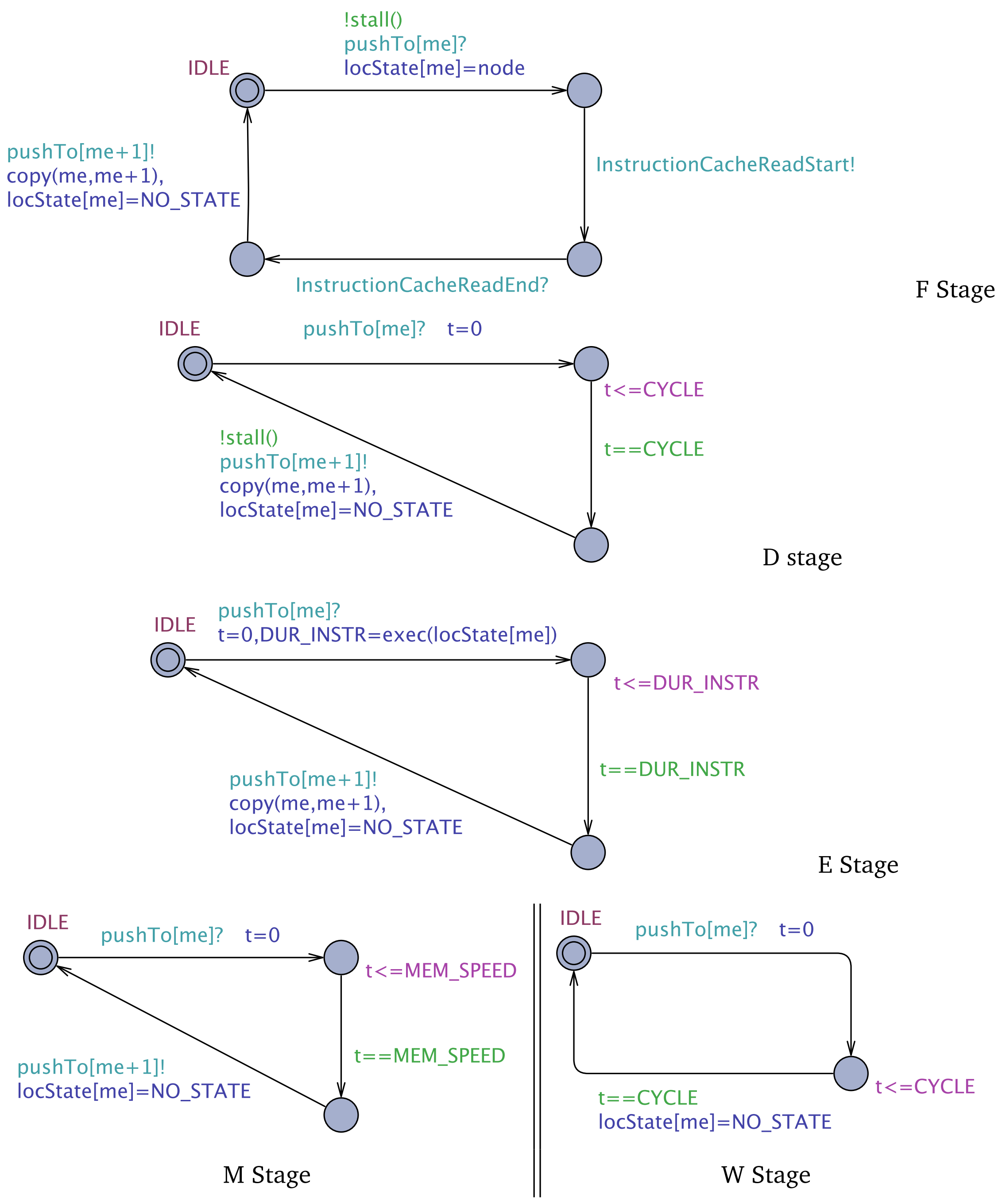


FIGURE 4.15: Timed Automata for F, D, E, M and W Stages (pipeline ARM920T).

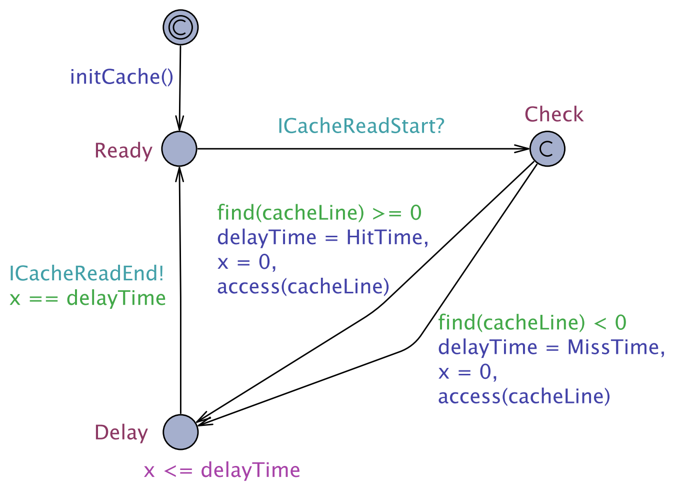


FIGURE 4.16: Instruction Cache automaton `FullCache`

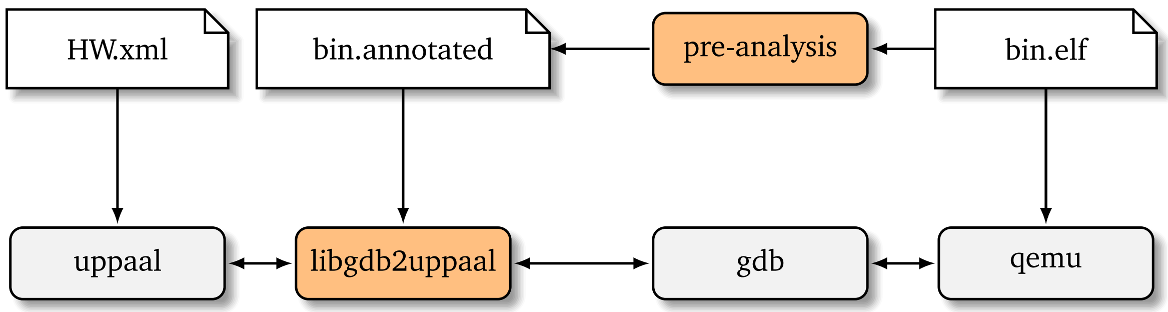


FIGURE 4.17: The tool chain of WUppaal. Orange blocks are the modules we implemented. Other blocks are existing modules.

- a Timed Automaton model of the hardware `HW.xml` for the pipelines, main memory and instruction cache.

- UPPAAL for computing the worst-case execution time given a sequence of nodes.

Computing the WCET for a given binary program `bin.elf` using our framework is a two-stage process. In the first stage, we compute an annotated program (e.g., a CFG and the set of variables needed to generate the annotated language) by using `pre-analysis` as described in Chapter 2). In the second stage we use UPPAAL to drive a search through the state space, interfacing (by proxy of gdb and `libgdb2uppaal`) with the emulator of the hardware.

TABLE 4.4: The experimental results, time is given in seconds and includes startup overhead from initializing gdb and qemu. The *loc* measure is the number of lines of assembly, $|\mathsf{Tree}^a_H(P)|$ measures the size of the representation of possible runs of the program.

Program	**Loc**	$|\mathsf{Tree}^a_H(P)|$	**Time**	**WCET**
duff	145	1750	4.51	61215
fibcall	48	553	2.91	19320
insertsort	84	7	2.09	210
janne_complex	67	360	3.21	12565
lcdnum	100	250	2.52	8715

4.6.6 Results

We have experimented our technique using some of the standard benchmarks [GBEL10] from Mälardalen University, for computing WCET [3].As we can see in Table 4.4, we are achieving a reasonable computation time (less than 5 seconds for all experiments), demonstrating the feasibility of our approach.

4.6.7 Hardware refinement

Model-checking is very sensitive to the number of states, and this can hinder the analysis of large systems. This is even more important when model-checking timed automata: the state space is composed of a discrete part and clock constraints, so it is even more important to try and reduce the state space of the model while preserving enough details to represent the property to be checked faithfully. To avoid the state-explosion problem resulting from a naive synchronization between the program states and the hardware states, this section proposes the usage of trace refinement in the hardware state. Experimental results show that the most demanding elements regarding state space are the models of the cache.

[3]Note that, even when the programs from the Mälardalen benchmark may seem small, they are specifically designed to highlight and test the complexity of binary-program verification. In this sense, the source code is usually modified to force the compiler to introduce non-standard constructions. This is why the complexity of providing a verified bound for the WCET in these programs is not directly dependent on the number of lines-of-code

Using an accurate state model for the caches forces to store an explicit representation to obtain the hits and misses. Next section presents a technique based on trace refinement to reduce the space needed to represent instruction caches.

Reducing the state space, minimal cache models.

As described previously, the execution time of a sequence of instructions depends on the sequence of cache hits or misses, and the time it takes for the CPU to execute each instruction. Assume that the time to execute each instruction is 0 and only the cache hits and misses impact the execution time. With a model that explicitly represents the cache, we can precisely track the hits and misses. For instance for the sequence $\sigma_1 = 1, 2, 3, 1$ and a cache of size 3, the successive states of the cache are $, 1, 1.2, 1.2.3$ and we obtain the following sequence of pairs (instruction, Hit/Miss) : (1, M).(2, M).(3.M).(1, H). This sequence fully determines the execution time. It turns out that the sequence of instructions $1, 4, 5, 1$ would produce the same execution time (if 4 and 5 have the same execution time as 2 and 3). But if we generate the sequences of pairs using the full state of the cache, the same sequence will be analyzed twice.

An example.

Consider the program of Figure 4.18. We fix the maximum number of iterations to $M = 5$. We can compute the WCET of the program with an explicit cache of size 2 and with an ideal model of the cache which is given by Figure 4.19. With a cache of size 2 all the runs in the program generates sequences of `Hit` and `Miss` of the form $(M.H.M.M)$. Hence the automaton of Figure 4.19 is a good cache model for this program for any number of switches N. Table 4.5 gives the number of states explored to compute the WCET with the explicit cache and the small cache of Figure 4.19. As can be seen, many states of the cache are equivalent and this is captured by the small model whereas the explicit model generates all the configurations. The explicit state cache model can be used with any program to analyze but may inflate the state space to be explored for computing the WCET. In the next section we describe how to compute small cache models for a given

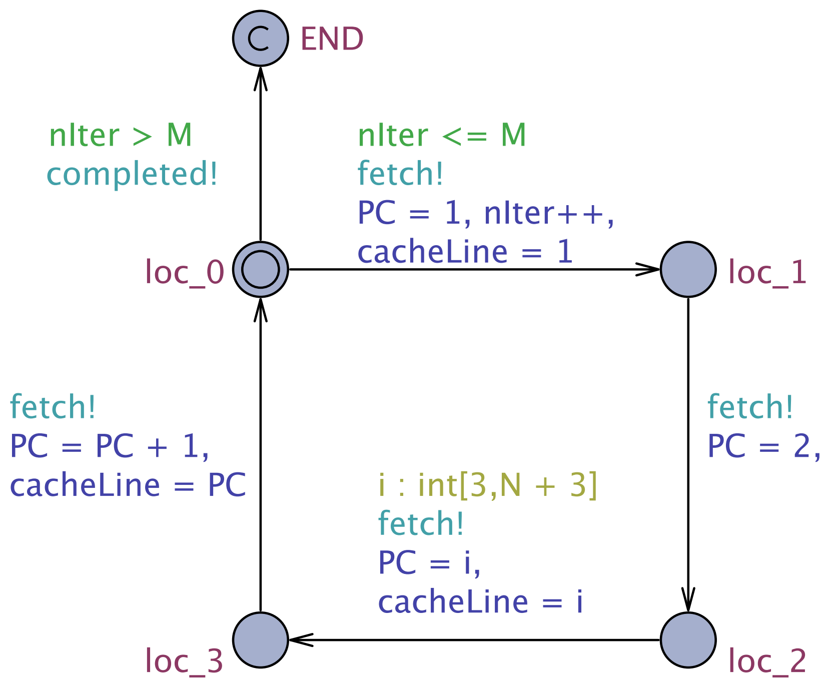


FIGURE 4.18: Simple program represented as an automaton in UPPAAL

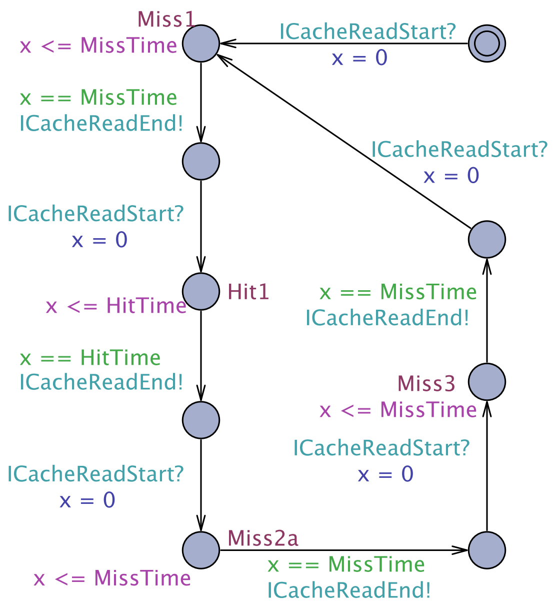


FIGURE 4.19: A small cache model

TABLE 4.5: States explored for computing the WCET.

N	States Explored		WCET
	Explicit Model	Small Model	
1	549	147	396
2	1055	196	396
3	1626	245	396
4	2267	294	396
5	2953	343	396
6	3699	392	396
7	4505	441	396
8	5371	490	396
9	6297	539	396
10	7283	588	396

program.

Computing abstract cache models

As demonstrated in the previous section, using an explicit cache model can be detrimental to the model-checking approach as many equivalent (time-wise) cache states may be explored. The objective of this section is to indicate how we can compute small cache models that are suitable abstractions for a given program. To do this, we use an abstraction refinement technique introduced in [HHP09b][HHP13]. The trace abstraction refinement technique was originally developed to analyze imperative programs. We adapt here it to compute the WCET of a program. Our method works as follows:

1. start with the most abstract model of the cache: every cache access can be either a hit or a miss;
2. compute the WCET T with the current model of the cache.
3. get a witness trace t i.e., a sequence of pairs (instructions, Hit/Miss) that yields this execution time T.
4. check that t is feasible in a concrete model of the cache. t contains the sequence of instructions and using the specification of the cache we can compute the corresponding sequence of hit and miss. This reference sequence is compared against

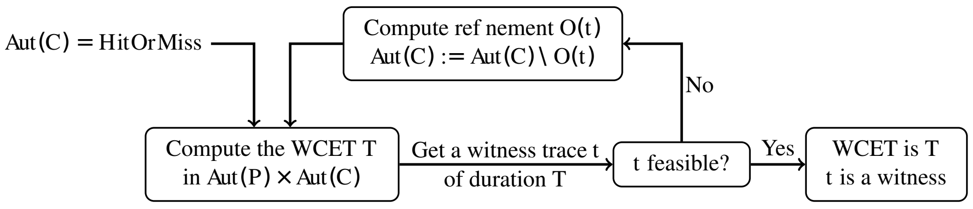


FIGURE 4.20: Trace Abstraction Refinement Algorithm for computing WCET

the sequence of hit and miss in t.

5. if the reference sequence and t match, t is feasible, the WCET is T and t is a witness trace.

6. otherwise t is infeasible in a concrete model of the cache, we refine the abstract cache. The refined abstract cache model should not allow t. We iterate this process and re-start at step 2.

Step 6 above is the central point of the trace abstraction refinement method: from one infeasible trace, we can compute a set of traces that are infeasible for the same reason. From t we can obtain a regular language of traces that are infeasible and represent it by a finite automaton $O(t)$. This crucial step is based on the computation of interpolants. We have defined a logic for caches and the corresponding notion of interpolants that enables us to compute $O(t)$ for each infeasible trace t.

An algorithm to compute the WCET of a program P using a trace abstraction refinement loop is given in Figure 4.20. The initial model for the cache is the timed automaton `HitOrMiss` which allows the cache to generate a Hit or a Miss for each memory access. When an infeasible trace is discovered, we refine the cache model by removing a set of infeasible traces given by $O(t)$.

It is important to notice that the sequence of worst-case execution times obtained by this refinement is monotonic, so the process can be stopped at any time and still produce a safe bound of the worst-case execution time.

4.6.8 Conclusions

In this section, we have presented a method to compute the WCET of binary programs based on timed automata and real-time model-checking with Uppaal. The method we designed is generic and can accommodate arbitrary hardware. The proposed tool chain allows us to achieve a modular approach to WCET-computation, reducing the overhead needed to support new binaries and new architectures.

Our technique does not rely on the computation of loop bounds or the assumption that the hardware is free of timing anomalies: this is one the strengths of the model-checking method. Another strength is that it generates a witness program trace that produces the WCET. Other attractive features of this approach include its generality: we do not need to assume that the initial state of the caches is known.

Our technique is also general enough to be paired with program refinement and hardware refinement techniques. This enables us to define an iterative method to compute better and better over-approximations of the WCET and ensure that one witness trace exists.

We experiment the idea of hardware refinement and provide an iterative algorithm to approximate the WCET monotonically; we can start with a very simple model of the caches where every transaction is either a Hit or a Miss. Once we compute a WCET with Uppaal, we can check whether the witness trace is feasible in the program and the caches. If the cache behavior that is in the witness is spurious (infeasible), we can refine it as well. This enables us to get some control on the accuracy of the computation via model-checking.

4.7 Combination with UML/OCL Specification

4.7.1 Introduction

The use of formal models to describe early versions of the structure and the behavior of a system has become common practice in the industry. UML and OCL are the de-facto specification languages for these tasks. They allow for capturing system properties and

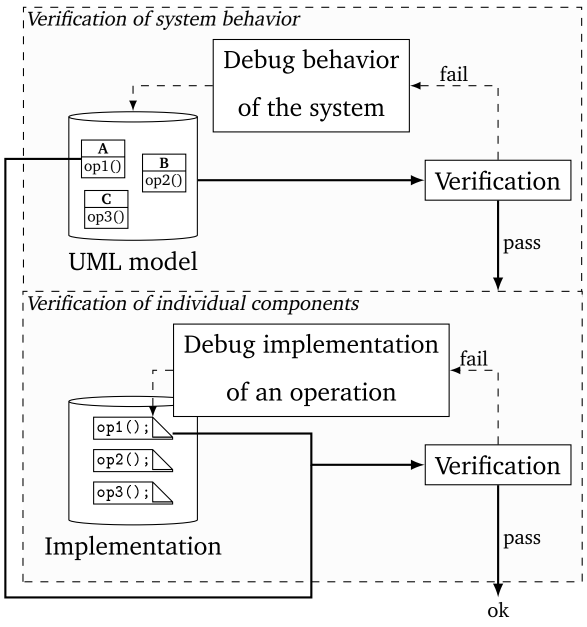

Verification of system behavior
Debug behavior of the system
fail
A
op1()
B
op2()
C
op3()
Verification
pass
UML model
Verification of individual components
Debug implementation of an operation
fail
op1();
op2();
op3();
Verification
pass
Implementation
ok

FIGURE 4.21: Envisioned verification flow

module behavior in an abstract but still formal fashion. At the same time, this enables designers to detect errors or inconsistencies in the initial phases of the design process-even if the implementation has not already started. Corresponding tools for verification of formal models got established in the recent past. However, verification results are usually not re-used in later design steps anymore. In fact, similar verification tasks are applied again, e. g. after the implementation has been completed. This is a waste of computational and human effort. In this section, we address this problem by proposing a method which checks a given implementation of a system against its corresponding formal method. This allows for transferring verification results already obtained from the formal model to the implementation and, eventually, motivates a new design process which addresses verification across abstraction levels. To this end, we propose a design and verification flow as depicted in Figure 4.21 in which the following two stages are conducted.

In the first stage, the formal specification (provided by a stakeholder or designer in abstract modeling languages such as UML, SysML, MARTE, etc.) is verified by a UML/OCL verifier ("modelchecker") [SWD11b] . This includes consistency checks such as checking whether it is possible to instantiate the desired system considering all constraints and requirements, but also first behavioral checks such as checking whether it is feasible to reach a prohibited state. For this purpose, a variety of (automatic) verification approaches is already available, e. g. [CCR08] or [SWK+10] for consistency checking or [SWD11c], [CCR09] for the verification of behavior. These methods allow for the detection of design flaws already in very early design steps - even in the absence of a precise implementation. That is, if the verification of a formal specification fails, design bugs can be determined and fixed without considering implementation details. Moreover, also the interaction of various components of a complex design can be verified by these approaches. Consequently, further checks can often focus on the implementation of sole components only. We conduct this step in the second stage. More precisely, this stage does not consider the entire system and the interactions of their components anymore but checks whether the implementations of its individual components are functionally correct with respect to the formal specification. For this purpose, the implementation of the respectively considered component (e. g. for operation op1() in the sketch of Figure 4.21; provided in C, C++, SystemC, or other means) is compared against its corresponding formal specification (i. e. the formal constraints for class A and operation op1() provided in UML, SysML, MARTE, or other means).

Following such a flow yields the following advantages:

- An “as early as possible” verification scheme is employed, i. e. errors are detected as soon as possible. The more errors detected in the UML/OCL level, the fewer errors that are propagated to lower levels.
- Debugging loops are kept small. In fact, if an error has been detected, identifying its source remains in one abstraction level (either the formal specification or the implementation is debugged). Loops over several abstraction levels may only occur due to design exploration or unsatisfied non-functional requirements.
- The efficiency is significantly increased due to the clear separation of concerns. The interaction of the respective components is mainly verified in Step 1 using the abstract descriptions at the formal specification level. In contrast, the (much more precise) implementations are solely considered in Step 2. This is a clear

improvement compared to existing approaches such as [BUZC11a], [CDE08a], [CKL04a], [dAE15a], [GLD10], [LGHD13] where the implementation as a whole has to be considered, which frequently leads to complexity problems.

However, the realization of such a design and verification flow requires further CAD solutions. While for the first stage, a variety of (automatic) verification approaches is already available, no formal solution for the second step exists yet.

This section is structured as follows; in “UML/OCL model-checking in modelchecker”, the formal definitions of the UML and OCL elements that are used in this section will be introduced as they are used by the tool [SWD11b]. In “Our combined approach” we present our combination of the symbolic representation of the formal specification with the symbolic representation of the implementation. This integration starts by defining a combined encoding of both the OCL/UML specification as well as the implementation of each action. The satisfiability of that formula implies that the design is not implemented according to the specification. This encoding of the design includes constraints derived from the UML/OCL domain as well as the implementation domain. In “Symbolic Representation of the Specification”, we summarize the former. Our approach constraints an abstract symbolic state of the specification by adding several restrictions related to UML/OCL elements. In “Combination with Symbolic Execution”, we discuss how to integrate this restricted symbolic state with the encoding of an implementation as described in Section 4.4. In “Case Studies” we present several examples that prove the validity and usability of our approach. We end the section with “Conclusions”.

This section is a summary of the paper I co-authored with Nils Przigoda, Robert Wille, Rolf Drechsler, and Pablo Sanchez [GdAPW$^+$16], from which I have taken the definitions of the sections 4.7.2 and 4.7.2.

4.7.2 UML/OCL in “modelchecker”

This introduction covers the basics of UML models and textual constraints provided by OCL. At the same time, we are introducing the notation as applied in the remainder of the section. This section represents a summary of the more extensive formulation described in [GdAPW+16]. The interested reader is referred to that publication for more details and examples.

UML Modeling

UML was introduced as a language for specifying systems in a unified way and has quickly become the de-facto standard for describing the behavior of complex systems. In the following, the resulting models as well as their instances are considered as follows:

Definition 1 *A* model $m = (\mathcal{C}, \mathcal{R})$ *is a tuple of* classes $\mathcal{C}$ *and* relations $\mathcal{R}$ *(also known as* associations*). A* class $c \in \mathcal{C}$ *with* $c = (\mathcal{A}, O, \mathcal{I})$ *is a 3-tuple composed of attributes* $\mathcal{A}$*, operations* O*, and invariants* $\mathcal{I}$*.An* operation $o \in O$ *is a 5-tuple* $o = (P, r, \lhd, \rhd, \mathcal{F})$ *composed of a (possibly empty) set of parameters* P*, a return value* r*, preconditions* $\lhd$*, postconditions* $\rhd$*, and frame conditions* $\mathcal{F}$*.*[4]

All invariants $i \in \mathcal{I}$ *for all classes* $c \in \mathcal{C}$ *as well as pre-and postconditions* $\lhd$*,* $\rhd$ *of all operation are* OCL constraint expressions *[Obj14]. The frame conditions* $\mathcal{F}$ *are a list of those model elements (with a specific navigation scope), which are allowed to change their value by executing the corresponding operation.*

A relation $r = (c_1, c_2, (l_1, u_1), (l_2, u_2)) \in \mathcal{R}$ *consists of two classes* c_1 *and* c_2 *in* $\mathcal{C}$ *as well as two tuples representing the multiplicities between the classes. The tuples* (l_i, u_i) *with* $i \in \{1, 2\}$ *represent the* lower and the upper bound*, i. e. each instance of* c_1 *(*c_2*) shall be connected with at least* l_1 *(*l_2*) and at most* u_1 *(*u_2*) instances of* c_2 *(*c_1*). Such pairs of bounds are called* UML constraints *in the following. The lower bound is an arbitrary natural number, while the upper bound is either a positive natural number or infinity. These bounds are used in a first step in our procedure to refine the set of feasible states.*

Object Constraint Language

The *Object Constraint Language* (OCL, [Obj14]) is a declarative language which allows for the formulation of *constraint expressions*. Constraint expressions are used together

[4] In order to specify which elements of the state are not allowed to change when an operation is called, additional framing constraints are assumed in verification [NHGW15a],[NHGW15b].

with the model in order to add further restrictions that cannot be expressed by the given model notation itself. The OCL mainly consists of

- navigation expressions to access elements in the model,
- logic expressions (i.e., , con-/disjunction, negation, etc.),
- arithmetic expressions (i.e., , addition, subtraction, multiplication, division, etc.), and
- collection expressions (i.e., , intersection, union, element containment, etc.).

A comprehensive overview on all OCL expressions as well as keywords is given in [Obj14]. Precise semantic definitions can also be obtained from [Obj14]. For a model m, a system state σ, and an arbitrary OCL expression e, we say that σ is *valid*, if all invariants (as well as UML constraints) $i \in \mathscr{I}$ holds in σ.

In order to execute a valid operation call, the preconditions must be satisfied in the calling system state, while the postcondition restricts the resulting successor state σ'.

4.7.3 Our combined approach

In this section, we discuss in more detail the existing gap which prevents the application of a design and verification flow as depicted above by means of Figure 4.21. Afterward, we propose the general idea of a solution which closes this gap. Based on that, the remainder of this work provides a detailed description of the implementation and application of the suggested solution.

As already mentioned in Section 4.7, many approaches for the verification of the code of embedded systems have been proposed in the past [BUZC11b], [CDE08b], [CKL04b], [dAE15a], [GLD10], [LGHD13]. They are helpful in identifying the existence of flaws in late stages of the design process and may prevent the commercialization of a system based on a flawed implementation. They, however, have the drawback that a verification of the *entire* system behavior is attempted and this usually leads to overly under-approximated analysis of the entire system. In particular, for detailed implementations, verifying the entire system behavior is a computationally complex task. Moreover, it is also a redundant task, since, e. g., the interaction of the system's components could have already been verified by means of the UML/OCL model. But existing methods do not allow for transferring such verification results from the formal specification level to the implementation level – a clear disadvantage.

In this section, we are proposing such a solution for this. More precisely, for a given formal model m (provided in UML/OCL and composed of several operations O) and an implementation impl of m (provided in C, C++ or SystemC), an approach is presented which automatically checks whether the implementation impl_o of an operation $o \in O$ has been realized as specified in m. Using this solution, the design and verification flow as shown in Figure 4.21 becomes applicable.

In order to address the problem above, the following check needs to be conducted: For all possible system states σ that lead to the succeeding state σ' when executing the implementation impl_o of o, eventually a succeeding state σ' must result which satisfy the invariants $\mathscr{I}$, satisfy the postconditions $\rhd_o$, and satisfy the frame conditions $\mathscr{F}_o$ between itself and the calling state σ.

More formally, an implementation impl_o of an operation $o \in O$ has been realized as specified in m if

$$\forall \sigma, \sigma' \in \Sigma : \left(\left(\bigwedge_{i \in \mathscr{I}} [\![i]\!]^{\sigma} \right) \wedge \left([\![\lhd_o]\!]^{\sigma} \right) \wedge \left(\sigma' = \text{impl}_o(\sigma) \right) \right) \Rightarrow \left([\![\rhd_o]\!]^{\sigma,\sigma'} \wedge \bigwedge_{i \in \mathscr{I}} [\![i]\!]^{\sigma'} \wedge \mathscr{F}_o(\sigma, \sigma') \right) \tag{4.4}$$

holds, whereby

- impl_o is a function which takes the calling state σ and maps it to the successor state by applying all atomic instructions of the implementation in the given order. The derivation of this formula from the source-code has been described in Section 4.1.

- $\mathscr{F}_o$ is a function which evaluates to `false` iff the frame conditions of operation o are violated between the states σ and σ'.

Since SMT solvers are highly optimized for determining a (satisfying) assignment for a given formula or to prove that no such assignments exist, the negation of equation (4.4) is applied instead, namely:

$$\begin{aligned}\exists\, \sigma, \sigma' \in \Sigma : \Bigg(\Bigg(\bigwedge_{i \in \mathscr{I}} [\![i]\!]^{\sigma} \Bigg) \wedge \Big([\![\lhd_o]\!]^{\sigma} \Big) \wedge \Big(\sigma' = \mathrm{impl}_o(\sigma) \Big) \Bigg) \\ \wedge \Bigg(\neg [\![\rhd_o]\!]^{\sigma, \sigma'} \vee \bigvee_{i \in \mathscr{I}} \neg [\![i]\!]^{\sigma'} \vee \neg \mathscr{F}_o(\sigma, \sigma') \Bigg) \end{aligned} \tag{4.5}$$

Equation (4.5) eventually states a classical satisfiability problem which represents the considered problem. What remains open is how to precisely create an instance representing equation (4.5) for an (arbitrarily) given model m.

Our approach for creating such a formula is divided into two steps;

- First, a symbolic pair of states σ, σ' is created by applying constraints to refine the UML/OCL specification iteratively. In a first instance, we consider a symbolic formulation that contains *all* the states of the system, we then apply constraints to restrict the symbolic states to those in which the specification is consistent regarding the links that connect classes in the UML specification. For those (symbolic) states that comply with the restrictions as specified by their link relations, we add corresponding bit vector constraints from the OCL specification enforcing that only valid assignments are employed, i. e. assignments satisfying all invariants as well as the pre- and postconditions associated with the respectively considered operations.

- Once the pair of states are obtained, the implementation as derived in Section 4.4 is added to further constraint the state space.

- If equation (4.5) is then satisfiable, an assignment (i. e. two system states) exists which violate the constraint stated in equation (4.4). These system states are counterexamples showing that, although a valid operation call has been conducted in σ (all invariants and preconditions are satisfied), the implementation impl_o yielded a succeeding system state σ' which violated at least one invariant, postcondition, or frame condition. This shows that the implementation has not been realized as specified in m. Vice versa, if the SMT solver can prove that no such assignment exists, it can be concluded that no such counterexample exists and, hence, the implementation has been realized as specified in m (equation (4.4) has been proven).

Symbolic representation of the formal specification

This section summarizes the first step (i. e. the refinement of the symbolic state from the UML/OCL specification). For further details refer to [GdAPW+16].

- In a first instance, we use the link associations to restrict states in which UML constraints hold. To this end, we use the bounds of the relations and introduce symbolic bit-vector variables representing arbitrary instantiations and assignments of links. We then propose an encoding in which we consider instances to be linked if the corresponding bits of the variables are set to 1. This is necessary because we do not know anything about a satisfying assignment of Equation 4.5 and, thus, have to ensure that combinations of links can be represented with the variables. Note that the bit vector constraints derived from the link associations must also satisfy the corresponding OCL constraints, as well as bounds of the relation. Further constraints to include those relations are also added.

- Additionally, all invariants must hold for a system state. For this, all invariants are translated to a corresponding bit vector constraint. Fortunately, most of the OCL operations can be translated directly, e. g. logic expressions such as conjunction, disjunction, and negation have counterparts called `and`, `or`, and `not`. Arithmetic expressions can be directly represented with bit vector constraints such as `bvuadd` for an unsigned addition and `bvadd` for a signed addition. A detailed list of how to map OCL constraints to bit vector constraints is provided in [SWD11a].

- Similar to the representation of invariants, similar bit vector constraints are additionally created for the preconditions of o (employed in the representation of σ) and for the postconditions of o (employed in the representation of σ'). Passing the resulting formulation to an SMT solver, only system states which satisfy the respective constraints would be derived.

Based on these concepts, a symbolic representation for two system states, namely σ and σ' is created.

In order to create a precise instance representing Equation 4.5 for a given model m, we propose a formulation which, first, symbolically represents *all* possible system states. To this end, a symbolic representation based on SMT and its theory of Quantifier-free Bit Vectors (QF_BV) is applied in which assignments, e. g., for attributes or links are represented in terms of bit vector variables. Afterwards, we add corresponding bit vector constraints to these variables which enforce that only valid assignments are employed, i. e., assignments satisfying all invariants as well as the pre- and post-conditions associated to the respectively considered operation o. Using such a formulation, a solving engine, i. e., an SMT solver, is able to determine the existence of corresponding assignments and, hence, the existence of corresponding system states σ, σ' satisfying the OCL constraints stated in Equation 4.5. However, before such a symbolic formulation can be created, several bounds have to be defined. In fact, UML/OCL models inherently allow, e. g., for an infinite number of object instantiations. Also data-types of the attributes are usually not bounded, i. e., an integer is usually not restricted in its bit width. Such assumptions obviously would make a consideration of Equation 4.5 undecidable (since, in the worst case, an infinite number of cases has to be checked in order to prove the non-existence of the system states σ, σ'). Consequently, we need to assume bounds for both object instantiations as well as data-types. This is not a serious restriction of the proposed approach, since such bounds also must have been defined when creating the implementation of the model. In fact, correspondingly applied bounds can directly be derived from the implementation or the platform in many cases.

Using these bounds, bit vector variables representing arbitrary instantiations and assignments of links and attributes, respectively, can be created. To this end, formulations as already proposed in the past can be utilized: For an instance $\upsilon \in \Upsilon$ of a class $c \in \mathscr{C}$, an attribute $a \in \mathscr{A}$ (of the corresponding class c) is represented by a variable ${}_{\sigma}\vec{\alpha}_{\upsilon}^{a} \in \mathbb{B}^{k}$, where σ is the corresponding system state of Υ. By this, ${}_{\sigma}\vec{\alpha}_{\upsilon}^{a}$ can represent 2^{k} different values. Thus, for Boolean attributes k is set to 1 and for integers to 8. For a relation $r = (c_1, c_2, *, *)$, variables ${}_{\sigma}\vec{\lambda}_{v}^{r} \in \mathbb{B}^k$ are introduced for all $v \in \Upsilon(c_1) \cup \Upsilon(c_2)$. For links, the value of k depends not only on the classes of the relation but also on the problem bounds, but it does *not* depend on bounds of the relation. As an instance of c_1 can be linked to every instance of c_2, we set $k = |\Upsilon(c_2)|$ for all variables corresponding to $v \in \Upsilon(c_1)$; and vice versa. Then, a link between two instances is represented by a so-called hot encoding, i. e., instances are linked if the corresponding bits of the variables are set to `1`. This is necessary because we do not know anything about a satisfying assignment of Equation 4.5 and, thus, have to ensure that combinations of links can be represented with the variables.

Passing this formulation, e. g., to an SMT solver, arbitrary assignments to all bit vector variables would be obtained. They, in turn, would represent arbitrary system states. However, we are not interested in arbitrary system states, but states which satisfy the invariants, pre-, and postconditions. Hence, in another step, bit vector constraints are added which restrict the possible assignments of the bit vector variables to those which satisfy the corresponding OCL constraints.

Again, formulations from previous work can be utilized for this purpose. Consider, e. g., the formulation of links. If two instances are linked in a system state, the bits of their respective ${}_{\sigma}\vec{\lambda}_{v}^{r}$-variables must both be set to `1`. If they are not connected, both bits must be set to `0`. Furthermore, the bounds of an association must be represented by constraints: For a relation $r = (c_1, c_2, (l_1, u_1), (l_2, u_2))$ all instances of c_1 must be connected to at least l_1 and at most u_1 instances of c_2. This can be done adding the following inequality:

$$l_1 \leq \sum_{i=0}^{|\Upsilon(c_2)|-1} \lambda[i] \leq u_1,$$

where $\lambda[i]$ is the ith bit of a variable λ which stands here as representative for the ${}_{\sigma}\vec{\lambda}_{v}^{r}$-variables.

Additionally, all invariants must hold for a system state. For this, all invariants are translated to a corresponding bit vector constraint. Fortunately, most of the OCL operations can be translated directly, e. g., logic expressions such as conjunction, disjunction, and negation have counterparts called `and`, `or`, and `not`. Arithmetic expressions can directly be represented with bit vector constraints such as `bvuadd` for an unsigned addition and bvadd for a signed addition. A detailed list of how to map OCL constraints to bit vector constraints is provided in [SWD11a].

Based on these concepts, a symbolic representation for two system states, namely σ and σ' is created. Similar to the representation of invariants, corresponding bit vector constraints are additionally created for the preconditions of o (employed in the representation of σ) and for the postconditions of o (employed in the representation of σ'). Passing the resulting formulation to an SMT solver, only system states which satisfy the respective constraints would be derived – a significant part of the formulation of Equation 4.5 is covered.

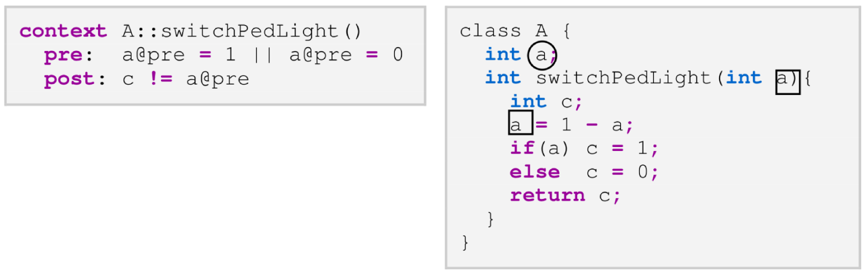
Combination with Symbolic Execution

The symbolic representation of the implementation as described in Section 4.4 is the last piece missing in equation 4.5. This symbolic formulation represents all possible executions of the given implementation. As a final step, this formulation has to be integrated into equation 4.5. To this end, all bit vectors representing variables of the implementation have to be mapped to the bit vectors representing the corresponding attributes of the model. While this can, in principle, easily be conducted by mapping the respective identifiers, the different states a variable may assume in the implementation has to be considered. An example illustrates the issue.

Example 1 *Consider an operation specified in terms of UML/OCL as shown in Figure 4.22 (left) and its implementation as shown in Figure 4.22 (right). In UML/OCL, attributes only argue over two consecutive system states σ and σ', which, e. g., for the attribute a, is clearly distinguished by a and a@pre for σ' and σ, respectively.*

However, the variable a may assume more states in the implementation (in fact, after each assignment, a new value may be assumed). Moreover, these stages are not explicitly denoted but have to be derived from the order of statements in the implementation.

To solve this problem, we add two pseudo functions `prestate(a, ‘‘a’’)` and `poststate(a, ‘‘a’’)` at the beginning and the end of the implementation of the operation, respectively. These functions have no effect on the implementation, however, they can be identified



```
context A::switchPedLight()
  pre:  a@pre = 1 || a@pre = 0
  post: c != a@pre
```

(a) Modeled Operation

```
class A {
  int a;
  int switchPedLight(int a){
    int c;
    a = 1 - a;
    if(a) c = 1;
    else  c = 0;
    return c;
  }
}
```

(b) Implemented Operation

FIGURE 4.22: Correlation of variables

when processing the LLVM syntax tree. The first parameter helps to identify the corresponding LLVM variable and the second parameter assigns the respective identifier to the variable.

4.7.4 Case Studies

The proposed methodology has been implemented using existing frameworks from [SWD11b] as well as the tool FOREST proposed in [dAE15a]. More precisely, the translations of the formal specification as well as the implementation as described in Section 4.7.3 and Section 4.1, respectively, are realized utilizing previously proposed solutions for UML/OCL verification and C validation. Afterward, the applicability of the resulting verification flow has been evaluated by means the following two examples:

- In the first example, we study the scalability of the proposed approach. For that, we apply the proposed methodology to a case study in which the complexity of the verification can be easily modified. This allows for the evaluation of the verification time with a conventional approach.

- Secondly, the approach is evaluated considering an industrial example; i. e. a UML description of a turn indicator used in Mercedes Benz cars (taken from [PLL$^+$]).

In the following, we summarize the results of these evaluations.

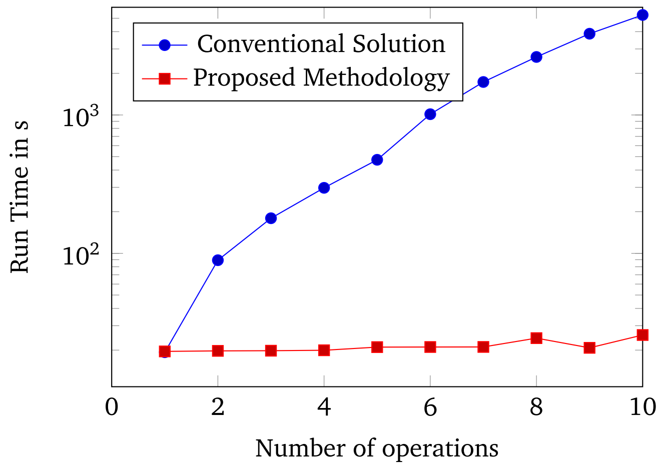
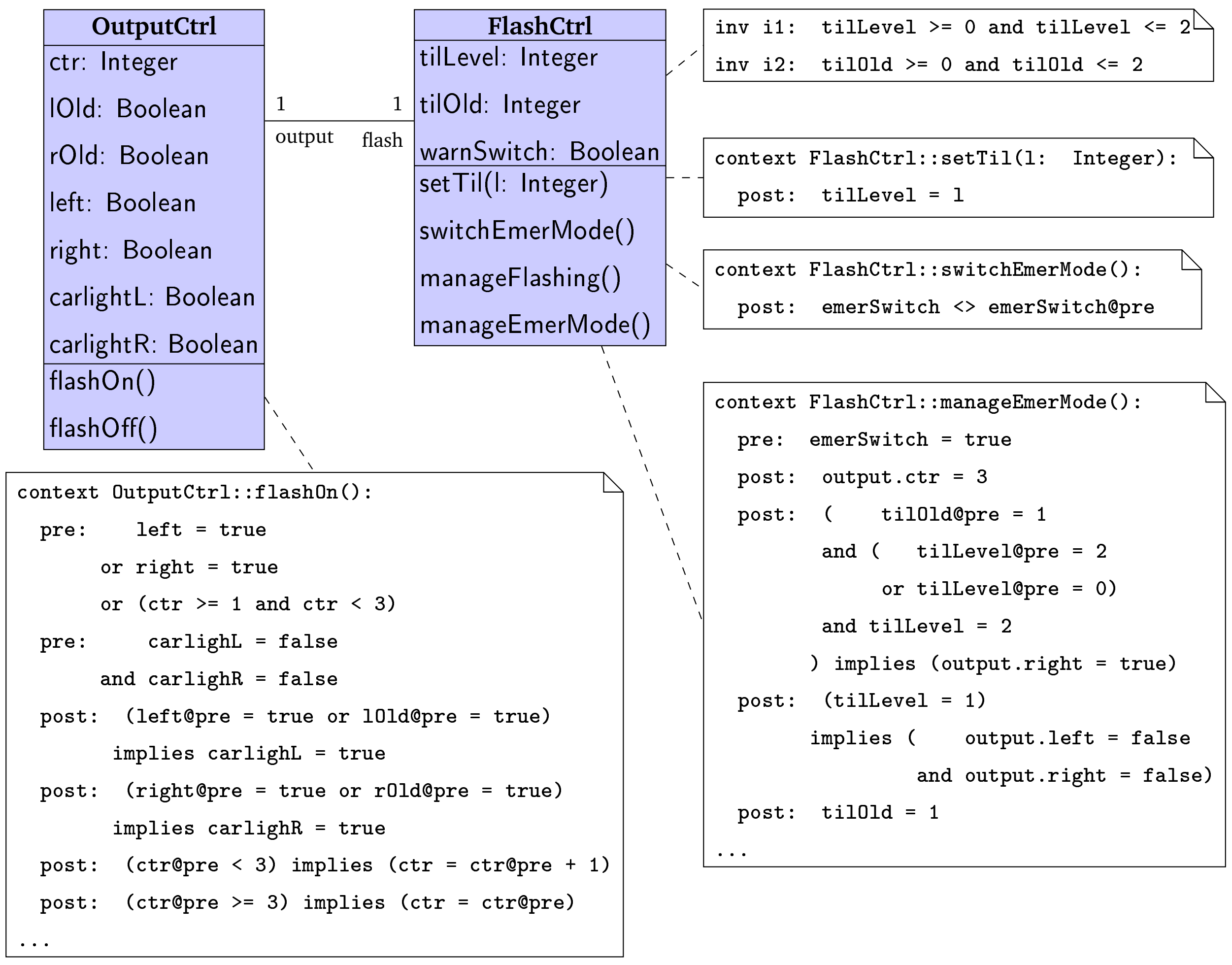


FIGURE 4.23: Needed run-times to verify the scalable example

Scalable Example

First, an example is considered which allows for a scalable evaluation of the proposed methodology (and, hence, also a comparison to currently applied approaches). More precisely, implementations are considered which sorts an array of size 10 according to different sort criteria (ascending, descending, odds-first, evens-first, alphabetical, capital-letters-first, and their combinations). Since each of these sort functions have a rather homogeneous design space, this allows for a consideration of different systems which can be scaled with respect to a different number of operations.

Figure 4.23 summarizes the resulting run times in CPU seconds (y-axis) needed to verify corresponding systems with a different number of sorting operations (x-axis). The blue values denote the time required using a conventional approach (i. e. techniques described in section 4.1), while the red values denote the time needed by the proposed methodology. The numbers clearly show the multiplicative effect that an increasing number of operations (and, hence, possible interactions) has on conventional verification methods (see blue values). In contrast, if the proposed methodology is applied, the overall system can be considered significantly faster on the UML/OCL level and, afterward, the detailed implementations can be verified in isolation. This leads to a significant improvement in the verification time (see red values).



```
inv i1:  tilLevel >= 0 and tilLevel <= 2
inv i2:  tilOld >= 0 and tilOld <= 2
```

```
context FlashCtrl::setTil(l:  Integer):
  post:  tilLevel = l
```

```
context FlashCtrl::switchEmerMode():
  post:  emerSwitch <> emerSwitch@pre
```

```
context FlashCtrl::manageEmerMode():
  pre:  emerSwitch = true
  post:  output.ctr = 3
  post:  (    tilOld@pre = 1
         and (   tilLevel@pre = 2
              or tilLevel@pre = 0)
         and tilLevel = 2
        ) implies (output.right = true)
  post:  (tilLevel = 1)
        implies (    output.left = false
                 and output.right = false)
  post:  tilOld = 1
...
```

```
context OutputCtrl::flashOn():
  pre:     left = true
       or right = true
       or (ctr >= 1 and ctr < 3)
  pre:      carlighL = false
       and carlighR = false
  post:  (left@pre = true or lOld@pre = true)
        implies carlighL = true
  post:  (right@pre = true or rOld@pre = true)
        implies carlighR = true
  post:  (ctr@pre < 3) implies (ctr = ctr@pre + 1)
  post:  (ctr@pre >= 3) implies (ctr = ctr@pre)
...
```

FIGURE 4.24: An industrial example: turn indicator

Industrial Example

Finally, the proposed methodology has been applied to verify a system from an industrial context; more precisely a turn indicator as used in Mercedes Benz cars and specified in [PLL+]. The turn indicator offers functionality for controlling the flash signal of the car by a lever (indicating left or right) and a switch for the warning lights. Once the driver moves the lever up or down, the corresponding light is turned on and off at least three times. The light keeps flashing for as long as the lever is left in the respective position. Pushing the switch turns on both lights simultaneously. As an additional function, the warning lights can always get interrupted by the regular flash signal, i. e. when the warning lights are active and the lever is pulled, the respective flash signal is turned on instead.

This specification leads to a UML/OCL description of the desired functionality as shown in Figure 4.24. More precisely, two classes (*OutputCtrl* and *FlashCtrl*) represent the structure of the system. The integer attributes `tilLevel` and `tilOld` represent the current and previous status of the lever, respectively, i. e. whether the level is/was on the left position (`0`), the neutral position (`1`), or the right position (`2`). The invariants `i1` and `i2` ensure that only valid values are assigned.

The Boolean attribute `warnSwitch` represents the value of the switch for the warning lights. If its value is **`true`** (**`false`**) the warning lights are switched on (off).

The attributes of the class *OutputCtrl* represent the respective "output signals" of the turn indicator. More precisely, the integer attribute `ctr` counts the number of flashes of the indicator (this is needed to keep track on whether the lights turned on and off at least three times). The attributes `left` and `right` represent internal signals to control the car lights. Besides this, the two attributes `lOld` and `rOld` store the indication of the direction in case it got interrupted by the warning lights. The car lights itself are represented by the Boolean attributes `carlightL` and `carlightR`.

The behavior of the turn indicator is specified by the operations as well as their corresponding pre- and postconditions. More precisely, `setTil` and `switchEmerMode` specify the actions of the driver, i. e. the activation of either the turn indication or the warning lights. The remaining two operations of class *FlashCtrl*, `manageFlashing` and `manageEmerMode`, enforce that the correct signals are activated in the output controller (represented by class *OutputCtrl*). Here, the car lights are switched on/off by executing the operations `flashOn` and `flashOff`.

Based on the model, we have implemented the operation(s)/method(s) and, now, want to check whether the implementation represents a valid instantiation of the functionality described in the model. We have implemented all operations in the presented model and validated the resulting implementation against it. To evaluate the capabilities of the proposed methodology, we have also introduced some flaws. In the following, we consider how the proposed methodology assists designers in detecting (and fixing) these flaws.

```
char ctr;

void FlashOn(){
  if(lOld||left) carlightleft  = 1;
  if(rOld||right)carlightright = 1;
  if(ctr < 3) ctr++;
}
```

```
char ctr;

void FlashOn(){
  if(lOld||left) carlightleft  = 1;
  if(rOld||right)carlightright = 1;
  if((unsigned char) ctr < 3) ctr++;
}
```

FIGURE 4.25: Faulty (and corrected) implementations of the running example.

One of the most concise bugs detected is related with method `flashOn`. This function implements the activation of the output controller. The pre- and postconditions of this operation can be seen in Figure 4.24, and can be read like this:

- If the left/right signal is true in the previous state, or in the actual one, then the corresponding output has to activate.
- The counter has to increase if it is smaller than three.
- If the counter is three (or bigger) it should not change its value.
- The preconditions state that the operation can only be called, iff the signal `left` or `right` in the previous state is assigned **`true`** or the value of the counter is between 1 and 3.

A first implementation of this function in C can be seen in Figure 4.7.4 (left). At first glance, the implementation looks like a trivial translation from postconditions to C, but an error has been introduced in it. As a result, the framework produces a counterexample (provided in Figure 4.26), in which the preconditions hold, but applying the input to the implementation, the postconditions do not.

The counterexample shows the set of violated postconditions and can be used to debug the implementation. For example, if the input is $0x83$ (which is in principle greater than 3), the counter should not change its value but the counterexample shows that it is increased. This flaw comes from the fact that $0x83$ is expressed as the following bit string:

```
-- values in the pre state --
ctr   = #0x83
lOld = 0      rOld    = 1
left = 0      right   = 1

-- values in the post state --
ctr   = #0x84
lOld = 0      rOld   = 1
left = 0      right  = 1
```

FIGURE 4.26: Counterexample showing the error in List. 4.7.4

10000011_2. This is only greater than 3 if we consider *signed* comparison semantics, but the symbolic formulas for the postconditions are expressed in terms of *unsigned* values. More precisely, the u in the prefix bvu states that unsigned bit vector operations are used. The details of this bug motivate the fix shown in Figure 4.7.4 (right). Checking this new version using the proposed methodology proves the correctness of the implementation. In a similar fashion, all remaining implementations have been verified.

4.7.5 Conclusions

In this section, we propose a two-level methodology for the verification of complex systems. In the first level, established approaches are used to guarantee that the model is specified as desired before the implementation phase. In a second level, we check the implementation of each operation against its contracts, i. e. the pre- and postconditions of the corresponding operation in the model.

We show how this staged verification methodology can help in the verification of complex integrated designs. We exploit the fact that the OCL pre- and postconditions succinctly capture the behavior intended by the developer and conduct a first verification of the integration of components using only those pre- and postconditions. If this does not warn about potential bugs, the second phase imposes further constraints that are derived from the implementation. Here, only local reasoning is needed, since the first phase already considered global reasoning. As shown in the examples, this can lead to significant reduction in the overall computational verification time.

On top of that, if a bug is detected in the second phase, the symbolic formulation of the implementation enables a solver to construct a test vector that demonstrates the feasibility of the path that leads to the bug. By this, the methodology offers a proof that a bug is in the code. This reasoning is performed *locally* on each function, which also greatly simplifies the human effort devoted to debugging and maintenance.

5 Conclusions

The results obtained during this Ph.D. have been disseminated by the publication of twelve conference papers, four journal manuscripts, and two workshop contributions.

This dissertation focuses on the analysis of non-functional properties of embedded real-time systems (ERTSs) and has advanced the state-of-the-art development by providing new techniques, tools, and ideas to analyze ERTSs.

The thesis proposes a combination of the techniques of "native simulation", "symbolic execution", and "trace refinement" and shows that this combination offers several advantages in the analysis and verification of non-functional properties of embedded systems:

- It provides a framework that can be used at different stages of the design process, balancing the requirements of the early and the late stages of the design.

- Thanks to automated instrumentation, the simulation, combination with static analysis, dynamic execution, and computation of worst-case execution time only require minimal user intervention.
- The bounds for the worst-case execution time are tight and mathematically proven, so they can be used to verify properties of safety-critical systems.

The thesis comprises three parts:

Part 1 (Chapter 2) Program instrumentation and analysis

The first part of the thesis describes the software instrumentation and analysis to implement the techniques detailed in this thesis. This dissertation emphasizes the importance of considering intermediate representations that lie between high-level languages and machine code. We also acknowledge the need for other representations of the program when the intermediate representation does not provide enough data for the analysis, such as the derivation of the worst-case execution time. This thesis provides, for those cases, a technique and a tool to reconstruct the control-flow-graph from the binary program. The primary outcomes of this part are:

- A tool that instruments LLVM Intermediate Representation adding function calls that are useful for implementing simulation and verification techniques. This tool has been developed by me and includes around 10,000 lines of C++ code.
- A tool to extract control-flow-graphs directly from binary code. This tool has been developed by me during my stay at Macquarie University (iMQRes Ph.D. grant) in collaboration with the co-author of publication [W2] and includes around 5,000 lines of Scala code.

Different analyses described in this dissertation have been successfully applied on top of the described instrumentation, demonstrating that the tools are appropriate for implementing analyzers of non-functional properties for Embedded Systems.

Part 2 (Chapter 3) Simulation of non-functional properties

This chapter describes several advances in *native simulation* that obtain estimates about non-functional requirements in the early stages of the design of ERTS. This particular focus on the initial stages of the design is motivated by the need for choosing a valid platform when the application has not been optimized yet for any particular target. In this part, we describe our approach about how to split the simulation domain into three sub-domains (single-core, many-core platforms with a NOC, and hardware accelerators implemented in an FPGA) and combine them to obtain fast and accurate results. This work has been developed in collaboration with the authors of papers [C1], [J1], [J2], [J3], [C2], [C3], [C4], [C5], [C6] and [C7]. I emphasize the following conclusions of this chapter.

- The combination of these techniques provides a good balance between the requirements of a framework in the early stages of the design (when tools are required to be fast, loosely accurate, and abstract) and the late stages of the design (when the tools require to be exact, formal, and precise).
- Experimental measurements demonstrate that the presented techniques outperform other state-of-the-art tools and techniques by orders of magnitude regarding simulation time with a small penalty in accuracy.
- The tools and techniques have been tested with industrial and research designs. The case studies are interesting in their own right and illustrate some of the real problems that are faced when designing embedded real-time systems and high-performance systems as well as the benefits that these techniques can bring to the design.
- In the case of single-core simulation, the techniques described in this chapter were used to win the third prize in the 11th Memocode design Competition [MSSS14].
- In the case of many-core simulation, the techniques described in this chapter were used to win a first prize in the Intel Modern Code Challenge, organized by Intel, CERN, and Newcastle University.

- In the case of FPGA simulation, the techniques described in this chapter were used to optimize a computer-vision application obtaining results that are three times faster than other state-of-the-art implementations for the same conditions [PAS12].

- These optimization techniques highlight the importance of the selected trade-offs in simulation tools and demonstrate that the presented techniques and selected parameters of the architecture are useful for analyzing the non-functional properties of embedded systems.

- The chapter concludes with the application of the selected tools and techniques to the analysis of the thermal behavior of Systems-On-Chip [CGD+11].

The results of this chapter have been published in the following journals and conferences:

[C1] **Pablo Gonzalez de Aledo Marugan**, Alvaro Diaz Suarez, Luis Diaz Suarez and Pablo Sanchez. "Profiling and optimizations for Embedded Systems." Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2014. pp 194–197 Lausanne 2014. ISBN: 978-1-4799-5338-7

[J1] **Pablo Gonzalez de Aledo Marugan**, Javier Gonzalez Bayon, Pable Sanchez Espeso and Juan Casal. "OpenMP performance analysis for many-core platforms with non-uniform memory access" International Journal of Computer Science Issues vol 10 Issue 2 pp 463–470 2013. ISSN: 1694-0784

[J2] Jesus Perez, **Pablo Gonzalez de Aledo Marugan** and Pablo Sanchez "Real-time voxel-based visual hull reconstruction" Microprocessors and Microsystems vol 5 pp 439–447 Elsevier Science Publishers, 2012. Amsterdam ISSN: 0141-9331

[J3] Daniel Calvo, **Pablo Gonzalez de Aledo Marugan**, Luis Diaz, Hector Posadas, Pablo Sanchez, Eugenio Villar, Andrea Acquaviva and Enrico Macii. "A Multi-Processing System-On-Chip Native Simulation Framework for Power and Thermal-Aware Design" ASP Journal on Low-Power Electronics JOLPE vol 4 - Special Issue on Low

Power Design and Verification Techniques. pp 2–16 American Scientific Publishers, 2011. ISSN: 1546-1998

[C2] **Pablo Gonzalez de Aledo Marugan**, Javier Gonzalez Bayon and Pablo Sanchez, 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE "An approach for algorithm parallelization oriented to a many-core implementation" pp 841–842 2012. ISBN: 978-1-4673-1631-6

[C3] **Pablo Gonzalez de Aledo Marugan**, Javier Gonzalez-Bayon and Pablo Sanchez. Digital System Design (DSD) "A virtual platform for performance estimation of many-core implementations" vol 15 pp 541–544 Izmir, 2012. ISBN: 978-1-4673-2498-4

[C4] **Pablo Gonzalez de Aledo Marugan**, Javier Gonzalez Bayon and Pablo Sanchez "A virtual Platform for performance estimation of OpenMP Programs" Proceedings of the XXX Conference on Design of Circuits and Integrated Systems 2012.

[C5] **Pablo Gonzalez de Aledo Marugan**, Javier Bayon Gonzalez and Pablo Sanchez Espeso. Specification and Design Languages (FDL), 2011 Forum on Hardware "Performance estimation by Dynamic Scheduling" pp 1–6 Oldenburg, 2011. ISBN: 978-1-4577-0763-6

[C6] **Pablo Gonzalez de Aledo Marugan** and Pablo Sanchez Espeso "Hardware performance estimation by Dynamic Scheduling" XXV Conference on Design of Circuits and Integrated Systems pp 453–458 Albufeira, 2011. ISBN: 978-9729918131

[C7] **Pablo Gonzalez de Aledo Marugan**, Pablo Sanchez Espeso and Luis Diaz. "Embedded software execution time estimation at different abstraction levels" XXV Conference on Design of Circuits and Integrated Systems, DCIS-10 pp 532–537 Lanzarote, 2010. ISBN: 978-84-693-7393-4

The optimization results of the Intel Modern Code Challenge also led to a (submitted) publication that I co-authored with the authors of reference [J4] as a collaboration between CERN, Intel, and Newcastle University.

[J4] **Pablo Gonzalez de Aledo Marugan**, Pablo Sanchez, Marco Manca, Jerry Baugh, Andrey Vladimirov, Ryo Asai, Marcus Kaiser and Roman Bauer. "An optimization approach for the computational modeling of biological development" Advances in Engineering Software. **(Under Review)**

Part 3 (Chapter 4) Verification of non-functional properties

In this chapter, we focus on the problem of *verification* of non-functional properties (i.e. how to prove properties of the non-functional behavior of a system for all possible inputs and initial hardware states). The problem is challenging because both the program and the hardware are complex systems that interact in a parallel manner. In this chapter, we propose tools and techniques to address these difficulties.

The main outcomes of this part of the thesis are:

- A tool (FOREST) that, given a C source, can construct an under-approximated model of the behavior of the program that is useful for finding bugs. This tool has been created by me inside the project DREAMs (TEC2011-28666-C04-02) and includes around 30,000 lines of C++ code.

- The tool has competed in the International Software Verification Competition and obtained the best score in the loops sub-category in 2015 [dAE15b][Bey15][1].

- The tool has been integrated with a commercial static analyzer (Goanna) to improve both the capabilities of the static analyzer to detect bugs as well as the capabilities of FOREST to prove that a program does not have bugs. This integration was implemented during my Endeavour Fellowship [GdASH15] in collaboration with the authors of publication [W1].

- The same techniques that were used to prove functional properties in the software were extended to derive bounds for the worst-case execution time. This extension

[1]https://sv-comp.sosy-lab.org/2015/results/Loops.table.html

was implemented on top of the tool UPPAAL by the authors of publication [C8] during our stay at Macquarie University.

- In collaboration with the authors of publication [J5], FOREST was also integrated with a model checker that checks the consistency of a UML/OCL specification during my visit to Bremen University.

The results of this part of the thesis have been published in:

[C8] Franck Cassez, **Pablo Gonzalez de Aledo Marugan** and Peter Jensen. “WUPPAAL: Computation of Worst-Case-Execution-Time for Binary Programs with UPPAAL” **(Under Review)**

[J5] **Pablo Gonzalez de Aledo Marugan**, Nils Przigoda, Robert Wille, Rolf Drechsler and Pablo Sanchez. “Towards a Verification Flow Across Abstraction Levels: Verifying Implementations Against Their Formal Specification” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2016). DOI: 10.1109/TCAD.2016.2611494

[W1] **Pablo Gonzalez de Aledo Margugan**, Pablo Sanchez Espeso and Ralf Huuck, “An Approach to Static-Dynamic Software Analysis” Proceedings of the 4th International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS), 2015. Paris, France, November 6-7, 2015. pp 225–240. DOI: 10.1007/978-3-319-29510-7_13

[W2] Franck Cassez and **Pablo Gonzalez de Aledo Marugan**, “Timed Automata for Modelling Caches and Pipelines”, Proceedings Workshop on Models for Formal Analysis of Real Systems (MARS) 2015 Suva, Fiji, November 23, 2015. pp 37–45. DOI: 10.4204/EPTCS.196.4

[C9] **Pablo Gonzalez de Aledo Marugan**, Alvaro Diaz Suarez, Pablo Sanchez and Ralf Huuck. “Discovering and Validating Concurrency Specification from Test Executions”, Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering. 2016.

[C10] **Pablo Gonzalez de Aledo Marugan** and Pablo Sanchez. “Framework For Embedded System Verification”, Proceedings of the 21st Conference on Tools and Algorithms for the Construction and Analysis of Systems. vol 9035 pp 429–431 Springer-Verlag Berlin Heidelberg, 2015. ISBN: 978-3-662-46681-0

[C11] “Virtual platform for power and security analysis of wireless sensor networks”, A. Diaz, J. Gonzalez-Bayon, **Pablo Gonzalez de Aledo Marugan**, P. Sanchez Proceedings of SPIE Vol 8764, 2013. pp 87640I–1

[C12] Franck Cassez, Anthony Sloane, Matthew Roberts, Matt Pigram, Pongsak Suvanpong and **Pablo Gonzalez de Aledo Marugan** “Skink: Static Analysis of Programs in LLVM Intermediate Representation (Competition contribution)”, Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2017 Uppsala, Sweden, April 22-29, 2017. LNCS. Springer

5.1 Application example

At risk of oversimplifying the design and implementation of an industrial design methodology, the following example illustrates how the techniques developed in this thesis could help in such scenario, and at the same time summarizes the main contributions of this work over a cohesive example. The implementation of such example is left for future work.

Suppose an industrial assembly chain in which one of the steps in the chain consists on classifying different pieces of a mechanical design based on their shape. Due to the variety in shapes, the identification of pieces needs to be done in 3D because a planar projection is usually not enough to reliably identify different pieces. At the same time, the high throughput of the assembly chain imposes several timing constraints in the recognition; for real-time applications these constraints vary, but are considered to be between 30 and 60 frames-per-second. To supervise the correct operation of the assembly chain, a camera is also installed to transmit a video stream to an operator via a wireless link.

Even in such a simplified example, there are different non-functional constraints. Some of the elements of the design, such as the 3D projection and identification require strong real-time constraints because missing the deadline can make the assembly chain to malfunction and lock the whole process. On the other hand, the streaming of video in this case is used as a control measure, and the design decisions are probably more influenced by the latency of the video encoding, transmission and decoding as well as their subjective quality. On top of this, without going to the extreme case of environments that are hazardous or inaccessible to humans, the installation of electronic equipment in these scenarios is usually not easily accessible (therefore difficult to repair in case of fault), while being at the same time under thermal stress and vibrations.

We have already seen in Section 3.6.4 an example of 3D volume reconstruction based on images provided by several cameras that reconstruct an approximation of the convex hull silhouette of a 3d object. As we have seen, design decisions both in the software and in the hardware domain can lead to differences of several orders of magnitude in the execution time of the reconstruction, which can mean the difference between being able to process 60 frames per second, or requiring several seconds to process one frame. As we have seen, analysing some of these design decisions for one single algorithm and one single platform is already a challenging task. However, one might imagine that in the mentioned example we might be interested on analyzing several possible algorithms under several target platforms, some of them which might even not have been created yet, or are designed ad-hoc for our example. A flexible way of evaluating the impact of these design decisions without much manual effort in implementing all the details of each solution might help in this evaluation. Indeed, some benchmarks for evaluating SLAM algorithms [NBZ+15] take the idea one step further and automate the search of the best configuration of parameters through a design-space-exploration (DSE) process that uses machine learning to find the pareto-optimal solution in the domain of algorithm, compiler and hardware. These approaches, however are limited by the enormous amount of time that is required to evaluate one single solution (in the order of days), or the need of having a physical platform in which we can evaluate each configuration. We could imagine therefore using instead the lightweight Instrumentation and simulation techniques described in this thesis to prune the infeasible designs to ease the DSE in the early stages of the design flow.

At the end of the day, however, we need to prove that the results of the simulation are valid for any possible input of the program (the results of a simulation do not prove the correctness of the design). For that we need to integrate a formal model of the computing platform with a model of the software as obtained from the binary code. We have shown that the 3D reconstruction of the convex hull can be performed by matrix multiplications, whose binary control-flow-graph we can reconstruct as demonstrated in Section 2.2. Also, in Section 4.6.6 we explain how (knowing a model of the platform that we have previously chosen) we can employ the combination of the software and hardware models to verify such temporal constraints. This would enable us to formally prove the correctness of the design; something that we were not able to do before.

In the case of the video streaming we might face similar challenges; a video codec is generally parameterized by several dozens of parameters, whose modification alters the quality of transmission, the latency and the throughput. As we see in Section 3.7, the simulation platform described in the first part of this thesis enables a holistic simulation of designs like this having in mind not only the architectural parameters but also physical ones such as the thermal behavior, which would maximize the useful life of the design under thermal and mechanical stress. Again, having a flexible way of evaluating several designs can help in finding good trade offs; such as implementing redundant encoders in a multi-core platform or an FPGA such that the system can recover from a malfunction, or using instead a more robust sequential implementation over a single CPU.

Future Work

Based on the results reported in this dissertation, the following directions are relevant to improve the efficiency and applicability of our techniques:

- **Modeling hardware elements.** As a result of further progress in the design of embedded systems such as branch prediction and speculative and out-of-order execution, new architectural elements in modern Systems-On-Chip continue to complicate the analysis and simulation. A future line of research is developing models to simulate those elements. This would enable the described techniques to be used in more platforms, as well as reduce the simulation error.

- **Leveraging many-core developer platforms.** Regarding the native-simulation of many-core platforms, as the main bottleneck is the simulation of concurrency, an interesting field of research is how to leverage the multiple cores of the host platform to simulate the multi-core architecture of the target platform. Several state-of-the-art techniques are starting to apply this idea to improve the simulation time of those systems [DGVS15].

- **Instrumentation for newer parallelization APIs.** In the simulation of heterogeneous designs, new languages and APIs have appeared in recent years that aim to model the massively concurrent parallelism of hardware using a high-level API from C/C++. One of these APIs that is gaining adoption is OpenACC. Supporting this API in a similar way as we did for OpenMP will provide more capabilities to the tool and would simplify the estimation of non-functional properties in the early phases of the design of heterogeneous CPU/GPU systems.

- **Formalizing parallel platforms.** Formalizing these models as we did for a single-core processor will enable us not only to simulate non-functional properties but also to *verify* them. The formalism of timed automata is especially well suited for the verification of timed concurrent systems and, as we have demonstrated, can be used for verifying timing properties of multi-processor systems.

- **High and low-level verification for schedulability.** In the same way that OCL enables designers to describe functional specifications, several extensions have been added to the UML language (Profile for Schedulability, Performance and Time) [Dou03] to specify real-time constraints. Several tools allow the analysis of schedulability of systems based on the worst-case execution time of the individual tasks [HGGM01]. In a similar way that our techniques have been integrated to bridge the gap between the high and low-level verification of functional properties, they can be combined to decide the schedulability of a system with tasks described at a low level.

References

[AD94] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994. 134

[AH08] M. B. Abdelhalim and S. E D Habib. Fast FPGA-based area and latency estimation for a novel hardware/software partitioning scheme. *Canadian Conference on Electrical and Computer Engineering*, pages 775–780, 2008. 78

[arm] Arm1176jzf-s technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/ddi0301h_arm1176jzfs_r0p7_trm.pdf. Accessed: 2017-02-21. 54, 55

[BBB+05] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *The Journal of VLSI Signal Processing*, 41(2):169–182, 2005. 92

[BBC+10] A Bessey, K Block, B Chelf, A Chou, and B Fulton. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the*, 2010. 109

[BBF16] Nicola Bombieri, Federico Busato, and Franco Fummi. A fine-grained performance model for gpu architectures. In *Design, Automation & Test* *in Europe Conference & Exhibition (DATE), 2016*, pages 1267–1272. IEEE, 2016. 40

[BC11] Jean-Luc Béchennec and Franck Cassez. Computation of WCET using program slicing and real-time model-checking. *CoRR*, abs/1105.1633, 2011. 135

[BCF+12a] Mark Bradley, Franck Cassez, Ansgar Fehnker, Thomas Given-Wilson, and Ralf Huuck. High Performance Static Analysis for Industry. *ENTCS*, 289(0):3–14, 2012. 125

[BCF+12b] Mark Bradley, Franck Cassez, Ansgar Fehnker, Thomas Given-Wilson, Ralf Huuck, and Maximilian Junker. Goannasmt–a static analyzer with smt-based refinement. *Tools for Automatic Program AnalysiS (TAPAS 2012)*, 2012. 120

[BCTB10] A Bartolini, M Cacciari, A Tilli, and L Benini. A virtual platform environment for exploring power, thermal and reliability management control strategies in high-performance multicores. *Proceedings of the 20th*, 2010. 92

[BDVS15] E Borin, PRB Devloo, GS Vieira, and N Shauer. Accelerating engineering software on modern multi-core processors. *Advances in Engineering*, 2015. 61

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. 139

[Bey15] Dirk Beyer. Software Verification and Verifiable Witnesses. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *LNCS*, pages 401–416. 2015. 120, 128, 172

[BGF+10] Markus Becker, Giuseppe Di Guglielmo, Franco Fummi, Wolfgang Mueller, Graziano Pravadelli, and Tao Xie. RTOS-Aware Refinement for TLM2 .

0-based HW / SW Designs. *Proceeding of Design Automation and Test in Europe (DATE2010)*, pages 1053 – 1058, 2010. 45

[BLL$^+$96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal–a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, pages 232–243. Springer, 1996. 134

[bou] Bound-t. `http://www.tidorum.fi`, `http://www.bound-t.com`. 111, 134

[BS08] J Burnim and K Sen. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society. 122

[BTM00] D Brooks, V Tiwari, and M Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. 2000. 44

[BUZC11a] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. *Proceedings of the sixth conference on Computer systems - EuroSys '11*, page 183, 2011. 150

[BUZC11b] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *European Conference on Computer Systems*, pages 183–198, 2011. 152

[Cas11] Franck Cassez. Timed games for computing WCET for pipelined processors with caches. In *11th International Conference on Application of Concurrency to System Design, ACSD 2011, Newcastle Upon Tyne, UK, 20-24 June, 2011*, pages 195–204. IEEE, 2011. 135

[cat] Catapult c synthesis, mentor graphics. `http://www.mentor.com/products/c-based_design`. 76, 83

[CB13a] F Cassez and JL Béchennec. Timing analysis of binary programs with UPPAAL. *Application of Concurrency to*, 2013. 132

[CB13b] Franck Cassez and Jean-Luc Béchennec. Timing analysis of binary programs with UPPAAL. In *13th International Conference on Application of Concurrency to System Design, ACSD 2013*, pages 41–50. IEEE Computer Society, July 2013. 135

[CC77] Patrick Cousot and Radhia 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*, pages 238–252. ACM, 1977. 109

[CCR08] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of {UML/OCL} Class Diagrams using Constraint Programming. In *Int'l Conf. on Software Testing Verification and Validation*, pages 73–80, 2008. 149

[CCR09] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verifying UML/OCL Operation Contracts. In *Integrated Formal Methods*, pages 40–55, 2009. 149

[CdAM15] Franck Cassez and Pablo González de Aledo Marugán. Timed automata for modelling caches and pipelines. In Rob J. van Glabbeek, Jan Friso Groote, and Peter Höfner, editors, *Proceedings Workshop on Models for Formal Analysis of Real Systems, MARS 2015, Suva, Fiji, November 23, 2015.*, volume 196 of *EPTCS*, pages 37–45, 2015. 135, 139

[CDE08a] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, 2008. 107, 150

[CDE08b] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. pages 209–224, 2008. 152

[CGD$^+$11] Daniel Calvo, Pablo González, Luís Díaz, Héctor Posadas, Pablo Sánchez, Eugenio Villar, Andrea Acquaviva, and Enrico Macii. A multi-processing systems-on-chip native simulation framework for power and thermal-aware design. *Journal of Low Power Electronics*, 7(1):2–16, 2011. 92, 93, 170

[CGJ$^+$00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000. 109

[CGK11] C Cadar, P Godefroid, and S Khurshid. Symbolic execution for software testing in practice: preliminary assessment. *Proceedings of the 33rd*, 2011. 108

[CGWm04] LN Chakrapani, J Gyllenhaal, and WH Wen-mei. Trimaran: An infrastructure for research in instruction-level parallelism. *on Languages and ...*, 2004. 78

[CHO12] F Cassez, RR Hansen, and MC Olesen. What is a Timing Anomaly? *OASIcs-OpenAccess Series*, 2012. 134

[Chv83] Vašek Chvátal. Linear programming. a series of books in the mathematical sciences, 1983. 110

[CKL04a] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. *Tools and Algorithms for the Construction and Analysis of Systems*, 2988:168–176, 2004. 150

[CKL04b] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. pages 168–176, 2004. 107, 152

[CP00] A Colin and I Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 2000. 110

[CP01] A Colin and I Puaut. A modular and retargetable framework for tree-based WCET analysis. *Time Systems, 13th Euromicro Conference on ...*, 2001. 110

[CPVM07] Juan Castillo, Héctor Posadas, Eugenio Villar, and Marcos Martínez. Energy consumption estimation technique in embedded processors with stable power consumption based on source-code operator energy figures. In *XXII Conference on Design of Circuits and Integrated Systems*, 2007. 46, 48

[CPVM10] J Castillo, H Posadas, E Villar, and M Martinez. Fast instruction cache modeling for approximate timed HW/SW co-simulation. *Proceedings of the 20th*, 2010. 49

[CSC+09] Jianjiang Ceng, Weihua Sheng, Jeronimo Castrillon, Anastasia Stulova, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A high-level virtual platform for early MPSoC software development. *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis - CODES+ISSS '09*, page 11, 2009. 41

[CSR+17] Franck Cassez, Anthony M. Sloane, Matthew Roberts, Matthew Pigram, Pongsak Suvanpong, and Pablo González de Aledo Marugán. Skink: Static analysis of programs in llvm intermediate representation (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017. Proceedings*, 2017. forthcoming. 110

[CWLW15] Y Cai, G Wang, G Li, and H Wang. A high performance crashworthiness simulation system based on GPU. *Advances in Engineering Software*, 2015. 61

[dA11] Pablo González de Aledo. Simulación nativa para la estimación de rendimiento en entornos de codiseño Hardware/Software. Master's thesis, University of Cantabria, 2011. 82

[dAE15a] Pablo González de Aledo and Pablo Sánchez Espeso. FramewORk for Embedded System verification - (Competition Contribution). In *Tools and*

Algorithms for Construction and Analysis of Systems, pages 429–431, 2015. 150, 152, 159

[dAE15b] Pablo González de Aledo and Pablo Sánchez Espeso. FramewORk for Embedded System verification - (Competition Contribution). In *Tools and Algorithms for Construction and Analysis of Systems*, pages 429–431, 2015. 172

[dAMGBE11] Pablo González de Aledo Marugán, Javier González-Bayón, and Pablo Sánchez Espeso. Hardware performance estimation by dynamic scheduling. In *Specification and Design Languages (FDL), 2011 Forum on*, pages 1–6. IEEE, 2011. 82

[DDSL08] S Dhouib, JP Diguet, E Senn, and J Laurent. Energy models of real time operating systems on FPGA. *OSPERT 2008*, 2008. 40

[DGVS15] Luis Diaz, Eduardo González, Eugenio Villar, and Pablo Sanchez. VIPPE: Parallel simulation and performance analysis of complex embedded systems. In *HiPPES4CogApp: High Performance, Predictable Embedded Systems for Cognitive Application.*, 2015. 177

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, jul 2008. 123

[DLT03] Vassilios V Dimakopoulos, Elias Leontiadis, and George Tzoumas. A portable c compiler for openmp v. 2.0. In *Proc. EWOMP*, pages 5–11, 2003. 62

[Dou03] Bruce Powel Douglass. Real time uml. In *Formal Techniques in Real-Time and Fault-Tolerant Systems: 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002. Proceedings*, volume 2469, page 53. Springer, 2003. 177

[EESG03] J Engblom, A Ermedahl, M Sjödin, and J Gustafsson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on*, 2003. 112

[FAM08] S Johann Filho, A Aguiar, and CAM Marcon. High-level estimation of execution time and energy consumption for fast homogeneous mpsocs prototyping. *, 2008. RSP'08. The ...*, 2008. 45

[FH04] Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004. 111, 134

[FHJ07] A Fehnker, R Huuck, and P Jayet. Model checking software at compile time. *Aspects of Software ...*, 2007. 123

[FHS10] A Fehnker, R Huuck, and S Seefried. Counterexample guided path reduction for static program analysis. *Concurrency, Compositionality, and*, 2010. 123

[FHW04] C Ferdinand, R Heckmann, and R Wilhelm. Analyzing the worst-case execution time by abstract interpretation of executable code. *Automotive Software Workshop*, 2004. 111

[GADSSE10] Pablo Gonzalez-Aledo, Luis Diaz Suarez, and Pablo Sanchez Espeso. Embedded software execution time at different abstraction levels. In *Proceedings of the 25th Conference on Design of Circuits and Integrated Systems*, pages 532–537, 2010. 11, 46, 47

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010. 31, 142

[GdAPW$^+$16] Pablo Gonzalez-de Aledo, Nils Przigoda, Robert Wille, Rolf Drechsler, and Pablo Sanchez. Towards a verification flow across abstraction levels: Verifying implementations against their formal specification. *IEEE Transactions* *on Computer-Aided Design of Integrated Circuits and Systems*, 2016. 118, 150, 151, 155

[GdASH15] Pablo Gonzalez-de Aledo, Pablo Sanchez, and Ralf Huuck. An approach to static-dynamic software analysis. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 225–240. Springer, 2015. 172

[GDGN03] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466. IEEE, 2003. 76

[GDWL12] DD Gajski, ND Dutt, ACH Wu, and SYL Lin. High—Level Synthesis: Introduction to Chip and System Design. 2012. 78

[GE05] J Gustafsson and A Ermedahl. Towards a flow analysis for embedded system C programs. *, 2005. WORDS 2005. 10th . . .* , 2005. 112

[GGBS12] Pablo González, Javier González-Bayon, and Pablo Sánchez. An approach for algorithm parallelization oriented to a many-core implementation. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 841–842, 2012. 62

[GGBSEC13] Pablo González, Javier González-Bayon, Pablo Sanchez Espeso, and Juan Casal. OpenMP performance analysis for many-core platforms with non-uniform memory access. *International Journal of Computer Science Issues*, 10:5–11, 2013. 62

[GGD$^+$02] AB Abril Garcia, J Gobert, T Dombek, H Mehrez, and F Petrot. Cycle-accurate energy estimation in system level descriptions of embedded systems. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, volume 2, pages 549–552. IEEE, 2002. 61

[GGP08] Patrice Gerin, Xavier Guérin, and Frédéric Pétrot. Efficient implementation of native software simulation for MPSoC. *Proceedings -Design, Automation and Test in Europe, DATE*, pages 676–681, 2008. 45

[GGS12] Pablo González, Javier González, and Pablo Sánchez. A virtual platform for performance estimation of many-core implementations. In *Proceedings of Digital System Design (DSD)*, volume 15, pages 541–544, 2012. 62

[GKS05] P Godefroid, N Klarlund, and K Sen. DART: directed automated random testing. *ACM Sigplan Notices*, 2005. 116

[GLD10] Daniel Große, Hoang M Le, and Rolf Drechsler. Proving transaction and system-level properties of untimed SystemC {TLM} designs. In *Int'l Conf. on Formal Methods and Models for Codesign*, pages 113–122, 2010. 150, 152

[Gra] GrammaTech. {C}ode{S}urfer. `http://www.grammatech.com/`. 109

[GS02] S Gurumurthi and A Sivasubramaniam. Using complete machine simulation for software power estimation: The softwatt approach. *High-Performance*, 2002. 44

[GUA11] E Gebrewahid and Z Ul-Abdin. Mapping Occam-pi programs to a Manycore Architecture. *MCC-2011, Fourth*, 2011. 61

[Gus00] J Gustafsson. Analyzing execution-time of object-oriented programs using abstract interpretation. 2000. 112

[Hag13] RJ Hagerman. Epilepsy drives autism in neurodevelopmental disorders. *Developmental Medicine & Child Neurology*, 2013. 65

[HAM99] CA Healy, RD Arnold, and F Mueller. Bounding pipeline and instruction cache performance. *IEEE Transactions*, 1999. 112

[HCW14] WF Hu, MH Chahrour, and CA Walsh. The diverse genetic landscape of neurodevelopmental disorders. *Annual review of genomics*, 2014. 65

[HCZD04] B Han, D Comaniciu, Y Zhu, and L Davis. Incremental density approximation and kernel-based bayesian filtering for object tracking. *CVPR (1)*, 2004. 86

[HGGM01] M González Harbour, JJ Gutiérrez García, JC Palencia Gutiérrez, and JM Drake Moyano. Mast: Modeling and analysis suite for real time applications. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 125–134. IEEE, 2001. 177

[HHP09a] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *International Static Analysis Symposium*, pages 69–85. Springer, 2009. 110

[HHP09b] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *Static Analysis Symposium*, number Sas in LNCS, pages 69–85. Springer Berlin Heidelberg, 2009. 145

[HHP13] M Heizmann, J Hoenicke, and A Podelski. Software model checking for people who love automata. *International Conference on*, 2013. 145

[HR05] M Holzer and M Rupp. Static estimation of execution times for hardware accelerators in system-on-chips. *System-on-Chip, 2005. Proceedings. 2005*, 2005. 76

[HSR98] C Healy, M Sjodin, and V Rustagi. Bounding loop iterations for timing analysis. *Symposium, 1998. ...*, 1998. 112

[HTHT09] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. 17:242–254, 2009. 82

[Huu15] Ralf Huuck. Technology transfer: Formal analysis, engineering, and business value. *Sci. Comput. Program.*, 103:3–12, 2015. 109

[iem] Intelligent energy management (iem). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0375a/Cegdcfdi.html. 95

[Ins10] TR Insel. Rethinking schizophrenia. *Nature*, 2010. 65

[JHFK12] M Junker, R Huuck, A Fehnker, and A Knapp. SMT-based false positive elimination in static program analysis. *International Conference on*, 2012. 123, 124, 130

[Jin13] T Jin. Gradient sensing during chemotaxis. *Current opinion in cell biology*, 2013. 66

[KHH03] Praveen Kalla, Jörg Henkel, and Xiaobo Sharon Hu. SEA. In *Proceedings of the 2003 conference on Asia South Pacific design automation - ASPDAC*, page 600, New York, New York, USA, 2003. ACM Press. 45

[KKAD06] K Khouri, F Kurdahi, M Abadir, and N Dutt. Floorplan driven leakage power aware IP-based SoC design space exploration. *Codesign and System ...*, 2006. 91, 92

[KT14] Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014. 116

[KWC$^+$04] S Kang, H Wang, Y Chen, X Wang, and Y Dai. Skyeye: An instruction simulator with energy awareness. *International Conference on*, 2004. 46

[LA04] C Lattner and V Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the international symposium on Code*, 2004. 78

[Lau94] A Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Transactions on pattern analysis and*, 1994. 85, 86

[LB08] A Ladikos and S Benhimane. Efficient visual hull computation for real-time 3D reconstruction using CUDA. *Computer Vision and*, 2008. 85, 90

[LBN08] Alexander Ladikos, Selim Benhimane, and Nassir Navab. Efficient visual hull computation for real-time 3d reconstruction using cuda. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–8. IEEE, 2008. 90

[LGHD13] Hoang M Le, Daniel Große, Vladimir Herdt, and Rolf Drechsler. Verifying SystemC using an intermediate verification language and symbolic simulation. In *Design Automation Conference*, pages 116:1—-116:6, 2013. 150, 152

[Lis14] Björn Lisper. *SWEET – A Tool for WCET Flow Analysis (Extended Abstract)*, pages 482–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. 108

[LJ03] T Li and LK John. Run-time modeling and estimation of operating system power consumption. *ACM SIGMETRICS Performance Evaluation Review*, 2003. 39, 40, 42

[LLSV99] M. Lajolo, M. Lazarescu, and a. Sangiovanni-Vincentelli. A compilation-based software estimation scheme for hardware/software co-simulation. *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99) (IEEE Cat. No.99TH8450)*, pages 85–89, 1999. 45

[LM95] YTS Li and S Malik. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Notices*, 1995. 110

[LMW95] YTS Li, S Malik, and A Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. *-Time Systems Symposium, 1995. ...*, 1995. 110

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997. 133, 134

[LS99] T Lundqvist and P Stenstrom. Timing anomalies in dynamically scheduled microprocessors. *Real-time systems symposium, 1999*, 1999. 134

[LSP08] MP Lawitzky, DC Snowdon, and SM Petters. Integrating real time and power management in a real system. *Proc. OSPERT*, 2008. 40, 42

[Lun02] T Lundqvist. A WCET analysis method for pipelined microprocessors with cache memories. 2002. 108, 134

[MBM01] W Matusik, C Buehler, and L McMillan. Polyhedral visual hulls for real-time rendering. *Rendering Techniques 2001*, 2001. 86

[MBRG00] W Matusik, C Buehler, R Raskar, and SJ Gortler. Image-based visual hulls. *Proceedings of the 27th*, 2000. 86

[MCE+02] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002. 92

[Mer03] J Merrill. Generic and gimple: A new tree representation for entire functions. *Proceedings of the 2003 GCC Developers' Summit*, 2003. 78

[MRR07] A Muttreja, A Raghunathan, and S Ravi. Automated energy/performance macromodeling of embedded software. *IEEE Transactions on*, 2007. 40

[MRR12] MD McCool, AD Robison, and J Reinders. Structured parallel programming: patterns for efficient computation. 2012. 69

[MSSS14] Pablo González Aledo Marugán, Luis Díaz Suárez, Álvaro Díaz Suárez, and Pablo Sánchez. Profiling and optimizations for embedded systems. In *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*, pages 194–197. IEEE, 2014. 52, 169

[NBZ+15] Luigi Nardi, Bruno Bodin, M Zeeshan Zia, John Mawer, Andy Nisbet, Paul HJ Kelly, Andrew J Davison, Mikel Luján, Michael FP O'Boyle, Graham Riley, et al. Introducing slambench, a performance and accuracy benchmarking methodology for slam. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 5783–5790. IEEE, 2015. 175

[ncs] Incisive enterprise simulator. `https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/incisive-enterprise-simulator-ds.pdf`. 83

[Net04] N Nethercote. Dynamic binary analysis and instrumentation. 2004. 9, 10

[NHGW15a] Philipp Niemann, Frank Hilken, Martin Gogolla, and Robert Wille. Assisted Generation of Frame Conditions for Formal Models. In *Design, Automation and Test in Europe*, pages 309–312, 2015. 151

[NHGW15b] Philipp Niemann, Frank Hilken, Martin Gogolla, and Robert Wille. Extracting frame conditions from operation contracts. In *Int'l Conf. on Model Driven Engineering Languages and Systems*, pages 266–275, 2015. 151

[NM01] A Nandi and R Marculescu. System-level power/performance analysis for embedded systems design. *Proceedings of the 38th annual Design*, 2001. 40

[NNH99] F Nielson, H Riis Nielson, and C L Hankin. *Principles of Program Analysis*. Springer, 1999. 123

[NR00] M Narasimhan and J Ramanujam. On lower bounds for scheduling problems in high-level synthesis. *Proceedings of the 37th Annual Design*, 2000. 78

[Obj14] Object Management Group. *Object Constraint Language*, 2014. 151, 152

[OKS+08] Neal Orman, Hansung Kim, Ryuuki Sakamoto, Tomoji Toriyama, Kiyoshi Kogure, and Robert Lindeman. Gpu-based optimization of a free-viewpoint video system. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, page 15. ACM, 2008. 90

[OKST08] N Orman, H Kim, R Sakamoto, and T Toriyama. GPU-based optimization of a free-viewpoint video system. *Proceedings of the*, 2008. 90

[PAS12] Jesús M Perez, Pablo G Aledo, and Pablo P Sanchez. Real-time voxel-based visual hull reconstruction. *Microprocessors and Microsystems*, 36(5):439–447, 2012. 86, 87, 88, 170

[PDV11] H Posadas, L Díaz, and E Villar. Fast data-cache modeling for native co-simulation. *Proceedings of the 16th Asia and South*, 2011. 42, 45, 63

[PLL$^+$] Jan Peleska, Florian Lapschies, Helge Löding, Peer Smuda, Hermann Schmid, Elena Vorobev, and Cornelia Zahlten. Turn Indicator Model Overview. 159, 161

[PP11] Pierre Paulin and Pierre. Programming challenges & solutions for multi-processor SoCs. In *Proceedings of the 48th Design Automation Conference on - DAC '11*, page 262, New York, New York, USA, 2011. ACM Press. 59, 62

[PPB07] G Paci, F Poletti, and L Benini. Exploring temperature-aware design in low-power MPSoCs. *International journal of*, 2007. 43, 92, 96, 97

[QR11] X Qu and B Robinson. A case study of concolic testing tools and their limitations. *Empirical Software Engineering and*, 2011. 108

[RWT06] J Reineke, B Wachter, and S Thesing. A definition and classification of timing anomalies. *OASIcs-*, 2006. 134

[SCB16] AM Sloane, F Cassez, and S Buckley. The sbt-rats parser generator plugin for Scala (tool paper). *Proceedings of the 2016 7th ACM*, 2016. 30

[Sha89] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989. 110

[SKV13] Anthony M. Sloane, Lennart C L Kats, and Eelco Visser. A pure embedding of attribute grammars. *Science of Computer Programming*, 78:1752–1769, 2013. 30

[SLD$^+$03] H Su, F Liu, A Devgan, E Acar, and S Nassif. Full chip leakage estimation considering power supply and temperature variations. *Proceedings of the 2003*, 2003. 43, 93

[SMA05] K Sen, D Marinov, and G Agha. CUTE: a concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*, 2005. 116

[SOF00] E SOFlWARE. ARM System-on-Chip Architecture. 2000. 50

[SPS$^+$02] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 51:02110–1301, 2002. 139

[SSS04] K Skadron, MR Stan, and K Sankaranarayanan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions on*, 2004. 92

[SW02] Connie U Smith and Lloyd G Williams. Introduction to software performance engineering. In *Int. CMG Conference*, pages 7–14, 2002. 5

[SWD11a] Mathias Soeken, Robert Wille, and Rolf Drechsler. Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In *Tests and Proof*, pages 152–170, 2011. 155, 158

[SWD11b] Mathias Soeken, Robert Wille, and Rolf Drechsler. Verifying Dynamic Aspects of UML models. In *Design, Automation and Test in Europe*, pages 1077–1082, 2011. 148, 150, 159

[SWD11c] Mathias Soeken, Robert Wille, and Rolf Drechsler. Verifying Dynamic Aspects of UML Models. In *Design, Automation and Test in Europe*, pages 1077–1082, 2011. 149

[SWK+10] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using Boolean satisfiability. In *Design, Automation and Test in Europe*, pages 1341–1344, 2010. 149

[TR01] TK Tan and A Raghunathan. High-level software energy macro-modeling. *Design Automation*, 2001. 40, 48

[VB14] AC Velivelli and KM Bryden. Domain decomposition based coupling between the lattice Boltzmann method and traditional CFD methods—part I: formulation and application to the 2-D Burgers'. *Advances in Engineering Software*, 2014. 61

[VK] Asai Vladimirov and Karpusenko. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. 67

[Wei84] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984. 27

[WMMR05] N Williams, B Marre, P Mouy, and M Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. *European Dependable*, 2005. 122

[YLB+08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable Shape Analysis for Systems Code. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 385–398, Berlin, Heidelberg, 2008. Springer-Verlag. 109

[YZA+09] Yuang Zhang, Zhonghai Lu, Axel Jantsch, Li Li, and Minglun Gao. Towards hierarchical cluster based cache coherence for large-scale network-on-chip. In *2009 4th International Conference on Design & Technology of Integrated Systems in Nanoscal Era*, pages 119–122. IEEE, apr 2009. 59

[ZZWY16] S Zhang, T Zhang, Y Wu, and Y Yi. Flow simulation and visualization in a three-dimensional shipping information system. *Advances in Engineering Software*, 2016. 69