Precise and Adaptable Worst-Case Execution Time Estimation in Hard Real-Time Systems

Vladimir-Alexandru Paun

Precise and Adaptable Worst-Case Execution Time Estimation in Hard Real-Time Systems

A dissertation presented

by

Vladimir-Alexandru PAUN

to

The Department of UIIS, ENSTA ParisTech

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Ecole Polytechnique

Palaiseau, France

December 2014

PhD thesis defended the 18th of December 2014 in front of the examining committee:

Prof. Liliana Cucu-Grosjean	**Referee**
INRIA Researcher	
Dr. Renaud Sirdey	**Referee**
CEA Director of Research	
Prof. Bruno Monsuez	**Co-Supervisor**
ENSTA ParisTech	
Prof. Michel Mauny	**Co-Supervisor**
ENSTA ParisTech	
Mr. Philippe Baufreton	**Examiner**
Sagem Senior Scientist	
Dr. Franck Vedrine	**Examiner**
CEA Research Engineer	
Prof. Kamel Barkaoui	**Examiner**
CNAM, VESPA Research Team Leader	
Prof. Florin Popentiu	**Examiner**
UPB, City University London, UNESCO Chair	

Thesis advisors
Prof. Bruno Monsuez
Prof. Michel Mauny

Author
Vladimir-Alexandru PAUN

Precise and Adaptable Worst-Case Execution Time Estimation in Hard Real-Time Systems

Abstract

Nowadays real-time systems are omnipresent and embedded systems thrive in a variety of application fields. When they are integrated into safety-critical systems, the verification of their properties becomes a crucial part. Dependability is a primary design goal in environments that use hard real-time systems, whereas general-use microprocessors were designed with a high performance goal. The average-throughput maximization design choice is intrinsically opposed to design goals such as dependability that benefit mostly from highly deterministic architectures without local optimizations.

Besides the growth in complexity of the embedded systems, platforms are getting more and more heterogeneous. With regard to the respect of the timing constraints, real-time systems are classified in two categories: hard real-time systems (the non respect of a deadline can lead to catastrophic consequences) and soft real-time systems (missing a deadline can cause performance degradation and material loss). We analyze hard real-time systems that need precise and safe determination of the worst-case execution time bounds in order to be certified. The validation of their non-functional properties is a complex and resource consuming task. One of the main reasons is that currently available solutions focus on delivering precise estimations through tools that are highly dependent on the underlying platform (in order to provide precise and safe results, the architecture of the system must be taken into account).

In this thesis we address the above issues by introducing a timing analysis method that maintains a good level of precision while being applicable to a variety of platforms. This adaptability is achieved through separating as much as possible the worst-case execution time (WCET) estimation from the model of the hardware. Our approach consists in the introduction of a new formal modeling language that captures the complex behaviour of modern hardware and is guided by the timing analysis in order to achieve the needed pre-

cision to scalability tradeoff. The analysis drives a conjoint symbolic execution of the program's binary and the processor model using a dynamic prediction module that decides what states to merge in order to limit the state space explosion. Several state merging algorithms are introduced and applied that can also give an estimation of the introduced precision loss.

Keywords: real-time systems, worst-case execution time, timing analysis.

Directeur de Thèse
Prof. Bruno Monsuez
Prof. Michel Mauny

Auteur
Vladimir-Alexandru PAUN

Détermination des Pire-temps d'Exécution des Systèmes Temps Réels Durs

Abstract

La détermination précise de pires temps d'exécution (WCET) est un sujet de grand intérêt pour les systèmes embarqués critiques. Le sujet de la Thèse adresse des problèmes qui ne sont pas résolus dans la littérature notamment l'inertie notable dans le changement de plateformes cibles des analyseurs existants et la perte de précision lors du passage à l'échelle. Nos travaux se concentrent justement sur une souplesse au changement du processeur à analyser et la maitrise de la perte de précision à l'aide d'un nouvel langage de modélisation du matériel qui se trouve en étroit lien avec l'analyseur même.Une estimation sûre du WCET nécessite la prise en compte du matériel sur lequel le programme est exécuté. Les processeurs embarqués dans les systèmes critiques présentent des composants qui ont été conçus non pas pour faciliter leur analyse mais pour maximiser les performances moyennes, introduisant une variabilité temporelle significative. Le temps d'exécution est donc dépendent, entre autres des valeurs effectives de données mais aussi de l'historique d'exécution. Le but étant d'estimer le pire temps d'exécution, l'option de supposer qu'à chaque fois l'optimisation ne se produit pas et que le pire temps possible est nécessaire conduit à une surestimation trop importante. A ce problème se rajoute aussi le fait que nous ne pouvons pas supposer qu'un pire temps local (par exemple le nombre de cycles pendant une micro-opération du pipeline) contribuera au vrai pire temps global à cause des anomalies temporelles des processeurs. Tous les chemins d'exécution, engendrés par la totalité des entrées possibles doivent donc être analysés. Le fait de devoir gérer cette explosion combinatoire, nous a guidé dans les choix de conception du modèle utilisé pour simuler le processeur. Nous avons conçu une méthode de spécification et d'analyse de systèmes, basée sur la méthode des abstract state machines (ASM). L'extension HiTAsm

consiste en l'incorporation des notions de hiérarchie et de temporalité, pour pouvoir gérer l'explosion combinatoire en choisissant une définition d'un composant parmi plusieurs niveaux d'abstraction possibles et pour pouvoir estimer le temps écoulé entre chaque transition des états du système. Cela permet l'estimation de la consommation temporelle et la variation dynamique du niveau d'abstraction du système analysé, afin d'analyser un grand nombre d'états, tout en minimisant la perte de précision. Le modèle du processeur et l'analyseur sont complètement séparés, ce qui est nécessaire pour rendre l'outil adaptable aux changements de plateforme. L'analyseur est basé sur une exécution symbolique conjointe du binaire et du modèle de processeur spécifié avec HiTAsm. En partant des valeurs symboliques, toutes les entrées possibles du binaire sont analysées, en utilisant des sur-approximations uniquement quand c'est nécessaire de manière à minimiser la perte de précision et fournir le pire-temps d'exécution du programme le plus proche de la valeur théorique.

Mots-clés: pire-temps d'exécution, analyse statique, temps réel, sûreté, certification.

Contents

List of Figures

List of Tables

Citations to Previously Published Work

Large portions of Chapter 3 have appeared in the following two papers:

> **V.-A. Paun**, B. Monsuez, P. Baufreton, ***Hierarchical Timed Abstract State Machines for Hard Real-Time Embedded Processors***,
> Verification and Evaluation of Computer and Communication Systems (VECoS), 2013.

> **V.-A. Paun**, B. Monsuez, P. Baufreton, ***Hierarchical Timed Symbolic Abstract State Machines for precise WCET estimation***,
> WiP Session of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013.

Parts of Chapter 6 and 7 have appeared in the paper:

> **V.-A. Paun**, B. Monsuez, P. Baufreton, ***On the Determinism of Multicore Processors***,
> French Singaporean Workshop on Formal Methods and Applications (FSFMA), 2013.

> **V.-A. Paun**, B. Monsuez, P. Baufreton, ***Adaptable and Precise Worst Case Execution Time Estimation Tool***,
> WiP Session of Languages, Compilers, Tools and Theory for Embedded Systems (LCTES), 2012.

A publication on the material presented in Chapter 3 (together with B. Monsuez, P. Baufreton) has been accepted for publication and will be published in the first issue of the following journal:

> **International Journal of Critical Computer-Based Systems (IJCCBS)**.

Acknowledgments

First of all, I would like to thank Professor Bruno Monsuez for his continuous scientific and technical support offered throughout the course of the PhD thesis. He provided me with enough freedom for approaching my goals and guidance to complete this thesis. I appreciated the excellent opportunity to work in his research group, an elevated professional environment and an enthusiastic creative ambiance.

I wish to thank Professor Michel Mauny, my thesis co-supervisor, for the critical constructive discussions designed to improve the quality of this thesis.

It is my pleasure to thank Professor Liliana Cucu-Grosjean, and Dr. Renaud Sirdey, for accepting to review my PhD thesis as well for their valuable remarks and subtle suggestions. I extend my thanks to the other members of the jury, Dr. Franck Vedrine and Professor Kamel Barkaoui.

The PhD thesis work was carried out in the CIFRE framework of a project supported and funded by the Association Nationale de la Recherche et de la Technologie (ANRT), with an industrial partner (Sagem - SAFRAN Electronics). I thank the Research Program Manager, Mr. Philippe Baufreton, from Sagem for the interest, good collaboration and expert advices that offered me a thorough and valuable industrial insight which completed my vision and knowledge of the subject.

In addition I would like to thank Mrs. Christine Ferret for her help since my enrolment to the Ecole Doctorale Polytechnique to the present. Also, I thank Mrs. Catherine Le Golvan and Mr. Thierry Pouliquen, for their kind help in various administrative situations over the years as well as for their great human qualities.

Also, a friendly consideration goes to my colleagues whose names were not explicitly mentioned. Sincerely I confess that they occupy the same central place in my memories as the other.

Not least, I mention here the opportunity offered to do a PhD thesis at ENSTA ParisTech, an university with international prestige and high scientific level. I consider it an extraordinary experience.

Finally, I express my entire gratitude to my family. My parents, my sister and my grandparents have been really supportive all the time and they represent a wonderful team.

Dedicated to my father Viorel,
my mother Jenica,
and my sister Maria.

Chapter 1

Introduction

With regard to the respect of the timing constraints, real-time systems are classified in two categories namely hard real-time systems (the non respect of a deadline can lead to catastrophic consequences) and soft real-time systems (missing a deadline can cause performance degradation and material loss). We hereby analyze hard real-time systems that need precise and safe determination of the worst case execution time (WCET) bounds that are crucial in the certification process. Two approaches exist for the timing analysis, namely dynamic and static methods, [WEE+08]. We propose a static method in order to deliver safe estimations for modern processors that contain, for example, pipelines or cache memories that make the analysis challenging.

In order to give a safe estimation of the WCET, all the interactions and reachable states of the system must be considered or over approximated, hence the need of an analysis that takes into account the exact underlying architecture. We also focus on the separation, as much as possible, of the system model from the analysis part in order to achieve the flexibility needed to adapt to new hardware.

In our approach we start from the system's model and the binary that will be executed on the final platform. An extension of the Symbolic Execution (SE), [LS99a], the *conjoint* SE, will generate all the reachable states of the processor, under the supervision of a *prediction module* that will *fusion* identical and similar states in order to contain the state space explosion and give details regarding the global precision loss of the WCET estimation.

In the following we first take a look into the state of the art concerning timing analysis and

we continue with the description of the high level architecture of our tool. Subsequently we take a closer look into the formal model used to simulate the hardware that gives us the edge in the adaptability of our tool followed by a presentation of the WCET estimation steps and the transformations needed to contain the combinatorial explosion.

1.1 WCET and hard real-time systems

1.1.1 Real-time systems

The increased use of computers to control safety-critical real-time functions in the past decade has made their study the focus of numerous research. A real-time system can be seen as an information processing system which must respond to external inputs within a finite and specified period, [You82]. The concept of *time* is essential in real-time application systems, and since such systems involve sharing resources among various processes, the concept of *scheduling* is equally important.

Definition 1 (Real-time systems) The computer systems where the correctness of a computation is dependent on both the logical results of the computation and the time at which these results will be produced are called *real-time systems*.

A real-time system usually consists of a number of real-time *tasks* that generate *jobs* in a predictable manner. These tasks and jobs have upper bounds upon their worst-case execution requirements, and associated deadlines, by which all execution must be completed. As we already pointed out, the consequences of a task missing its time bounds may vary from task to task and are often expressed as the criticality of the task.

Definition 2 (Real-time task) A real-time task is a task with real-time constraints associated with it.

Definition 3 (Jobs) Let $j = (t_a, e, t_d)$ be a real-time job, characterized by the three parameters - arrival time t_a, execution requirement e, and a deadline t_d, with the interpretation that the job consumes e execution units over the interval $[t_a, t_d)$ and let J be a finite or infinite collection of jobs with $\mathrm{J} = \{j_1, j_2, \cdots\}$.

A number of different types of deadlines can be associated with real-time systems:

- a *hard deadline* is critical to the operation of the system as a whole - missing one can result in failure;
- a *firm deadline* must be met, and failure to do so will render the result of execution useless therefore wasting resources;
- a *soft deadline* is less stringent, a late result will not be as valuable but may still be useful.

Based on the classification of deadlines, we can distinguish between real-time systems:

- *hard real-time system* - systems with safety-critical constraints - missed deadlines may be catastrophic;
- *soft real-time system* - missed deadlines reduce the value of the system.

Hard real-time systems span many application areas like space, nuclear, avionics, automotives, process control, robot systems, just to mention a few. The logical and timing correctness must be explored or proven if possible during the design implementation and test passes and additionally actions must be taken to handle run-time that may occur from transient or permanent hardware errors. A dependability requirement of the hard real-time system imposes that the system be kept in a *safe state*.

Definition 4 (Dependability). Dependability is a measure of a system's availability, reliability, and its maintainability.
The dependability states that a system should remain fully operational even after a permanent hardware fault occurs and remain in a safe state even if a second fault occurs. The probability of failure of a dependable system must be tolerable.

Hard real-time systems must undergo a certification process that will assign further *safety insurance levels* to each function of the system based on the gravity of the consequences of a deadline miss.

Failure in hard real-time systems has unacceptable consequences, therefore the task must execute correctly and usually their deadline cannot be exceeded without resultant failure. Scheduling has as its aim the correct allocation of temporal computational resources between competing tasks in a system so that each task will perform its computational overhead within a deadline. The confirmation that this is possible means that the system is schedulable, otherwise not.

Definition 5 (Schedule) Let S be a schedule for any collection of jobs J defined as:

$$S : R \times J \rightarrow \{0, 1\},$$

a mapping from the cartesian product of the real numbers and the collection of jobs to $\{0, 1\}$, where $S(t, j)$ equal to one if schedule S assigns the processor to job j at time-instant t, and zero otherwise.

Real-time task scheduling essentially refers to determining the order in which the various tasks are to be executed by the operating system. Operating systems rely on one or more schedules to prepare the execution schedule of various tasks that it needs to run.
A schedulability analysis must be performed in order to assess certain properties of the tasks and their behaviour in the system. This implies that the execution time of the individual tasks of the system must be determined. In order to consider the worst possible combination of features that can take place, the guarantees must be pessimistic. Therefore, if the worst-case occurs there will be enough time in the schedule to ensure completion of all tasks.
Schedulers are characterized by the scheduling algorithm they use. A popular scheme classifies the real-time task scheduling algorithms based on the definition of the scheduling points:

1. Clock Driven:
    - Table-driven
    - Cyclic
2. Event Driven:

- Simple priority-based
- Rate Monotonic Analysis
- Earliest Deadline First

3. Hybrid
   - Round-robin

An example of a highly used scheduler in the industry is the cyclic scheduler. It gets its popularity thanks to its simplicity, efficiency and the programming ease.
Cyclic scheduling is static, computed offline and stored in a table, therefore it repeats a precomputed schedule that is stored for only one *major cycle*. Each task in the task set J to be scheduled repeats identically in every major cycle that is divided into *minor cycles* also called *frames*, Figure. 1.1.

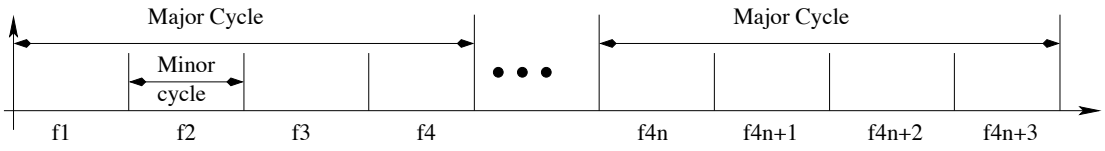


Figure 1.1: Major and minor cycle (frame) in a Cyclic Scheduler

The frame boundaries are defined through interrupts generated by a periodic timer, a frame being assigned to run in one or more frames following a *schedule table* as shown in table 1.2.
An important design parameter of the scheduler is the size of the frame that needs to be carefully chosen in order to satisfy the following three constraints:

- **Minimum context switching.** Ensures that a minimum number of context switches occur during the task execution. This means that the task instance must complete execution within its assigned frame. We can formally express this constraint as: $max(\{e_i\}) \leq F$ where e_i is the execution time of the task T_i, and F is the frame size.
- **Minimization of table size.** Requires that the number of entries should be minimal in order to minimize the storage requirements of the schedule table.

- **Satisfaction of task deadline.** This constraint is necessary to ensure the task meets its deadline. It imposes that between the arrival and the deadline of the task at least one full frame must be available.

Task Number	Frame Number
T_3	F_1
T_1	F_2
T_3	F_3
T_4	F_2

Figure 1.2: Schedule Table for Cyclic Scheduler

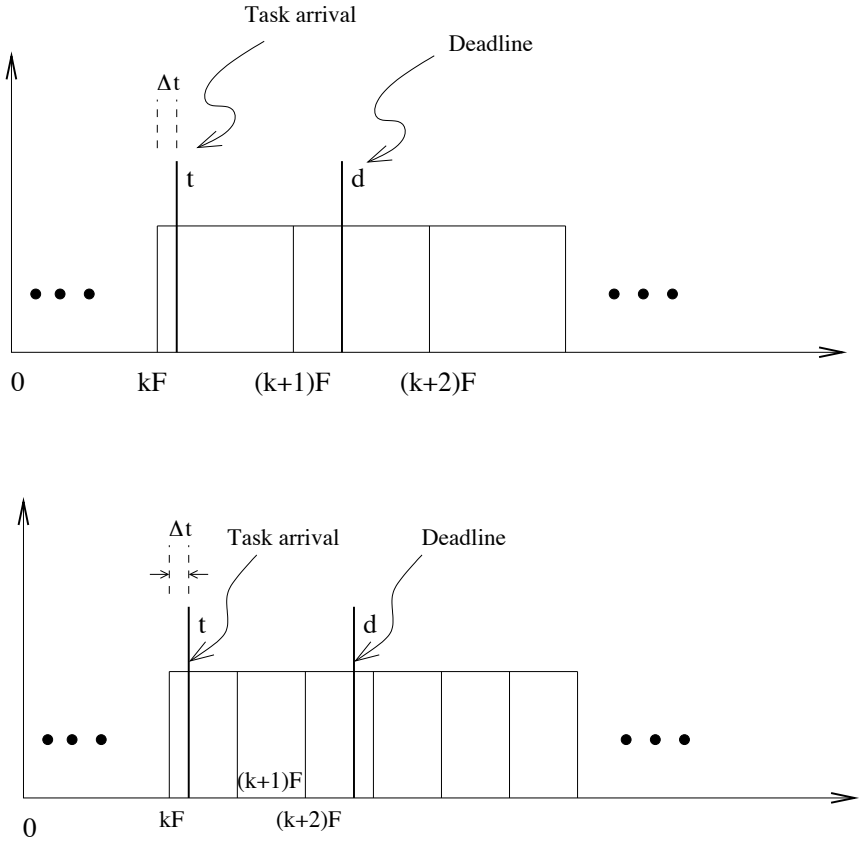


Figure 1.3: A Full Frame exists between the Arrival and Deadline of the Task

As we can see in the scheduler example above, the execution time of a task (expressed as e, the execution requirement in Definition 3) is crucial in order to ensure that the deadline is expected and to design a valid schedule of the task collection. In the following we introduce the execution time and timing analysis of real-time systems.

1.1.2 Worst-case execution time

The *execution time* (ET) of a task is the amount of computation time required to obtain its result. For simplicity, the original scheduling techniques assumed that ET did not vary - considered to be fixed for each executing task. Since this is a severe restriction there is an obvious way to extend this model. By assuming that a task with variable ET has at least a finite maximum which it can take we can therefore use the value of this *worst-case execution time* (WCET) in the analysis instead of a constant execution time.

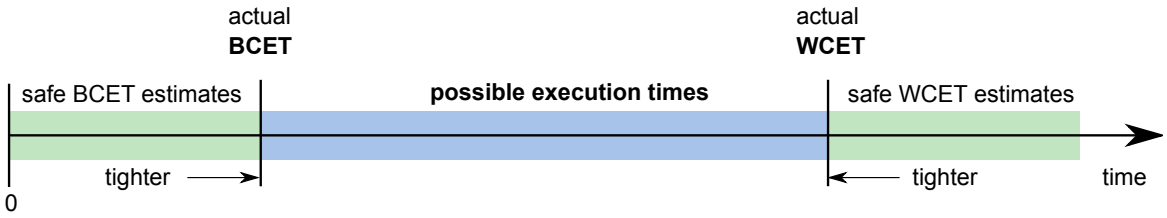


Figure 1.4: Execution time of a task

Definition 6 (WCET) The worst-case execution time is an upper bound on the greatest time that a task will take to perform its execution when it has sole access to all computational resources and under the complete set of all possible environmental conditions.

When the WCET value is used and the time frame in the scheduler is chosen accordingly, no task will exceed this value and the deadlines will be met. The obvious drawback is that computational resources will be wasted whenever the task will take less than the WCET. Moreover, as we will show in the following, WCET can only be estimated and not computed because of the nature of the underlying hardware in the real-time system.
The execution time of a task is no longer a constant value since the introduction in embedded systems of processors featuring modern architectural units. These units were conceived to improve the average execution time with little to no regard to the worst-case.

The heuristics employed take advantage of data locality and benefit from certain hardware configurations in order to gain processing time by using either best effort techniques in pipelining resources or complex cache replacement policies, [Rei09; HLTW03].
The value of the WCET estimation is of great importance in real-time systems having multiple uses:

- Temporal validation
- Schedulability analysis schedulability guarantees (worst-case)
- System dimensioning Hardware selection
- Optimization of application code
- Early in application design lifecycle

Another useful concept in real-time systems is the worst-case response time employed, for example, to determine whether a system model is schedulable.

Definition 7 (Worst-case response time). The worst-case response time is defined as an upper bound on the greatest time duration starting from the occurrence of the event generating the task (task arrival time) until the time the task produces its results.

Let S be a system defined by the collection of tasks $\bigcup_{i=1}^{n} T_i$. For a given task T_i if $RT < t_d$ meaning that the response time is less than its deadline then T_i is schedulable. If all of the tasks in a system are schedulable, T_i is schedulable $\forall i \in [0 \cdots n]$, then the system S is said to have a schedulability guarantee [LL73b].
The link between the WCRT and the WCET and its importance in the case of the scheduling analysis can be seen in the following equation which gives the WCRT r_i of a task i through a worst-case number of interrupts by all higher-priority task $j \in hp(i)$:

$$r_i = C_i + \sum_{\forall j \in hp(i)} n_j(\tau_i) \times C_j, \tag{1.1}$$

where $n_j(\tau_i)$ is the maximum number of activated events for the task j during r_i and C_i, C_j are the WCET (therefore assuming no interrupts of task i and j. We generally compute r_i

iteratively, as shown in the following algorithm, [JHE04].

Algorithm 1: Iterative WCRT computation algorithm

Data: T_{low}

Result: $t_{newResp}$, the new response time and the worst-case response time, r_i

```
1 t_newResp ⟵ T_low.WCET;
2 while t_newResp > t_oldResp do
3     t_oldResp ⟵ t_newResp;
4     for i ← 0 to T_high List.size do
5         T_high ⟵ HighPriorTaskList.get(i);
6         MaxActivations ⟵max number of activations of T_high during t_oldResp;
7         T_interr ⟵ MaxActivations ∗ T_high.WCET;
8         t_newResp ⟵ t_newResp + T_interr;
```

1.1.2.1 Estimation of Execution Time

Several types of execution time measures can be used in order to describe the timing behaviour of a system as shown in Figure 1.5.

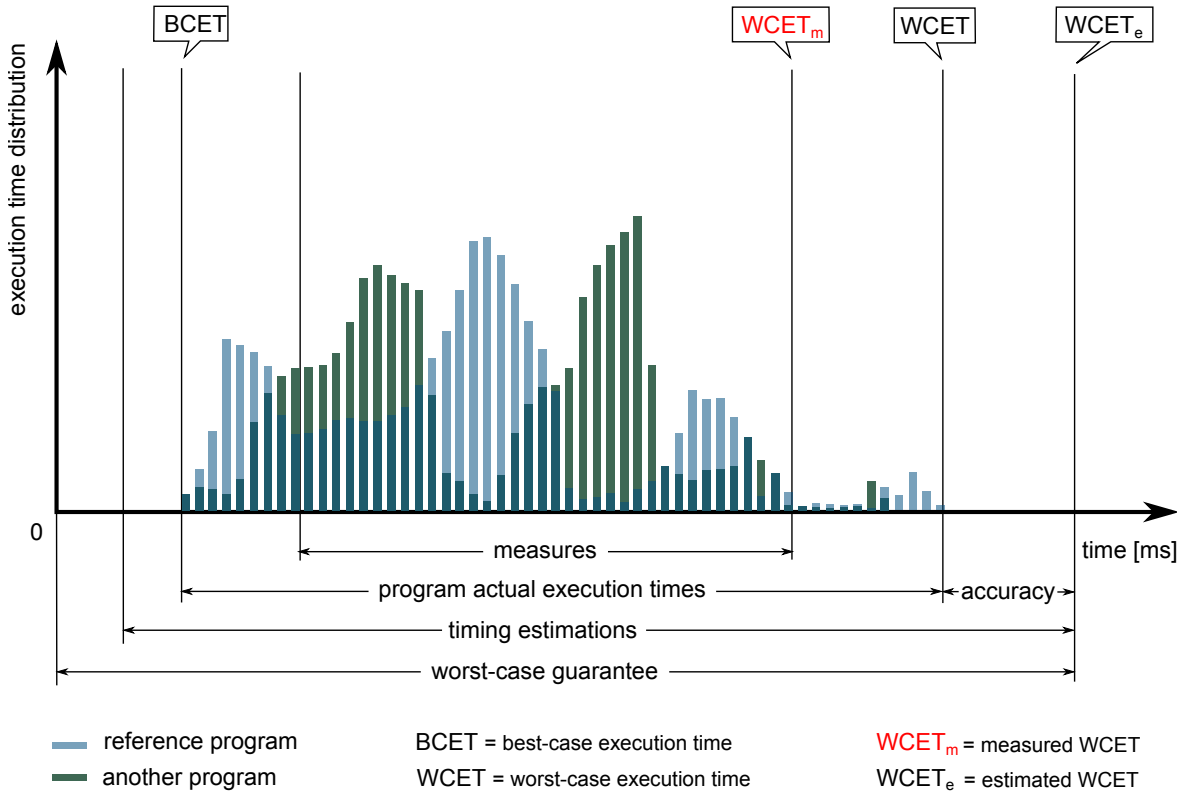


Figure 1.5: Distribution of execution times

The worst-case execution time is the longest possible execution time of the program when it runs in its production environment. At the opposite spectrum, the *best-case execution time* (BCET) is the shortest execution time that a program takes to execute. BCET can sometimes be estimated with the same methods as the WCET just by changing the goal function in the optimisation problem of the path analysis. Besides giving an idea of the minimum computational requirement of a system, the BCET can be used in reducing the pessimism of the schedulability analysis by reducing the estimation of jitter in the activation times of tasks and messages for certain scheduling algorithms. The *average-case execution time* (ACET) lies somewhere in-between the WCET and the BCET, and depends on the execution time distribution of the program. A low ACET would indicate that the algorithm generally incurs low computation overheads. This measure is more important in embedded systems with weaker timing constraints with soft real-time applications, since they quantify the achievable sustained throughput (number of processed images/detected

objects per second by an image-processing system, number of requests handled by a server per time unit, etc.).

Modern processors introduce more and more heuristic methods in order to improve the ACET. This means that in order to determine the WCET or the BCET, one must take into account the interactions between the software and hardware meaning between all the possible program states and all the possible hardware states (configurations) for all possible inputs.

Exploring all reachable system's states leads to a combinatorial problem through the inherent state space explosion. The problem of determining the WCET is twofold. On one hand measurements will not provide the guarantee of an ET over-approximation, because it will simply consist in giving one of the different possible times from the distribution shown in Figure 1.5 and on the other hand exploring all reachable states in order to determine the worst-case is impossible.

Therefore timing analysis aims to provide estimates of both WCET and BCET. One of the essential constraints imposed to the estimate is for it to be *safe* which means for the worst-case that an upper bound should be determined and a lower one for the best-case. Any other estimates are unsafe, for example an underestimation of the WCET will lead to an unsafe scheduler that will take into account a deadline inferior to the worst-case might stop the task before the outputs are obtained. Therefore subsequently scheduled tasks that use that result as an input will be executed with outdated values and produce incoherent outputs.

The safety provided by the overestimation is crucial in designing hard real-time system, however in order for an estimate to be useful it must also have a low degree of pessimism, and be as *tight* as possible. In other words the upper-bound must be as close as possible to the actual value, which will guarantee the efficient use of, often scarce, resources ensuring the greatest utilization of computation possible.

1.1.2.2 The Use of WCET Estimates

As previously stated the role of the WCET is of great importance for schedulability analysis and scheduling [LL73a; ABD+95; CRTM98] however WCET estimates have a much broader application domain, mainly whenever timing is important.

The WCET estimation can be used to identify which are the most resource consuming tasks and to focus the development effort in order to reduce the computational overhead. Tools for modeling and verifying systems represented as timed automata, like Uppaal [LPY97] can benefit from the WCET estimates in order to obtain timing values from the real implementation of a system [BPPS00].
The WCET estimation is also used in the feasibility test of the project and in selecting the appropriate hardware to embed in the final system. Making sure that the hardware platform is not under-dimensioned ensures that the feasibility test will pass. On the other hand, ensuring that the system is not too over-dimensioned can be of a great financial importance. Based on the WCET estimation the system can also be tuned, like for example adjusting the clock frequency of the selected processor.

1.1.2.3 Calculation of WCET Estimates

As inferred in the above sections, a key feature of the real-time system is the value of the WCET that is used in the construction and analysis of a schedule. The schedulability of a system depends on all variables of the system, where deadline and period are usually fixed in the specification. The calculation of computation time is more challenging, mainly due to two assumptions that are made to obtain its value, over-approximation and tight estimation.
Traditionally, methods from two main families are used namely *dynamic methods* and *static methods*. The justification behind the use of dynamic methods revolves around facts like - the best model of the system is the actual system itself and that the worst possible input configuration will lead to the worst possible outcome. Therefore timing measurements can be performed, with the use of different techniques, in order to obtain the execution time while keeping track of the worst encountered value (technique called *high-water marking*). Indeed, running the program with the worst-case input may select the worst-case path in the program's execution, however finding them requires expert-level knowledge of the system and the actual executed code. Identifying these particular inputs could also be made through an exhaustive exploration of the refined input state space. Nowadays, this kind of approach is becoming hard to implement as the input set is probably infinite and the complexity of the

code increases. However one of the most important drawbacks is that hardware has evolved in such a way that the WCET value is highly dependent on the hardware architecture, and execution history, therefore the same code executed for the same input values may generate different execution times.

Statement 1 *The execution time is history-sensitive, depends on the execution state and cannot be determined in isolation.*

The architectural hardware features that introduce heuristics in the execution time determination will be introduced in the next sections and throughly analyzed throughout the following chapters. Dynamic methods try to eliminate some of the difficulties by usually adding a *safety margin* to the worst measured execution time in order to hopefully capture the real WCET. Nevertheless, no guarantee that the real WCET will be found can be given, measurements can only produce statistical evidence regarding the WCET, with no complete certainty, [Gan01].
In order to obtain a safe estimations of the WCET, mathematically founded method like the *static analysis* should be used. Safety is achieved through the analysis of all possible program behaviors. As opposed to the dynamic methods, the program is not executed but analyzed and the inferred properties characterize all possible outcomes. A hardware model is generally used in order to precisely take into account the software-hardware interactions. The main technique in order to achieve safe results and formulate answers about the program properties is through the use of abstractions. Therefore we will no longer consider only reachable architectural states but broaden the exploration set also with unfeasible paths. The advantage is that we search to transpose the problem into a space where the properties will be computable. In the widening of the exploration space lies the explanation of the safety and also of the precision loss. Indeed new paths are added, however none of the existing feasible paths are disconsidered and they will all be analyzed. As only the greatest value of the execution time is retained, only a possible over-approximation can be made. The difference between the real WCET and the computed value gives the imprecision of the method.

1.1.2.4 On the Characterization of Estimations

In order to evaluate the quality of the WCET estimation or measure we employ the terms *safeness* and *tightness*.
The tightness is a well known concept in the theory of measures, defined for different types of measures like the regular one. In mathematics, a regular measure on a topological space is a measure for which every measurable set can be approximated from above by open measurable sets and from below by compact measurable sets, [Bil99].

Definition 8 (Measure (regular).) Let (X, T) be a topological space and let Σ be a σ-algebra on X. Let μ be a measure on (X, Σ). A measurable subset A of X is said to be inner regular if

$$\mu(A) = \sup\left\{\mu(F) \mid F \subset A, F \textit{ compact and measurable}\right\}$$

and said to be outer regular if

$$\mu(A) = \inf\left\{\mu(G) \mid G \supset A, G \textit{ open and measurable}\right\}$$

We therefore define the tightness of measure as follows, [Bil95]:

Definition 9 (Tightness of measures.) Let (X, T) be a topological space, and let Σ be a σ-algebra on X that contains the topology T. (Thus, every open subset of X is a measurable set.) Let M be a collection of measures defined on Σ. The collection M is called tight if, for any $\epsilon > 0$, there is a compact subset K_ϵ of X such that, for all measures μ in M,

$$|\mu|\,(X \setminus K_\epsilon) < \epsilon,$$

where $|\mu|$ is the total variation measure of μ.
The safeness of a WCET estimation demands that the result of the analysis be over-approximations so that no possible execution time will be greater than the estimation. We therefore define the upper bound of an ordered set in the following:

Definition 10 (Upper bound.) An upper bound of a subset S of some partially ordered set $(K, \leq)$ is an element of K which is greater than or equal to every element of S. Let $u \in K$ such that $\forall x \in S \subset K.x \leq u$, then u is called the upper bound of the subset K.

Properties By transitivity, any element $e \in K.e \geq u$ is again an upper bound of S. This leads to the consideration of least upper bounds (or suprema).
The bounds of a subset S of a partially ordered set K may or may not be elements of S itself. If S contains an upper bound then that upper bound is unique and is called the greatest element of S. The greatest element of S (if it exists) is also the least upper bound of the set S.
For introductory purposes we choose to define the *timed program state* in a general form. We throughly define later our interpretation of delayed transition in chapter 3.

Definition 11 (Timed Program State.) Let the tuple $s_t = (s, \delta)$ be the timed program state where $\delta \in \mathbb{T}$ is the time that state s_t takes to finalize its execution and $\mathbb{T}$ be the associated time set.

We introduce a special function called *time* that will return the execution time of a program state, defined by:

$$time : S_t \rightarrow \mathbb{T}, \tag{1.2}$$

where S_t is the set of timed states:

$$time\left(s_t^{(i)}\right) = \delta_i. \tag{1.3}$$

Definition 12 (Timed Program Run.) Let us define the run of a program as the union of all program states that will execute during that run, $\rho_t = \bigcup_i s_t^{(i)} = \{s_t, s_t', s_t'', \cdots\}$, starting with the initial state s at time t. If the set is finite, we say that the run terminates and a final state s_t^f exists.

We only analyze programs for which we have a proof of termination, therefore we always reach a final state $s_t^f = \left(s_f, \delta_f\right)$. We now want to characterize a program as the union of all possible paths that can ever be taken.
The program execution is parametrized by a context run $C = \{c \mid c \textit{ is an execution context}\}$ defined by the unit of all possible execution under all the possible input combination,

$$C = \bigcup_i c_i.$$

Given the fact that the run is finite, we can calculate its total execution time in a program context $c \in C$ represented by the value of the inputs:

$$ext\,(\rho_t, c) = \sum_{i=0}^{n} time\Big(s_t^{(i)}\Big). \tag{1.4}$$

Definition 13 (Program Paths.) Let us define the program paths as the set of all possible runs of program P

$$P = \bigcup_{i=0}^{f} \rho_t^{(i)}. \tag{1.5}$$

For the program paths P we can define the WCET as:

$$wcet(P) = max \bigcup_{i=0}^{f} ext\Big(\rho_t^{(i)}, C\Big). \tag{1.6}$$

As presented in section 1.1.2.3, the methods used to determine the WCET come from two main families: dynamic and static methods. The first one is a measurement-based method. The program will be run with many different inputs determined through code review and expertise and the greatest value of the execution time is kept.

Let M_d be the union of measures corresponding to the value of every measured program run,

$$M_d = \left\{ \delta_i \in \mathbb{T}.i \in (1 \ldots n) \mid \delta_i \in \bigcup_{i=0}^{f} \rho_t^{(i)}(c_i). \right\}, \tag{1.7}$$

where n is the number of measures and c_i a contexts from the tested execution contexts C_d.

Definition 14 ($WCET_{dynamic}$). Let $WCET_{dynamic}$ be the WCET of a program computed using dynamic methods.

$$WCET_{dynamic} = max(M_d) \tag{1.8}$$

The safety issues of dynamic methods Only trivial or program under serious constrains have a finite or a decent number of elements in the input set. We must therefore suppose that the cardinality of the measurement set, $|M_d| = n < |C|$, therefore

$$\exists c_i \in C. \forall c_j \in C_m, ext(\rho_t, c_i) > ext(\rho_t, c_j), \tag{1.9}$$

where C_m is the set of contexts of the measured runs.
We can therefore state that the dynamic methods are not safe. The industrial practice is however based on this kind of methods. *De facto* the value of the WCET is used only after a safety margin is added to the measured value:

$$WCET_d = WCET_{dynamic} + \delta_{safety}. \tag{1.10}$$

It is obvious that it is still impossible to give strong guaranties that this new value is a safe bound. However if sufficient constrains are applied to the software and the hardware, if the processor in simple enough and other internal specific practices are used the bound will be accepted in the certification process.
Nevertheless as the hardware is becoming more complex, the difficulty of measuring a value that is close enough to the WCET and also the choice of the safety margin is increasing. Therefore static methods are getting the spotlight having the safety property ensured by construction. These methods estimate the WCET value through an *upper bound approximation* defined as follows:

Definition 15 (WCET upper bound approximation). Let $WCET \leq C + \epsilon . \forall \epsilon > 0$. Then we can say that $WCET \leq C$, where WCET is the theoretical worst-case execution time of the program and C is the estimated value, with precision ϵ.

Proof 1 *The proof by contradiction is trivial. We will assume that $WCET > C$.*

- *There is a $\delta > 0$ such that $WCET > C + \delta$. For example, let $\delta = (WCET - C)/2$.*
- *As given, $WCET \leq C + \epsilon$ for all ϵ, so in particular $WCET \leq C + \delta$.*

We therefore have a contradiction, because $WCET \leq C + \delta$ and $WCET > C + \delta$ cannot both be true simultaneously.

By definition, static methods can only be imprecise on the safe side, [KF12], the accumulation of imprecision is additive, therefore $\epsilon \geq 0$.

1.1.3 Hardware Considerations in WCET Estimation

Hard real time systems are evolving in order to respond to the increasing demand in complex functionalities while taking advantage of newer hardware. Software development for safety critical systems has to comply with strict requirements that will facilitate the certification process. During this process, each part of the system is evaluated, requiring a certain level of assurance in order to provide confidence in the product. In particular there must be a level of confidence that the system behaves deterministically that may be based on functionality, resources and time. The success of system verification depends greatly on the capacity to determine its exact behavior. Nonetheless, hardware evolved in order to maximize the average computation power throughput, therefore modern architectural features of processors, like pipelines, cache memories and co-processors, make it hard to verify that all the needed properties are respected. Some of the units that generate behaviors that are hard to take into account can be deactivated, but it is not always easy to predict the impact on the performance. Nevertheless the features that cannot be disabled (such as the out of order execution or some nondeterministic crossbar access policies) must be analyzed and taken into account in the processor model and the timing estimation.
The hardware platform is a central point when analyzing a system. Therefore it is essential to dispose of a precise model of the processor in order to determine its effective behavior. Most of the available processors were not especially designed for the hard real-time systems. Data communication and synchronization between the different units are optimized for maximum throughput of executed instructions. Therefore multiple execution paths can be taken depending on the execution history, the current state of the processor or even local choices based on random decisions.

1.1.4 Certification

The use of complex computers in safety-critical systems creates the need to ensure that the embedded systems act in the way they are supposed to and that consequences of a malfunction are completely handled in a safe manner. Different standards apply according to the danger level of the system failure. These standards are presented in a collection of guidelines to follow in order to empower the system with a necessary confidence level. The

respect of these recommendations determine the success of the certification process necessary for the software approval.
The WCET estimation method presented in this thesis WCET was developed baering in mind the avionics standards of software certification. For the assurance of commercial avionics systems a document called ”Software Considerations in Airborne Systems and Equipment Certification” is used. Bearing the name DO-178B [Rad] in the US and ED-12B in Europe, it describes the objectives of software life-cycle processes, process activities and the evidence of compliance required at different software levels. Safety standards like DO-178B and IEC-61508 [Int10] explicitly call for the identification of functional and non-functional hazards and for software compliance with the relevant safety goals. In these standards three important non-functional software characteristics related to safety are mentioned: absence of run-time errors, execution time and memory consumption.
The IEC-61508 has a great impact on the hardware selection as it requires the absence of unpredictable timing-related interferences which might affect real-time functions. Nevertheless, this type of interferences are quite common and must be dealt with.
The WCET estimation consists of two main steps, namely the control-flow analysis that determines the feasible paths in a program, and the processor-behavior analysis based on low-level analysis, therefor we need to throughly determine the hardware behavior. The choice of a hardware platform is therefore greatly influenced by the visibility on the device internal structure as precise architectural implementation details are proprietary data often undisclosed.
Identifying the processors that are to hard to analyze or which are the part that can greatly impact the analysis performance, will greatly influence the precision of the estimation. Section 6.1.1 acts as a guideline, exposing the inherent problems of each processor’s component.

1.2 Problem definition

The quality of an embedded system does not only rely on the quality of the hardware architecture and the integrated software but also in the capability to thoroughly and hope-

fully simply prove functional and non-functional properties. Hard real-time systems, like the ones used in the avionic field, are subject to various certification constraints, such as the DO-178 standard for the software and the DO-254 for the hardware. The timing analysis has an important place in the embedded project development, for example the DO-178, clearly states that a Worst-Case Execution Time (WCET) analysis is required for every task. The quality of execution time estimates for the software is very important in determining the timing behavior of a system before it is deployed.

Hard real-time systems are used in a variety of industry branches, housing specific project design practices and heterogenous certification procedures even for similar criticality levels. This also implies the use of different types of hardware systems, namely processors, depending on the project type, application branch, etc. However they all demand and benefit from precise WCET estimations. Therefore it is very important for a tool or model that aims to be used in real life to be efficiently *retargetable* so that it can be adapted to different types of processors with a reasonable effort.

The safety property is crucial for the usability of the method in such systems and is provided by the use of static methods that will take into account all the achievable states of the system and provide an over-approximation of the WCET estimates.

The adaptability, as previously stated is an important factor for the usability of the method, saving time and resources as the underlying platform can evolve and change even during the same project's development cycle.

The precision is a well known term in different sciences and fields. The precision of static methods in the estimation of the WCET relates to the difference between the obtained over-approximation and the theoretical WCET value. Since the exact value of the WCET is often impossible to determine it is also impossible the compute the exact value of the difference. The quantification of this difference is however important in order to characterize the quality of the WCET estimation method. Tight bounds ensure a good resource use and allow for the selection of cheaper processors and on the contrary loose bounds render the result useless.

1.3 Research goal

The research effort of this thesis revolves around WCET estimation through a novel approach that will be usable in an industrial context covering most of the necessary steps of a static method and giving all the building blocks needed for the implementation.
Our aim is to define a *safe*, *adaptable* and *precise* WCET analysis of processors embedded in systems with hard real-time constraints.
The method must be safe therefore it will provide an over-approximation of the WCET through the use of a formal model in the processor definition. As it was shown in the introductory part, in order to completely characterize the behavior of the system and to take into account all the possible temporal outcomes of the software we must consider the interaction with the hardware, for all possible inputs. Our intuition was that the choice of the modelling language for the processor is key for the rest of the analysis. The analysis of all execution paths of the processor executing the target program generates a combinatorial explosion, therefore it is impossible to look into each individual state. Safe abstractions of the hardware model must be possible in order to be able to deal with large systems. The compositionality of the WCET analysis is compromised for modern processors because of the presence of timing anomalies. A good processor model has to be able to naturally take into account this behavior with minimum effort.
Regarding the program to analyze it is important to start with the actual binary that will be deployed in the actual system in order to make sure that its precise execution behavior is taken into account because the compiler used to generate it has an impact on its WCET. A value analysis is to be performed on the program's binary as to give a first information about the value domains of the variables and the loop counters.
The generation and analysis of all the reachable states of the processor is ensured through a conjoint symbolic execution of the binary and the processor model.
The WCET analyser will guide the flow analysis through the hardware model parametrized by the binary and will decide what level of abstraction is best suited along the way. The separation of the model from the WCET analysis will allow for a smooth transition to another platform, hence the aimed adaptability.
State merges will be performed whenever needed. Whenever a state merging will be per-

formed, a precision loss is introduced. A threshold will be used in order to determine if the precision loss is acceptable and by storing all the cumulated imprecision, estimates will be given regarding the precision loss of the method.

1.4 Contributions

We propose a novel approach for the WCET estimation and we revisit most of the steps of the analysis.
The centrepiece of our work is the extension of the abstract state machine (ASM) formal method. This framework was throughly formalised and all the crucial aspects for the timing estimations were defined and conceived with regard to this model.

We start by acknowledging that even though the presented method is a complete and detailed static timing analysis approach the purpose of this work was not to fully implement, in the sense of coding, the timing analyser, which is the object of an undergoing effort, but rather to introduce a novel formal language and dedicated WCET estimation framework.

Throughout the numerous definitions and the clear semantics of all the parts of the model and framework, we disambigously define our approach and plug-in novel methods to enable the safe and scalable timing analysis. Some existing state of the art analysis techniques are also discussed and applied using the defined framework being revisited and extended whenever needed, like some of the techniques in the timing anomalies section.

One of the original extensions and redefinition of existing techniques is the abstract state merging. The intuition behind state merging is that each instruction constrains the processor in certain state. An instruction sequence will generate a conjunction of constrains, further refining the processor configuration. Giving the finite set of possible configuration on a processor, we can suppose that the processor will recurrently find itself in identical and mostly similar states.
Original contributions of the thesis:

- *Extension of the ASM model with temporal and hierarchical features - HiTAsm.* The advantages of the ASM framework are manifold and defended in the next chapters.

We enrich the framework with some key features and provide seamless and correct integration into the formalism. The features are custom tailored for the timing analysis of large and complex systems.

- *Extension of the ASM framework with symbolic terms and symbolic runs for the symbolic execution of HiTAsm*. We present a seamless integration of the symbolic execution into the ASM framework and give the inductive definition of the semantics of Symbolic HiTAsm.
- *Integration of an ASM based hardware model in the WCET analysis*. ASM usage spans across a multitude of real-time domains and applications. We take it forward by using the HiTAsms for timing analysis.
- *Definition of the syntax and semantics of a HiTAsm Language (HiTAsmL).*
- *Definition of a graphical UML-style syntax for the HiTAsmL.*
- *Implementation of a HiTAsmL interpreter*.
- *Implementation of an interpreter for the binary value analysis results*. We interpret the results of an external value analysis of the binary and create a CFG with nodes as basic blocks of linear assembler instructions, their addresses and values.
- *Modelling of a full RISC processor featuring modern components*. We provide a test-case for our model by the implementation of the Motorola MPC555 processor using our HiTAsmL language.
- *Definition of an Oracle that is capable to dynamically change the abstraction level of the processor model*. The framework used to model the processor enables hierarchical abstraction levels that can adapt to the execution context. This adaptability is governed by an *Oracle* that has a few fixed, possibly processor-dependent, strategies aimed at reducing the state-space explosion. This strategies can be broadened and updated at in a simple way.
- *Definition of an WCET analysis based on the WCET model and the conjoint symbolic execution of the binary and the processor model.* In a nutshell, our analysis technique

consists in the generation of all reachable processor states for a program through their conjoint symbolic execution. This analysis controls the state abstraction levels through the Oracle and identifies, on the fly, similar states to be *merged* using the *Prediction Module*.

- *Definition of ad-hoc state merging strategies based on the HiTAsm component definition*. State merging is another technique that we use in order to constrain the state space explosion of the analysis. Based on our formal framework we define the *state merge* and also methods of merge validation.

- *Equivalence classes for the analysis state-space partition* Because the state space that must be analyzed is so vast, reducing the number of state comparisons needed in order to merge them is crucial. We therefore partition the space using equivalence classes defined using the HiTAsm formalism.

- *Timing anomalies identification and safe window definition* Again, using the HiTAsm model we define portions of code where compositionally of the analysis or use state of the art algorithms applied to our model to determine a safe time difference to add as a penalty in case of local worst-case assumptions.

- *Dynamic state merging (fusion) and Prediction Module based on the HiTAsm model*. Once again, the HiTAsm framework is used to formally define research strategies that not only identify state that have better merging success probability but are also able to validate them. Additionally, validated state merges will become starting points for other predictions.

1.5 Organization of the thesis

This work is structured into 7 chapters, including the introduction and is organized as follows.

Chapter 2 follows the introduction of the thesis and presents the state of the art in the of the worst-case execution time analysis. A in-depth view of the methods and practices used

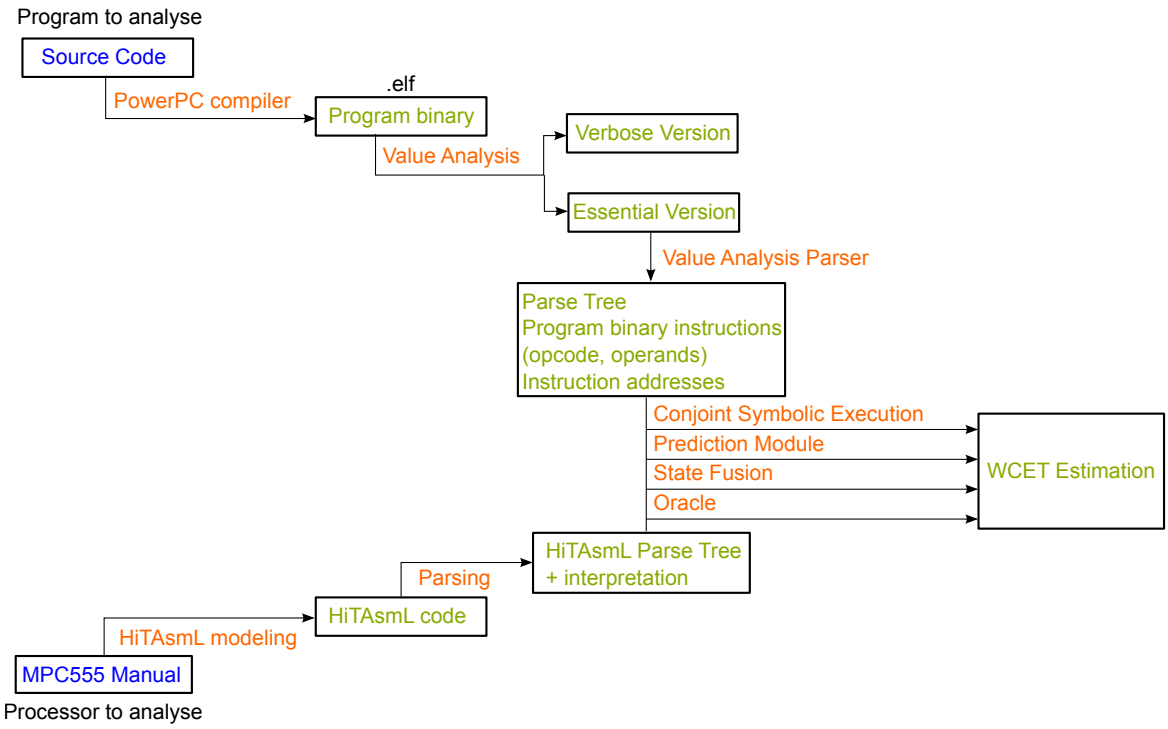


Figure 1.6: Global structure of our WCET estimation method

is given before presenting actual tools either academic or industrials.

Chapter 3 starts with a fundamental introduction of the ASM formal framework that we used as a starting point and defines the extensions.

Chapter 5 gives the definition and implementation of the language created to support the introduced formal model.

Chapter 6 gives the processor model used in the timing analysis, defined in the HiTAsmL language. We choose to describe and implement the Motorola MPC555 processor, featuring a RISC architecture with a five stage pipeline, out of order execution, branch prediction unit, etc. therefore complex enough to show the full strength and flexibility of the method.

Chapter 7 presents the WCET estimation method based on the formal framework previously introduced. A global view is presented in figure 1.6 It uses the hierarchical abstraction of the processor definition in order to selected the best adapted component definition for the actual context. The Prediction Module is detailed, and state merging as a method to reduce the state space explosion is introduced.

Chapter 8 summarizes the achievements of this thesis and gives an intro to the future perspectives of this work.

Chapter 2

Related Work

Many embedded systems require hard or soft real-time execution that must meet rigid timing constraints. As we focus on hard real-time systems, we must ensure that these constraints are always respected. Two main approaches exist for the timing analysis, namely dynamic and static methods, [WEE+08]. At the crossroad of both static and dynamic approach we find the hybrid method, presented in the last part of this chapter.

Regarding the dynamic methods, the choice of the measuring technique is very important, and impacts the quality of the timing result. We present a variety of techniques, that we classify based on pertinent attributes like the *resolution*, *accuracy*, *granularity* of their measure and the *difficulty* to use them in practice. The presentation order is given in ascending order based on the mean of these attributes. Measurements provided by the dynamic methods can be used as the basis for real-time scheduling analysis, for identifying timing problems, or to know what code needs to be optimized. However the safety of their result is often subject to questioning.

As opposed to the dynamic approach, the static methods ensure by construction the safeness of their results. However, most of them are too dependent on the underlying hardware making it difficult to adapt to new processors. Many of the available time analysis tools show a list of compatible hardware and present each new platform taken into account as a new feature. One of existing methods, OTAWA, [ERT06], differentiates itself by making a first step towards adaptability as it uses a basic parametric model of a generic platform that can address a variety of architectures. AbsInt, one of the leading WCET analyzers and

overall solutions for timing analysis, also takes a step towards adaptability by looking to use an HDL processor description in order to generate an abstract processor model.

2.1 Dynamic methods

Dynamic methods rely on run-time information in order to estimate what is generally an under-approximation of the WCET. However, input at both functional and hardware levels that leads to worst-case execution path of a task is unknown and impossible to find in any case. The method performs timing measures of individual tasks, and establishes the overall real-time performance of the system. Only a sub-set of possible executions of the program are covered and the instance corresponding to the worst observed case is retained. Depending on the type of tools used to measure the execution time, the dynamic methods can be either hardware or software. The industry employs a variety of measurement tools, including emulators, time-accurate simulators, logic analyzers, oscilloscopes, timer readings inserted into the software, software profiling tools, time functions provided by the operating system or third party time measurement tools [Ive98]. Some of the available functions are suited for interactive programs and can even give finer granularity than a function for example execution time of a code segment or a loop. However, in order to call these functions, additional lines of code must be added that are against the principle *test what you fly and fly what you test*. This section presents a variety of representative dynamic methods, featuring measurement techniques at both coarse-grain and fine-grain levels.

2.1.1 The choice of the measuring method

The system designer is presented with a variety of choices regarding the methods and technologies to use in order to measure the WCET. The methods differentiate themselves based on the following attributes:

- *Resolution* - concerns the limitations of the timing hardware. Measurement resolutions vary from a 0.01 sec resolution in the case of the stop watch, to 50 nsec in the case of a logic analyzer;

- *Accuracy* - the difference of the closeness measured value using a given method of measuring, and the actual time obtained with an ideal measure. When repeated several times, measures undergo an amount of error.
- *Granularity* - the part of the code that can be measured:
  - coarse-grain - measure execution time on a per-process, per-procedure, or per-function basis.
  - fine-grain - can be used to measure execution time of a loop, small code segments or even a single instruction.
- *Difficulty* - the effort to obtain measurements, defined subjectively. Usually, software-only methods are easier, however they only provide coarse-grain results. Hardware-assisted methods are considered hard, and yield fine-grain results of high accuracy.

Like the ease of setup, the granularity of the measure and the time needed for the result, all subject to cost-efficiency metrics. Therefore, depending on the project type, the project's lifecycle stage, system configuration or criticality of the application we can choose the most adapted measure method.

2.1.2 Stopwatch method

The simplest and most obvious time measuring method is the so called *stopwatch method*, that we present in order to have a complete referential regarding the granularity and precision of the method. It consists, as stated by its name, in the use of a chronometer that will be started concomitantly with the program execution and stopped when the program terminates. It can be used in a preliminary phase to obtain an order of magnitude of the execution time.
The disadvantages of this method are manifolds. For instance it is impossible to ensure that no delays introduced by interruptions were counted into the measure. It is therefore recommended to be used only on systems without interruptions, preemptions and with a single and very simplified processing unit.

2.1.3 Date and time OS commands

This method can be used on any system featuring an operating system that implements the *date* and *time* system commands, like UNIX systems. A script is generally written with a loop of a program call surrounded by two *date* instructions and make a mean of the results. Similarly, we can directly use the *time* system instruction. In listing 2.1 we can see the use of the two functions on a binary called *progBinToMeasure.elf* that simply prints a message on the standard output.

Listing 2.1: Measures with date and time in the UNIX shell

```
1  mpc555board: paun$ date
2  Thu Oct 23 15:05:24 CEST 2014
3  mpc555board: paun$ ./progBinaryToMeasure.elf
4  Hello WCET world
5  mpc555board: paun$ date
6  Thu Oct 23 15:05:51 CEST 2014
7  mpc555board: paun$
8  mpc555board: paun$
9  mpc555board: paun$ time progBinaryToMeasure.elf
10
11 real  0m0.005s
12 user  0m0.002s
13 sys 0m0.003s
```

Once again, preemptions and interruptions are not taken into account, therefore the method can only be used to get a rough estimation of the execution time.

2.1.4 Prof and Gprof (UNIX)

The *prof* and $gprof$ methods measure execution on a per function basis, a finer granularity than the already presented methods. Available in UNIX system, the methods provide a set of timing measurements for all the code with clock resolution ($10^{-2}s$).

These methods are intrusive meaning that the measured execution time will be greater than the real execution time of the program when it is not being profiled.
As listed in 2.2 the `prof` method is applied by using the compile options `-p` with the platform's compiler and running the program followed by a few other commands. The binary file `mon.out` is generated automatically and contains the timing data by function for the program. It can be viewed with the integrated `prof` command.

Listing 2.2: Using prof

```
mpc555board: paun$ gcc  p  o progToMeasure progToMeasure.c
mpc555board: paun$ ./progToMeasure
mpc555board: paun$ ls
mon.out
progToMeasure
progToMeasure.c
mpc555board: paun$ prof progToMeasure > progToMeasure.prof
mpc555board: paun$ cat progToMeasure.prof

Function list, in descending order by time

[index] secs % cum.% samples function (dso: file, line)

[1]1020.180 43.6% 43.6% 102018  __pow (libm.so: pow.c, 198)
[2]721.570 30.8%  74.4% 72157  __log (libm.so: log.c, 139)
[3]358.100 15.3%  89.7% 35810  __exp (libm.so: exp.c, 102)
[4]235.820 10.1%  99.8% 23582  asa (asa_run: asa.c, 59)
[5]1.310 0.1%  99.9% 131  wr_lst_bst (asa_run: getpd.c,566)
[6] 0.990 0.0%  99.9% 99  loadlists (asa_run: getpd.c, 172)
[7] 0.960 0.0%  99.9% 96  cost_der (asa_run: asa.c,2229)
[8] 0.540 0.0% 100.0% 54  print_state (asa_run: asa.c, 2762)
[9] 0.210 0.0% 100.0% 21  _free (libc.so.1: malloc.c, 903)
[10] 0.100 0.0% 100.0% 10  wr_prf (asa_run: getpd.c,629)
```

2.1.5 Timer and Counter

This method consists in measuring small segments of code by programming the system's counters or times. To provide greater precision for timing measurements, we must use a timer that operates at the clock cycle level. This timer is actually a special register that gets incremented every single clock cycle. Therefore the processor must implement the functionality in order for this method to be applcable. Special machine instructions can be used to read the value of the counter.
The method consists on calling the function timer:

$$timer \rightarrow ClockTicks \tag{2.1}$$

before and after the code segment. The obtained results τ_i and τ_f correspond to the total number of elapsed clock cycles since the system startup. After we compute the difference of the two values we obtain the total amount of cycles elapsed between the initial timer launch and the final one. In order to get the elapsed time we must divide by the frequency of the processor, f, the number of clocks per second.

$$\Delta t = \frac{\tau_f - \tau_i}{f} \tag{2.2}$$

The Pentium processor features a cycle counter, the IA32, accessed with the **rdtsc** instruction. This instruction sets register **edx** to the high-order 32 bits of the counter and register **eax** to the low-order 32 bits. The obtained cycle counter stored on 64 bits can store enough values that it only cycles around once every 570 years.
For the ease of use, we generally want to be able to access these values directly from the C code. Therefore we must provide a C program interface, by encapsulate this instruction within a procedure like the one defined in listing 2.3.

Listing 2.3: Implementation of the counter read interface

```
1 void read_counter(unsigned *hi, unsigned *lo)
2 {
3   asm("rdtsc;␣movl␣%%edx,%0;␣movl␣%%eax,%1"//Read counter
4     : "=r" (*hi), "=r" (*lo)          //move results to
```

```
5       : // No input
6       : "%edx", "%eax");              //the 2 outputs
7   }
8   void start_counter()
9   {
10    read_counter(&cyc_hi, &cyc_lo);
11  }
```

Listing 2.4: Using the clock counter

```
1   double time_cold()
2   {
3     progSourceToMeasure(); /* Warm up instruction cache */
4     clear_cache(); /* Clear data cache */
5     start_counter();
6     progSourceToMeasure();
7     return get_counter();
8   }
```

The function presented in listing 2.4 measures the execution time of the function: `progSourceToMeasure()`. It starts by running the program once in order to warm up the cache and proceeds to a data cache content clear through different data writings. Once no more program data is in the cache, the timer in started and the function run once again, followed by the return of the counter corresponding to the program end. The execution time found this way roughly corresponds to the worst-case.
In order to obtain a best-case execution time, the `clear_cache()` routine is removed.

2.1.6 Software Analyzer

RTOS and tool vendors propose a series of software tools designed specifically for measuring execution time entitled software analyzers like CodeTest [cod11], TimeTrace [Tim], and WindView [Win].

Software analyzer can be based on the system clock (resolution of the order of a millisecond) or some other hardware-based method, such as using an onboard timer/counter chip (resolution might be in the microseconds range).
The software analyzers can sometimes provide a timing trace that shows precisely what process is executing and at what time. If the timing trace is also correlated to the source code, then the tool can identify what part of code is responsible for extended periods of execution.
A main drawback of software analyzers is the used resources as they may slow down the code. Most part of the analyzers needs a lot of memory to work, a problem in embedded system that already fully use the available memory.

2.1.7 Logic Analyzer

Among the hardware methods, the logic analyzers provide a good tool while being non-intrusive. It can obtain data in two modes, which are called state mode and timing mode. The timing mode gives a better resolution in the signal. However, the timing mode requires more memory to store a larger number of information that is sampled. The logic analyzer dose not give access to the internal state of the processors, nevertheless it gives a good view of the externally visible signals. The result obtained by this method is not a temporal bound but merely a measure that corresponds to the observed execution time for the executed inputs, one path at a time. Therefore the difficulty relies in finding the worst-case input, which is impossible in the general case, especially without any program flow information.

2.1.8 Summary of execution time measurement measures

In table 2.1 we summarize the methods and attributes of each presented measurement method with estimative and sometimes subjective values.

Table 2.1: Summary of measurement measures

Method name	**Resolution**	**Accuracy**	**Granularity**	**Difficulty**
Stop watch	$10^{-2}s$	$5 \cdot 10^{-1}s$	program	easy
Date/Time	$2 \cdot 10^{-2}s$	$2 \cdot 10^{-1}s$	program	easy
prof and gprof	$10^{-2}s$	$2 \cdot 10^{-2}s$	subroutines	moderate
Time/Counter	$5 - 40 \cdot 10^{-6}s$	$1 - 8 \cdot 10^{-6}s$	statement	very hard
Software Analyzer	$10 \cdot 10^{-6}s$	$20 \cdot 10^{-6}s$	sub-routine	moderate
Logic Analyzer	$50 \cdot 10^{-9}s$	$5 \cdot 10^{-7}s$	statement	hard

2.1.9 Advantages and weaknesses of dynamic methods

The downsides of dynamic methods, and timing measuring in general, became obvious through the presentation of the different measuring methods in the previous sections. However, they still found their way in numerous industrial projects, at any criticality level. This is explained through the good-enough performance when used in conjunction with best practices, coding restrictions, hardware restriction, human expertise, safety margins and other additions that provide sufficient guarantees for use in projects up to hard-real time system.

These methods align, nevertheless, undeniable qualities through the fact that the best hardware ”simulator” is the actual system itself. Measures are taken directly on the hardware that will be embedded in the final system, on the same binary that will run on the end project. Indeed, the system in occurrence must physically exist in order to be able to perform any measures, even though experimental configuration on development versions can be made.

Even though the call for a future transition to static methods is clearly stated by the certification guidelines, and therefore needed by the industry, dynamic methods are still pervasive in application and the *de facto* choice for system certification. This might end even sooner,

as, besides the certifiability concern, the intrinsic nature of the application field makes cost-efficiency a clear goal and dynamic methods on modern platforms require more and more testing efforts and time. It must be also stated though, that the success of such methods is often tributary to such product- and company-dependent efforts that no real adaptability can be defended. Moreover, an increasing number of dynamic methods are evolving into so called hybrid, semi-dynamic methods where a part of the job is done using static methods, helping the tester identify critical paths that can pin-point a better input set for the measures. Hybrid methods are further detailed in section 2.3.

2.2 Static methods

The respect of deadlines is important in safety-critical systems. Therefore the WCET must be either exact, which is impossible in the general case, or an over-approximation of that value in order to ensure the safety of the system. The majority of the current embedded systems use hardware components that have a data or history dependent execution time. In order to estimate the WCET, a direct measure is no longer possible because local optimizations are used in order to maximize the platformÕs maximum throughput.
Static methods are conceived to take into account not only the program but also the platform and possible hardware interactions. In order to obtain a safe WCET estimation we must analyze the time generated by every possible program execution path for every possible values. This is impossible to realize for non-trivial cases because of the state space explosion. Therefore a common feature of static analyzers is to use approximations that will ensure the computability of the result by providing an over-approximation of the WCET. The safety of the static analysis is a given whereas the precision of the result is an evaluation of the performance of the method. Another evaluation criteria is the scalability of the method, feature usually in tradeoff with the precision of the estimation. The adaptability is also a good feature, as the computation part of methods tends to be overly dependent on the architectural model.
These methods consists of several steps that will start from the program/task itself, usually the actual binary that will run on the final platform, compute all the possible control flow

paths through the program (associated with the high-level analysis), combine them eventually with information from the value analysis and use them on a hardware model, usually abstract, in order to generate all the possible states that will lead to an approximation of the WCET.

Among the modern features that harden the WCET estimation are the pipeline, out-of-order execution, cache memory, branch prediction unit and others. One of the problems that they generate are the timing anomalies that violate an intuitive, but incorrect, assumption that whenever presented a choice, choosing the transition that takes the longest time, will lead to the Worst-case execution time. In other words, the monotonicity assumption is compromised for processors sporting modern features.

The static methods offer great expectations, nevertheless the practice shows that it is hard to achieve a completely automatic method. Annotations are often required and can become an issue. The dependency with the analyzed platform is another problem as the platform choices in the embedded world are quite heterogeneous.

The downsides of the method get amplified in the case of multicores as new problem arise among which even the determinism can be questioned.

Static methods are going through a transition in order to achieve maturity but show promising results and offer crucial features like the safeness of the estimated results. Some representative examples of the static methods are presented below.

2.2.1 AbsInt Advance Analyzer

An example of a great success story of a static analyzer is AbsInt Advance Analyzer™ (a^{3}™), [a3it].

The wrapper application a^3 integrates a number of static program analysis tools like:

- aiT for worst-case execution time analysis
- StackAnalyzer for stack usage analysis
- ValueAnalyzer for value analysis
- TimingExplorer for ECU-level architecture exploration

- Custom-built products (e.g. TimeWeaver)

In the following we will focus on the presentation of *aiT*, a modular WCET analyzer, as described in [FH08] and depicted in figure 2.1.

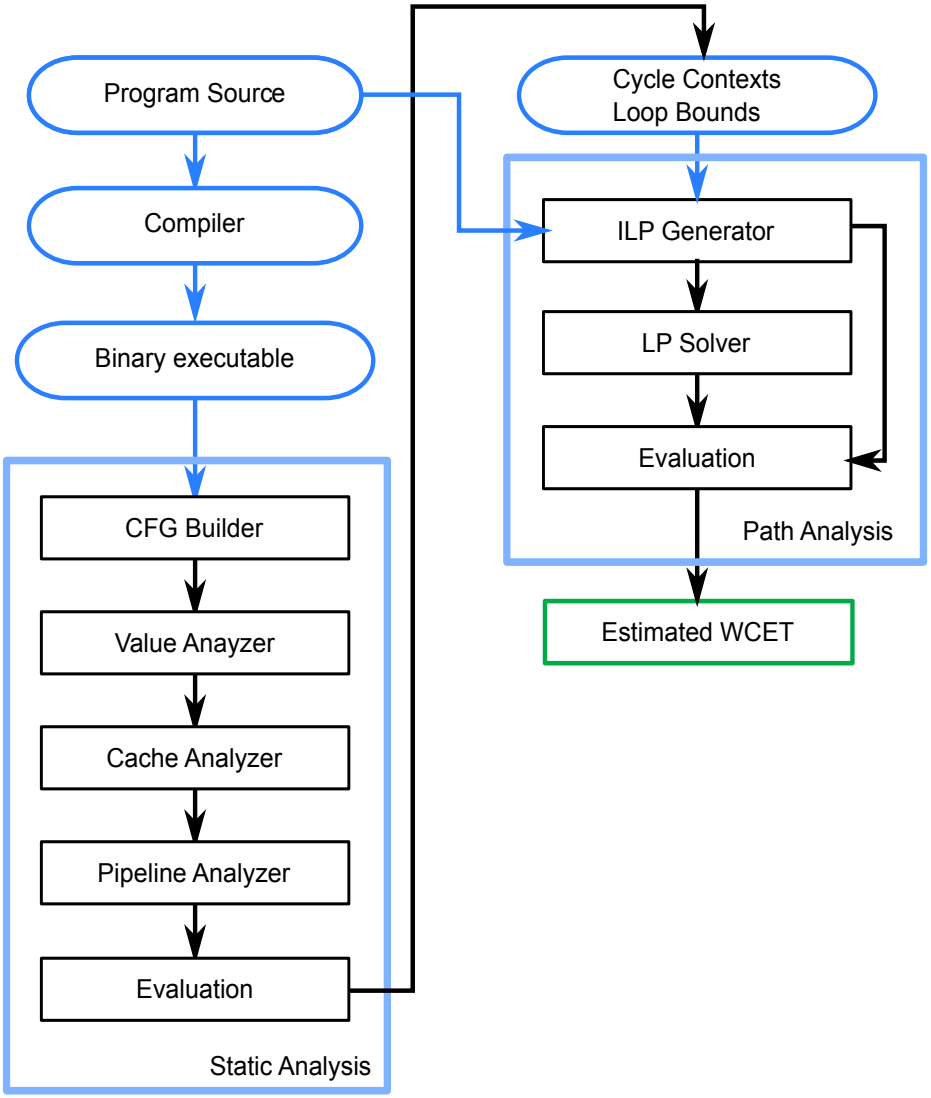


Figure 2.1: Global structure of the aiT tool

CFG During the *control-flow reconstruction* the analysis creates the control flow graph, CFG of the program's binary file. Information like: operations, instructions, basic blocks

and routines is organised hierarchically. A basic block contains instructions that execute linearly, meaning that no branching instruction can be contained except for the last line. To simplify the cache analysis, the size of the basic block is chosen so that it can be contained on a single cache line, therefore every CFG node is associated to a single memory reference.

Value analysis A *value analysis* is then performed that will produce an over-approximation of the accessed memory locations. The subsequent cache analysis needs to know the addresses of the program's data, [WW08]. Given the fact that precise addressees will only be known during the execution, an interval analysis is made, [CC77], that computes the possible addresses of registers.
The value analysis provides access information to data-cache/pipeline analysis and detects unfeasible paths through *interval analysis* - lower and upper bounds calculation for the values occurring in the machine program addresses, register contents, local and global variables).

Loop analysis A static *loop analysis* is further performed on the code. The first purpose is to identify, using a pattern matching mechanism, the presence of loops in the code and the second is to perform a *data flow* analysis, interpreting the machine instructions inside the loops in order to infer the loop bound value, [Cul06]. Not all loops can be bounded by the analyzer, therefore the user must manually annotate the rest.

Component abstraction Based on this result, the *cache analysis* classifies the memory references into the following categories:

- *always hit* - the block is always present in the cache;
- *always miss* - the block is never present in the cache;
- *persistant* - the referenced memory block is loaded once at most;
- *unreachable* - the code cannot be reached.

- *not classified* - the memory reference couldnÕt be classified in any of the above categories

Cache/Pipeline analysis uses the register values ranges computed by the value analysis and tries to statically determine in advance all the possible cache content. Through the *must/may analysis* memory references are classified into cache hits and potential cache misses.
The pipeline analysis uses an abstract model to simulate the behavior of the pipeline, [The04]. The pipeline is considered as a finite state machine (FSM) and the analysis is defined as a calculation of sets of states of FSM [59]. The efficiency of the approach is streightened through the use of the binary decision diagram (BDD) representation of the FSM.

Path Analysis is used to determine upper bounds for the program's execution time based on the IPET approach that uses ILP on the previous CFG as well as results from the cache/pipeline analysis.
a^3 targets a vast array of platforms: Am486, ARM, C16x/ST10, C28x, C33, ERC32, HC11, HCS12, i386DX, LEON2, LEON3, M68020, PowerPC 5xx, e200 (55xx, 56xx, 57xx), e300 (603e, 82xx, 83xx), 7448, 7448s, 750, 755, 755s, TriCore, V850E.
The tool has however some limitations:

- no support for dynamic allocation (malloc and such)
- **setjmpg/longjmp** not supported
- code must follow standard ABI calling conventions
- code must be generated by restricted subset of ANSI C;
- function return addresses must no be modifed

2.2.2 OTAWA

OTAWA (Open Tool for Adaptive WCET Analysis), is a framework of C++ classes dedicated to static analyzes of programs in machine code and to the computation of WCET.

OTAWA emerged from the Traces Research group on Architectures and Compilers for Embedded Systems at IRIT (Institut de Recherche en Informatique de Toulouse).
With its modular architecture, OTAWA is adaptable to various WCET calculating techniques. It is based on a set of analyzers that can perform various tasks, from the construction of the CFG from the binary to the final calculation of the WCET. An proprietary annotation system stores information like the data entry, intermediate or final results, on the analyzed program. For the effective WCET estimation, the tool uses an off the shelf ILP solver called *lp_solve*.
The tool is structured in three main steps:

- identifying infeasible paths and bounding loops through the flow analysis
- low-level analysis: determining the global effects of the processor on execution times and at deriving the worst-case execution times of code snippets;
- the flow and low-level analyzes are used to estimate the WCET.

One of the main advantages of this tool is the adaptability to new architectures, taking full advantage of the modularization and separating the WCET estimation part from the processor model. The characteristics of the processor are provided in the form of an XML file. However the analysis does not fully take into account all the processor characteristics and component interactions, therefore the over-approximation is quite important.

2.2.3 SWEET

SWEET (SWEdish Execution Time Analysis) is a research prototype tool focused on flow analysis, developed at Uppsala and Malardalen University since 2001, [Lis14]. The tool computes BCET and WCET estimations. SWEET is divided into three major parts, as fallows:

- detection of program flow constraints through a flow analysis, which is the main function that tries to calculate flow information as automatically as possible;
- a low-level analysis - pipeline analysis is limited to in-order pipelines and does not consider timing anomalies;

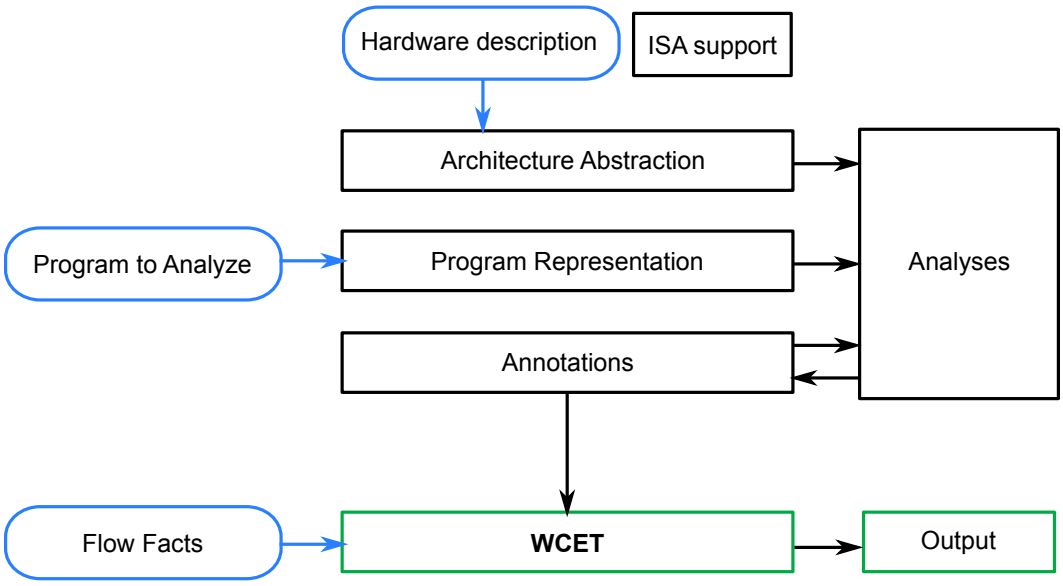


Figure 2.2: Global structure of the OTAWA tool

- WCET calculation.

The flow analysis part of SWEET analyzes intermediate code produced by a research compiler of a general intermediate program language called Artist Flow Analysis language (ALF) as it is coupled to a research compiler which is only able to process a subset of ANSI C. SWEET offers:

- powerful loop bound analysis;
- infeasible paths identification;
- derives the explicit WCET and BCET execution paths through the program,

using three different WCET calculation methods:

- a fast path-based method;

- a global IPET method;

- a hybrid clustered method.

Target architectures include: ARM9 and NEC V850E processor.

2.2.4 CHRONOS

Chronos is an open source, academic software developed at National University of Singapore (NUS) which performs timing analysis of embedded software through static analysis, [LLMR07], [CR09]. The tool is divided into five main steps, visible in figure 2.3:

- Binary generation from the program source using the GCC compiler from the *SimpleScalar* toolset.

- CFG reconstruction and control flow analysis. Paths are represented as linear constraints called *flow constraints*. The user can also introduce annotations through a GUI taken into account in the *user constraints*.

- The tool uses a processor model configured by the user via a GUI and generates timing information for basic blocks in execution contexts and constraints on the occurrences of execution contexts (architectural state). Together with the flow contraints this data contributes to a ILP problem solved in the next step.

- The ILP problem is solved using either *CPLEX* or *lp_solver* and thus the WCET estimation is calculated.

- An observed WCET is also calculated using sim-outorder simulator in the SimpleScalar toolset.

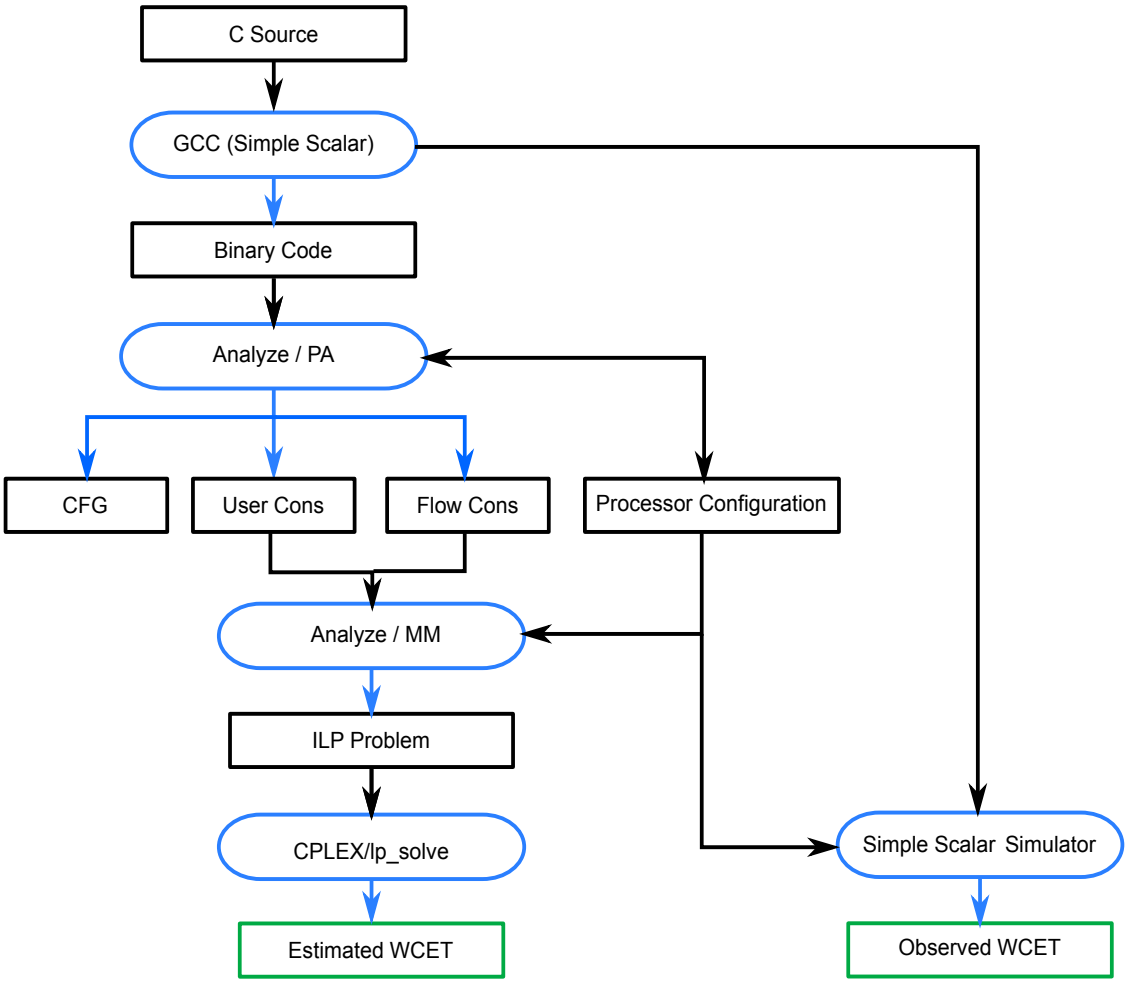


Figure 2.3: Global structure of the Chronos tool

The tool shows the following limitations:

- doesn't support any real HW platform;
- doesn't analyze data caches.
- deployment difficulties (building issues and difficulties)

2.2.5 BoundT

The Bound-T is a WCET Analysis Tool working on machine code, [bou]. Relying on Presburger arithmetic using the Omega Calculator [Pug91], the tool performs:

- Determination of loop bounds (identifies loop-counter variables and computes a bound on loop iterations from the counterÕs initial value) - can be replaced by user assertions;
- Dynamic jumps resolution.

The worst-case path is determined with IPET using the *lp_solve* program.
We identified some of the following limitations for Bound-T:

- no cache analysis
- task must not be recursive
- CFG must be reducible
- dynamic calls analysis only if an unique target address found
- weak aliasing analysis
- bounds of an inner loop cannot depend on the index of outer loop
- loop-bound analysis doesn't cover multiplication, division, bitwise ops
- task must use standard calling conventions

2.2.6 Advantages and weaknesses of static methods

Static methods will soon become industrial standard for product certification. The strong guaranty on the safety of the WCET estimation makes them an ideal candidate in systems that must comply with rigorous safety-related constrains.
However all the existing methods share at least one of the following drawbacks:

- lack of accuracy;
- total or partial adaptability issues to new hardware models;
- lack of support for multi-core architectures;
- prohibitive costs;

- scalability issues;
- lack of processor support;
- few fully automatic, push-button results.

2.3 Hybrid methods

Hybrid methods are at the crossroad of dynamic and static approaches, presented in sections 2.1 and 2.2. Given the hybrid nature, two genders of approaches fall into this method namely the dynamic methods that incorporate static analysis and, symmetrically, the static methods that evolved by integrating some dynamic features or timing measurements.
These methods witnessed the most natural genealogy and evolution. Measure-based approaches have always been around since the early days of embedded systems. Real-time systems evolved, in pair with the hardware modernization, making it more difficult to capture all the temporal behaviors of intricate interactions and increasing the testing effort needed to achieve an equivalent level on insurance as before. As a direct consequence, more advanced techniques of program and hardware analysis were progressively introduced in order to more easily take into account the platform's complexity. Ideally a total code coverage would be needed through input sets that generate an *all the path* control flow graph transversal. Given the fact that it has become impossible to achieve, a less complete execution path coverage can still be of great help in order to guide, improve and reduce the coding efforts. Let us take a closer look on this approach.
Hybrid approaches identify the program paths consisting of a sequence of basic blocks where the execution is invariant to input data, [WSE02]. To find these paths starting from the source code level, symbolic analysis on abstract syntax trees can be used. After this analysis, the execution time of the path is measured on real hardware or by cycle-accurate simulators. Branches that are dependent on the input need input data for a complete branch coverage. Finally, techniques from the static approaches are used in order to determine the longest path and to estimate the WCET.
The behavior of a program can be represented by a control flow graph, where the nodes are

the programÕs instructions and the arcs represent program flow information. Code analysis is used to produce inputs that will generate the different program paths. The values for each variable can be split in different sets for each path, grouping inputs that generate the same path. However, we do not always have the guarantee that all the values of a variable from the same set (that generate the same path) will lead to exactly the same functional or non-functional behavior. For the functional part, certain values may lead to an overfull error, for example, and most of them not, hence it is important to ensure the absence of such cases. The timing behavior is highly dependent on the architecture. Hence, even if a value generates the same path, the timing will depend on the presence in the cache memory, or on the possibility to pipeline other instructions (for example a test that can be validated by an integer or a float value which changes the timing because of the floating point unit).
This type of method is safe only if all the inputs of the program can be tested or if complete path coverage of the control flow graph (CFG) is made. Furthermore, the previous assumption holds only for a certain type of hardware architecture, excluding most of the processorÕs modern features. Moreover, complete path coverage of the CFG is impossible to compute for complex programs.
Program flow information together with the timing of basic blocs can be translated into a linear programming problem. Integer linear programming (ILP) is used in approaches as IPET (implicit path enumeration technique) to bound calculations. However ILP is a NP-hard problem so it should only be used in subproblems of timing analysis.
The extraction of control flow information from the program code and the automatic generation of test data are the challenges of this method, as both cannot be fully automated for any program code. Approximations are used and human intervention is needed. Based on its code expertise, the contributor can reduce the number of paths that must be covered. Human expert analysis is also required in order to estimate a temporal safety margin that will over-approximate the eventual omissions of the estimation. An additional problem is that it is increasingly difficult to foresee the impact of the overseen cases and to estimate the impact of pathological cases in a large program or on processors with modern features.
A few representative examples of the hybrid methods are presented in the following.

2.3.1 FORTAS

FORTAS (FORmal Timing Analysis Suite) is measurement-based tool for timing analysis of embedded real-time software and is a joint effort of the Real Time Systems group at Vienna University of Technology and the Formal Methods in Systems Engineering group at Technische Universitat Darmstadt, [vHHL+11; ZBK11; BK08].
The tool, that targets C code, fills the gap between ad hoc testing, and classical static analysis, which requires detailed knowledge of the target hardware architecture and significant human effort.
FORTAS will use abstraction methods (model checking) to extract abstract models of the software from which test data can be derived automatically and independently of the target hardware. Timing data is gathered to obtain a timing model as an annotated state machine by executing tests on the target hardware and the process reiterated.

2.3.2 Heptane

Heptane (Hades Embedded Processor Timing ANalyzEr), developed by IRISA is an academic, open-source static WCET analysis tool targeted at a fairly recent processor: the Intel™ Pentium™. Heptane implements an Implicit Path Enumeration Technique (IPET) and includes cache analysis techniques for many cache architectures. Heptane also integrates a branch prediction analysis. A study on the effect of the branch prediction on WCET evaluation is presented in [CP01]. The number of timing penalties introduced by branch target miss-prediction is statically bounded by collecting informations on branch target buffer.

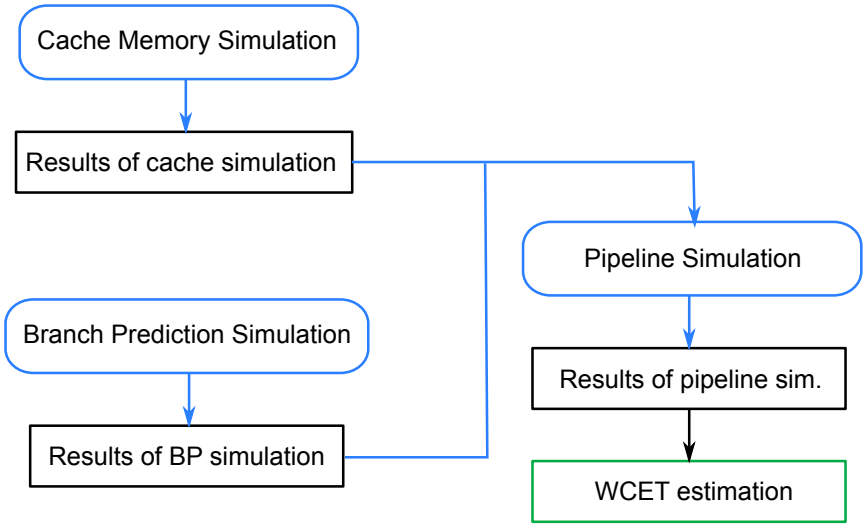


Figure 2.4: Global structure of the Heptane tool

Structure of the tool, as depicted in figure2.4:

- *Program representation* In the first step, the assembly code of the program is divided in basic block. Thus these basic blocks are used in conjunction with information collected during program compilation to generate two representations of source code:
  - a syntactic tree,
  - a CFG.
- *Loop analysis* The presented method defines a hierarchical data structure. This gives it the ability to identify each loop of the program and its level of nesting.
- *Cache analysis* A static cache analysis is performed, [Mue94]. Instructions are classified based on their behaviour related to the cache. This ranking is based on the presence (lack) of probable instruction in the cache when they are required by the processor, as well as the impact of the behavior of this instruction on probable presence/absence of other instructions. Thus, the purpose of this classification is to identify instructions that may cause a conflict causing a cache miss.

- *Branch analysis* A technique similar to the one used during the cache analysis is also used to categorized the conditional branch instructions. It identifies the directions in which the branching unit does not already dispose of a history for the branching instruction in treatment.

- *Pipeline analysis* The results obtained from the analysis of cache memories and branch instructions are used by the pipeline simulator that will finaly provide an estimate of the WCET.

The WCET can be computed either at source code level (tree-based approach) or via an ILP-based method that operates on a CFG extracted from the binary. Target architecture includes fairly simple processors as: StrongARM 1110 or Hitachi H8/300.

2.3.3 Probabilistic worst-case execution time

The probabilistic methods are gaining a lot of focus given the fact that hardware is increasing in complexity and still having to comply with safety-critical constraints. A good example is the multi-core processors where the actual hardware is not completely deterministic, [PMB13c], therefore a concrete model and a classic timing analysis would fail.
This type of methods also aim at reducing the cost of obtaining knowledge about the WCET estimation, which is a major drawback on applying the timing analysis on a larger scale.
Measurements offer an exact image of the underlying hardware, therefore in this context the software is the major actor dictating the execution time. A series of reasoning is made to ensure that the wright type and amount of measures are conducted thus, in conjunction with the right probabilistic model, safe temporal bounds can be extracted. The reduced amount of necessary runs of the application increase the applicability of the method and contribute to the cost reduction.
The work presented [LNBCG11] describes the tool support for a framework for performing statistical WCET analysis based on a sound measurement-based probabilistic timing analysis technique based on Extreme Value Theory. This method reduces the dependence of the execution history on the execution time using a combination of time-randomisation introduced to architectural features, [BG11], and the novel Probabilistic Timing Analysis

(PTA) techniques [CGSH+12]
The effect of *dependences* (relating system elements and their concurrent execution) on the task execution time using probabilistic models extracted from measures and the observation of the system in general, are explored in [MNS13]. A methodology in order to ensure the *safety* of probabilistic WCET (pWCET) estimation is also described.

Intuition In order to compute a safe estimation of the probabilistic WCET, we must be able to estimate the impact and inter-dependency of each unit.
Through measurements we obtain a distribution of the execution times. As we cannot make exhaustive measurements, we must find a series that over-approximates the WCET. Again, as we cannot have an infinite series we can approximate the result with an exceedence probability using the Rare Events. In order to use this theory, the events must have certain properties. Events associated with the components of a processor do not generally have these properties. Therefore another approximation is introduced, which represents a part of the goal of the method. The time series is shifted with a difference that represent the cost needed to create that independence.
Let C_i be a distribution of execution times for a task τ_i as a discrete random variable and f_{C_i} the probabilistic representation of C_i through probabilistic distribution function (pdf), [MNS13] defines the *pWCET* as:

Definition 16 (probabilistic Worst-Case Execution Time C_i^*) The pWCET C_i^* of a task τ_i is defined as the least upper-bound on all the distributions C_i^j of the execution time of τ_i, where C_i^j is generated for every possible combination j of the input data to the program running an infinite number of times. Thus $\forall j, C_i^* >= C_i^j$.

Overall approach The methodology of the approach can be summarized as following:

1. Define the temporal behavior of a task in terms of probabilities → use a probabilistic representation through probabilistic distribution function.

2. The Execution Time Profile (ETP), based on the distribution of execution times, uses exhaustive measurement of the task's execution time. The Cumulative Distribution

Function (CDF) is introduced.
From this point the pWCET is extracted.

3. The pWCET is defined as a least upper-bound for all the distribution of execution time (generated for every possible input combination after an infinite run). Therefore it is impossible to obtain.
   The rare events theory is introduced in order to safely approximate this pWCET. In order to explain this, a series of theoretical elements are introduced: independent random variables, a partial order on these random variables, rare events).

4. EVTs are used in order to approximate the pWCET (the partial order is used to give an upper bound on the pWCET).
   Apparently the *glue* behind this is that the rare events theory is used to estimate exceedence probabilities of 10^{-n}. This should mean that the approximation's *safety degree* is of the order of 10^{-n}.

5. EVTs need certain hypothesis in order to ensure safe pWCET approximation:

   - independence (i.)
   - identical distribution (i.d.)

   This forms the i.i.d criteria.

6. The dependence and independence of CDF are defined.

7. We can directly apply the above theory if a task runs on a system O described as a set of events $O = \{O_1, \cdots, O_i\}$ under the *in-dependence* assumption. This is not the case in reality, therefore the notion of copulas is redefined to introduce a distance that will help find a new time distribution with a CDF that bounds the original and offers the in-dependence.
   In other words we admit that we do not have independent events and through measurements we identify the difference between O_i and O_j by computing the shift of their time distributions. This distance is regarded as the cost required in order to create independence between the time series of the event i and the event j.

pWCET remarks Analyzing high-complexity hardware, like multi-core processor for example, in the classical manner is a difficult or impossible task. The probabilistic approach is suited for such systems where we cannot reason in therms of deterministic behavior and moreover in systems where the cost of the analysis is a crucial factor. We think that the success of the method is in tight relation with the ability to identify safe probabilistic models for highly complex interactions between units and to separate the measured times series of the different interacting components (before and after activation). Another challenge might be the system decomposition unit by unit, as even though all components are activated, they might not participate in all measure series.

2.3.4 RapiTime

Available either as a stand alone tool or as part of Rapita Verification Suite™ (RVS), Rapi**Time**™ is an on-target timing verification and optimization tool for aerospace/automotive from RAPITA Systems ltd. The tool collects execution traces to provide code coverage metrics and execution time measurement statistics in a cost-efficient solution. It also integrates probabilistic methods presented in section 2.3.3 through the integration of the `pWCET` method, [BCP03; pWC].
The main features of the tool, depicted in figure 2.5 are as following:

- *Execution Time Measurement* - execute fewer time measurements that classic dynamic methods to highlights potential problem areas;
- *Worst-case Calculation* - identifies worst-case hotspots in the code that contribute to the global WCET;
- *Performance Optimization* - assistance in the code optimization based on the identification of the hot-spots identified in the previous part;
- Code coverage and debugging.

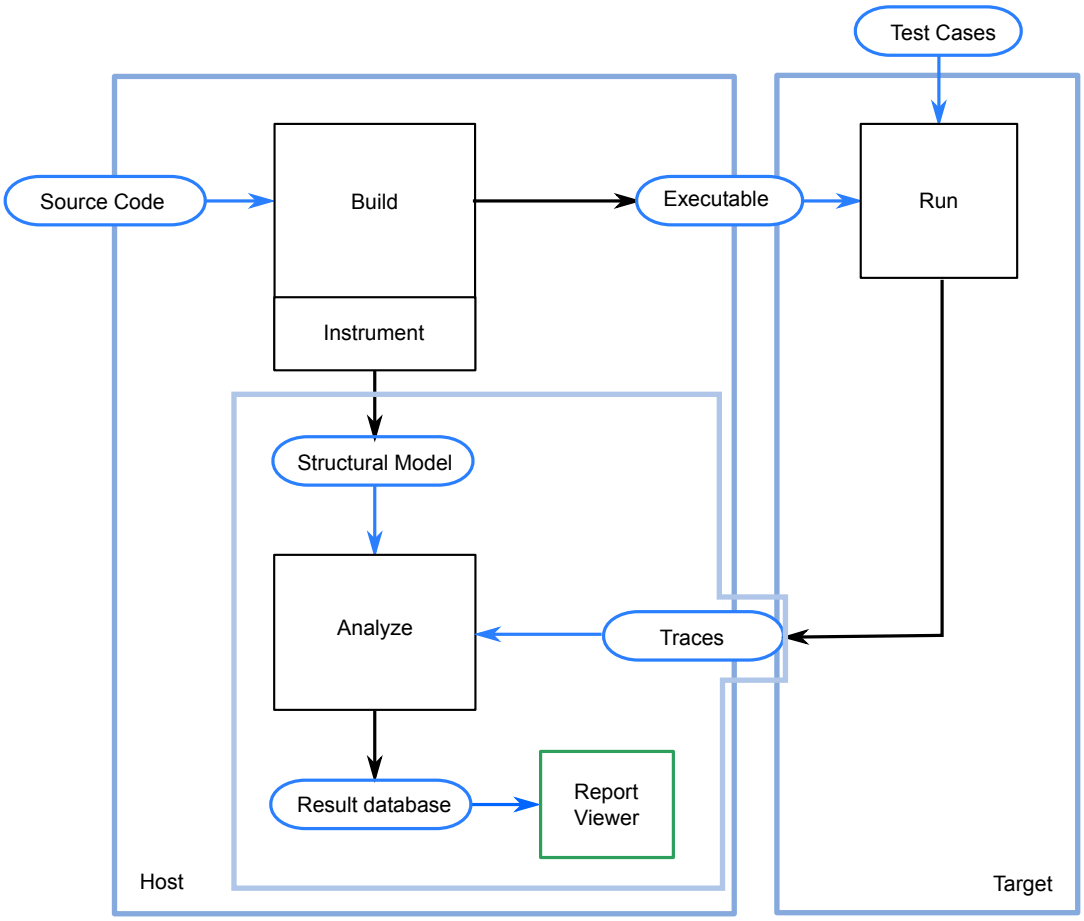


Figure 2.5: Global structure of the RapiTime Tool

2.3.5 Advantages and weaknesses of hybrid methods

The hybrid methods were introduced in order to compensate the lack of complete coverage of dynamic methods and the combinatorial explosion of some complicated static features. One of their main advantages is that they do not use complex abstract models of the hardware architecture, one of the main challenges of static methods and also a source of accuracy loss. Being at the crossroad of the two main method families they also inherit numerous disadvantages from both sides. A majors disadvantage is the lack of strong guarantees regarding the safeness of the assumed worst-case behavior that is covered by measures for which worst-case inputs can not be assumed in all cases. Another drawback is the need, in most cases, of code annotations. Besides the obvious added difficulty, code

annotations may interfere with some temporal analysis requirements as they may not be allowed in some particular certification scenarios.

2.4 Comparison of existing methods

In this section we present a brief schematic overview of the most important methods and tools, introduced in the previous part.

Table 2.2: WCET estimation methods overview

Method name	**Type**	**Entries**	**License**	**Key features**
Stopwatch	*d*	test entry set		
Date/Time	*d*	test entry set		needs OS support
prof and gprof	*d*	test entry set		needs OS support
Time/Counter	*d*	test entry set		needs processor with internal counter
Software Analyzer	*d*	test entry set	commercial	version dependent accuracy
Logic Analyzer	*d*	test entry set		
aiT	*s*	binary	commercial	most complete, might need loop counters, processor dependent version
BoundT	*s*	source code	licensed	IPET, *lp_solve*
Chronos	*s*	source code binary	academic	ILP, processor model, computes an additional observed WCET value.
OTAWA	*s*	binary & partial architectural description	academic	easy architectural description (XML file)
SWEET	*s*	ALF intermediate	academic	focused on flow analysis (as fast as possible),

Table 2.2 – continued from previous page

Method name	**Type**	**Entries**	**License**	**Key features**
		language		
FORTAS	*h*	source code	academic	Model Checking, time annotated state machine.
HEPTANE	*h*	binary	commercial	Has a simulator; ILP.
RapiTime	*h*	source code test entry set	commercial	measurements, fast, probabilistic model, code optimization
probabilistic	*h*	source code		improves the test entry set, alternative for hardware model of complex systems.

Table 2.2 gives an overview comparaison of the presented methods and table 2.3 summarizes their capabilities and limitations in a check list form. This table shows the properties discussed above for each family method. Namely that the *dynamic methods* naturally take into account all the processor features because the *model* used is the actual physical processor. The check list has an asterisk to make the difference from other methods that statically take into account the full extent of the hardware component effect. The methods earn their accuracy check in comparison with the static methods, thanks to the fact that in practice, good results can be achieved in controlled environments featuring a series of restrictions. Static methods are mainly safe but lack accuracy with the exception of aiT which is one of the most mature and complete timing analysis solutions. Hybrid methods can also use the actual hardware as a hardware model which earns them a starred check on the first part. Probabilisitic methods show good result and might be the only solution to take into account less deterministic hardware like multi-cores.

Table 2.3: WCET estimation methods summary - the * stands for a partial fulfillment or a certain difference compared to the general interpretation of validation.

Method name	**Loop**	**Pipeline**	**Hardware Model**	**Adapt.**	**Accuracy**	**Safety**
Stopwatch	✓*	✓*	✗	✓	✗	✗
Date/Time	✓*	✓*	✗	✓	✗	✗
prof and gprof	✓*	✓*	✗	✓	✓	✗
Time/Counter	✓*	✓*	✗	✓	✓	✗
Software Analyzer	✓*	✓*	✗	✓	✓	✗
Logic Analyzer	✓*	✓*	✗	✓	✓	✗
aiT	✓	✓	✓	✓	✓*	✓
BoundT	✓	✗	✗	✗	✗	✗
Chronos	✓	✓	✓	✗	✗	✓
OTAWA	✓	✓	✓	✓	✗	✓
SWEET	✓	✓	✓	✗	✗	✓
FORTAS	✓*	✓*	✗	✓	✗	✗
HEPTANE	✓*	✓*	✗	✗	✗	✓
RapiTime	✓*	✓*	✓*	✗	✗	✓*
probabilistic	✓*	✓*	✓*	✗	✗	✓*

2.5 Conclusions

The state of the art presented in this chapter shows a variety of approaches to determine the WCET of embedded systems from two main families of methods. Even though the dynamic methods represent still the industrial reality of the WCET estimation, static methods are emerging as a solution to the growing complexity of embedded processors, where measurement based methods are showing their limitations.

The most successful static methods use advanced models of hardware in order to finely take into account the software - hardware dependency for execution time estimation. Techniques as abstract interpretation are used to create abstract hardware models ensuring the calculability of the timing result with a certain precision. Some steps are also made to facilitate the modeling of the processor and enable the method to easily take into account additional processor, like the use of parametric processor models (xml, etc.).
The graal of the WCET estimation is precision and adaptability. The following chapter introduces a novel method centred on a formal framework for processor definition that will take a step forward in the adaptability of the precise worst-case analysis.

Chapter 3

HiTAsm Formal Framework

Certification standards, like the ones that can be found in avionics, give precise recommendation regarding the confidence level that functional and non-functional properties must provide. Regarding the non-functional aspects, distinct focus is granted to the bounding of resource consumption. Of particular interest is the timing aspect or the ability to estimate a tight worst-case execution time (WCET) of tasks on a given system. Nevertheless, modern processors have evolved in order to improve the maximum performance throughput with little to no regard to the determinism of their components. Such modern features influence the instruction timings that can be context or history dependent. Therefore the local worst case no longer suffices in the estimation of the global worst case execution time. In order to safely and precisely estimate the WCET of a processor we need a versatile model that can take into account all the possible component interactions and offer the means to confine and control the inherent state space explosion of exploring all the execution scenarios.

In this chapter we introduce a novel extension of the Abstract State Machines, which has a number of interesting features and will be used in our timing estimation analyzer:

- adaptability of the analysis, given by the separation of the processor model and the analysis;
- ease of use;
- preservation of the formal background of the model extensions;

- adaptability to imprecise value analysis;
- state space explosion confinement techniques through hierarchical abstractions and fusions;
- built-in capturing of timing anomalies;

which make our model suitable for the WCET estimation.
Abstract State Machines (ASMs) have been used with success in processor modelling and verification [HC97], and are a good candidate to describe the underlying architecture for system analysis and worst-case execution time estimation. Despite the formal background which makes it suited for proofs, the ASM model can be seen as a simple language and used accordingly with a minimum time to take in hand. The ASM method bridges the gap between the two ends of system development: human understanding of problems and deployment of their solutions by code executing machines. In this sense, ASM is human readable and machine executable which makes it even more suited for system design and analysis. Many approaches exist for integrating time into ASMs, however they are either focused on the verification of the correctness of the specification or on the flexibility of design of embedded systems. Based on the richness of ASMs, we create a model better suited for the analysis of large systems, where a permanent tradeoff between precision and state space explosion can to be made through the selection of component abstraction levels. Our approach is different from others as we use the time information in conjunction with the notion of dynamic turbo-jumps that can vary the duration of the transition from one state to the next one. Furthermore, the temporal model serves as a base to introduce hierarchical levels, similar to refinements however extended to run-time, dynamically controlled by the language in order to optimize the aforementioned tradeoff.
We choose to represent the delay as a semantic information that increments a special *location* used to store the current time associated to a *state*. As for the hierarchical part, we introduce a special oracle-based transition semantics that dynamically selects the most appropriate definition between several levels of component abstraction. We focused on simplicity and specialisation as we only target a subset of real-time systems, like processors and closely connected parts of the environment. Special attention is given to preserving the

mathematical foundation of the original basic ASM model.

3.1 Abstraction and Computer Science

Some of the most interesting properties in computer science are undecidable problems. Some of the problems of theoretical computer science can be descriebed briefly as follows:

Language and Problems: any problem can be restated as a language recognition problem;

Undecidability: undecidable languages that cannot be decided by any Turing Machine (TM);

Rice's Theorem: all nontrivial properties about the language of a TM are undecidable;

Church-Turing Thesis: any mechanical computation can be done by some TM.
We can therefore state that *any nontrivial property about general mechanical computations cannot be decided.*
A useful technique to infer properties about programs or systems is through abstractions. Abstraction in the general sense implies simplification, the replacement of a complex and detailed real-world situation by an understandable model within which we can solve a problem. Finding a good abstraction is not obvious because we are forced to confront the fundamental limitations on the tasks computers can perform and the computation speed of those computers.
Abstraction allows to scale and model all possible runs of a system, however they must:

- be conservative
- try to balance precision and scalability - Flow-sensitive, context-sensitive, path-sensitive.

On the other hand static analysis abstractions do not cleanly match developer abstractions.

In order to analyze safety-critical systems, we need to prove that certain properties are respected for all possible executions of the program on the given platform. Often, the

solution to those problems is quite simple to find, however it may require computational resources that are unavailable or might never be.
To address this computability problem, abstractions can be used in order to simplify the computations. For example, one could choose to use abstractions that despite the fact they model an extended research space, will eventually enable a most efficient property validation.
Any abstraction leads to information loss. Let us take the example of semantics, where all answers based on the abstract semantics are always correct with regard to the concrete semantics. Termination of a program can be proven using the relational semantics for example. However, on the account of the information loss, not all questions can be answered using abstract semantics.
Choosing the right abstraction technique or abstract domain is very important in capturing the needed behavior of the program, in a flexible, adaptable and computable way.
In this chapter we introduce a novel way of hardware abstraction using a temporal *abstract state machine* representation featuring hierarchical abstraction levels that can dynamically change the definition of the processor depending on the needed precision level and on the richness of information on the manipulated values.

3.2 Related Work

Numerous approaches to integrate time into the ASMs exist in the literature, all for a relatively different purpose. The main directions in the related works are focused around the notion of time as a durative action or as an instantaneous action. Timed ASM with instantaneous actions were first introduced in [GH96]. Both paradigms are further developed in [BS02], [CS08] and [OL08] with semantics oriented on verification. In [OL08] the Timed ASM is presented as the moves of agents, synchronised using a system clock. Their concurrency semantics is based on synchronous multi-agent ASMs. Moves can take time and are associated to durative actions. Time is used to specify the duration of a step and it is chosen non-deterministically from the specified interval. The parallel with the analysis of the system design for worst-case and best-case behavior, such WCET is also made. A

detailed presentation of the use of ASMs for precise WCET computation can be found in [BM09b]. The works in [AMMS10] deal with continuos and discrete time, introduce time constraints as linear inequalities, instantaneous actions and delayed actions with the delay chosen non-deterministically. The model is tailored in order to enable first order timed logic (FOTL) to automatically describe runs of ASM. This is achieved mainly for instantaneous actions in a form apt for formal analysis like model checking.
The previous approaches deal with time as an information needed primarily for system verification which imposes certain choices regarding the semantics of the ASM. The approach presented in this work incorporates concepts from previous approaches from the ASM community however it represents the time in the simplest useful manner, close to the basic ASM but on the other hand adapted to generate runs that can reduce the number of processed information. The model is intended to give a clear vision of semantics of timed algorithms and enables easy abstraction of external or internal components.
Regarding the hierarchical aspect of our model, no other attempts were made to integrate a dynamically adaptable level of abstraction of the ASM, to the best of our knowledge.

3.3 Motivation

The model presented in this work is conceived in order to facilitate the abstraction of components rather than to ease the stepwise refinement for system design. We do not choose a dense time domain as we are not interested in arbitrary action refinement (making more internal steps visible). The ability to insert, between any two moves, an intermediate move, at an intermediate time point is an useful feature from the design point of view. However we focus less on the ease of conception as our model will be less used to conceive the target system but rather for *refinements* as abstractions from a more detailed definition to a more abstract one. Nonetheless, our framework is still able to refine a processor model down to a unitary interval that can by associated to a cycle which is sufficient for our purpose.
The ability to model a processor in basic ASMs as a straightforward step was proven in [Gau95] and besides, as we stay close to this model semantically and syntactically, we can

always start the modelling of the processor without the timing information, following the classic stepwise refinement technique if ever needed.
Even though not presented explicitly in this paper, composition mechanisms that enable partitioning and the reuse of components, like the ones presented in [Anl00] and [OL08] are possible through the use of the update set composition presented in definition 25.
Based on the compositional operation we can also generate different runs of the model corresponding to abstract traces. Every time a certain imprecision in the value analysis of the code introduces an interval delay, the execution is split into different runs according to the information on the timing anomalies occurrence. A list of locations that can generate timing anomalies is used in order to trigger the parallel execution only for those values of the interval that can cause problems. For all the other locations the monotonic assumption on the execution time is safe, therefore we can take the maximum of the delay interval, avoiding unnecessary runs. Whenever the execution is split, we can accumulate the update sets generated by each parallel consecutive state without applying it to the splitting state. If we do not have any guarantee on the global timing influence of local times we verify all the options and after a certain number of steps (related to the particular architecture and mostly to the pipeline's depth), when we identify the right trace, we can rollback to the initial state. The mathematical foundation of the timed ASM states as algebras will also ease the identification of relations between components and states. In this manner the equivalent or the less time consuming states are merged by default.

3.4 Notational preamble

In the following chapter we used an unified notational system to give the semantics of our framework, mainly inspired by the reference book in abstract state machines, [BS03]. The set of states (ASM, Timed ASM, HiTAsm) will be described using $\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \ldots$, precisions will be made about the precise type of state. States from these sets are denoted by $\mathfrak{A}', \mathfrak{A}'', \ldots \mathfrak{A}^{(i)}$ where $\mathfrak{A}' \stackrel{\text{not}}{=} \mathfrak{A}^{(1)}$.
We will use P, Q, R to denote rules, U, V for update sets freshly generated by rules in a given state and $\mathcal{U}, \mathcal{V}$ for delayed update rules sets that accumulate all the previous not yet

fired updates. The interpretation of a rule R in state $\mathfrak{A}$:

$$[\![R]\!]^{\mathfrak{A}} \stackrel{\text{def}}{=} U_R^{\mathfrak{A}} \tag{3.1}$$

yields a new state $\mathfrak{A}'$ with:

$$\mathfrak{A}' = \mathfrak{A} + U_R^{\mathfrak{A}}. \tag{3.2}$$

When it will be obvious in which state the update set was generated, we will omit $\mathfrak{A}$ from $U_R^{\mathfrak{A}}$.

The semantics will further be described using:

$$\frac{\textit{further interpretations}}{\textit{things to interpret, in which state, under what constraints}}\textit{condition} \tag{3.3}$$

For example, the interpretation of rule *skip* in state $\mathfrak{A}$ will generate the update set ϕ:

$$\frac{}{\textit{yields}\,(\textbf{skip}, \mathfrak{A}, \phi)} \tag{3.4}$$

The interpretation of the sequential ASM rule, P **seq** Q, generating the update sets U and V respectively under the variable assignment ζ and the consistency condition is written:

$$\frac{\textit{yields}\,(P, \mathfrak{A}, \zeta, U)\ \textit{yields}\,(Q, \mathfrak{A} + U, \zeta, V)}{\textit{yields}\,(P\ \textbf{seq}\ Q, \mathfrak{A}, \zeta, U \oslash V)}\ U \textit{ and } V \textit{ consistent} \tag{3.5}$$

meaning that we must first interpret the P rule that yields the update set U and then interpret rule Q in state $\mathfrak{A} + U$.

3.5 Abstract State Machine

The ASM thesis was introduced by Yuri Gurevich as a reaction to the Turing thesis, [Gur95b]. One of the most general notions of states and dynamic state changes modern mathematics can offer is (static) algebras as states and guarded destructive assignments for abstract functions as basic dynamic operations.

Gurevich thought of computation as the evolution of a state. It can also be seen as a finite state machine where states are Tarski structures. Static algebras are first-order structures with purely functional vocabulary. Therefore the notion of Tarski's structures was tweaked

by adding the Boolean values and using the boolean functions to represent relations.
In the science of universal algebra, first-order structures with only functional vocabularies are called algebras. An algebra is a Tarski structure without the relations. It is known from the vast mathematical logic experience that any static mathematic reality can be faithfully represented by a first-order structure.

3.5.1 ASMs in a nutshell

Abstract State Machines are a computational model based on mathematical structures. The choice for the mathematical structure is static algebra which can be seen as a state. The model of the run are groups of finite updates that make transition from one state to another. The run proceeds either until a final state is reached or for an infinite number of steps. The evolution of states is achieved through the notion of dynamic algebras by the use of dynamic functions that change the content at certain locations. All the function names form a set called the *vocabulary*. A special function defined on this set is in charge with the interpretation of function names. For example, every vocabulary has the Boolean set $Bool = \{true, false\}$ as well as the natural boolean relation names as static functions. The interpretation of static functions is fixed throughout the run, and translates for the Boolean set as the natural boolean interpretation. Locations can be defined as functions applied to the argument. The interpretation of the functions is the content of the locations. Changing (or defining, if there is none) the value location (represented by the functions at the given parameters)

$$f(t_1, ..., t_n) := t \tag{3.6}$$

is done through *updates*. The interpretation of a function $f = functionName$ is depicted in figure 3.1 where v_i is the interpretation of the term t_i.

The 0-ary function has therefore the following interpretation: The ASM is a finite set of guarded updates:

```
1 if Condition
2   then Update
```

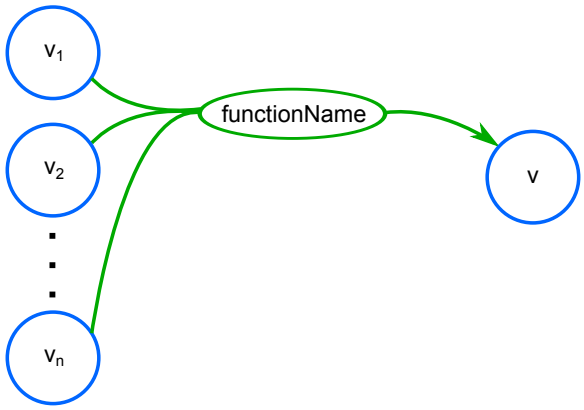


Figure 3.1: The interpretation of the Asm n-ary function $f(t_1, ..., t_n) := t$

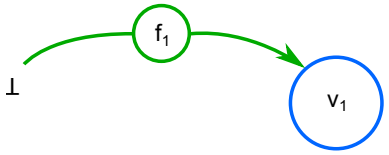


Figure 3.2: The interpretation of the Asm 0-ary function $f_1 := t, [\![t]\!]^{\mathfrak{A}} = v_1$

ASMs can describe algorithms at their appropriate abstraction level. System design can be performed incrementally with the help of simultaneous updates which help avoiding an explicit description of the intermediate storage.

An example of an ASM code of swapping the values of two functions is presented in Listing 3.1.

Listing 3.1: ASM example of instantaneous updates - variable value swap

```
1 rule valueSwap = {
2   a := b
3   b := a
4 }
```

The interpretation of the update rule involves an update set. Therefore, prior to their *execution*, all known updates are collected and stored into an *update set* and *fired* simultaneously. The firing of updates modifies the interpretation of nullary functions a and b.

Therefore there is no need for the explicit description of a memory container c storing

the intermediate value of a. We say that ASMs abstract the memory interpretation of the system.

3.5.1.1 Turing Machines

As we stated at the begining of the chapter, the ASMs were introduced as a reaction to the Turing thesis, so let us have a look on how to use the ASMs to model a state of a Turing machine.
The base set of the structure: $Control \cup Alphabet \cup Tape$ which is disjoint from $\{true, false, undef\}$.

- $Control$ is the set of states of the finite control. Each element of $Control$ has a function name in the vocabulary. In addition, there is a dynamic distinguished element $CurrentControl$ in Control.
- $Alphabet$ is the tape alphabet. One of these elements is called $Blank$.
- $Tape$ is the set of integers representing tape cells. It has the unary operations $Succ$ and $Pred$. There is also a dynamic function $Head$ that takes values in $Tape$ (which is considered infinite).

Finally, we have a unary function

$$Content : Tape \implies Alphabet \tag{3.7}$$

which assigns $Blank$ to all but finitely many cells (and which assigns $undef$ to every element outside of $Tape$). The following self-explanatory rule reflects a Turing instruction:

Listing 3.2: ASM example of instantaneous updates - variable value swap

```
1 if CurrentControl = q1 and Content(Head) = s1 then
2   doInParallel
3     CurrentControl := q2
```

```
4     Content(Head) := s2
5     Head := Successor(Head)
```

The whole program of a Turing machine can be written as a *do − in − parallel* block of rules like that.

3.5.2 ASMs and hardware modelling

Abstract State Machines were first introduced under the name of evolving algebras. Indeed, algorithms, in the general meaning, are seen as computation steps applied to states, as algebras, that change the interpretation of *functions* names at the specified *locations*. *Moves* from one state to another consist in the *simultaneous* evaluation and execution of all the *update rules* that are applied in parallel, in the same time, changing locations in the previous state.

A *signature*, Σ, is a finite collection of function names. Signatures are also called *vocabularies*. Each function name has an *arity*, a non-negative integer.

$$n\text{-}arry\ functions \begin{cases} if\ n = 0 \quad , & variable,\ value\ of\ a\ memory\ address; \\ if\ n = 1 \quad , & content\ of\ a\ memory\ address,\ of\ a\ register; \\ if\ n > 1 \quad , & memory\ mappings. \end{cases} \tag{3.8}$$

All function interpretations are an element from the *domain* or *universe* of the function, $f : X^n \implies X$. ASM functions are *total*, however *partial* functions can be represented through the use of predefined value *undef*. In order to do that, values where the function is not defined will return $undef$.

Relations are represented as functions interpreted in the *Bool universe*, $f : X^n \implies Bool$, where $Bool = \{true, false\}$ is a predefined ASM universe. The universe of a relation f is defined by $\{\forall x \in R^n | f(x) = true\}$. A *location* is defined as the tuple $l = (f, \bar{t})$, where f is a function name from the vocabulary and $\bar{t}$ the argument of the function. An ASM *moves* from one state to another through the evaluation of *rules* and the application of the *update* rule. An *update*, $u = (l, v)$, changes the interpretation of a function at the specified location l with the value v. Functions can be *static* or *dynamic* and are further differentiated into *internal*, *external*, *controlled*, *monitored*, etc. Static functions can be seen as *constants*,

their interpretation does not change throughout the ASM's run. On the other hand dynamic functions can have their interpretation changed through updates.

3.5.3 ASMs and hardware abstraction

The ASM theory states that the internal structure of the superuniverse is not important. In the following we present such a case, which develops the example introduced in [**?**]. The example consists in the implementation of logical functions through ASM functions.

Let Σ_{Bool} be the vocabulary of our boolean algebra with $\Sigma_{Bool} = \{And, Or, Not, Equal\}$ and Bit the its univers, $Bit = \{0, 1\}$.

$$\begin{aligned} True &\Longrightarrow Bit \\ False &\Longrightarrow Bit \\ And : Bit \times Bit &\Longrightarrow Bit \\ Or : Bit \times Bit &\Longrightarrow Bit \\ Equal : Bit \times Bit &\Longrightarrow Bit \\ Not : Bit &\Longrightarrow Bit \end{aligned} \tag{3.9}$$

with the following interpretations for the state $\mathfrak{A}$:

$$\begin{aligned} True^{\mathfrak{A}} &:= 1 \\ False^{\mathfrak{A}} &:= 0 \\ And^{\mathfrak{A}}(0,0) &:= 0 \\ And^{\mathfrak{A}}(1,1) &:= 1 \\ &\cdots \\ Not^{\mathfrak{A}}(0) &:= 1 \\ Not^{\mathfrak{A}}(1) &:= 0 \end{aligned} \tag{3.10}$$

Let $\mathfrak{B}$ be a new state with the superuniverse $|\mathfrak{B}|$, the power set of the set of binary values. $\Sigma_{Bool} = \mathcal{B} = \{\{\}, \{0\}, \{1\}, \{0, 1\}\}$ Our function names have now the following interpretations:

$$\begin{aligned} True^{\mathfrak{B}} &:= \{0, 1\} \\ False^{\mathfrak{B}} &:= \{\} \end{aligned} \tag{3.11}$$

Let $NOT(t)$ be all the elements of the superuniverse except t. We can define *Not* formally as:

$$
\begin{aligned}
Not^{\mathfrak{B}}(t) &= \mathcal{B} \setminus t = \{n \in \mathcal{B} | n \neq t\} \\
And^{\mathfrak{B}}(t_1, t_2) &= t_1 \cap t_2 = \{\forall n \in \mathcal{B} | n \in t_1 \wedge n \in t_2\} \\
And^{\mathfrak{B}}(\{\}, \{\}) &:= \{\} \\
And^{\mathfrak{B}}(\{\}, \{0\}) &:= \{\} \\
And^{\mathfrak{B}}(\{1\}, \{0\}) &:= \{\} \\
And^{\mathfrak{B}}(\{1\}, \{1\}) &:= \{1\} \\
&\cdots \\
And^{\mathfrak{B}}(\{0,1\}, \{0,1\}) &:= \{0,1\} \\
Or^{\mathfrak{B}}(t_1, t_2) &= t_1 \cup t_2 = \{\forall n \in \mathcal{B} | n = t_1 \vee n = t_2\} \\
Equal^{\mathfrak{B}}(t_1, t_2) &= \{\forall n \in t_1 | n \in t_2\}
\end{aligned}
$$

Let's now introduce a function $\alpha : |\mathfrak{B}| \implies |\mathfrak{A}|$, for a subset $X \subset |\mathfrak{B}|$ we define

$$
\alpha(X) = \begin{cases} 1 & , \quad if\, X = \{0,1\}; \\ 0 & , \quad otherwise. \end{cases} \tag{3.12}
$$

We say that α is a *homomorphism* from $|\mathfrak{B}|$ to $|\mathfrak{A}|$ if $\alpha\left(\mathfrak{B}(l)\right) = \mathfrak{A}(\alpha(l))$ for each location l in $\mathfrak{B}$, where $\mathfrak{B}(l)$ represents the value at the location l. We need to extend the function α to locations, or more precisely to function names. Let $l = (f, (b_0, \cdots, b_n))$ be a location of $\mathfrak{B}$, where f is a function name and a_i are elements of $|\mathfrak{B}|$. The content of the location l in $\mathfrak{B}$ is $f^{\mathfrak{B}}(b_0, \cdots, b_n)$ therefore we define

$$
\alpha(l^{\mathfrak{B}}) = \alpha(f^{\mathfrak{B}}(b_0, \cdots, b_n)) = f^{\mathfrak{A}}(\alpha(b_0), \cdots, \alpha(b_n)) = f^{\mathfrak{A}}(a_0, \cdots, a_n). \tag{3.13}
$$

For $0 - ary$ functions f, $\alpha\left(f^{\mathfrak{A}}\right) = f^{\mathfrak{B}}$, which actually means that we get the corresponding function in the other state.

The figure 3.3 illustrates the ASM principle of information hiding, stating that the inner structure of the state it's not important. The operations that can be performed on the elements of the state are what matters. We can also see in the figure that the superuniverses of

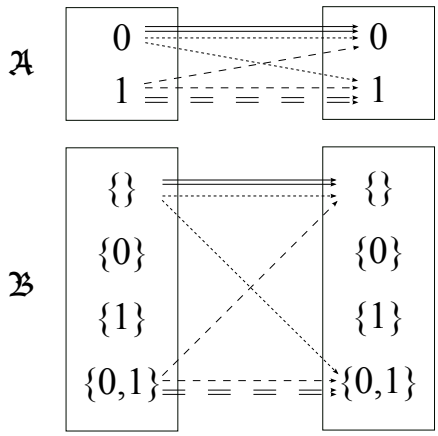


Figure 3.3: Binary function application on two isomorphic states

the two abstract states can be mapped to each other and the interpretations of the functions agree on corresponding elements.

$$\alpha(\mathfrak{B}(l)) = \alpha\left(\mathfrak{B}((And^{\mathfrak{B}}, (\{0,1\},\{0,1\})))\right) = \alpha((And^{\mathfrak{B}}((\{0,1\},\{0,1\})) = \alpha(\{0,1\}) = 1 \tag{3.14}$$

We must now compute $\mathfrak{A}(\alpha(l))$ where

$$\begin{aligned}\alpha(l) = \alpha((And^{\mathfrak{B}}((\{0,1\},\{0,1\})) = \alpha((And^{\mathfrak{B}}, (\{0,1\},\{0,1\}))) = \\ = (\alpha(And^{\mathfrak{B}}), (\alpha(\{0,1\}), \alpha(\{0,1\})))) = (And^{\mathfrak{A}}, (1,1))\end{aligned} \tag{3.15}$$

therefore

$$\mathfrak{A}(\alpha(l)) = \mathfrak{A}(And^{\mathfrak{A}}, (1,1)) = 1. \tag{3.16}$$

From equations 3.14 and 3.16 we obtain $\alpha(\mathfrak{B}(l)) = \mathfrak{A}(\alpha(l))$, that can be verified for all the locations.

In our HiTAsm method we extend the information hiding principle to dynamic refinements of ASMs and exploit it to model hardware at different abstraction levels in a processor's analysis that guides the needed precision level of the hardware component. For more details on our HiTAsm method, please refer to [PMB13a].

3.5.4 Stepwise Refinement of ASMs

The ASM method enables stepwise refinement abstract operational modeling that bridges the gap between virtually all the system design steps, [Bor03].
The starting point is the contract with the costumer, modeled in the human readable language of ASMs that doesn't require specifics of the inner math foundation of the model. The process ends with the generated code from the detailed, refined specification or even the ASM as an executable model.
The method starts thus with the *ground model*, a correct and complete task formulation derived from the application-domain understanding through the requirements analysis. The model is easy to understand and can be validated by the costumer and will constitute the starting base for the system designer.
The ground model follows a series of steps corresponding to the incremental refinements. These transformations can be proven correct at every iteration. The final model will be linked to the *generated code* that will run on the target device. By proving that every transformation step is equivalent to the previous, validated model, it is shown that the end model correctly solves the problem formulated in the ground model.

3.6 Time and Abstract State Machines

The notion of time is important to model real-time systems. Hard real-time systems emphasize the notion of safety with regard to the respect of deadlines. Determining the WCET is essential for this kind of systems and therefore the notion of time in the hardware model used for the estimation. Several approaches can be taken to integrate time in the ASMs.

As a function in the Basic ASM The most natural extension is to start from the basic ASM, expanding the notion of instantaneous updates that generate update sets in parallel with simultaneous effect, if they are consistent. The effect of the updates is to change the interpretation of the updated functions at the specified location. The consistency of the updates implies that parallel updates do not affect the interpretation of the same function at

the same location and must be ensured by the user.

The time is a function with predefined interpretation that is updated in every new state, giving the *current time*. The semantics of the time update can take several forms.

- A simple increment corresponding to the time of the longest update. At every fixed time interval Δt actions occur like in a processor where the *clock* signal activates events. Nevertheless a *time-accurate* model can also be easily extended by adding condition for firing updates only if the next state is different (with regards to the Superuniverse or a monitored subset). In this case we have a Δt_i associated to every state $\mathfrak{B}_i$.

  The next choice is to either use the time function inequalities in guards or not. Disregarding the time function in guards enables us to keep the same design paradigm as in basic ASMs, which was successfully used to model several processors. Adding them will change the processor modelling as we can have events triggered at a certain time.

- Through the notion of *submachines* in the Sequential Turbo ASMs, by adding time information to each submachine.

A parallel can be made between the two approaches. If we disregard the hierarchical composition of submachines, the two models are equivalent. The time for each transition is given by the update that takes the maximum amount of time. The use of sub machines is a way to voluntarily hide away internal updates. The resulting time is the time of the parent or, if not present, the maximum time of the children submachines. Certain constraints apply to the update sets, namely in order to be consistent, the updates of the submachines must not interfere with locations exterior to the parent. Therefore the sequential ASM approach can be regarded as a more compact basic ASM. A state of the compact ASM represents several states of the sequential one where only certain rules were activated and the rest of the locations were left unchanged.

3.6.1 Adding time in basic ASMs

ASM is a state-based model, characterized by an explicit notion of state as static algebras. The state is a first class citizen while the events are secondary level citizens in the form of moves between states. The run of basic ASMs consists in a succession of moves. A move applies the update set generated by all the update rules with valid guards in that particular state. All updates are instantaneous and applied in parallel. This influences the way the system is designed, in our case the processor. In order to describe its exact behaviour, we must give a *step by step* definition of all the interacting components.

Even though there is no explicit notion of time in the basic ASM, there is a clear notion of sequentiality that can be very easily applied to the model of a processor where all *changes* are governed by a central clock and applied in the same time after a clock tick. Therefore the basic ASM model can be seen as a transition system that gives a picture of the state before and after the clock tick. By adding a simple counter we can represent the notion of clock tick.

We introduce a simple way to abstract certain components that take several basic ASM moves to complete by adding the time information to the final group of updates. Therefore we introduce *delayed* updates that model a whole set of rules in a *blackbox* style. The execution of delayed rules implies the use of a global time scale that will be discussed in the following.

We use the notation δ to refer to the update delay, that is semantically connected to the basic ASM through a special location, a 0-ary function CT, holding a term that evaluates to a value defined on $\mathcal{T}$. Let $(l, v, \delta) \in U$ be an *delayed update*, $l \in \mathcal{L}$ a location of the HiTAsm state $\mathfrak{A}$ where δ is the value of the delay after which the location l will take the value v. We can now introduce the $delay : U \times \mathcal{L} \implies \mathcal{T}$ function that applies to the update set associated to an update rule in U and extracts the time information of that rule. If the rule has no delay information, then it returns 1.

$$delay(U, l) = \begin{cases} \delta & , \ if \ \exists (l, v, \delta) \in U, \ with \ \delta = [\![t]\!]^{\mathfrak{A}}_{\zeta} \in \mathcal{T}; \\ 1 & , \ otherwise. \end{cases} \tag{3.17}$$

Note that in this case t has the normal interpretation of terms in the state $\mathfrak{A}$, given by ζ and can be any term that evaluates to a time value in $\mathcal{T}$. From the system design point of view, interpreting durations in this way has the advantage of enabling the scheduling of a certain location's update depending on the update of another. For example, let us consider two location updates $u_1 = (l_1, v_1, \delta_1)$ and $u_2 = (l_2, v_2, \delta_2)$ with u_2 depending on the result of the u_1 update. We can therefore define $\delta_2 = delay(U, l_1) + t_{offset}$, with $t_{offset} > 0$ ensuring that l_2 will be updated after the location l_1.

For example, it could be used to express pipeline stalls in order to avoid data hazards in a pipelined implementation of a processor. Let us consider the consecutive pipeline stages instruction decode, IF/ID and instruction execution ID/Ex stages and their pipeline registers. A data hazard situation could be an *add* instruction whose operands are dependent of a *load* instruction that takes multiple cycles to execute. In this case, data forwarding can not avoid the addition instruction's stall. Normally a scoreboarding mechanism is implemented for the dynamic pipeline scheduling. However if we write the *add*'s delay for the updates in the ID/Ex stage depending on the delays of correspondent *load* updates (for the forward registers or write back stage) we no longer need to explicitly give the definition of the scorebording unit.

Time also enables the notion of time-accurate transitions. The idea is to compact the run by only making moves that change the locations of the state besides the current time. In this case we obtain a conjunction of update sets that makes a move to an equivalent state (the first that is *different*) with the same equivalent duration $\delta = \sum \delta_i$.

If every rule can have an associated duration, then we can have in the same state update sets associated to different durations and the following scenarios.

Let $\mathfrak{A}_0, \mathfrak{A}_1 = \mathfrak{A}_0 + U_0, \ldots, \mathfrak{A}_{i+1} = \mathfrak{A}_i + U_i, \ldots$ be a run. The next state $\mathfrak{A}_{i+g}$ is obtained by firing all the updates from the update set U_i in state $\mathfrak{A}_i$. Depending on the delay information found in the update set, we can have the following scenarios:

3.6.1.1 No timed updates

The run is therefore equivalent to the one of a cycle-accurate model.

$$\mathfrak{B}_0, \mathfrak{B}_1 = \mathfrak{B}_0 + U_0, \ldots, \mathfrak{B}_{i+1} = \mathfrak{B}_i + U_i, \ldots \tag{3.18}$$

$$U_{j\in[0,i]} = (l, v) i.e. \delta = 0. \tag{3.19}$$

If no time update functions are present, the CT function is by default incremented by 1 after all the updates are effective.

3.6.1.2 Single timed updates

Only one timed update occurs in the update set $U_{j\in[0,i]} = \{(l, v, \delta)\}$, equivalent of a time accurate run, as we make the move directly to a state that has *significant* updates.

$$\mathfrak{B}_0, \mathfrak{B}_1 = \mathfrak{B}_0 + U_0, \ldots, \mathfrak{B}_{i+1} = \mathfrak{B}_i + U_i \tag{3.20}$$

with $U_{j\in[0,i]} = (l, v, \delta)$.

3.6.1.3 Mixed updates

Both timed and untimed updates, or timed updates with different durations are specified,

$$U_i = \{(l_i, v_i, \delta_i), (l'_i, u'_i, \delta'_i), , \ldots\}, with\ \delta_i^{(m)} \neq \delta_0^{(n)}. \tag{3.21}$$

In this case we first apply the update set associated with the smallest $\delta_{min} = min(\overline{\delta_i})$ and subtract δ_{min} from all the other durations. After each update, a move is made to a new state, meaning that all the rules are evaluated again and new updates are added to the update set. This is a way to encapsulate rules in a single black-box rule or in other words to replace a rule definition with a less refined version that hides the inner actions. When all the updates in a state take one cycle, we can say that we have a precise definition of all the units.
Updates that take several cycles hide away some inner updates and use less detailed definitions of the respective unit.
When all the updates in a state take the minimal amount of time (that exists according to the temporal properties of the model, listed below), one cycle for example, we can say that we have a precise definition of all the units. Updates that take several cycles can conceal some inner updates and look like less detailed definitions of the respective unit. Certain constraints apply to the use of such *delayed* versions, like the absence of *critical locations* with regards to update sets that must be applied earlier.

In order to make a valid delayed move, we must have by conception disjoint sets between the delayed update set and the update sets that will be applied before it. This check can be automated. Given the fact that we only have a limited number of rules that describe the processor, we can create dependency lists of rules sharing locations.

For a delayed rule two types of dependencies exist: on intermediate and on final locations. If a dependency on the intermediate locations exists, the use of the delayed rule is forbidden. If only dependencies on the final locations exists, the execution is natural as the updates of the depending rules will be delayed by their updates. After the delayed update takes place, the rule guards are evaluated and that will either validate or invalidate the guards of the depending rule.

```
1  if not(delayedActions) then
2    forall u in U
3      forall l in u
4        l := v
5      endforall
6    endforall
7    CT := now + 1
8  else
9    let u = min(updateSets) in
10     forall l in u
11       l := v
12     endforall
13     CT := now + dur(u)
14     forall v in U-u
15       delay(v):=delay(v)-delay(u)
16     endforall
17   endlet
18 endif
```

Adding delays to update sets generates the following run cases:

1. all delays are equal to one (clock cycle)

2. one component has a *n*-cycle delay and no dependencies with other locations

3. several components have a delay superior to one

For the first case we can even eliminate the delay information as the minimum and basic delay would be one. Concerning the semantics of the run, all update sets are applied at every move except for those who have a non-unitary delay associated to their update set. Their delay is a multiple of a clock cycle and gets decremented by the number of clocks that the move takes.
The second case can be used to simulate a time accurate model, making a move directly to the state that brings a significant update. The whole interest of this model is to have a more compact run, eliminating useless states.
Therefore the model needs to know when to automatically make a larger move and advance the execution with the respective number of cycles.
Our semantics is simpler and closer to the basic ASMs as we see the delayed rules as a black box rule and not as a sequential composition of the potentially hierarchic submachines, performing sub-computations on locations accessible by the other rules like in the Turbo ASM presented in [**?**].
One way to automatically make the turbo-transitions is to check the update set. The update set is the union of all the locations updates and their associated delays, as we can see in the detailed definition. If no other one cycle update exists but only a *n*-cycle delayed update set (for two consecutive cycles) then we can make the move directly after *n*-cycles.
We first give the detailed mathematical definition of our model and then prove some temporal properties of the timed state transit system.

3.6.1.4 Detailed definition

The semantics of the delayed updates can be completely simulated with basic ASMs constructs and the introduction of the pre-interpreted sort $Time$ and the external pre-interpreted function CT that gives the time associated to the current state.

```
1 executedStatus := true
2 ...
```

```
3  if C then
4    if executedStatus = true then
5      CD := CT + delay
6      executedStatus := undef
7    endif
8    if CT = CD then
9      delayedUpdateRule
10   endif
11 endif
```

The semantics in the above listing imposes that the guard C remains valid throughout the whole delay period, which might be a necessary constraint. Otherwise, we can use a definition similar to the *control state ASMs* to simulate the delayed application of the rule like in the listing below, where `new(CTD(ruleName))` is a function that generates a new location to store a delay for the rule `ruleName`.

```
1  if Cr then
2    if ctl_state = 1 then
3      CTD(ruleName) := CT + delay
4      ctl_state := 0
5    endif
6  endif
7  if CT = new(CTD(ruleName)) then
8    delayedUpdateRule
9    ctl_state := 1
10 endif
```

Other than the verbosity of this approach it would be furthermore difficult to express the delay as an interval, where the delay can take all the values from the set $\{\delta_{min}, \ldots, \delta_{max}\}$. One of the natural constraints of consistency regarding the updates is that we cannot have multiple updates for the same location (in the same time i.e more than one value for the same location in the same update set). Adding a delay to the update raises the question of

allowing updates during that delay period or not. Forbidding the firing of the same rule until the previous delay has elapsed, eliminates the possibility to pipeline the delayed rule from the design point of view.

3.6.1.5 States and Update Sets

For a processor modelling, a discrete interpretation of time is sufficient as all time informations of the description are multiples of a cycle. Therefore we can only react to external actions after the next cycle tick, which we consider sufficient in our WCET estimation context.
We extend the definition of the signature from [**?**], Section 2.4, in order to handle time.

Definition 17 (Signature): A signature Σ is a finite collection of function names. Each function name f has an arity, a non-negative integer. Every ASM signature contains the static constants *undef*, *true*, *false* and also the external dynamic pre-interpreted function CT that gives the value of the *current time* from the pre-interpreted sort of time, denoted by $\mathcal{T} \in X$.

The definitions of the state and location remain unchanged and are presented like in [**?**] from which we also adopt some of the notations.

Definition 18 (State). A state $\mathfrak{A}$ for the signature Σ is a non-empty set X, the superuniverse of $\mathfrak{A}$, denoted by $|\mathfrak{A}|$, together with interpretations of the function names of Σ.

Definition 19 (Location). A *location* of $\mathfrak{A}$ is a pair $(f, (a_1, \ldots, a_n))$, where f is an n-ary function name and $a_1, \ldots a_n$ are elements of $|\mathfrak{A}|$. The value $f^{\mathfrak{A}}(a_1, \ldots, a_n)$ is called the content of the location in $\mathfrak{A}$.
We use a special location called the current time, $CT \in \Sigma$, to store the time progress of the run by adding all the delays δ associated to updates. In cycle accurate models, we can obtain the current time solely from the number of elapsed clock ticks. However the ticks are no longer constant in our case.

We write $\mathfrak{A}(l)$ for the content of the location l in $\mathfrak{A}$ and also $\mathfrak{A}(\delta) = [\![\delta]\!]^{\mathfrak{A}}$ for the value (interpretation) of the delay in the state $\mathfrak{A}$. Depending on the interpretation of the duration

δ that we allow, we can introduce a first order temporal logic in our model. This can be accomplished by allowing the delays to be terms and associate the same formulas like for the other locations.

The definition of updated set is modified in order to allow delayed updates.

Definition 20 (Timed update and update set). A timed update for $\mathfrak{A}$ is a pair (l, v, δ), where l is a location of $\mathfrak{A}$, v is an element of $|\mathfrak{A}|$, the value of l after the delay δ with regards to the current time, CT. An update set groups all updates that are scheduled to be fired with regard to the current state.

The delay δ cannot be infinitely small and is multiple of a smallest time interval, that can be associated to a system clock tick, for example. The location l keeps its old value until the delay δ has elapsed with regards to the current time when the update was scheduled to be fired. The update is trivial, if v is the content of l in $\mathfrak{A}$.
Different delays for updates imply that the update set of a state is not always empty after the move to the next state. However adding the notion of update set more closely in the state would modify too much the basic ASM structure. We prefer to consider that the update set is partially *consumed* in the current state and the rest is *passed* to the next one. In other words, the next state can already have a non-empty update set before starting the evaluation of its rules. Therefore we distinguish two kinds of update sets.

- $U_{\mathfrak{A}}$ - yielded by the machine's main rule, in state $\mathfrak{A}$;
- $\mathcal{U}$ - updates inherited from the last move that did not fire, with their *delay* decremented by the *minimum delay* δ_{min} cf. definition 22.

Due to the parallelism of the updates, we must avoid the update *clash*, when a function is updated at the same arguments in the same time. We therefore modify the definition of the *consistent* update set.

Definition 21 (Consistent update set). An update set U is called consistent, if it has no clashing updates, i.e. if for any location l, and elements $v_1, v_2, \delta_1, \delta_2$ it is true that if $(l, v_1, \delta_1) \in U$ and $(l, v_2, \delta_2) \in U \cup \mathcal{U}$, if $\delta_1 = \delta_2$ then $v_1 = v_2$. If the delays are different, $\delta_1 \neq \delta_2$, they are no constraints on the location's update value, in other words, the

update set accumulate scheduled updates for locations.
The consistency of an update set $U_{\mathfrak{A}}$ generated in state $\mathfrak{A}$, must be checked against locations in $U = U_{\mathfrak{A}} + \mathcal{U}$.

If an update set U is consistent in a given state, then it can be fired. In the generated new state the dynamic functions, that had associated updates with delay values equal to the time step of the move, are changed according to U. All the other delays from the update set are decremented by δ_{min}.

Definition 22 (Minimum delay) Using the *delay* function we can define the minimum delay δ_{min} of an update set $U = U_{\mathfrak{A}} \cup \mathcal{U}$ for the state $\mathfrak{A}$ which is the value of the duration needed to make a move from that state to the next state $\mathfrak{A} + U$:
$\delta_{min} \in \{\delta_i = delay(U, l_i) \mid delay(U, l_i) \leq delay(U, l), \forall l \in |\mathfrak{A}|\}$.

Definition 23 (Firing of updates). The result of firing a consistent update set U in a state $\mathfrak{A}$ is a new state $\mathfrak{A} + U$ with the same superuniverse as $\mathfrak{A}$ such that for every location l of state $\mathfrak{A}$:

$$(\mathfrak{A} + U)(l) = \begin{cases} v & , \quad if\ (l, t, \delta) \in U.\delta = \delta_{min}, [\![t]\!]^{\mathfrak{A}}_{\zeta} = v \\ v_t & , \quad if\ l = CT, [\![t]\!]^{\mathfrak{A}}_{\zeta} + [\![\delta_{min}]\!]^{\mathfrak{A}}_{\zeta} = v_t \\ \mathfrak{A}(l) & , \quad if\ (l, v, \delta) \in U.\delta > \delta_{min} \\ \mathfrak{A}(l) & , \quad otherwise. \end{cases}$$

The state $\mathfrak{A} + U$ is called the *sequel* of $\mathfrak{A}$ with respect to U, therefore $(\mathfrak{A} + U)(l)$ is the content of the location l after firing the updates in U. Since U is consistent, the state $\mathfrak{A} + U$ is still determined in a unique way, and some locations will have a new content in $\mathfrak{A} + U$ with respect to $\mathfrak{A}$. If we do not have an *immediate* update (i.e. $\delta = 1$), the locations having $\delta = \delta_{min}$ will be updated directly after the *minimum delay*.

The *state difference* is defined as the unique set of non-trivial updates that can be fired to reach one state from another state with the same signature and the same superuniverse.

Definition 24 (State difference). Let $\mathfrak{A}$ and $\mathfrak{B}$ be two states with the same superuniverse. Their difference is defined by $\mathfrak{B} - \mathfrak{A} = \{(l, \mathfrak{B}(l), \delta_{\mathfrak{A}\mathfrak{B}}) \mid \mathfrak{B}(l) \neq \mathfrak{A}(l)\} \cup \{(CT, \mathfrak{A}(CT) + \delta_{\mathfrak{A}\mathfrak{B}})\}$,

where $\delta_{\mathfrak{AB}} = [\![\delta_{min}]\!]^{\mathfrak{A}}$ is the duration value of the move from $\mathfrak{A}$ to $\mathfrak{B}$, therefore the *minimum delay* of $\mathfrak{A}$.

The original ASM lemma still holds,

$$\mathfrak{A} + (\mathfrak{B} - \mathfrak{A}) = \mathfrak{B}. \tag{3.22}$$

Through the difference operator, $-_{state}$, applied to the two states, we obtain an update set containing all the redefinitions of locations that will occur in the new state, in other words all the locations that are scheduled for update with the minimum delay δ_{min} and therefore also the redefinition of the curent time after the minimum delay increment.
Let $\mathfrak{A}$, $\mathfrak{B}$ and $\mathfrak{C}$ be states such that $\mathfrak{A} + U = \mathfrak{B}$ and $\mathfrak{B} + V = \mathfrak{C}$. We define two subsets of the update set such that $U = U_l + U_{CT}$, which separates the timed update of locations from the update of the current time location.
The composition of update sets, which corresponds to the sequential application of the updates, is defined in the following way for basic ASMs:

$$U \oplus V = V \cup \{(l, v) \mid \textit{there is no } w \textit{ with } (l, w) \in V\}. \tag{3.23}$$

In other words, the composition of the update sets is the set of all the updates in V, some that override locations in U, others that are generated by U and all the updates unique to U and V.

The composition of delayed update sets U and V is the set $U \oplus V$ and follows a similar principle, containing:

- the updates that should have fired in U and are not overridden in V at the next state, therefore after a $[\![\delta_{min}]\!]^{\mathfrak{A}+V}$ delay;
- all the updates that are unique to V, with the delay incremented by $[\![\delta_{min}]\!]^{\mathfrak{A}+U}$.

Definition 25 (Composition of timed update sets).
$U \oplus V = \{(l, v_1, \delta_1) \in U \mid (l, v_2, \delta_2) \notin V\} \cup \{(l, v, \delta + [\![\delta_{min}]\!]^U) \mid (l, v, \delta) \in V\}$.
We define by $[\![\delta_{min}]\!]^U$ the value of the minimum delay when applying the update set U which is different from the minimum delay in the update set U because we must also

consider the inherited update set $\mathcal{U}$. Therefore $[\![\delta_{min}]\!]^V$ will be the minimum delay from the new inherited set (that has elements from the update set U also) and the update set V.

All updates from V were incremented with $[\![\delta_{min}]\!]^U$ in order to maintain the consistency of the count time function CT, that has to be equal after the firing of the two rules to the sum of their minimum delays.

Lemma 1 *Let U, V and W be update sets.*

1. $(U \oplus V) \oplus W = U \oplus (V \oplus W)$ *The equality is obvious because of the associativity of the union operation on sets and the associativity of addition on the time domain.*
2. *If U and V are consistent, then $U \oplus V$ is consistent. The consistency of the composition is ensured by definition and it verifies the definition 21.*
3. *If U and V are consistent, then $\mathfrak{U} + (U \oplus V) = (\mathfrak{U} + U) + V$.*

A move of an ASM consists of firing the updates produced by the main rule of the machine and by the update set inherited from the previous state, if they do not clash.

3.6.1.6 Transition rules and runs of the HiTAsm

States in computer science are dynamic, evolving through updates during the computation. The updates performed on an ASM state will change the interpretation of some functions from the underlying signature. A set of rules describes how these updates are performed (under what assumptions, in what order, how many times).
In the definition of our extension we kept most of the ASM rules, adding the timed update rule and, as presented in the following sections, a hierarchical composition together with their operations.

Timed update rule

$$f(s_1, \ldots, s_n) := t, \delta \tag{3.24}$$

Syntactic condition f is an n-ary dynamic function name of Σ, t can be any term and δ a positive integer (and as it will be introduced later, an interval of positive integers).

Meaning Change the value of f at arguments $(s_1, \ldots, s_n)$ to the term t after the delay δ, s_i, t, δ being interpreted in the current state.

Semantics The semantics of the timed update rule are given in the following semantic equation, where $\mathfrak{A}, \zeta, \mathcal{U}, l, v$ and δ are introduced in definition 26:

$$\frac{}{yields(f(s_1, \ldots, s_n) := t.\delta, \mathfrak{A}, \zeta, \mathcal{U}, \{(l, v, d)\})} \quad , \quad \begin{array}{l} where\ l = (f, (\llbracket s_1 \rrbracket^{\mathfrak{A}}_{\zeta}, \ldots, \llbracket s_n \rrbracket^{\mathfrak{A}}_{\zeta})), \\ v = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta}\ and\ d = \llbracket \delta \rrbracket^{\mathfrak{A}}_{\zeta} \end{array} \tag{3.25}$$

In equation 3.26 we give the execution semantics of a rule featuring delayed updates, where $\sum_{\oplus}$ is the composition-sum of update sets. It shows the behavior of timed updates, meaning that update sets $\mathcal{U}_{i-1}$ with smaller delays δ_{i-1} will be applied before update sets with larger delays.

$$\frac{yields(R_1, \mathfrak{A}, \zeta, \mathcal{U}_1) \ldots\ yields(R_n, \mathfrak{A} \oplus \sum_{\oplus}^{n-1} \mathcal{U}_i, \zeta, \mathcal{U}_n)}{yields(R_\delta, \mathfrak{A}, \zeta, \bigcup_i \mathcal{U}_i)} \quad , \quad \begin{array}{l} \forall u = (l, v, \delta) \mid u_i \in R_i, \\ u_{i-1} \in R_{i-1} \implies \delta_i^{\mathfrak{A}(i)} > \delta_{i-1}^{\mathfrak{A}(i-1)} \end{array} \tag{3.26}$$

Definition 26 (TAsm). A timed abstract state machine $TM = \{\mathfrak{A} = (\Sigma, X, \zeta), \mathcal{R}, \mathcal{U}\}$ consists of a signature Σ, a set of initial states for Σ, a set of rule declarations, including the main rule name of the machine and a possibly empty set of scheduled updates $\mathcal{U}$, accumulated between the runs.

Definition 27 (Move of an TAsm). We say that a TAsm TM makes a move from a state $\mathfrak{A}$ to another state $\mathfrak{B}$ (written $\mathfrak{A} \overset{M}{\Longrightarrow} \mathfrak{B}$) when the main rule yields a consistent update set $U = U_i + U_s$ in state $\mathfrak{A}$, $\mathfrak{B} = \mathfrak{A} + U_i$, $U_s \subseteq \mathcal{U}_{\mathfrak{B}}$ and $\mathfrak{B}(CT) = \mathfrak{A}(CT) + \llbracket \delta_{min} \rrbracket^{\mathfrak{A}}$.

Interpretation of delayed update sets A visual interpretation of the delayed update semantics is given in Figures 3.4 - 3.7. In the left side, in black we have function names whose interpretation will change to the values of the y_i terms after the δ_i delay. The red

Table 3.1: Inductive deduction of the semantics of HiTAsm rules

$$\frac{}{yields\,(\mathbf{skip}, \mathfrak{A}, \zeta, \phi)}$$ no time progression

$$\frac{yields\,(\mathbf{skip}, \mathfrak{A} + \{(l, v, \delta)\}, \zeta, \phi)}{yields\,(\mathbf{skip}, \mathfrak{A}, \zeta, \{(l, v, \delta)\})}$$

$$\frac{yields\left(\mathbf{skip}, \mathfrak{A} + \bigcup_{i=1}^{|\mathcal{U}|} \{(l_i, v_i, \delta_{max})\}, \zeta, \mathcal{U} \setminus \bigcup_{i=1}^{|\mathcal{U}|} \{(l_i, v_i, \delta_{max})\}\right)}{yields\,(f(s_1, \ldots, s_n) := t.\delta, \mathfrak{A}, \zeta, \mathcal{U} \oplus \{(l, v, d)\})}$$ $l = (f, (\llbracket s_1 \rrbracket^{\mathfrak{A}}_{\zeta}, \ldots, \llbracket s_n \rrbracket^{\mathfrak{A}}_{\zeta}))$, $v = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta}$ *and* $d = \llbracket \delta \rrbracket^{\mathfrak{A}}_{\zeta}$

$$\frac{yields\,(P, \mathfrak{A}, \zeta, \mathcal{U})}{yields\,(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, \zeta, \mathcal{U})}$$ $\llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta[x \mapsto a]} = true.$

$$\frac{yields\,(Q, \mathfrak{A}, \zeta, \mathcal{V})}{yields\,(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, \zeta, \mathcal{V})}$$ $\llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta[x \mapsto a]} = false.$

$$\frac{yields\,(P, \mathfrak{A}, \zeta, \mathcal{U})\ yields\,(Q, \mathfrak{A}, \zeta, \mathcal{V})}{yields\,(P\ \mathbf{par}\ Q, \mathfrak{A}, \zeta, \mathcal{U} \bigcup \mathcal{V})}$$

$$\frac{yields\,(P, \mathfrak{A}, \zeta, \mathcal{U})\ yields\left(Q, \mathfrak{A} + \llbracket P \rrbracket^{\mathfrak{A}}_{\rho}, \zeta, \mathcal{V}\right)}{yields\left(P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta, \mathcal{U} \bigcup_{\rho} \mathcal{V}\right)}$$ $\mathcal{U}^{\mathfrak{A} + \llbracket P \rrbracket^{\mathfrak{A}}_{\rho}} = \phi$

$$\frac{yields\,(P, \mathfrak{A}, \zeta, \mathcal{U})\ yields\,(Q, \mathfrak{A}, \zeta, \mathcal{V})}{yields\,(P\ \mathbf{abs}\ Q, \mathfrak{A}, \zeta, \mathcal{U} \bigcup \mathcal{V})}$$ $\mathcal{U} = U \oslash V$, $\mathcal{V} = U \oslash V$

$$\frac{yields\,(P, \mathfrak{A}, \zeta[x \mapsto a], U_a)}{yields\,(\mathbf{forall}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ P, \mathfrak{A}, \zeta, \mathcal{U} = \bigcup_{a \in I} U_a \cup \mathcal{U} = \bigcup_{b \in J} U_b)}$$ *if* $\exists x \varphi : a \in X_{\mathcal{SE}}, \llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta[x \mapsto a]}$.

bounding boxes are a quantitative representation of the delay. We first identify the update sets with the minimal delay and then fire all updates from the set. The delays of unfired updates $\cup\delta_i.\delta_i \neq \delta_{min}$ is then diminished by the minimal delay δ_{min} as depicted in figure 3.6.

Once all the updates corresponding to the minimal duration have been fired, the process

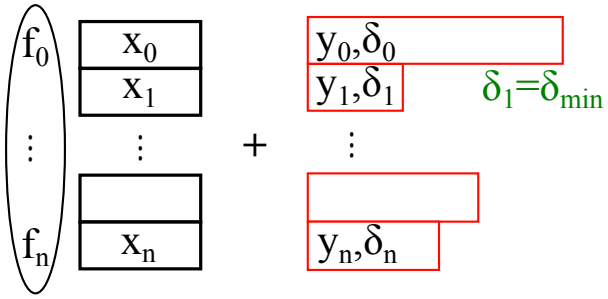


Figure 3.4: Determining the update with the minimal duration

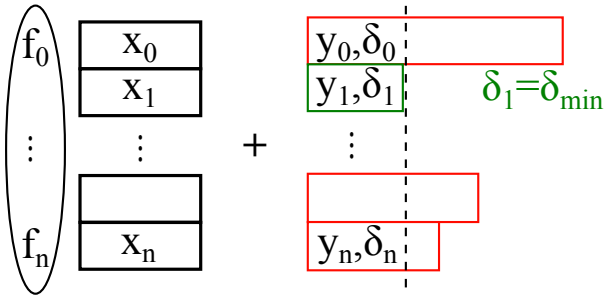


Figure 3.5: Selecting the update set corresponding to the minimal duration

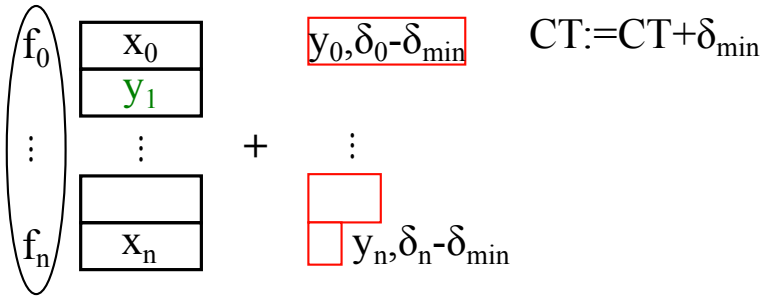


Figure 3.6: Applying the update set and updating the remaining durations

continues, figure 3.7.

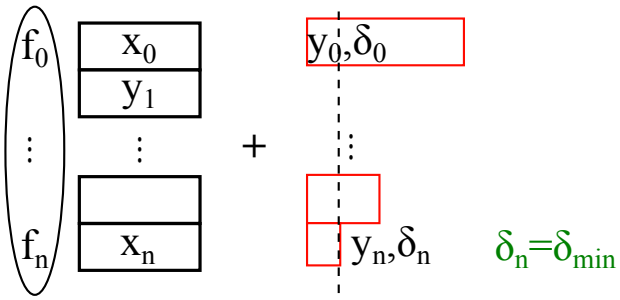


Figure 3.7: Selecting the new minimum delay from the remaining update sets

3.6.2 Equivalence with the basic ASM

Adding the delay δ to the updates, (l, v, δ) can be explained through classical locations. If we introduce a mapping function $\zeta_\delta : \Sigma_\delta \implies \Sigma \setminus \Sigma_\delta$ where $\delta : \mathcal{T}$ are location names from a subset of the ASM vocabulary Σ we can associate to any location from $\Sigma \setminus \Sigma_\delta$ a duration δ. In order to represent the advancement of the current time we can use a rule that will be fired at each step that computes the minimal delay δ_{min} and adds it to the controlled function CT.

3.6.3 Timed ASM defined by a set of Axioms

In this section we prove the respect of the time properties presented in [GP07]. Let $when : M \implies \mathcal{T}$ be a function that gives the time of a move for every move of the ASM. A move takes place at the moment when at least one guard of a rule is satisfied and the associated delay is equal to δ_{min}. Therefore the value of the function *when* is the current time after the move to the new state, which is the current time plus the minimum delay of the updates from the start state.

Property 1 *The move from a state to another takes place at a point in time defined by the function when:*

$$when : M \implies Time.$$

Let $\mathfrak{A}_0 \overset{m_1}{\Longrightarrow} \mathfrak{A}_1$ be a move from state $\mathfrak{A}_0$ to $\mathfrak{A}_1$, we have

$$when(m_1) = \mathfrak{A}_1(CT) = \mathfrak{A}_0(CT) + \mathfrak{A}_0(\delta_{min}).$$

Property 2 *Global Time: all time stamps of a run are totally ordered, i.e. the partial order of the moves is extended to a total preorder for their occurrence times,*

$$\forall m_1, m_2 \in M.when(m_1) < when(m_2) \vee when(m_2) < when(m_1)$$

Obviously true, from the definitions: if $\mathfrak{A}_0 \overset{m_1}{\Longrightarrow} \mathfrak{A}_1 \overset{m_2}{\Longrightarrow} \mathfrak{A}_2$ then $when(m_1) = \mathfrak{A}_1(CT)$ and $when(m_2) = \mathfrak{A}_1(CT) + \mathfrak{A}_0(\delta_{min})$, with $\mathfrak{A}_0(\delta_{min}) \geq 1$ since all time progress is positive and non-null, therefore $when(m_1) < when(m_2)$. Same reasoning if the moves occur in the opposite order.

Property 3 *Strict time progress along causal chains: whenever two moves are causally ordered, their occurrence times are strictly ordered, i.e.*

$$\forall m_1, m_2 \in M.m_1 < m_2 \Longrightarrow when(m_1) < when(m_2).$$

According to definition 24, a move takes at least one unit of time, therefore the property above is trivial.

Property 4 *Timed order is not stronger than causal order: causally non related events are not comparable in the timed order, i.e.*

$$\forall m_1, m_2 \in M.m_1 \leq m_2 \Longleftrightarrow when(m_1) \leq when(m_2).$$

For the time being, we forbid conditions on the delays, therefore the time order is induced only by the casual order.

Property 5 *Absence of Zeno computations: In any infinite run, there is no upper bound of the time values attached with moves, i.e.*

$$\forall R\, isInfinite(R) \Longrightarrow \forall t \in Time\, \exists m \in R.when(m) > t.$$

The Property 5 is satisfied because if the number of moves is infinite we will always have a new move with the current time, $when(m)$ superior at least with one time unit than the previous one.

Property 6 *Minimal time distance: There is a lower bound of the time differences between non-simultaneous causally ordered moves, i.e.*

$$\exists \delta \in Duration, \forall R, \forall m_1, m_2 \in R. m_1 < m_2 \Longrightarrow when(m_2) - when(m_1) > \delta.$$

The minimal duration allowed in our update semantics is one, therefore such a δ exists.

Property 7 *Events at discrete steps: Any two moves occur either at the same instant or the time differences between their occurrence times are a multiple of a given value,*

$\exists \delta \in Duration \forall R \forall m_1, m_2 \in R \exists k \in N. when(m_1) - when(m_2) = k * \delta.$

According to our definition, the value of the minimal delay is 1. The value of the other delays δ_{min} is a multiple of the minimal delay hence $k = 1$.

Property 8 *Local urgency: The time of each state change of each run is minimal.*

Property 9 *Global Urgency: The earliest state change amongst all distributed agents is taken.*

Updates sets are composed of updates and different associated delays. According to definition 24, the time progress is defined as the minimal delay in the updates list.

Time The time is an integrated part in the ASM. We associate time to an update set in the following way. If no time information is present, the time associated to the update set is equal to one as in one cycle. We can hide away details of the implementation of a rule by associating to an update set a time superior to one. We use the term *timed* rule or update to distinguish the last one from the regular one-cycle case. The interpretation is that we see the more complex rule that would have executed in several cycles as a *turbo* rule where the only thing that matters is the result, or the final update set. Several conditions must be satisfied in order to be able to use a timed rule. From the refinement point of view, the one-cycle update corresponds to the most refined version of the ASM which is sufficient in our case to model the processor behavior. We will now take a look at the timing properties of our state transition system.

Delay as interval Our notion of duration interval is different from the one presented in [AMMS10] where the delay is chosen non-deterministically from the specified interval. We use the notion of interval in order to simulate traces of abstract runs. When such a delay interval is associated to an update, the pipelined model of the processor will generate different execution time results for different delays in the interval.
A known problem for the computability of the WCET estimation, on certain architectures, is that the analysis is not compositional, because the local worst case might not correspond to the global worst case. This is encountered in processors featuring timing anomalies, that can generate unobvious results such as a cache miss being more optimistic than a cache hit with regards to the execution time.

Interval Value Domain for the Delayed Updates In order to represent abstract states of the hardware in a timing model, the timing information is also extended. For concrete definitions of the hardware, a single delay information is enough to represent the time needed for the transition to be made. We introduce a natural extension of the natural value domain of time to the interval domain where the delay of an abstract component can take any values inside an interval.

Definition 28 (Interval domain). Let us define the domain of non-empty natural intervals $\mathbb{I}$, on the totally ordered set $(\mathbb{N}, \leq)$, with the property that any natural number that lies between two numbers in the set is also included in the set where

$$\mathbb{I} = \{[a, b] | a \in \mathbb{N}, b \in \mathbb{N} \cup \{+\infty\}, a < b\}. \tag{3.27}$$

We can also define the interval of naturals with regard to the natural set as all the natural elements between a lower and a upper bound:

$$\mathbb{I} = \{x \in \mathbb{N} | a \leq x \leq b\}. \tag{3.28}$$

The interval set $\mathbb{I}$ is totally ordered by $\sqsubseteq^{\mathbb{I}}$ defined as:

$$[a_1, b_1] \sqsubseteq^{\mathbb{I}} [a_2, b_2] \Leftrightarrow a_2 \leq a_1 \wedge b_1 \leq b_2. \tag{3.29}$$

The least upper bound and the greatest lowest bound are defined as follows:

$$[a_1, b_1] \sqcup^{\mathbb{I}} [a_2, b_2] = [min(a_1, a_2), max(b_1, b_2)], \tag{3.30}$$

$$[a_1, b_1] \sqcap^{\mathbb{I}} [a_2, b_2] = [max(a_1, a_2), min(b_1, b_2)], \ et\ max(a_1, a_2) \leq min(b_1, b_2) \tag{3.31}$$

Being able to use imprecise intervals for a certain action comes in hand when using the model in pair with a value analyzer that gives imprecise information. In this case a parallel execution of all the different scenarios must be made.
Whenever parallel executions, for different data flows, are needed, we proceed until a *merging* point is reached, where it would be safe to return to a single execution flow that will generate the highest global execution time.
We introduce the *stores* and *parallel stores* in order to handle the multiple generated runs, each time such an interval is encountered. The problem with this type of execution is the inherent state-space explosion. We therefore introduce the notion of *reference store* and at every execution point that deals with a duration interval we keep a history of all the updates that must be applied in order to get into that respective state of that particular run. This is equivalent to keeping a list of all the locations that have been modified since the splitting point if we merge all the updates. The merging consists of keeping only the final value for the same location. To further compact the update sets, we can compare the locations to the ones from the original store and eliminate the redundant information.
After several parallel executions the complementary update set will become:

- *empty*, meaning that except for the count time, we arrive in the same state as the one where the execution has split. In this case we can fusion it with the original run, keeping the last count time.

- *identical* to the one of another run, we can thus fusion and eliminate it, preserving the maximal count time.

- *similar* to one of another run. The notion of similarity is to be further discussed. The intuition is that we can reach two states that have differences only in non-interfering locations, so the two can be merged.

States of ASMs are static algebras, therefore we can use this fact to create equivalence classes between different states of the parallel runs in order to reduce the state space explosion when searching for similar states.

3.7 Hierarchical TASM foundation

In this section we formally introduce the concept of HiTAsm abstraction hierarchy. We start by a comparaison with the ASM step-wise refinements system engineering technique that implies the equivalence through homomorphism of the different ASMs. After giving the intuition and justification for the introduction of this new feature into the ASM framework we proceed to the rigorous definition of the hierarchical levels, and the Oracle that chooses between them, in the same formalism used to define the temporal feature. The last sections are dedicated to the use of HiTAsm, introduces an order on states and provides a formal definition for the state merging.

3.7.1 Preambule

The hierarchical notion of ASMs in already present in the ASM literature as a basis for the incremental design by refinements. Besides the compactness and good readability of specifications, ASMs offer a homogenous formalism for all abstraction levels. We believe that the concept can be further exploited as the level of abstraction of ASM is fixed at the beginning of the modelling phase. We therefore introduce the dynamic choice of the refinement granularity in what we call Hierarchical Timed ASM (HiTAsm). Therefore, the model itself, is able to chose on the fly the appropriate abstraction level for a given rule among several, user-defined, definitions. This can be particularly useful in the case of a processor design as the precision on the values of manipulated data, given by a value analyzer for example, is not always on pair with the level of detail of the processor model. The main idea is to adapt the level of abstraction of the processor description to the precision given by the value analysis in order to master or reduce the state-space explosion inherent to the WCET analysis.

3.7.2 Hierarchical ASMs

ASMs have the nice property of being able to describe any algorithm at its right abstraction level. One major difference in the concept of ASMs with regards to other state transition systems is that the values of the support set remain the same while the *transfer* functions change after an update.

The relation between functions and data is reversed, instead of having mutable data structures with immutable functions, we have immutable data which is operated on by mutable and immutable functions. A data selector can be seen as a single argument function over the selected data domain and the other way around. The assignment of an expression to a variable field f of a record x can be re-interpreted as an update of the mutable function f at the position x.

In other words instead of changing values, the interpretation of functions, for the right arguments gets changed, associating the function name at the respective arguments to a new value from the superuniverse. We have thus only immutable data serving as index structures for possibly mutable functions. The abstract state machines approach leads naturally to components which are adaptable.

As an extension to this logic, we introduce HiTAsms not only as transition systems that change the interpretation of functions according to the rules but also the interpretation of the interpretation of functions. This is achieved through the extension of the model, with an *oracle* that will modify the interpretation of function names, making them depend on different set of rules.

The decisions of the Oracle can be modelled either based on the internal state of the HiTAsm (watching certain locations can trigger a decision to use a more abstract state when suited - for example when no timing anomalies can occur) or on external, monitored locations (for example when we have insufficient precision on certain values we can switch to a definition of an unit that can work with the larger - more imprecise - domain).

Correctness of the hierarchical ASM A HiTAsm can have multiple definitions for a same (rule, module) name that corresponds to different levels of refinement that we call abstractions. During a same run either of the available hierarchy levels can be selected, there-

fore we must ensure that the run is correct with regard to the semantics of the processor and also with regards to the time estimation. Informally, the correctness of the model is granted by ensuring all the hierarchical level of the sub-ASM have the same input and output *interfaces*, that their initial and final states are equivalent and the intermediate runs produce also equivalent states, in the same order. The creation of the different hierarchy levels along with the functions that determine their choice is an important part in our WCET-centric HiTAsm model. Inspired by the stepwise refinement technique of the ASM, we introduce an additional workflow for the refinement generation. The ASM method starts with the a fairly abstract definition of the system, called the ground model, and advance towards a more concrete one up to the implementation of the *executable* model. This is a natural incremental design that is well suited for system engineering. Following this method might be sufficient to obtain the wanted levels of abstractions, whose properties we will describe in the following. Nevertheless fine-tuning them in order to achieve the appropriate level of abstraction might be necessary, hence the bottom-up approach, not to mention that the processor might be easier in some cases to describe directly at a level that approaches the executable one.

The bottom-up refinement approach

1. The starting point of the abstraction refinement is the final, concrete model of the processor, unit by unit.

2. Each functional unit will be refined to an equivalent more abstract one. The latter must do the same changes (eventually in a different number of steps) to the *target* locations (important algorithm-wise, in other words locations that are used by other sub-ASMs) until the final (equivalent) state is reached. The model itself doesn't need the guarantee to be finite under all inputs, running a finite program guarantees the finiteness of the run.

3. In the final step we must ensure that the eventual interference of the *additional* states will not affect the semantics of the processor execution and that the generated time is an over-approximation of the more concrete one.

The bottom-up approach is best adapted when a HDL version of the processor is already available, in which case an automated translation can be made. When starting with a top-down design, the model ends with the concrete version and the design can go back at any moment in order to give other abstract definition, most adapted to the context.

The top-down refinement approach

3.7.3 Cycle-accurate vs time-accurate model

Definition 29 (Cycle-accurate model) is an accurate description of the state of the model on each clock cycle. The cycle-accurate model is the pair $(S, \Longrightarrow)$ that associates a set of states S to another set of states S' at every rising clock edge, by applying the binary relation over S (of transitions) $\Longrightarrow \subseteq S \times S$.

Definition 30 Time-accurate model is a tuple $(S, \mathcal{T}, \Longrightarrow)$ that associates a set of states S to another set of states S' and the necessary time $t \in \mathcal{T}$, to arrive in the new state, by applying the binary relation over S (of transitions) $\Longrightarrow \subseteq S \times S$.

If $p, q \in S$ and $t \in \mathcal{T}$, then $(p, t, q) \in \Longrightarrow$ is written as

$$p \overset{t}{\rightarrow} q$$

Statement 2 *Let* $\mathfrak{A}$ *be a state and R a rule in state* $\mathfrak{A}$. *The normal semantics of HiTAsm will generate the following transition:*

$$\mathfrak{A} \xrightarrow[\bigcup_i u_i | delta(u_i)=\delta min]{\delta_{min}} \mathfrak{A} + u_i$$

where u_i *is the update set generated by the interpretation of the rule R in state* $\mathfrak{A}$.

The aim of the ASM extensions is to facilitate the timing analysis. A major problem in the WCET estimation is the inherent state-space explosion. In order to compute the WCET we would need to generate all reachable states and identify the branch corresponding to the worst case. This remains computationally impossible therefore abstractions are made that lead to approximations of the WCET. The analyzer that uses the processor model

makes constant attempts to reduce the state space exploration with minimum impacts on the precision. For example the history of states can be used in order to identify identical program points that will enable fusions as detailed in [PB12]. Therefore a cycle-accurate model will only clutter the analysis with otherwise useless information, as there is no need to keep track of consecutively identical states.

Using a *time-accurate* model eliminates unnecessary states with regard to the timing of the system. In other words if we have the knowledge about the time needed to get into a new state we can directly move the execution into that state. We use the definition of ASM-moves in order to control the run. We also need to show that the pseudo-steps made during that time will take into account the interactions of all the units. The time-accurate ASM can be seen as a simple case of "hierarchy" as the refinement is correct by definition because in the $(m, 1)$ refinement all the intermediate states are identical (the reason to use the time-accurate model in the first place). The top edge of the commutative diagram in

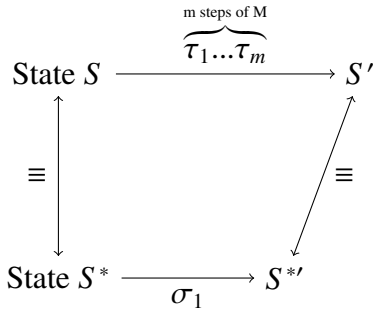


Figure 3.8: Time-accurate ASM as a refinement scheme

figure 3.8 represents the cycle-accurate model and the bottom part the time-accurate model. The equivalence notion $\equiv$ between locations of interest in corresponding states is trivial to prove and follows from the definition of the time-accurate model. Except for the time function, the pair of start and end states are identical so we have $S = S^*$ and $S' = S^{*\prime}$.

3.7.4 Extension of the ASM postulate

Definition 31 (Dynamic state changes) Static algebras as states and guarded destructive assignments for abstract functions as basic dynamic operations.

From the ground model to the code an *uniform algorithmic view* is maintained based upon an *abstract notion of run*.
Precise WCET estimation is dependent on the precision of the hardware model. However generating all the exact states of the processor at every step of the run triggers a state-space explosion. Therefore we should introduce a notion of abstract state runs that approximate an upper-bound on the execution time, while reducing the state-space explosion.
Our idea consists in handling a part of the abstraction notion directly in the model. The analysis that uses the processor model can thus use directly more abstract notions of state whenever needed.
The original sequential ASM thesis states that *every sequential algorithm can be step-for-step simulated by an appropriate sequential ASM*. It also implies that the level of abstraction for a given model should be fixed from the start. We modify the statement by confining the applications domain to certain systems and allowing the dynamic change of the abstraction level.

Definition 32 (HiTAsm postulate) A processor can be simulated by an appropriate HiTAsm at various abstraction levels generating *conservative* timed runs.

We introduce the notion of *conservativeness* with regard to time. By replacing an ASM with a more abstract one, at every step of the execution, the time taken by the more abstract ASM will be superior or equal to the less abstract one.
The postulate that we introduce is more conservative than the original one. We do not state that it applies to a processor for *every* abstraction level in all contexts. The correctness of the run of our model is based on some constraints applied to the choice and construction of the different ASM component definitions similarly to the notion of *consistent* updates of ASMs. The advantage of using such a model for the WCET estimation is two-fold. On the first hand we need a model that helps confining the state-explosion and on the other hand we need to prove that the processor model corresponds to the real processor.
By using the step-wise refinement technique inherited from the original ASM model, we can start the description of the processor from a very abstract form that can be directly inferred from the user manual of that processor and proof-read by experts. This represents the ground model that will de refined into the final HiTAsm as an executable model. This

means that we can prove that the executable model of the processor, used in the WCET estimation, corresponds to the initial, correct description.

3.7.5 Mathematical foundation of HiTAsm

According to the abstract-state postulate an algorithm does not distinguish among isomorphic types. A state is just a certain implementation of its isomorphism type. Looking at the way elements of the base set are accessed, allows us to identify a good manner to introduce multiple *views* of the same *operation*. Base set elements are accessed through ground terms that contain functions from the vocabulary of the state. By allowing multiple definitions for a function we access different ground terms leading to an equivalent operation with regarding to the semantics of the processor and its temporality for example.
The classic ASM refinement techniques provides us with the ability to build an abstract model with an equivalence notion between data in locations of interest in corresponding states. We want to be able to use multiple refinements levels during the same execution of the algorithm that ensure a correct result with regard to the target property.
Contrary to the refinement concept we want to make moves in both senses, from the refined version to the more abstract and vice-versa. This would not be possible in the general case, nevertheless thanks to the target property (the temporal over-approximation) we can make jumps in both ways. We hereby extend definition 18 of the *state* to take into account the

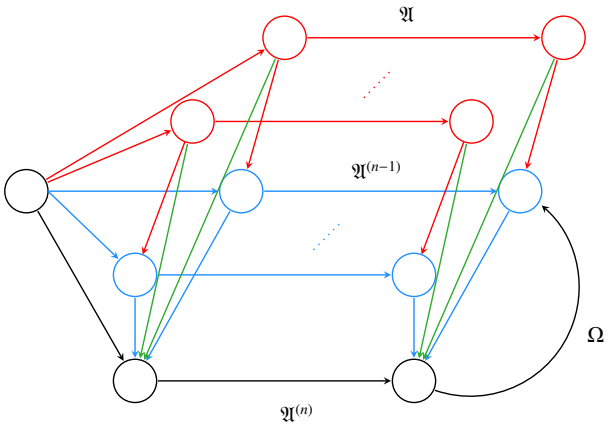


Figure 3.9: Dynamic HiTAsm abstraction level switch

different levels of abstraction.

Definition 33 (HiTAsm State). A state $\mathfrak{A}$ for the signature Σ is a non-empty domain $X = X_{Set} \cup X_{\alpha}$, the superuniverse of $\mathfrak{A}$, together with interpretations of the function names of Σ in one of the domains of $\mathfrak{A}$.

For each rule name we can give a definition for each universe of X. The values of terms, functions and their arguments, can be *imprecise*.

Definition 34 (Critical rule). A rule is called critical if it changes the interpretation of terms used in the guard of another rule, hence a rule dependency exists.

Definition 35 (Critical location). A location is called critical if it is involved in a critical rule.
Let r_i be a non trivial update rule (l_i, v) at location $l_i = (f, (a_1, \ldots, a_n))$ and let R be a guarded update if φ then r where φ is a function formula depending on a location l. If $l = l_i$ then l is a critical location.

$$cloc(r, \mathfrak{A}) = \left\{ l_i \in U : \exists l_j \in \varphi . l_i = l_j \right\} \tag{3.32}$$

To describe the behaviour of a precision guided HiTAsm abstraction level choice, we now introduce the ***abs***-construct which combines simultaneous atomic updates of basic TASMs in a global state with a choice of rules to apply.
We denote the abstraction level choice of two HiTAsm rules P, Q by P ***abs*** Q and define its semantics as the effect either executing P in the given state $\mathfrak{A}$ or Q in the same state $\mathfrak{A}$, depending on which domain critical locations belong to, where U is the set $[\![P]\!]^{\mathfrak{A}}$ of updates produced by P in $\mathfrak{A}$.

Definition 36 (Rule abstraction) Let P and Q be HiTAsm rules.

$$[\![P \textbf{\textit{abs}} Q]\!]^{\mathfrak{A}} = [\![P]\!]^{\mathfrak{A}} \oslash [\![Q]\!]^{\mathfrak{A}} \tag{3.33}$$

$$U \oslash V = \begin{cases} \{(l,t) \mid (l,t) \in U\} & , \ \textit{if } g \in guard(P, \mathfrak{A}) \implies Dom(t) \in Dom(g); \\ V & , \ \textit{otherwise}. \end{cases} \tag{3.34}$$

where $guard(P, \mathfrak{A})$ denotes the function that returns the last values of the locations used in the guards of P and $Dom(t)$ the functions that returns the type of domain of the super universe of t.

Abstract rule choice

$$P \textbf{ abs } Q \tag{3.35}$$

Syntactic condition Both P and Q must be defined as implementations of hmodules.

Meaning Depending on the execution context, the oracle decides which HiTAsm definition to choose for a certain rule.

Semantics The semantics of the abstract rule choice are given in the following semantic equation, where $\mathfrak{A}, \zeta, \mathcal{U}, l, v$ and δ are introduced in definition 26 and the $\oslash$-operator is formalized in definition 36.

$$\frac{yields(P, \mathfrak{A}, \zeta, \mathcal{U})}{yields(P \textbf{ abs } Q, \mathfrak{A}, \zeta, \mathcal{U})} \quad , \quad where\ \mathcal{U} = U \oslash V \tag{3.36}$$

$$\frac{yields(P, \mathfrak{A}, \zeta, \mathcal{V})}{yields(P \textbf{ abs } Q, \mathfrak{A}, \zeta, \mathcal{V})} \quad , \quad where\ \mathcal{V} = U \oslash V \tag{3.37}$$

3.7.6 Correctness proof outline

Safety-critical systems require a certain level of confidence regarding the respect of some constraints. In order to ensure a level of confidence to the system, tools are used to verify the respect of functional and non-functional properties. One of the advantages of using a formal model for the processor description is the foundation to build proofs of correctness. We provide in the following the intuition on how this can be achieved.
We distinguish two cases that both benefit from the formal support of the model. Firstly, we need to prove that the processor model is correct with regards to the real processor that will be used in the actual system and secondly that all the abstraction levels of the processor are correct refinements of HiTAsms. ASMs provide a stepwise refinement method that allows

the designer of the system to start with a high-level description of the system that is refined step by step into the final version which can be proven correct with regard to the initial one. If a processor SystemC description is available, we can automatically generate a correct HiTAsm model using techniques described in [MRR04] with minor modifications because all the additions to our language preserve the nice properties of ASMs.

In order to prove that the different abstraction levels of the HiTAsm are correct we can use similar techniques used to prove the correctness of compilers, [GZ00]. Program transformation used in compilers consists in transforming the control and data-flow graphs. In other words, the observable behaviour of the program must be preserved. In our case, we use as an input for the processor model the compiled code of the program. The transformation consists in two steps: data mapping and operation mapping. When compiling a source code, the initial graph depends on values that are known only at compilation time, and the stack and heap are being mapped into the internal register system of the target processor. Similarly we have a high level vision of the architecture in the form of the binary (after the value analysis step which provides information about loop counter, variable interval values, addresses, etc.) based on instructions from the ISA of the processor and a low level vision based on microcode operations that describe the exact behaviour at byte level of a particular ISA implementation (the actual processor). In figure 3.10, the top graph represents the most abstract model (the binary code) and the bottom one the most concrete model (the processor). Proving the correctness of the two models comes to independently prove the correctness of the data mapping and conditional graph rewrite rules. The data mapping assigns a new semantics (by means of an HiTAsm $\mathfrak{A}'_{\mu P}$) to the binary code using the concepts of the data part of the target language. The behavioural part is kept, therefore the corectenns of the mapping can be shown by proving that $\mathfrak{A}'_{\mu P}$ 1-1-refines $\mathfrak{A}_{\mu P}$.

ASMs are transition systems which transfer static algebras. The abstract state machines make use of the following abstraction principle, while limited to the notion of evolving algebras:

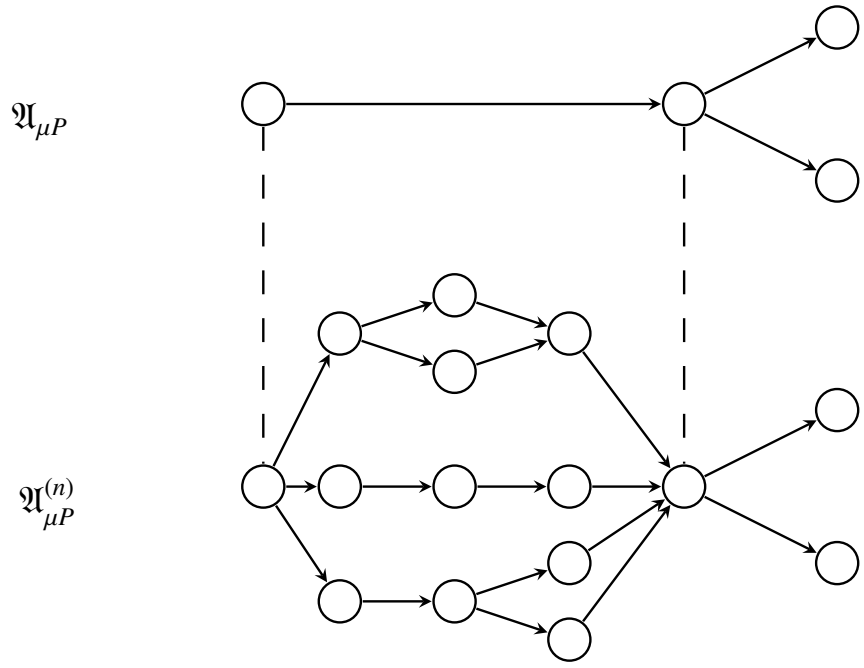


Figure 3.10: HiTAsm refinement

3.7.7 Abstract processor execution

Analyzing all reachable states of a processor makes the WCET estimation safe. Nevertheless, because of the state space explosion we must eliminate as much individual state handling as possible.

The HiTAsm model is custom tailored to confine the state space explosion of the undergoing analysis. After a number of safe, abstract steps the analysis goes back to a concrete state that corresponds to the global worst case. If the information regarding that state is lost or it is decided to be computationally expensive, the state is safely over-approximated by choosing a more pessimistic one.

We provide in the following a schematic view of the use of the model in the WCET estimation.

1. Complete the value analysis of the binary. We obtain information on the CFG, instructions, loop counters, register values, addresses, etc.

2. Start the conjoint symbolic execution (SE) on the HiTAsm model of the processor.

3. (a) If the value is exact $\Longrightarrow$ use the concrete HiTAsm $\mathfrak{U}^{(0)}_{\mu P}$.

   (b) If the value is imprecise (set, interval, etc.) $\Longrightarrow$ use abstract HiTAsm $\mathfrak{U}^{(i)}_{\mu P}$. We deal with symbolic values $\Longrightarrow$ all or some if the parameters of the functions (locations of the units functions) are intervals.

   What is the type of dependency?

   - unit type
   - functional (an instruction needs or depends on a certain value) $\Longrightarrow$ split the domain set (the universe of the HiTAsm) in different sub-domains that satisfy the constraint.

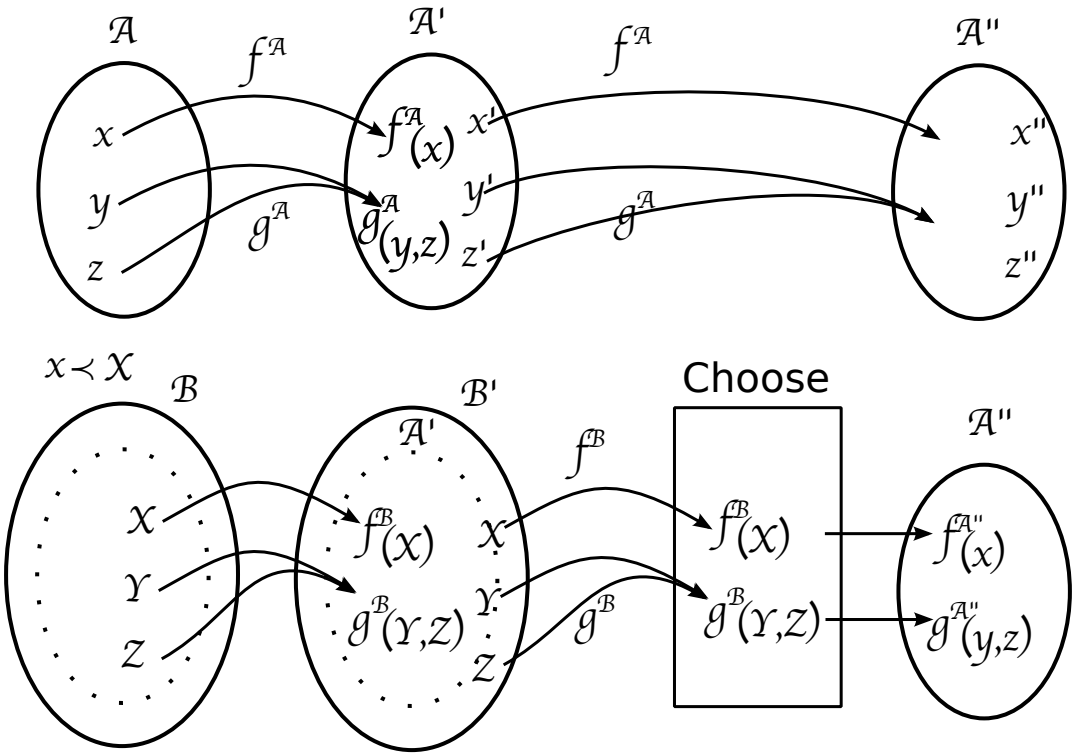


Figure 3.11: HiTAsm abstract execution

3.7.8 Dynamic choice of ASM refinements (the Oracle)

Possessing a precise and versatile model of the processor is very important. Nevertheless having access to an usable HDL code, is rarely the case for platforms used in hard

real-time systems, that are fairly outdated, and even if it exists, there is no common, unified description language. Ideally we should use the description of the processor as an input and generate an usable model for the analysis. As the lack of availability and standardization makes the task impossible, the need to create a model for each platform is mandatory. This is one of the bottlenecks in the adaptability of current tools, and we consider that the modeling part should be therefore a separated straightforward engineering task that can be made on the fly and without disposing of precise knowledge with regard to the rest of the tool. Therefore we chose to use the abstract state machine, a method that bridges the gap between human understanding and formulation of real-world problems and the deployment of their algorithmic solutions, in our case, the modeling of the processor, that showed its efficiency as a specification method in numerous practical applications (e.g. see [Mic], [BS03]).

Using a human readable and machine executable language makes the difference when it comes to speeding up the process of the hardware description. However some important features were not included in the original version of the ASMs [Gur95a] like the timing aspects hence updates are considered immediate. Ouimet et al. [OL07] introduced the concept of durative actions by adding delays directly in the syntax; our approach is similar. In [SV07] a prototype of a simulator for reactive timed ASMs that verifies the respect of requirements specifications is introduced. Besides the timing aspects we enrich the original model with hierarchical feature that enables us to give different definitions on several abstraction levels of the same processor component.

The goal of hierarchical ASMs is to provide at any time during the analysis, the right level of abstraction in order to prevent the combinatorial explosion. We know that we do not always dispose of precise information during the analysis (e.g. data memory address, availability in the cache, etc.) therefore using the most precise description of the fetching mechanism, for example, would be useless, on the other hand, a less precise, more abstract, definition could help reduce the number of generated states.

The hierarchical definition of components integrates seamlessly into the ASM formalism. Basically, the *oracle* is an ASM module that imports all the needed function definitions and exports the needed functions or rules. Each hierarchical module is defined as a control state ASM (cf. [BS03]) using in its condition the result from the *oracle* that decides which

implementation is appropriate for the current context.

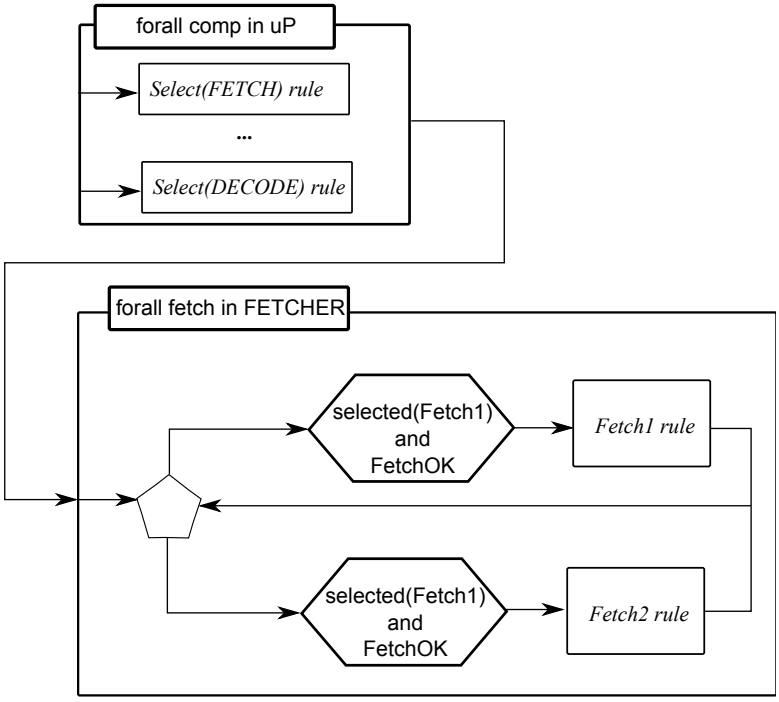


Figure 3.12: The oracle and the fetcher modules

FETCH =

forall fetch ∈ FETCHER **do** FETCH1(fetch), FETCH2(fetch)

In figure 3.13 we have two definition of the Fetch stage, the first one corresponding to the more abstract version that will typically be chosen if we have no precise information on the exact fetch address. Generally we have a family of abstraction for each component of the processor, $\alpha_{C_i} = \bigcup_{j=0}^{m} \alpha_j$ so that $C_i \overset{\alpha_j}{\Longrightarrow} C_i^{\alpha_j}$. Let $T(C_i^{\alpha_j})$ be the contribution of the abstract component to the global execution time. We must have $T(C_i^{\alpha_j}) \sqsupseteq T(C_i)$.

3.8 Conclusions

We have proposed an extension of the ASM model that handles time and dynamic abstraction in a simple manner. The possibility to make delayed transition is presented as a support for abstracting the processor components in order to achieve a more compact sim-

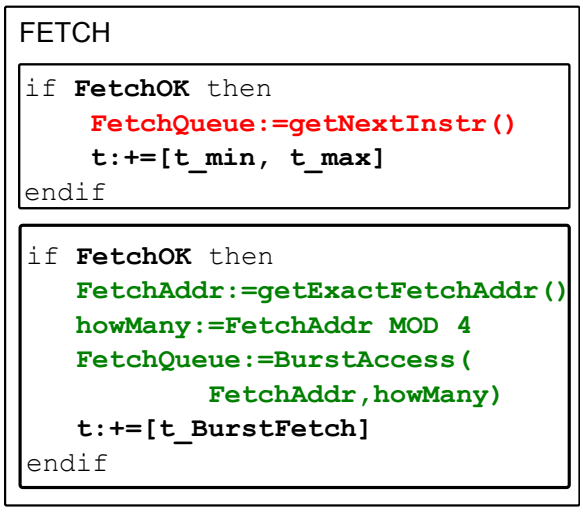


Figure 3.13: Different definitions of the fetcher

ulation.

Some temporal properties of the system were enumerated and discussed. We have also introduced a model that can dynamically refine the components of the processor, preparing a framework where run-time abstraction can be made in both ways between the concrete and the more abstract definition.

The adaptability of the analysis, given by the separation of the processor model and the analysis, the ease of use, the preservation of the formal background of the model extensions, the adaptability to imprecise value analysis, the state space explosion confinement techniques through hierarchical abstractions and fusions make our model suitable for the WCET estimation.

Chapter 4

HiTAsm at Work

In this section we present some additional application contexts of the HiTAsm that we further detail in chapter 7. We start by showing the handling of timing anomalies and continue with an overview of the HiTAsm integration into the timing analyzer.

4.1 On the hierarchical levels of abstraction

The execution semantics of delayed updates of abstract HiTAsms is based on timed moves corresponding to the updates that have the smallest associated delay, δ_{min}. This means that the full result of the execution of a rule might be visible in several steps.

Definition 37 (Finished Rule). Let U_R be the update set generated by the interpretation of the rule R, we say that the interpretation of R in state $\mathfrak{A}$ has *finished* if all the delayed updates of U_R were fired.

Definition 38 (Rule Finish Time). Let U_R be the update set generated by the interpretation of the rule R, we say that Δ_R is the *finish time* of R in state $\mathfrak{A}$ if:

$$\delta_{[\![R]\!]^{\mathfrak{A}}} \stackrel{\text{not}}{=} \sum_{i=1}^{|U_R|} \delta_i \mid (l_i, v_i, \delta_i) \in U_R.$$

Starting from definition 38 we can define the HiTAsm *rule semantics*.

Definition 39 (HiTAsm rule semantics). Let U_R be the update set generated by the interpretation of the rule R, we say that R is interpreted in state $\mathfrak{A}$ with the rule semantics and denote with $[\![R]\!]^{\mathfrak{A}}_{\rho}$, if:

$$\frac{yields\left(\mathbf{skip}, \mathfrak{A}_f, \zeta, \phi\right)}{yields\left(R, \mathfrak{A}, \zeta, U_R\right)}\rho$$

In other words, an *execution step* leads directly to a new state in which all the direct timed updates generated by that rule, namely U_R, are fired (all the delayed modifications stipulated in U_R are consumed).
The rule semantics from definition 39 can be easily extended to block-rule semantics.

Definition 40 (HiTAsm block-rule semantics). Let $\sum_{i=0}^{n} U_{R_i}$ be the composition of the update sets generated by the interpretation of the set of rules $R_0 \ldots R_n$, we say that the block of rules are interpreted in state $\mathfrak{A}$ with the rule semantics and denote with $[\![R]\!]^{\mathfrak{A}}_{\rho}$, if:

$$\frac{yields\left(R_0, \mathfrak{A}, \zeta, U_{R_0}\right) yields\left(R_1, \ldots, R_n\mathfrak{A}, \zeta, \sum_{i=1}^{n} U_{R_i}\right)}{yields\left(R_0, \ldots, R_n\mathfrak{A}, \zeta, \sum_{i=0}^{n} U_{R_i}\right)}\rho$$

Based on the above notations we can now give a new definition for the rule finish time:

$$\delta_{[\![R]\!]^{\mathfrak{A}}} = time(\mathfrak{A}_e) - time(\mathfrak{A}), \tag{4.1}$$

where $time(\mathfrak{A}) = [\![l_{CT}]\!]^{\mathfrak{A}}$.

Definition 41 (HiTAsm state transition system). Let $\mathfrak{A}$ be a set of HiTAsm states, $\Longrightarrow$ a set of state transitions labeled by the set of abstraction levels $\mathcal{L}$ and $\mathcal{T}_{\mathcal{L}}$ a time domain for the level of abstraction, we say that $(\mathfrak{A}, \mathcal{T}_{\mathcal{L}}, \mathcal{L}, \Longrightarrow)$ is a HiTAsm state transition system.
The fact that $(\mathfrak{A}_s, t, l, \mathfrak{A}_e) \in \Longrightarrow$ is written as:

$$\mathfrak{A}_s \xrightarrow[l]{t} \mathfrak{A}_e.$$

We can therefore also note, as a shortcut, the interpretation of rule R in the state $\mathfrak{A}$ in the rule semantics, and the interpretation of the block rule respectively:

$$\mathfrak{A} \xrightarrow[R]{\delta[\![R]\!]^{\mathfrak{A}}}_{\rho} \mathfrak{A}_e \tag{4.2}$$

$$\mathfrak{A} \xrightarrow[R_0,\ldots R_n]{\sum_{i=0}^{n} \delta[\![R_i]\!]^{\mathfrak{A}_i}}_{\rho} \mathfrak{A}_e \tag{4.3}$$

4.1.1 HiTAsm semantic level

In definition 39 we introduce a big step semantics based on the abstract HiTAsm model. Depending on the granularity of effects visible after a transition occurs, we can define several types of HiTAsm semantics like the cycle-level, δ-level (time accurate), rule-level, instruction level, block level, etc.

Cycle level Systems modeled in our framework can be as precise as needed, therefore we can model a processor cycle level, register transfer level (RTL) or even net-list level.

Definition 42 (Cycle Level). Let $\mathfrak{A}_s, \mathfrak{A}_e \in \mathfrak{A}$ be two HiTAsm states of $\mathfrak{A}_{cl}$ together with the superuniverse $X \supset \mathcal{T}_{cycle}$ and $\Longrightarrow_{cycle}$ the previously introduced transition rule, we have

$$\mathfrak{A}_s \xrightarrow[R]{\delta=1_{cycle}}_{cycle} \mathfrak{A}_e.$$

representing a move from the start state $\mathfrak{A}_s$ to the finish state $\mathfrak{A}_e$ after a delay of one cycle, 1_{cycle}.

Please note that depending on the vocabulary of the state $\mathfrak{A}_{cl}$ and the abstraction level of the rule implementation,

Theorem 1 *In a cycle-accurate semantics of the HiTAsm, two consecutive states are different iff that state generates or the delayed update set contains a delayed update with the delay on one cycle.*

$$\forall \mathfrak{A}_s, \mathfrak{A}_e \in \mathfrak{A}_{cl}. \mathfrak{A}_s \neq \mathfrak{A}_e \Leftrightarrow \exists (l_i, v_i, \delta_i) \in \mathcal{U}^{\mathfrak{A}_s} \mid \delta_i = 1_{cycle}$$

Delta level A delta-level model of the processor can capture the state modification that occurred in a lapse of time δ.

Definition 43 (Delta-accurate Level). Let $\mathfrak{A}_s, \mathfrak{A}_e \in \mathfrak{A}$ be two HiTAsm states of $\mathfrak{A}_\delta$ together with the superuniverse $X \supset \mathcal{T}_\delta$ and $\Longrightarrow_\delta$ the transition, we have

$$\mathfrak{A}_s \xrightarrow[R]{\delta}_{cycle} \mathfrak{A}_e.$$

representing a move from the start state $\mathfrak{A}_s$ to the finish state $\mathfrak{A}_e$ after a certain delay, δ.

Please note that depending on the vocabulary of the state $\mathfrak{A}_\delta$ and the abstraction level of the rule implementation,

Theorem 2 *In a delta-accurate semantics of the HiTAsm, two consecutive states are different iff that state generates or the delayed update set contains a delayed update with the delay δ.*

$$\forall \mathfrak{A}_s, \mathfrak{A}_e \in \mathfrak{A}_\delta . \mathfrak{A}_s \neq \mathfrak{A}_e \Leftrightarrow \exists (l_i, v_i, \delta_i) \in \mathcal{U}^{\mathfrak{A}_s} \mid \delta_i = \delta$$

Statement 3 *(Time accurate model). The default semantics of our HiTAsm framework is a specialization of the more general delta-accurate semantics where a transition is made only if it generates un update in the new state and the time of that transition is the minimum delay necessary to fire a delayed update.*

Instruction level The instruction level semantics captures the effects of the complete execution of an instruction on a processor state.

Definition 44 (Instruction Level). Let $\mathfrak{A}_s, \mathfrak{A}_e \in \mathfrak{A}$ be two HiTAsm states of $\mathfrak{A}_I$ together with the superuniverse $X \supset \mathcal{T}_I$ and $\Longrightarrow_I$ the transition rule, we have

$$\mathfrak{A}_s \xrightarrow[I]{\delta_I}_{instr} \mathfrak{A}_e.$$

representing a move from the start state $\mathfrak{A}_s$ to the finish state $\mathfrak{A}_e$ after the complete interpretation of the instruction I.

Instruction Block level The instruction level semantics can be naturally extended to a whole set of instructions.

Definition 45 (Instruction Block Level). Let $\mathfrak{A}_s, \mathfrak{A}_e \in \mathfrak{A}$ be two HiTAsm states of $\mathfrak{A}_B$ together with the superuniverse $X \supset \mathcal{T}_B$ and $\Longrightarrow_B$ the transition rule, we have

$$\mathfrak{A}_s \xrightarrow[I_0,\ldots I_n]{\delta_B}{}_B \mathfrak{A}_e.$$

representing a move from the start state $\mathfrak{A}_s$ to the finish state $\mathfrak{A}_e$ after the complete interpretation of the instructions $I_0, \ldots I_n$.

Abstract model remarks Please note that if the initial state in the transition is an abstract state (like above), several transitions and therefore several successor states may exist. We can therefore define the notion of worst case transition.

Definition 46 (Local worst case time).

$$max_\delta\,(\mathfrak{A}, R) \stackrel{\text{not}}{=} max\left(\delta[\![R]\!]^{\mathfrak{A}} \,\middle|\, \mathfrak{A} \xrightarrow[R]{\delta[\![R]\!]^{\mathfrak{A}}}_\rho \mathfrak{A}_e\right)$$

representing the maximum time taken by a move from the start state $\mathfrak{A}_s$ to the finish state $\mathfrak{A}_e$ after the complete interpretation of the rule R.

4.2 Timing Anomalies

The evolution of processor architecture has made the timing analysis more complicated. Among the consequences of modern processor features, timing anomalies (TA) have an important impact on the WCET estimation, by breaking the compositionality of the analysis. Timing anomalies in the context of WCET analysis were first described by [LS99b]. Hardware acceleration mechanism produce interferences that lead to timing anomalies, i.e., a local timing change causes an either larger or inverse change of the global timing.
The abstract execution problem comes to:

- being able to handle many potential states;

- at any moment either we have precise information about values $\implies$ next step or we don't and we must identify the worst case. If we don't know if the worst case can occur, we must suppose that it will.

Usually, the WCET analysis is confronted, at each execution step, to either a great overestimation of the WCET or a complex analysis of the precise worst case path.
We will now define the timing anomalies using the HiTAsm framework and the preamble in section 4.1 on page 109. The TA identification method is based on the definition from [RS09].

Definition 47 (Timing Anomaly). A HiTAsm semantics has a timing anomaly if it exists an instruction sequence $I_0, I_1, \ldots I_n$, with their associated update sets U_{I_i} such that for an abstract state $\mathfrak{A}$

$$\exists \mathfrak{A}', \mathfrak{A}'' \in \mathfrak{A}.\mathfrak{A}_s \xrightarrow[I_0]{\delta_1}_I \mathfrak{A}', \mathfrak{A}_s \xrightarrow[I_0]{\delta_2}_I \mathfrak{A}'' \mid \delta_1 < \delta_2 \implies$$
$$\implies \delta_1 + max_\delta(\mathfrak{A}', I_1, \ldots I_n) > \delta_2 + max_\delta(\mathfrak{A}'', I_1, \ldots I_n)$$

Every semantics that is characterized by definition 47 will not allow a safe compositional timing analysis. This means that local worst-case paths can not discard all others paths because they will not always lead to the global worst-case. In the following we extend the modeling of timing anomalies from [RS09].

Definition 48 (Local worst-case). Let $\mathfrak{A}^{(1)}, \ldots, \mathfrak{A}^{(n)} \in \mathfrak{A}$ be the possible successor states of $\mathfrak{A}_s$ after the interpretation of instruction I_0 such that

$$\mathfrak{A}_s \xrightarrow[I_0]{\delta_i}_I \mathfrak{A}^{(i)}$$

we define the local worst-case transition by:

$$lwc_\delta(\mathfrak{A}_s, I_0) \stackrel{\text{def}}{=} \left(\left(\mathfrak{A}_s, \mathfrak{A}^{(k)}\right) \in \xrightarrow[I_0]{\delta_i} \middle| \delta_k > \delta_i \forall k, i \in 1, \ldots n \right)$$

Definition 49 (Global worst-case path). Let $\mathfrak{A}_i^{(i)}$ be a path from that start state, under the interpretation of instructions $I_1, \ldots, I_n$ denoted by $(\mathfrak{A}^{(i)}, I_1, \ldots, I_n)$such that

$$\mathfrak{A}^{(i)} \xrightarrow[I_1,\ldots,I_n]{\delta_i}_I \mathfrak{A}_e$$

we define the local worst-case transition by:

$$gwc_{\delta}\left(\mathfrak{A}^{(i)}, I_1, \ldots, I_n\right) \stackrel{\text{def}}{=} \left(\left(\mathfrak{A}^{(i)}, \mathfrak{A}^{(0)}, \ldots, \mathfrak{A}^{(n)}\right) \middle| max\left(t \middle| t = \sum_{i=0}^{n} \delta_i.\mathfrak{A}_s \xrightarrow[I_1,\ldots I_n]{\delta_i} I\ \mathfrak{A}^{(n)}\right)\right).$$

TA-prone candidate validation As previously stated, TA characterization is given through necessary but not sufficient conditions, therefore, a search algorithm must identify TA-prone paths.We introduce a novel algorithm based on our framework that reduces the computation requirements to identify potential TA paths. The technique consists in validating a TA candidate as soon as the temporal gain is reduced meaning that the time of the path starts growing, even if it is not superior to the initial time difference.

TA identification Another contribution to the brute force algorithm presented in [RS09] is the use of our framework to identify transitions or paths that locations from units that are TA-susceptible and automatically labelling them as unsafe for compositional analysis.

4.2.1 Handling Timing Anomalies

Safely discard states The same work in [RS09] defines a temporal state difference function for each transition based on all possible executions of the instruction sequences.
Let us also define a similar delta function $\Delta_{\mathfrak{A}} : \mathfrak{A} \times \mathfrak{A} \implies \mathcal{T}_R \cup \top$ that will be used to bound the maximal worst-case timing difference between two states on any possible instruction sequence that will follow. Please not that no constant can bound transitions that generate a domino effect, hence the supremum $\top$ element is added in order to take this fact into account.

Definition 50 ($\Delta_{\mathfrak{A}}$ function). A function $\Delta_{\mathfrak{A}} : \mathfrak{A} \times \mathfrak{A} \implies \mathcal{T}_R \cup \top$ that bounds the worst case transition time towards two successor states is defined by:

$$\Delta_{\mathfrak{A}}\left(\mathfrak{A}', \mathfrak{A}''\right) \geq max_{\delta}\left(\mathfrak{A}', I_1 \ldots, I_n\right) -_{\mathcal{T}} max_{\delta}\left(\mathfrak{A}'', I_1 \ldots, I_n\right)$$

where the max_{δ} function is defined in definition 46 on page 113 and $-_{\mathcal{T}}$ is the subtraction operation defined on the time domain $\mathcal{T}$ of $\mathfrak{A}$. In the following we suppose that the time domain is the positive integers set with their natural operations.

Using the $\Delta_{\mathfrak{A}}$ function Let us suppose that a $\Delta_{\mathfrak{A}}$ function was computed for every state pair $(\mathfrak{A}^{(1)}, \mathfrak{A}^{(2)})$ following an abstract execution split, where a choice on the worst-case path must be taken. We want to discard as many states $\mathfrak{A}^{(i)}$ and pursue the execution on as few paths as possible.

Theorem 3 *(Safe state discard). Let $\mathfrak{A}^{(1)}, \mathfrak{A}^{(2)} \in \mathfrak{A}$ be two states and δ_1, δ_2 be the execution time needed to reach state $\mathfrak{A}^{(1)}$ and $\mathfrak{A}^{(2)}$ respectively, it is safe to discard the path induced by the state $\mathfrak{A}^{(2)}$ if:*

$$\delta_1 - \delta_2 \geq \Delta_{\mathfrak{A}}\left(\mathfrak{A}^{(1)}, \mathfrak{A}^{(2)}\right)$$

Proof 2 *(sketch) The proof is obvious, by supposing that it is not safe to discard $\mathfrak{A}^{(2)}$. This means that $\mathfrak{A}^{(2)}$ can generate a path such that:*

$$\exists \delta_i . \mathfrak{A}' \xrightarrow[I_1,\ldots,I_n]{\delta_i} {}_I\ \mathfrak{A}_e | \delta_i < \Delta_{\mathfrak{A}}\left(\mathfrak{A}', \mathfrak{A}''\right) \overset{\text{cf. def 46, 50}}{\Longrightarrow} \delta_i > max\left(\delta_j \,\middle|\, \mathfrak{A}' \xrightarrow[I_1,\ldots,I_n]{\delta_j} \mathfrak{A}_e\right)$$

which is a contradiction. Therefore it is safe to discard state $\mathfrak{A}^{(2)}$ ■.

Discarding the state $\mathfrak{A}^{(2)}$ means that we can take the path induced by $\mathfrak{A}^{(1)}$ by using the transition time:

$$\delta_1' = \delta_1 + \Delta_{\mathfrak{A}}\left(\mathfrak{A}', \mathfrak{A}''\right). \tag{4.4}$$

HiTAsm optimization techniques The computation of the $\Delta_{\mathfrak{A}}$ function for a pair of states is quite complex. We introduce several optimizations by defining *safe windows* and also by determining a likeliness factor of potentially timing anomalous states. We start from the observation that we do not need to compute the execution time of all the instructions sequence starting from that point. We can stop as soon as the path execution time is greater than the difference of time between the two compared states. We introduce the concept of *unsafe window*, as a sequence of rules during which it is not safe to assume the local worst-case was the global worst-case. We refer only to instructions in the present example but it can be easily extended to any atomic rule.

Definition 51 (Unsafe step). An unsafe execution step is a transition that produces a timing anomaly in other words, a transition starting in a step where we cannot suppose that the

longest temporal transition will lead to the longest execution path:

$$\delta' > \delta'' \implies max\left(\delta_i \left| \mathfrak{A}' \xrightarrow[I_1,\ldots,I_n]{\delta_i} \mathfrak{A}_e \right.\right) > max\left(\delta_j \left| \mathfrak{A}'' \xrightarrow[I_1,\ldots,I_n]{\delta_j} \mathfrak{A}_e \right.\right)$$

Definition 52 (Unsafe execution window). An unsafe execution window is an abstract path starting in an abstract state $\mathfrak{A}_s$ and ending after k_{TA} abstract transitions, as soon as a timing anomaly is produced:

$$k_{TA} \overset{\text{not}}{=} min(k|k \in 1,\ldots,n)|\delta' > \delta'' \implies max_\delta\left(\mathfrak{A}', I_1,\ldots,I_k\right) > max_\delta\left(\mathfrak{A}'', I_1,\ldots,I_k\right)$$

Analogously we can define the safe step and execution window.

Definition 53 (Safe execution step). A safe execution step is a transition that occurs after a timing anomaly occurred and before a another timing anomaly can occur.

Definition 54 (Safe execution window). We define the safe execution window as a subset of the run composed only of safe steps.

Property 10 *During a safe execution window the compositionally of the timing analysis is ensured.*

Based on the above definitions we can now introduce a heuristics that will cathegorize two states as timing anomalous with a certain probability, without computing all the k_{TA} steps needed to determine if a TA can occur. This will be useful when several path are being evaluated in the same time, in order to decide which one to traverse in priority.

Intuition The idea is to used the monotonicity of the time function and associate and increasing degree in the difference time function to a timing anomaly likeliness.

Safeness The only risk of this assumption is that we detect false positives, which will contribute to an over-approximation of the WCET estimation and introduce an imprecision. Therefore the estimation will remain safe.

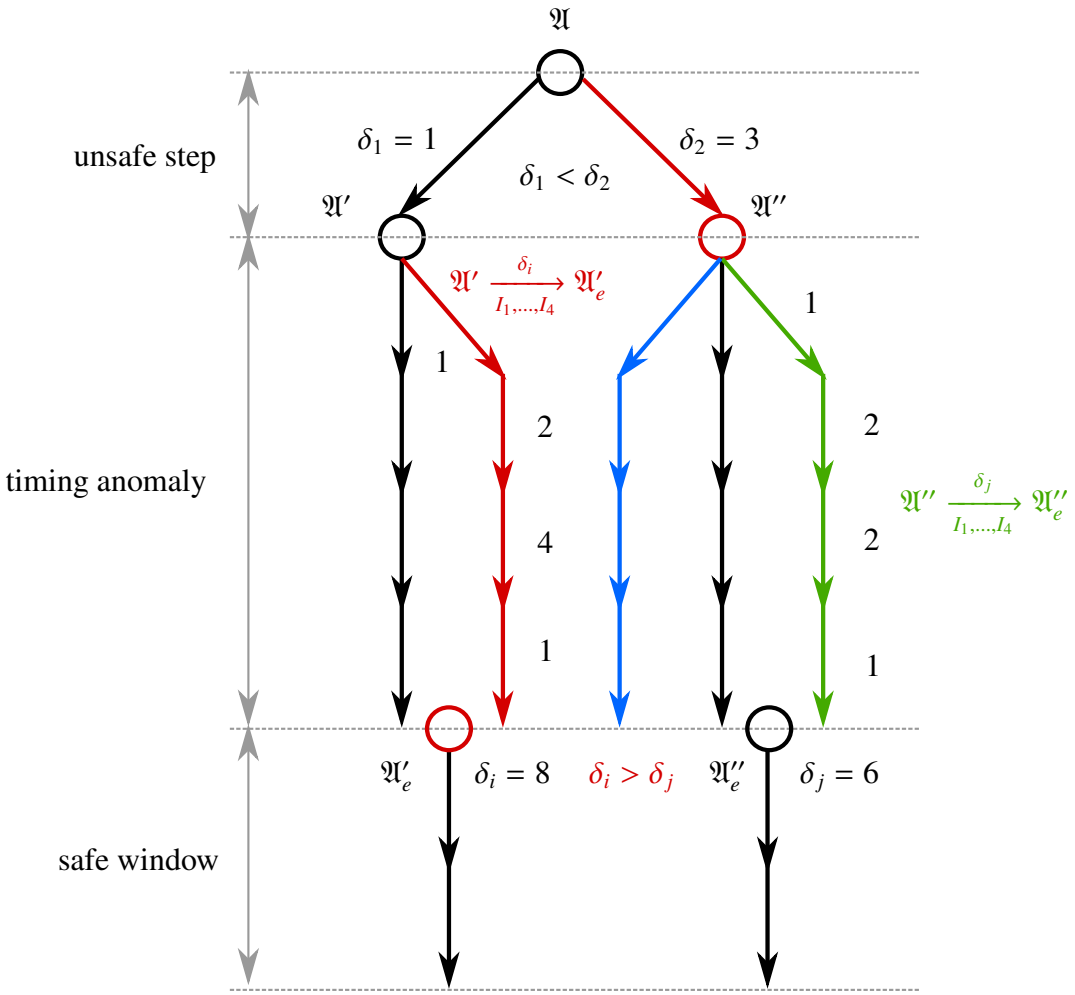


Figure 4.1: Timing anomalie example

Definition 55 (TA likeliness). After ek_{TA} abstract transitions we can decide that two states might produce a timing anomaly with a factor of p:

$$ek_{TA} \overset{\text{not}}{=} min(k|k \in 2,\ldots,n)|max_\delta(\mathfrak{A}', I_1,\ldots,I_k) - max_\delta(\mathfrak{A}'', I_1,\ldots,I_k)$$

$$> (max_\delta(\mathfrak{A}', I_1,\ldots,I_{k-1}) - max_\delta(\mathfrak{A}'', I_1,\ldots,I_{k-1})) + p \cdot (\delta' - \delta'')$$

where $p \in [0,1]$ gives an proportional information about the additive likeliness used to categorize the states.

An additional idea is to use the timing anomalies identification techniques, presented in [WKPR05b] and [EPB+06], to partition the analysis space in several categories corre-

sponding to the presence or absence of the TA.
[WKPR05b] also proposes a criterion that provides a necessary but not sufficient condition for timing anomalies to occur. Our model is based on relation between locations that are stored in functions and guarded updates, capturing by definition the timing anomalies. These relations are exploited to identify the necessary condition for the TA to occur. We can safely assume that if the necessary conditions are not satisfied, TA will not occur and we can use the *divide and conquer* analysis approach.
Therefore we create functions that evaluate the timing anomalies that can occur, obtaining the following cases:

- no TA are possible $\Longrightarrow$ choose the worst case;
- some TA might occur $\Longrightarrow$ analyze all the cases or use abstraction techniques.

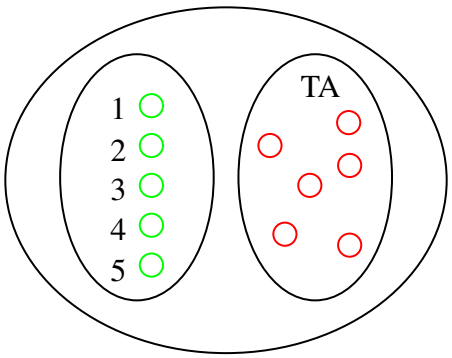


Figure 4.2: Timing anomalies partitioning

In the category where no TA can occur, we have an order on the states defined by a distance on abstract states based on the temporal impact and further relations on locations. This can also be used for defining similarities between states and perform merging based on techniques presented in [BM09b].

4.3 HiTAsm for WCET estimation in a nutshell

The HiTAsm method was introduced in order to facilitate the WCET estimation. The timed execution and the hierarchical abstraction levels of hardware components make this

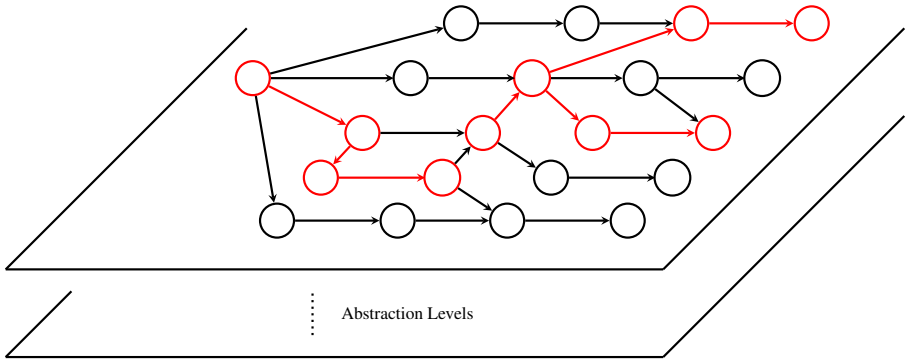


Figure 4.3: Timing anomalies identified paths through relations between locations

framework suited for timing analysis. In this section we give a few justifications and pinpoint some methods on how to use the HiTAsmL language in order to limit the state space explosion problem, inherent to industrial-size system analysis. This concepts will be detailed later in chapter 7.

The global structure of the WCET estimation tool is depicted in figure 4.4. The analysis consist in a *conjoint symbolic execution* of the program binary and the processor's HiTAsm model. The classical symbolic execution application is extended from the program level to a processors running a program. After the binary analysis, the CFG of the program will contain instruction block in each node. Each instruction that operates with unknown register values will receive symbolic values instead. The *symbolic instruction* are the entry for the processor model, generating all reachable states of the processor P under the program $\mathcal{P}$. State space explosion is handled through *state merging*, performed on similar symbolic processor. In order to identify additional similar state, a feature called the *Prediction Module* is used backtracking from known identical states in order to further reduce the analyzed state space, [PB12]. Merging can be set to allow non-identical state merging which consequently introduces a precision loss. By systematically keeping track of the introduced imprecision, the method can approximate the accuracy of the estimation.

4.3.1 Timing anomalies remarks

The above section not only proves that our framework can naturally capture timing anomalies but also that it is capable of taking into account state of the art TA identification

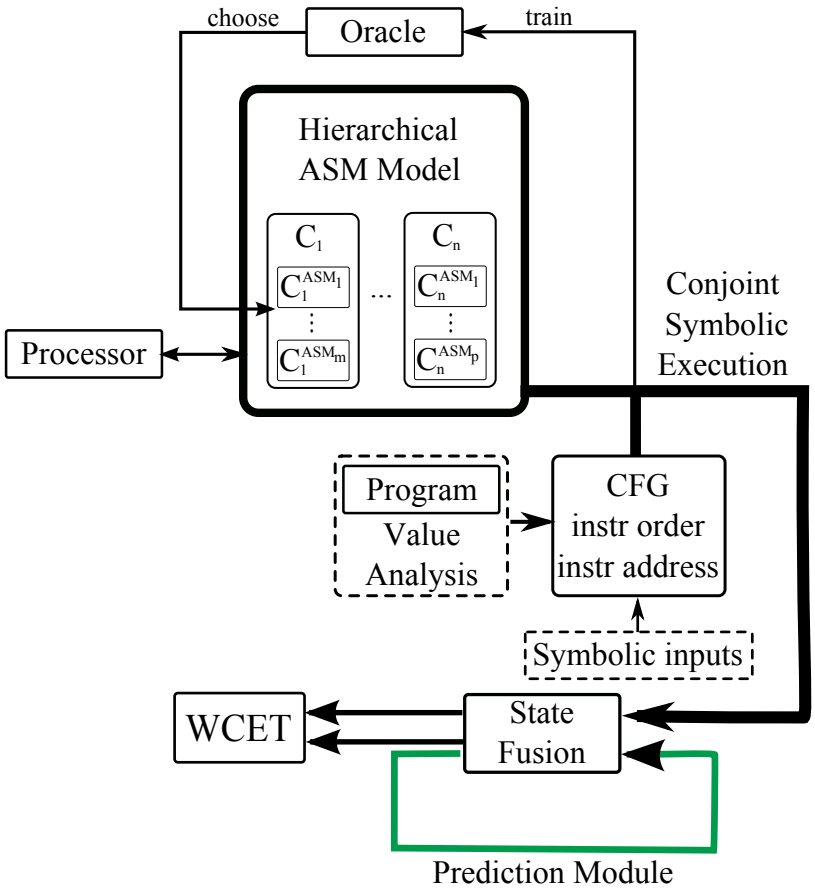


Figure 4.4: Global architecture of the WCET estimation tool

and discarding techniques but also improve them at least in the computational complexity.

4.4 Conclusions

In this chapter we have depicted the advantages and versatility of our HiTAsm formal framework. After a closer look into the hierarchical levels of abstraction, a HiTAsm state transition system is introduced that can adapt to multiple semantic levels. These semantic levels are also used to depict the way in which the timing anomalies are naturally captured and an approach to handle them based on state of the art timing anomalies capturing algorithm. Optimization techniques based on the HiTAsm framework are also introduced that further prove the suitability of our model for the WCET estimation.

Chapter 5

The HiTAsm Language Definition

The HiTASmL language implementation functions as a wrapper towards the main target language, *C#* used for the analyzer implementation. The abstract syntax tree of the parsed processor HiTAsmL code is further used by the analyzer to perform a conjoint symbolic execution of the program and hardware.
After an overview of the language syntax, the semantics of the language are introduced and code snipets of the Motorola MPC555 processor presented as example and discussion support.

5.1 Syntax of the language

In this section we present the main syntactical elements of our HiTAsmL language. We chose to remain as close as possible to the original ASM framework syntax in order to ease the adoption of the language.
The delay information, δ that will generate the timed update (l, v, δ) is simply specified after each delayed construct, separated by a comma. Hierarchical levels of abstraction are defined in a syntax similar to the object-oriented program class hierarchy constructs.

⟨*program*⟩ ::= ('htasm' | 'module') ⟨*ident*⟩ [':' ⟨*ident*⟩] ⟨*header*⟩ [⟨*body*⟩]

⟨*ident*⟩ ::= 'a..z,_' { 'a..z,_,0..9,unicode character over 00C0' }

⟨*header*⟩ ::= {'import' ⟨*ident*⟩} {'export' (⟨*ident*⟩ | '*') } 'signature' ':' ⟨*signature*⟩

⟨*signature*⟩ ::= ('static' | 'dynamic') ⟨*ident*⟩ ':' ⟨*varType*⟩ { '*' ⟨*varType*⟩ } '->' ⟨*varType*⟩

⟨*varType*⟩ ::= 'Int' | 'Bool' | 'RuleRef'

⟨*body*⟩ ::= 'definition' ':' (⟨*function*⟩ | ⟨*rule*⟩)

⟨*function*⟩ ::= 'function' ⟨*ident*⟩ ':' ⟨*domains*⟩ '->' ⟨*varType*⟩ ⟨*functionBody*⟩

⟨*functionBody*⟩ ::= 'initial' '{' ⟨*functionInit*⟩ {',' ⟨*functionInit*⟩} '}'

⟨*functionInit*⟩ ::= '[' ⟨*expr*⟩ ',' ⟨*expr*⟩ ']' '->' ⟨*expr*⟩

⟨*ruleDecl*⟩ ::= 'rule' ⟨*ident*⟩ '(' [⟨*ident*⟩ ':' ⟨*varType*⟩] {',' ⟨*ident*⟩ ':' ⟨*varType*⟩ } ')' '::=' ⟨*block*⟩

⟨*block*⟩ ::= '{' ⟨*updateRule*⟩ | ('call' ⟨*ident*⟩) '}'

⟨*rule*⟩ ::= ⟨*ifRule*⟩ | ⟨*updateRule*⟩

⟨*ifRule*⟩ :: 'if' '('⟨*expr*⟩ ')' ⟨*rule*⟩ ['else' ⟨*rule*⟩]

⟨*updateRule*⟩ ::= ⟨*ruleNameCall*⟩ ':=' (⟨*ruleNameCall*⟩ | ⟨*expr*⟩)

⟨*ruleNameCall*⟩ ::= ⟨*ident*⟩ ['(' argList ')']

⟨*argList*⟩ ::= [(⟨*argList*⟩ ',' ⟨*expr*⟩) | ⟨*expr*⟩]

⟨*expr*⟩ ::= term | Unexpr | Binexpr

⟨*term*⟩ ::= number

HiTAsmL syntax

5.2 Semantic essence of HiTAsmL

This section serves the purpose of illustrating, through a series of examples and explanations, the syntax and behaviour of the language. HiTAsmL was conceived with hardware modelling in mind, although it can be used for modelling the surrounding system as well, therefore it was unburden of unuseful language constructions. There is no notion of objects in HiTAsmL, even if the abstract definitions of components can be seen as class specialisa-

tion and inheritance or implementation of highly abstract interfaces. HiTAsmL is organised in *hitasms* and modules. The main difference between the two is that a *hitasm* contains a main rule that serves as a starting point for the program, dealing with the definition of the initial state and the termination of the execution. For example when an inconsistent update is generated or no more updates are generated during a state and the update set is empty, the program terminates. One of the advantages of the ASM programming model is the separation between the generation of new values and the committal of those values into the persistent state. This synchronous parallelism, allowing the performing of a collection of parametrized actions in parallel, gives rise to a cleaner programming style than it is possible in standard imperative languages.

Modules provide a mean to syntactically structure large HiTAsms. Like their ASM counterparts, they are defined by a signature, containing the function names in use, which defines the internal state of the module, and import and export clauses that interface with external environment. The transitive closure of the *import* clause is not allowed to be cyclic. Only identifiers defined inside or imported are allowed to be used in a module.

Modules are also used to provide syntactical representation of the hierarchical levels. Several option arise in expressing the hierarchy of the module's abstraction levels. We could of use a concept similar to class inheritance with a *base* module being more abstract than the *derived* one.

Listing 5.1: One way of expressing the abstraction levels

```
1 module unitA
2   import unitB, unitC
3   export instr
4 module unitA1 : unitA
5 module unitA2 : unitA1
6 module unitA3 : unitA2
```

We chose a different approach in order to easily impose a partial order relation on the abstract modules that provides information to the analyzer (the *Oracle* dynamically chooses the definition in order to adapt to the needed precision). The programmer specifies which are the modules that will be abstracted, by enumerating the names, and then defines abstractions by specifying the base module and also the order with regard to another existing abstract definition (module).

Listing 5.2: HiTAsmL syntax for abstraction level definition

```
1 htasm component
2     hmodules unitA, unitB //...
3     import unitC
4 hmodule unitA
5 module unitA1 : unitA > unitA
6 module unitA2 : unitA > unitA1
7 module unitA0 : unitA < unitA
```

The module's inheritance defines a hierarchy for abstraction levels. The higher in the hierarchy the module is, the more abstract its definition will be. It also allows to define modules that are on the same hierarchical level and thus constructing dependency chains.

$M_{base} \rightarrow m_{ch_1} \rightarrow m_{ch_2} \rightarrow m_{ch_3}$

$M_{base} \rightarrow m_{ch_1} \rightarrow m_{ch_{2b}} \rightarrow m_{ch_{3b}}$

Both syntax could be used to express the above hierarchy, however it would become complicated to express the following:

$M_{newbase} \rightarrow M_{base} \rightarrow m_{ch_1} \rightarrow m_{ch_2} \rightarrow m_{ch_3}$

In other words, the refinement can go in both directions with minimal syntactical changes in the rest of the code.
During this chapter, for the ease of presentation, we deal only with one way abstractions, corresponding with the classical ASM *refinement method*. The HiTAsm model is though capable of both ways refinements. This design method is sufficient for defining the processor as it is natural to start with a hight-level vision of components and further refine by adding new locations and update rules.

5.2.1 HiTAsmL-s the core of the HiTAsm Language

In order to illustrate our language concepts, we first implemented HiTAsmL-S, a simpler version of HiTAsmL.

Listing 5.3: HiTAsmL-S implementation of the Oracle

```
1 rule oracle(forceAbstraction : Int, noVAdata : Int) =
2   if (forceAbstraction or noVAdata )
```

```
3     call (AbsLevelChoice(0))
4   elif (not forceAbstraction or noVAdata)
5     call (AbsLevelChoice(1))
6 rule AbsLevelChoice(level : Int) =
7   case level {
8     0 -> call LevelZeroAbsPipeline()
9     1 -> call LevelOneAbsPipeline()
10  }
11 rule LevelZeroAbsPipeline() =
12  call FetchZero()
13  call DecodeZero()
14 rule LevelOneAbsPipeline() =
15  call FetchOne()
16  call DecodeOne()
```

The main difference between the two languages is that some features like module abstractions and the oracle implementation are not fully included in the language semantics. Therefore a part of the semantics of HiTAsmL is directly implemented in HiTAsmL-S which actually eases the presentation of the concepts.

5.2.2 Preambule

The only types present in the language are *Int* and *Bool* as *Enums* are pre-evaluated to *Int* values. The set of typed literals, *Literals*, contains values such as 1, *true* or $undef$. HiTAsmL-S uses expressions as a syntactic mean to write the executable specifications. Let e be an expression without free variables and v a literal, then $e \xrightarrow{v} v$ is the evaluation of e to the value v. Therefore the evaluation of simple expressions is: $0 + 1 \rightarrow v1$

---

1 **if** *true* **then** 1 **else** $0 \xrightarrow{v} 1$

---

let $x = 1$ **in** $x + x \xrightarrow{v} 2$

Let us define some sets of interest.

$$\begin{aligned} Value &= Id \cup Literal \\ TypeMap &= Id \rightarrow Type \\ Location &= Id \times Args \\ Content &= Location \rightarrow Value \\ Update &= (Location \times Value) \times Delay \\ UpdateSet &= SetOf(Update) \end{aligned} \tag{5.1}$$

The states of a computation are represented by *stores*. The store is defined as the triple $s = (\theta, \omega, u)$, where θ is a type map, ω is a content map and u is an update set.

5.2.3 Assignments

Assignments can be done through updates or function initialisation. HiTAsmL features timed updates that assign a delay δ to a location update.

signature : x $\rightarrow$ Int **definition** : function init() = x := 2; $1 \xrightarrow{\omega,u} \phi, \{(x, 2, 19)\}$

Initially the store ω is empty, represented by the symbol ϕ. An update in the form (l, v, δ), in our case $(x, 2, 19)$, is added to the update set that was initially empty.

The initialisation of the function is made in the **definition** part with the **initially** construct.

signature : MEM : Int $\rightarrow$ Int
definition : function MEM(addr : Int) = **initially** $\{0 \rightarrow 1\} \xrightarrow{\omega,u} \{(MEM, 0) \mapsto 1)\}, \phi$.

An example of a HiTAsmL code of swapping the values of two functions is presented in Listing 3.1.

Listing 5.4: ASM example of instantaneous updates - variable value swap

```
1 function a : -> Int
2 function b : -> Int
3
```

```
4 rule valueSwap = {
5   a := b
6   b := a
7 }
```

Algorithm 2: Interpretation of `valueSwap` rule

1 *Initial State* $\longleftarrow \{a = 0, b = 1\}$;

2 *Update Set* $\longleftarrow \{(a, 1), (b, 0)\}$;

3 *New State* $\longleftarrow \{a = 1, b = 0\}$;

The interpretation of the update rule involves an update set. Therefore, prior to their *execution*, all known updates are collected and stored into an *update set* and *fired* simultaneously. In listing 2 we can see the interpretation of updates and the `valueSwap` rule that modifies the interpretation of nullary functions a and b.

5.2.4 Firing updates

Firing updates in ASMs consists of putting the value v at location l, $u = (l, v)$. HiTAsm features timed updates, $u = (l, v, \delta)$, that are fully described in [PMB13a].
If u is a consistent update and $s = (\theta, \omega, u)$ is a store, we note by $\omega(lid)$ the value associated with the location defined by the identifier lid in the content map ω, and we define by $\hat{s} = (\theta, \hat{\omega}, \hat{u})$ the new store, where $\hat{\omega}$ and $\hat{u}$ have the following semantics

$$\hat{\omega}(lid) = \begin{cases} v & if\ (lid, v, \delta) \in u\ and\ \delta = \delta_{min} \\ \omega(lid) & if\ (lid, v, \delta) \in u\ and\ \delta > \delta_{min} \end{cases} \tag{5.2}$$

$$\hat{u}(lid) = \begin{cases} \phi & if\ (lid, v, \delta) \in u\ and\ \delta = \delta_{min} \\ (lid, v, \delta - \delta_{min}) & if\ (lid, v, \delta) \in u\ and\ \delta > \delta_{min} \end{cases} \tag{5.3}$$

Listing 5.5: Firing of timed updates

```
1 function rA : -> Int
2 function rB : -> Int
3 function rD : -> Int
```

```
4  function intr: -> Int
5
6  rule ALU_add = {
7    rA := mem(getFirstOpAddr(intr)), 2
8    rB := mem(getSndOpAddr(intr)), 2
9    rD := mem(getDestAddr(intr)), 2
10
11   mem(rD) := mem(rA) + mem(rB), 4
12 }
```

Interpretation of the `ALU_add` rule.

1. *Initial State* $\longleftarrow \{mem(1) = 3, mem(2) = 7, CT = 0, \dots\}$
2. *Timed Update Set 1* $\longleftarrow \{(rA, 1, 2), (rB, 2, 2), (rD, 3, 2),$
   $(mem([rD]), mem([rA]) + mem([rB]), 4), (CT, 0)\}$
3. $State_1 \longleftarrow \{rA = 1, rB = 2, rD = 3, mem(1) = 3, mem(2) = 7, CT = 2, \dots\}$
4. *Timed Update Set 2* $\longleftarrow \{(mem(3), mem(1) + mem(2), 2), (CT, 2)\}$
5. $State_2 \longleftarrow \{rA = 1, rB = 2, rD = 3, mem(3) = 10, CT = 4, \dots\}$

5.2.4.1 Module abstraction

We now introduce the semantics of module abstraction. A simple shifting example is presented that adds a value (10 in our example) to a specified address, *addr*. The abstract of modules inherits the signature of the extended modules. Let us consider a shifting unit called *ALUBFU* that has a *rA* function in its signature.

hmodules : ALUBFU $\xrightarrow{\theta,\omega,u} \{(\_ \mapsto ALUBFU)\}, \phi, \phi$
module ALUBFU
signature : rA : Int $\xrightarrow{\theta,\omega,u} \{(aluId_0 \mapsto ALUBFU)\}, \{(aluId_0, rA)\}, \phi$

Let $ALUBFU1$ be an extension of the $ALUBFU$ module that defines a *rAux* function. We can see that it also inherits the function name *rA* from the $ALUBFU$'s signature.

module ALUBFU1 : ALUBFU
signature : rAUX : Int $\xrightarrow{\theta,\omega,u} \{(aluId_1 \mapsto ALUBFU)\}, \{(aluId_1, rA), (aluId_1, rAUX)\}, \phi$

Modules also inherit rule names. We define the $shift(addr : Int)$ function in the $ALUBFU$ module and $shiftC(addr : Int)$ in the $ALUBFU1$ module.

module ALUBFU

definition : rule shift(addr : Int) rA := addr + 10, 7 $\xrightarrow{\theta,\omega,u} \{(aluId_0 \mapsto ALUBFU)\}, \phi, \{(aluId_0, add)\}$

module ALUBFU1 : ALUBFU

definition : rule shiftC(addr : Int) $\xrightarrow{\theta,\omega,u} \{(aluId_1 \mapsto ALUBFU)\}, \phi, \{(aluId_0, add), (aluId_1, addic)\}$

But most importantly they can override existing rule names. Let $shift(addr : Int)$ be a function redefinition in the $ALUBFU1$ module.

module ALUBFU1 : ALUBFU

definition : rule shift(addr : Int) rAUX := 10, 2 rA := addr + rAUX, 5
$\xrightarrow{\theta,\omega,u} \{(aluId_1 \mapsto ALUBFU)\}, \phi, \{(aluId_1, add)\}$

The effect of the rule name override is shown in the following through the **call** rule with the following semantics.

$$
\begin{array}{l}
{\scriptstyle \textbf{call}}\, add(11) \\
\xrightarrow{\rho,\omega,u} \begin{cases}
\{(aluId_0, shift)\}, & \{(aluId_0, addr) \mapsto 11\}, \{(rA, 21, 7)\} \,{\scriptstyle \text{if}}\, oracle(alu) = 0, \\
\{(aluId_1, shift)\}, & \{(aluId_1, addr) \mapsto 11\}, \{(rAUX, 10, 2), (rA, 21, 5)\} \\
 & \xrightarrow{\hat{\omega},\hat{u}} \{(aluId_1, addr) \mapsto 11, (aluId_1, rAUX) \mapsto 10\}, \{(rA, 21, 3)\} \\
 & \,{\scriptstyle \text{if}}\, oracle(alu) = 1.
\end{cases}
\end{array}
\tag{5.4}
$$

5.2.5 HiTAsmL semantics

In the following we define our language using a structural operational semantics (SOS), [Plo04],

- Simultaneous updates

$$
\frac{< U_1, \mathfrak{A} > \rightarrow \mathfrak{A}'}{< U_1; U_2, \mathfrak{A} > \rightarrow < U_2, \mathfrak{A} >} \; \mathcal{L}oc(U_1) \neq \mathcal{L}oc(U_1) \tag{5.5}
$$

or

$$\frac{\langle U_1, \mathfrak{A}\rangle \rightarrow \mathfrak{A}', \langle U_2, \mathfrak{A}\rangle \rightarrow \mathfrak{A}'}{\langle U_1; U_2, \mathfrak{A}\rangle \rightarrow \mathfrak{A}'} \tag{5.6}$$

- Timed updates

$$\frac{< \bar{t}_i, \mathfrak{A} >\Rrightarrow \bar{x}_i, < \bar{u}_i, \mathfrak{A} >\Rrightarrow \bar{y}_i, < CT, \mathfrak{A} >\Rrightarrow CT + \delta_i, < \delta_j, \mathfrak{A} >\Rrightarrow \delta_j - \delta_i}{< (f_1(\bar{t_1}) := u_0, \delta_0); \ldots; (f_n(\bar{t_n}) := u_n, \delta_n), \mathfrak{A} >\rightarrow< \mathfrak{A} \biguplus (f_i(\bar{x}_i) \mapsto y_i) >} \quad \delta_i = min(\delta_i), j = 0 \ldots n$$

5.2.6 A HiTAsmL graphical syntax

In order to facilitate the processor definition and the code reuse using our framework, we integrated a UML-style graphical syntax for the HiTAsmL language. It will be later used in practice for the modeling of the MPC555 processor in chapter 6 on page 139.

5.2.6.1 HiTAsm Module

The module regroups several language elements under the same namespace. It is depicted by a box with the label of the modules's name, a set of elliptic boxes with a rule name label and optionally, the delay corresponding to the execution of its set of rules. FIgure 5.1 shows the graphical interpretation of the HiTAsmL code in listing 5.6.

Listing 5.6: HiTAsmL module

```
1 module moduleName
2  definition:
3     rule ruleName(reg : Int) =
4       if reg = ALUdestReg then
5         dependency := true
```

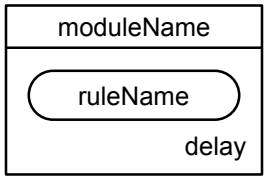


Figure 5.1: HiTAsmL module

5.2.6.2 HiTAsm Abstract Module

An abstract module is depicted as a module but labeled with the abstract module's name in italic font.

Listing 5.7: HiTAsm Abstract Module

```
1 hmodules abstractModule
```

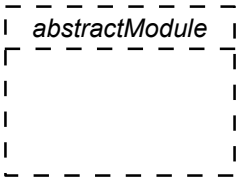


Figure 5.2: HiTAsmL abstract module

5.2.6.3 HiTAsm Hierarchic Module

A hierarchic module is depicted as a module but labeled with the abstract module's name, a colon and the name of the base or abstract module.

Listing 5.8: HiTAsm Hierarchic Module

```
1 hmodule absModuleName : moduleName > moduleName
```

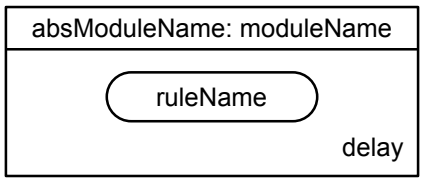


Figure 5.3: HiTAsmL abstract module

5.2.6.4 HiTAsm Function

A HiTAsmL rule can be depicted as a map between the HiTAsmL values. As a convention, the locations to the left are the signature of the function (inputs locations) and to the right, the result location.

Listing 5.9: HiTAsmL function

```
1 hmodule module1
2   signature:
3     static f1 : -> Int
```

```
4     static f2 : Int -> Int
5   definition:
6     function f1() =
7       -> v1
8     function f2(v : Int) =
9       v1 -> v2
```

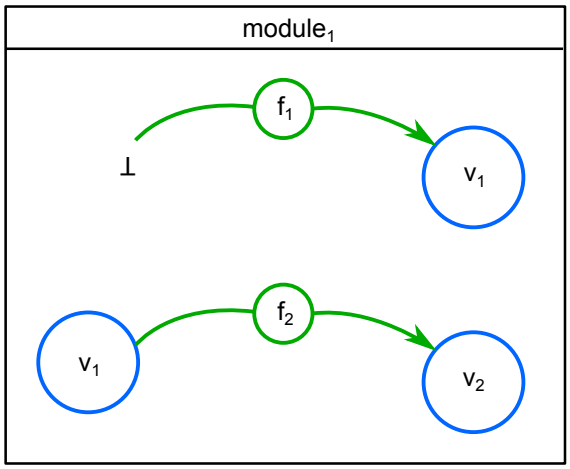


Figure 5.4: HiTAsmL nullary function

5.2.6.5 HiTAsm Rule

A HiTAsmL rule can be depicted as a map between the HiTAsmL locations. As a convention, locationsTo the left, the signature of the function and to the right, the result location.

Listing 5.10: HiTAsmL rule

```
1 hmodule module1
2   definition:
3     rule rule1(l1 : Int, l2 : Int, ln : Int) =
4       ld := l1 + l2 + ln, delay
```

5.2.6.6 HiTAsm abstractions

As we already stated, the HiTAsm framework enables incremental design through step-wise refinements.

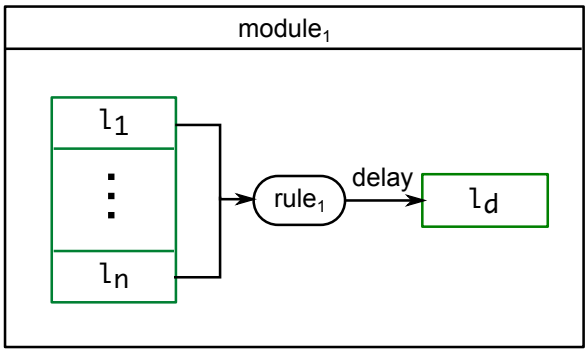


Figure 5.5: HiTAsmL rule

The first step will be the definition of a main module with a main rule, figure 5.7. This corresponds to the highest abstraction level in the model's hierarchy. Therefore the temporal label is depicted by top, $\top$ meaning that we have no delay information.

Listing 5.11: HiTAsmL rule

```
1 htasm hName
2   hmodules: absModule
```

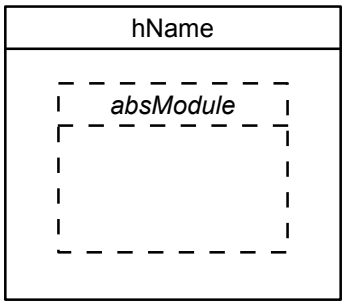


Figure 5.6: HiTAsmL htasm

Listing 5.12: HiTAsmL rule

```
1 htasm hName
2   hmodule mainModule : absModule
3     definition:
4       rule mainRule () , T
```

The model can be further refined with the delay information corresponding to the longest execution step of the processor between all the possible configurations. The squared head arrow represents a *concretization* of the definition.

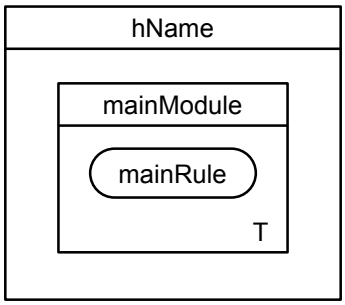


Figure 5.7: HiTAsmL hmodule

Listing 5.13: HiTAsmL rule

```
1
2 module mainModule#1 :mainModule > mainModule#1
3   definition:
4     rule mainRule () , maxdelay
```

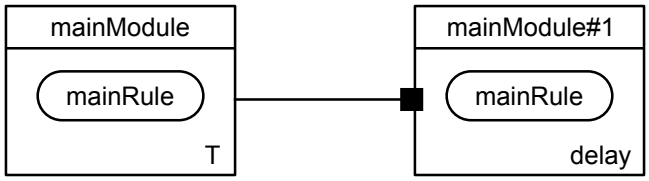


Figure 5.8: Lower level of abstraction

Listing 5.14: HiTAsmL rule

```
1
2 module mainModule#2 :mainModule < mainModule#2
3   definition:
4     rule mainRule () , maxdelay
```

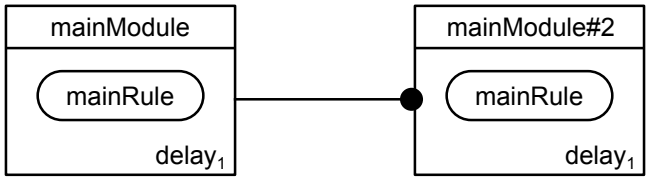


Figure 5.9: Higher level of abstraction

5.3 Implementation

The HiTAsm Language is implemented using a compiler of compilers based on the .Net technology called Irony [Iro]. As opposed to most existing yacc/lex-style solutions Irony does not use any scanner or parser code generation from grammar specifications written in a specialized meta-language. We choose to implement the analyzer in the .Net technology because of its versatility and especially because the coding times are quite short. The selected platform also justified the choice of Irony whose target language grammar is coded directly in *C#* using operator overloading to express grammar constructs. This means that defining the program syntax can be really short, for example:

- Mini-Python - 140 lines;
- Java - 130 lines;
- Scheme - 200 lines;
- JSON - 39 lines.

Besides the compactness of the defined syntax, the good integration into .Net proved as a real time saver with relatively no execution time penalty.

5.4 Conclusions

The HiTAsmL will provide a clear and easy to use language for the processor specifier. Its independence of the analyzer represents a forward step in the adaptability of the WCET estimation method. The migration to new hardware platform is also eased by the relatively short modeling times. HiTAsmL is an easy to use and learn language, with a simple syntax for program constructs and time annotations. The hierarchical abstraction levels of the hardware components can be easily defined in an *object oriented* style that reminds of class inheritance.
In the next chapter we present the aspects of a full processor implementation. The chosen processor features modern hardware units that provide a good use-case of the modelling language, as well as the analysis.

In terms of size the full processor implementation has under 1000 lines of code for the *concrete* definition and can go as low as 500 for an usable *abstract* definition. This is an advantage for the adaptability of the method, and proves that with relatively few implementation effort, new hardware can be quickly take into consideration by our analysis.

Chapter 6

The Hardware Model

In this chapter we present the implementation of the Motorola MPC555 processor in the HiTAsmL language. This processor is widely used in safety-critical embedded systems, and represents a good case study as it features most of the complex hardware components a WCET analysis should take into account.
The choice of the language, an extension of the ASM formalism, is intended to enrich the design with verification capabilities, [HC97], ease of use, a *human readable, machine executable* syntax and hierarchical timed abstractions to ease the WCET estimation performed by the analyzer. We can add to these features the advantage of an incremental design through definition refinements that eases the design or modeling stage of the processor.
After a brief introduction to processor modeling, we provide an analysis of the hardware architecture influence on the WCET estimation, predictability and analysis as a whole in section 6.1.1.

6.0.1 Global algorithm

The processor model will be of great importance in our timing analysis method. Its features and definitions, with their correspondent level of abstraction will be used throughout all the modules of the analyzer. Figure 6.1 repositions the processor model into the global approach of our method.

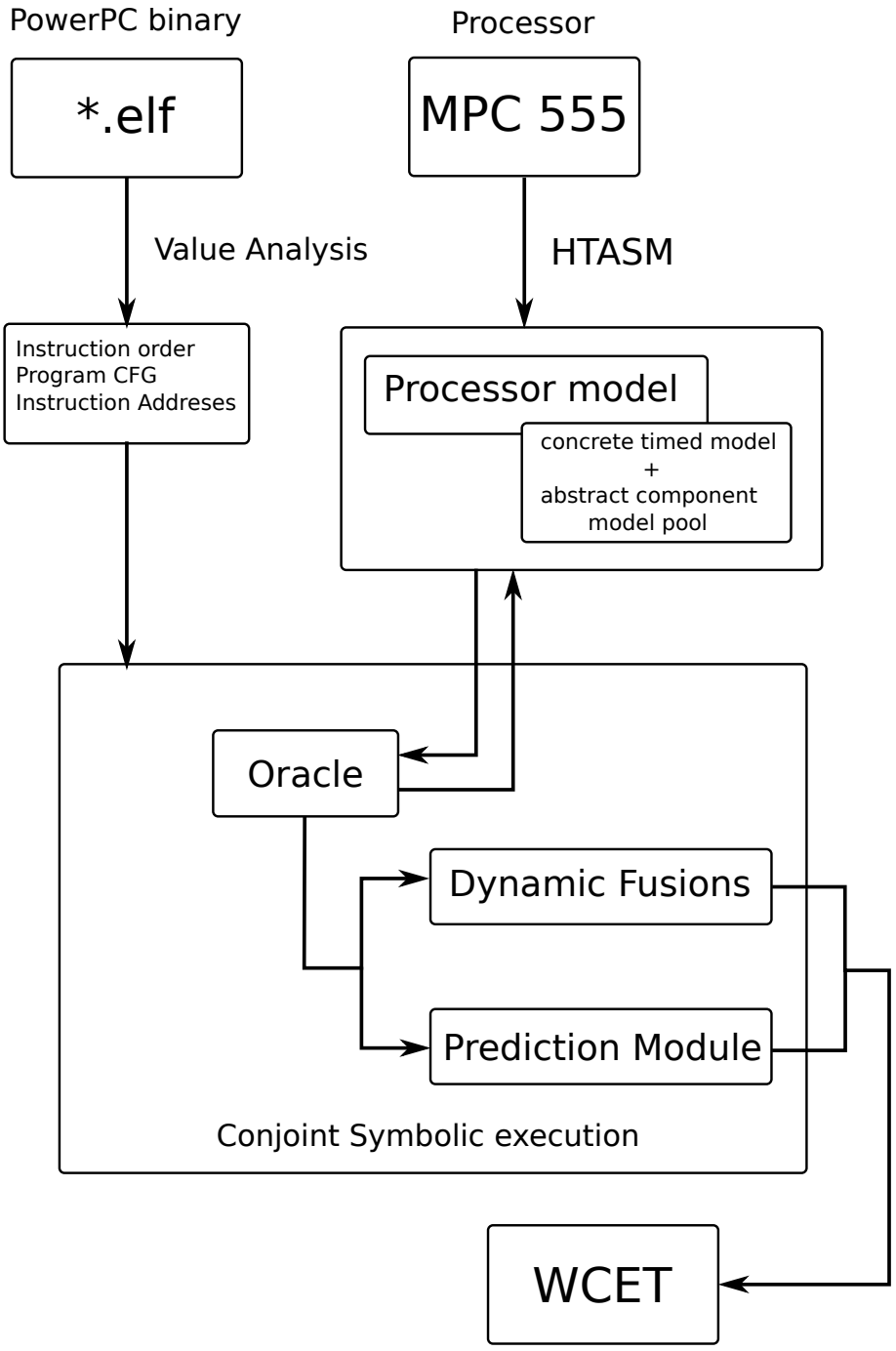


Figure 6.1: Global architecture of the WCET estimation tool

6.1 Modeling a Processor

A good processor model will make the difference in the timing analysis of modern hardware. Some key aspects that will prove to be important for this model are the following:

- perform lower level simulation than the instruction-level (cycle or time-acurate);
- capture all the timing properties of the hardware;
- capture the timing anomalies of the out-of-order execution;
- enable abstractions;
- clear and simple syntax.

6.1.1 Inherent analysis problems to the use of microprocessors in hard real-time systems

The hardware platform is a central point when analyzing a system. Therefore it is essential to dispose of a precise model of the processor in order to determine its effective behavior. The available information comes mainly in reference manuals and application notes that present the processor's architecture, how to interface it with the environment and how to configure its different function modes. Nevertheless, information present in the user manual is not intended for testing or verification purposes. Furthermore information relevant to the design method and the verification methodology are only briefly discussed, if not at all in these documents, mainly from the integrator point of view. Another issue is the questionable validity of the information presented in the reference document altogether as contradictory information is sometimes provided in different document.
The behavior of a microprocessor is challenging or even impossible to characterize. This is either due to the uncertainty of the effectively calculated result or the uncertainty on the actual time of the effective calculation. De facto, these two aspects are directly related to the notion of data availability.
The main consequence of the difficulty in architectural optimization is an interdependence

between the data and the instructions of a same task. In the context of multitask applications, an interdependence between various tasks coming from the commutation of the environments during the passing from one task to another, is introduced.

6.2 Hardware and its influence on temporal analysis

The majority of available processors was not especially designed for the real-time systems with safety-critical constraints, predictability coming after power consumption and mean-performance. Data communication and synchronization between the different units are optimized for maximum throughput of executed instructions. Therefore multiple execution paths can be taken depending on the execution history, the current state of units or even local choices based on random decisions. A natural way of analyzing the hardware components that influence the determinism would be to first look at those who give a local impact and proceed to components that have a global impact. One can also start by looking into local predictability effects and then into global effects. However, as shaped by this section, there is a thin frontier between the two as even classical, predictable units have a certain impact through interactions with more complex units. Therefore the analysis of such components can not be made isolated from the others.

6.2.1 Pipeline

Present in all modern processors, the pipeline was introduced in order to increase the average performance by ensuring that, whenever possible, an available hardware resource will be occupied. Nevertheless, different events can introduce pipeline stalls such as structural hazards, data hazards and control hazards.
Out of order execution (OoOE), a feature introduced in order to avoid pipeline stalls by decoupling the issue/dispatch and the execution/completion stages, allows an execution not following the instructions program order. A fetched instruction will be executed when the input operands and needed resources are available with no regard to whether it is the next in order instruction. The interaction between the cache memory and instruction scheduling influences the precision of the timing estimation.

Pipeline impact on the predictability

The impact of the pipeline varies from local influence with local monotonic optimizations to global influences with timing anomalies that cancel the monotonicity and compositionality. The size of the pipeline has an influence on the predictability of the WCET. A wrong branch prediction causes n cycles penalty, where n is the pipeline depth. The pipeline depth can further influence the predictability potentially generating more hazards as more instructions are being treated at the same time.
Besides intrinsic impacts on the predictability, the pipeline, in conjunction with other units, can lead to precision loss or nondeterminism. For example, in case of a L2 cache miss, the number of pipeline stages influences the memory access time [DSW95].

6.2.2 Branch Prediction Unit (BPU)

Through the BPU, processor attempts an early resolve of a branching instruction, before its time, by applying a strategy in order to anticipate the result. The BPU strategy can be either static or based on complex algorithms, un-deterministic in some cases. Based on this estimation, a speculative execution is initiated that will lead eventually to a significant time gain in the case the result is correct. The influence on the cache memory content is non negligible as a miss-prediction is not generally followed by a cache reorganization, therefore the cache configuration is polluted with information from the untaken path.

BPU impact on the predictability

The BPU can make incorrect branch predictions or incorrect branch target address lookup. It is systematically active and directly impacts the temporality of the instruction change. Furthermore, this unit relies on a set of data protection tables stored in tables. The impact is high because of the general unpredictable success rate of the early branching target resolution. It can be largely avoided in the case of statically resolved loops or branches that are not data dependent. Tailoring the condition of the jump taking into account the branching strategy in order to help it succeed in the majority of cases is also a solution as long as the WCET analyzer can take it into account. In this case, most techniques of adding

watermarks in the code with information that help or enable the prediction can be useful.

6.2.3 Floating Point Unit (FPU)

Floating point computation timing can also be hard to accurately estimate because of their implementation. A micro-pipelined unit takes advantage of consecutive instructions that can be pipelined. Units can have either a part of the FPU instructions pipelined or all of them. Therefore consecutive pipelined and non-pipelined instructions can cause stalls, making the timing difficult to compute especially in the case of the out of order execution. Floating-point data formats and instruction set generally conform to the IEEE Standard for Binary Floating-point Arithmetic, ANSI/IEEE Standard 754-1985.

FPU impact on the predictability

The impact of the FPU depends on its implementation. Instructions can take either a single cycle to execute or several cycles but they can also be pipelined. A combination of either way in parallel is also possible. In conjunction with the instruction rescheduling and thus with the change of data, cascade effects can occur and lead to pathological effects like it can be seen in some PowerPC architectures.

6.2.4 Level 1 Cache

Memories for instructions and data are implemented in order to make the most common case fast, benefiting from a program's spatial locality and temporal locality. Not taking this fact into account in the WCET estimation gives highly pessimistic timing estimations. Cache memory is usually organized in different levels, some local to the core and others situated outside the core. Different cache replacement strategies must be implemented in order to optimize the performance because a strategy that can fit all the possible cases is impossible to find. Therefore the average case performance is optimized. Commonly used strategies are LRU, pseudo LRU, FIFO or round robin and MRU each having a different impact on the predictability of the system.

The analysis of the Level 1 cache must be made in conjunction with the other units and is discussed within the timing anomalies in the following.

Impact on the predictability

Worst-case analysis on cache memories is a challenging problem, mainly because they are conceived in order to maximize the average performance. Achieving good results for data cache analysis, for example, is still an open problem, as they are difficult to statically analyze. An approach that enables time-predictable caching, is to lock cache blocks. Combining cache locking with cache partitioning for multiple tasks in the case of task preemption can improve the predictability in some cases [VLX03].
Unknown abstract cache states during the analysis generate loose WCET bounds. For example, unified cache for instruction and data can break down all the information on abstract cache states. After accessing n unknown addresses in an n-way set-associative cache all the cache lines will be unclassified in the analysis. Therefore, separation between the instruction and data cache memories should be chosen whenever possible (the problem still holds for shared caches and is discussed in the Level 2 cache section). For this reason Harvard architectures, with physically separate signal pathways for instructions and data, should be privileged despite of the von Neumann architecture. Context switch or cache misses can lead to a relatively high global impact due to timing anomalies or Translation Lookaside Buffer (TLB) strategies. In order to improve the performances of the cache memory, instruction and data locality could be increased using compiler techniques for example (code reposition, loop permutation, tiling [WL91] etc.). When performing the WCET analysis, the most problematic features to analyze are the replacement policies for set-associative caches [**?**]. Pseudo-round-robin and the 4-way associative cache is also a difficult combination in the Motorola ColdFire 5307 [Sch09]. In order to ensure the time-predictability of processors, locally deterministic update strategies for caches should be used. According to [RGBW07] the LRU strategy performs best in terms of predictability, far ahead of pseudo-LRU and FIFO.

6.2.5 Scratchpad

Scratchpad memories (SPMs) are used to guarantee a unit can work without main memory contention in a system employing multiple processors. As the memory access latencies are predictable, scratchpad memories have become popular for real-time embedded systems. However, the difficulty of allocating code/data to scratchpad memory lies now with the compiler. Scratchpad memory works like a local store and acts like ”software caches” therefore the strategy is implemented in software and the interactions in the global hardware must be analyzed. Timing anomalies with regard to the replacement strategy should be integrated into the hardware model. The most convenient approach to manage the SPM is using static allocation [**?**] but dynamic SPM allocation is more efficient (it can use profile-based optimization but multiple strategies exist). Analyzing dynamic strategies is challenging, especially the software implemented ones that give optimal allocation for the average execution time. Some WCET-centric techniques exist but they do not handle all architectures.

6.2.6 Memory Management Unit (MMU) and Translation Lookaside Buffer

The TLB is a cache that MMU uses to improve virtual addresses translation speeds. The time needed to determine the physical address depends on the number of performed operations. TLB time access is variable. In order to enforce the predictability, the MMU can be deactivated (however the performance loss is significant) or by reducing the size and thus complexity of the TLB (the TLB main entries can be blocked in order to ensure their persistence).
A solution is to increase the TLB size so that we only have hits but we still have the problem of an error in the translation that is detected late and takes an undefined (even if still reasonable) time to be corrected.
Typical user manuals [Aer12] give upper bounds in the TLB miss case. Timing anomalies invalidate the monotonicity assumption in the general case [WKPR05a], which means that we cannot directly use the upper-bound information as the worst-case scenario. There-

fore, without precise information on the exact behavior all possible cases must be analyzed, leading to a potential state space explosion. In order to reduce the potential temporal variability, the MMU should be disabled. Nevertheless due to the consequences on the global performances it is not recommendable.
In general, virtual memory raises predictability issues at two levels. First at the level of address translation that provides mapping between virtual to physical pages requires a TLB lookup. If the mapping is absent from the TLB a page table lookup is performed. The duration of address translation is hard-to-predict, because not all mappings can be stored in the limited capacity of the TLB or because the TLB might be shared between concurrent processes. Second at the level of paging activity as knowing whether or not a reference to a virtual page will result in a page fault. This is hard to predict because physical memory is shared between concurrent processes.

Impact on the predictability

Translation of virtual addresses to physical addresses is a large source of execution time variance. In large software systems virtual addressing is used by the applications to allow the object code to be relocatable.
Interrupts make the behavior of the TLB even more complicated to predict. When an interrupt occurs, there is an initial penalty for executing the service routine because none of its references hit in the TLB therefore all the references must be translated, [WD95].
It is the probabilistic nature of the TLB that adds significant uncertainty to the execution time of an application because such delays must be accounted for on every memory reference. For a hard real-time deadline, a worst case of three memory reference times must be assumed for every access because it is nearly impossible to predict whether an entry will be in the TLB when it is needed.

6.2.7 BUS

As competition for resources grows, the natural solution was to use techniques that enable the access from master to slave, and utilization of shared resources in general. Through the use of switching mechanism, permission is granted to one master or the other, which

introduces the need of a bus arbiter. Therefore a controller is usually implemented following different strategies that are more or less straightforward. The first difficulty comes from the implementation of the aforementioned strategies.

Impact on the predictability Because the main role is to grant access to different participants to the shared resources, certain properties must be ensured (fairness, deadlocks prevention) while still being able to ensure good average performances and timing predictability. Therefore when choosing a particular architecture for the hard real-time systems, certain bus architectures and arbitration algorithms should be privileged.
Nevertheless, the maximum length of a burst for each peripheral connected to the bus influences the maximum delay induced by bus contention. This may lead to high maximum delay bounds and may not be enough to provide firm real time guarantees for heavily loaded systems. Furthermore, having variable burst lengths, combined with the ability to pause them (split transfers), influences the predictability of the WCET. Bus contention can be avoided by using the TDMA bus arbitration with the cost of wasting bus band when the bus load is low. By manipulating the TDMA time slots, the maximum delay bound on transactions is controllable by the designer.

6.2.8 Direct Memory Access (DMA)

DMA allows access to the system memory independently from the CPU. Therefore the processor can proceed with its computations while waiting for relatively slow input/output data transfer.

Error handling

The DMA controller does not generally detect deadlocks in its communication channels, so it is up to the system to manually abort the DMA transfer. The DMA unit can be disabled not without a strong impact on the performances of certain class of applications.

6.2.9 Level 2 cache

Level 2 cache memory can be either private to each core or shared among cores. Timing anomalies render the result hard to predict like in the case when a cache miss from a core can reconfigure the memory in a state that is, timing wise, beneficial to the other. This also applies in other cache related scenarios. The problems that can occur in the analysis of the interactions with the pipeline are detailed in the timing anomalies part.

Level 2 cache impact on the predictability

The impact of the shared cache is high and global and gets amplified in the context switch case. Modeling the behavior of shared caches between cores is practically impossible because of the possible interactions between concurrent threads running on different cores [Sch09]. Under the assumption that the bus strategy can be statically analyzed, the second level of cache can be made predictable by partitioning the L2 cache for each core [SHP12].

6.2.10 Timing Anomalies remarks

Several types of timing anomalies exist. Some are inherent to instruction execution order and are generally caused by greedy scheduler that will change the instruction execution order causing inversion or amplification of the execution time difference. Others are caused by parallel decomposition and *divide et impera* approaches to WCET estimations. As the first ones cannot be avoided, the others may prove essential for the possibility to construct an efficient processor behaviour analysis that does not need to search the whole state space for the whole program at once.

6.3 The RISC processor Family

Introduced in the late 70s, the Reduced Instruction Set Computer (RISC) is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, as

opposed to a specialized set of instructions often found in other types of architectures. Processors from the RISC family generally share the following three features:

- *one cycle execution time*: RISC processors have a CPI (clock per instruction) of one cycle;
- *pipelining*: a technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
- *large number of registers*: the RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

As opposed to the CISC processor family, which emphasis the use of complex instructions in order to complete a task in as few lines of assembly as possible, RISC processors only use simple instructions that can be executed within one clock cycle.
RISC processors do not operate directly on the computer's memory banks, like CISC processors do, and requires the programmer to explicitly call loading and storing functions.

Definition 56 (The Performance Equation). In order to express a computer's performance ability the following equation is generally used:

$$\frac{time}{program} = \frac{time}{cycle} \times \frac{cycles}{instruction} \times \frac{instructions}{program} \tag{6.1}$$

The CISC focuses on minimising the number of instructions per program, sacrificing the number of cycles per instruction. The RISC approach does the contrary, reducing the cycles per instruction at the cost of the number of instructions per program.

6.4 Case study - Motorola MPC555 Processor

The PowerPC-based RISC processor (RCPU) used in the MPC500 family of microcontrollers integrates five independent execution units: an integer unit (IU), a load/ store unit (LSU), and a branch processing unit (BPU), floating-point unit (FPU) and integer multiplier divider (IMD), [Fre00]. MPC555 fully implements the PPC Instruction Set Architecture.

6.4.1 PowerPC ISA

PowerPC is an acronym for Performance Optimization With Enhanced RISC - Performance Computing, (also abbreviated as PPC) created by the 1991 Apple-IBM-Motorola alliance, known as AIM.
The PowerPC is designed along RISC principles, and allows for a superscalar implementation. Versions of the design exist in both 32-bit and 64-bit implementations.
The processor implements the instruction set, the storage model, and other facilities defined in this document. Instructions that the processor can execute fall into three classes:

- branch instructions
- fixed-point instructions
- floating-point instructions

Instructions used in the PowerPC Architecture are four bytes long and word-aligned.
The architecture provides for byte, halfword, word, and doubleword operand fetches and stores between storage and a set of 32 General Purpose Registers (GPRs). It also provides for word and doubleword operand fetches and stores between storage and a set of 32 Floating-Point Registers (FPRs). Signed integers are represented in two's complement form.
No computational instructions modifies the storage, [IBM05]. The contents of the storage operand must be loaded into a register, modified, and then stored back to the target location in order to use a storage operand in a computation and then modify the same or another storage location. figure 6.2 shows a logical representation of instruction processing.

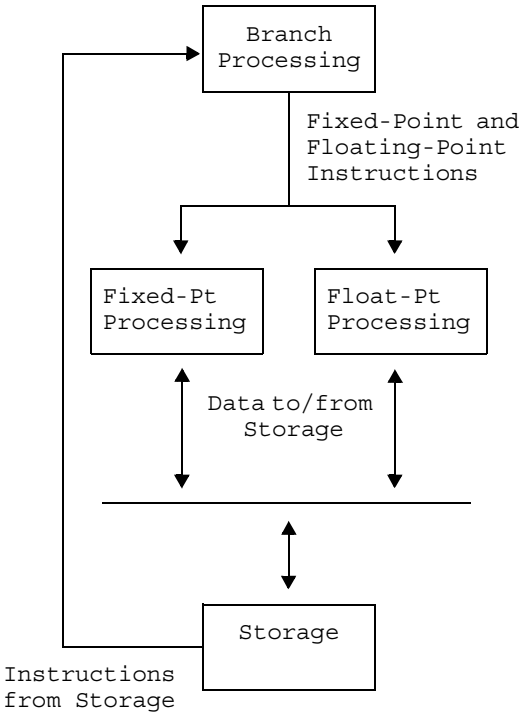


Figure 6.2: Logical Processing Mode

6.4.1.1 Instruction formats

All instructions are four bytes long and word-aligned. The opcode is systematically specified on bits 0:5 (OPCD). Many instructions feature an extended opcode (XO). The remaining bits of the instruction contain one or more fields with different operands or other parameters.

Some of the defined instructions have preferred forms that will execute in an efficient manner, but any other form may take significantly longer to execute than the preferred form. Instructions having preferred forms are:

- the *Condition Register Logical* instructions
- the *Load/Store Multiple* instructions
- the *Load/Store String* instructions
- the *Or* Immediate instruction (preferred form of no-op)

- the Move To Condition Register Fields instruction

Some instructions use virtual addressing therefore an effective address must be calculated.

6.4.2 Global architecture of the MPC555

The MPC555 is an out-of-order, single-issue, Harvard architecture design. A single instruction is issued per cycle to one of the five independent execution units that can execute out-of-order. The branch prediction unit ensures a high instruction throughput through by means of branch prediction. Updates of the processor registers are recorded in program order in the history queue. Most integer instructions execute in one clock cycle. Instructions can complete out of order for increased performance; however, the processor makes execution appear sequential through the use of the completion unit. Here are some general features of the processor:

- PowerPC core with floating-point unit
- 26 Kbytes fast RAM and 6 Kbytes TPU microcode RAM
- 448 Kbytes flash EEPROM with 5-V programming
- 5-V I/O system
- Serial system: queued serial multi-channel module (QSMCM), dual CAN 2.0B controller modules (TouCANTM)
- 50-channel timer system: dual time processor units (TPU3), modular I/O system (MIOS1)
- 32 analog inputs: dual queued analog-to-digital converters (QADC64)
- Submicron HCMOS (CDR1) technology
- 272-pin plastic ball grid array (PBGA) packaging
- 40-MHz operation, -40¡ C to 125¡ C with dual supply (3.3 V, 5 V)

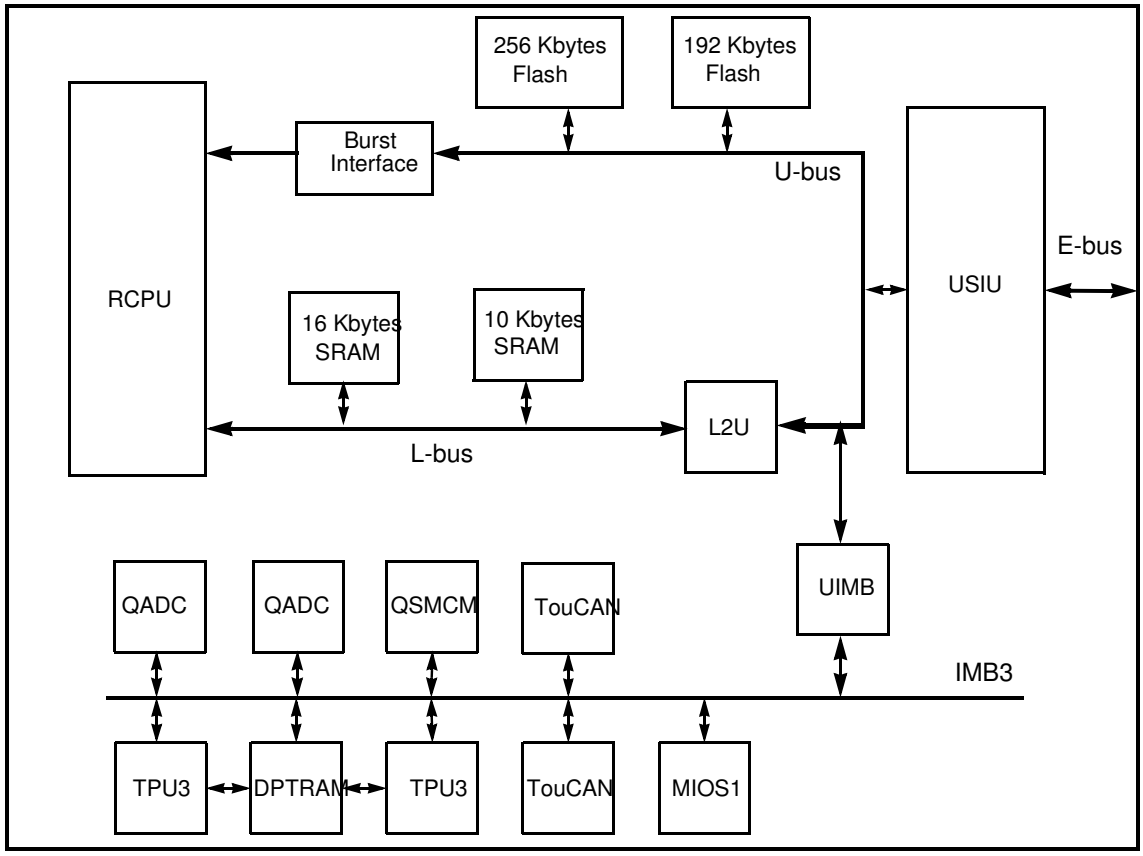


Figure 6.3: Motorola MPC555 view from the user manual

- MPC556 supports code compression to increase code density.

Major features of the RCPU include the following:

- High-performance microprocessor - Single clock-cycle execution for many instructions
- Five independent execution units and two register files
  - Independent LSU for load and store operations
  - BPU featuring static branch prediction
  - A 32-bit IU
  - Fully IEEE 754-compliant FPU for both single- and double-precision operations

  - Thirty-two general-purpose registers (GPRs) for integer operands
  - Thirty-two floating-point registers (FPRs) for single- or double-precision operands

- Facilities for enhanced system performance
  - Programmable big-and little-endian byte ordering
  - Atomic memory references

- In-system testability and debugging features

- High instruction and data throughput
  - Condition register (CR) look-ahead operations performed by BPU
  - Branch-folding capability during execution (zero-cycle branch execution time)
  - Programmable static branch prediction on unresolved conditional branches
  - A pre-fetch queue that can hold up to four instructions, providing look-ahead capability
  - Interlocked pipelines with feed-forwarding that control data dependencies in hardware

6.4.3 Instruction Sequencer

The instruction sequencer (IS) implements the pipeline behavior, controlling the flow from the memory to the execution units and the other way around. As we can notice in figure 6.5, the fetch initiated by the IS gets instructions, through the instruction data path, from the *Instruction Memory System* and stores them in the *Instruction Buffer*. What we will see shortly is that MPC555 implements a Burst Buffer, therefore all access to the instruction data path are burstable.
Depending on the nature of the instruction in the Instruction Buffer one of the following three actions in performed:

- the instruction is stored in the *Instruction Pre-fetch Queue* (IPFQ);

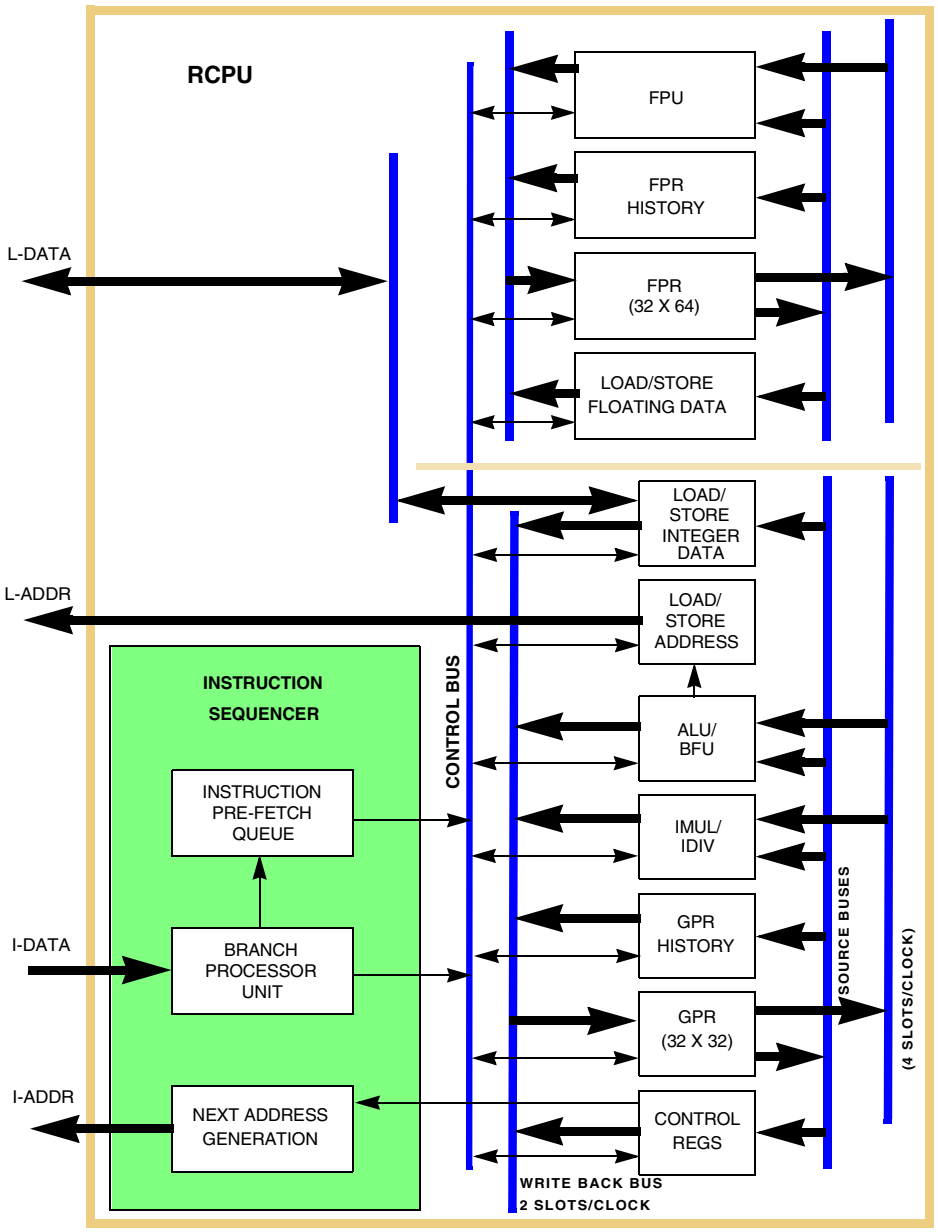


Figure 6.4: RCPU Block Diagram from the user manual

- the instruction is a branch instructions (the opcode gets verified) therefore the BPU gets activated and performs a Branch Condition Evaluation - in order to attempt an early resolve of the branching;

- in both cases the address of the next program instruction to be fetched must be de-

termined, therefore in case of a normal instruction, a simple shift is performed to the ”PC” in order to get the next linear instruction or, with the help of the branch prediction a new address is generated. The BPU tries to resolve the jump condition and the jump address. Depending on the result of the condition, on branch of the execution or the other will be performed. The BPU, based on a bit parity check decides what branch to take and therefore informs the IS to fetch addresses from the address corresponding to the presumed case.

Once an instruction arrives in the IPFQ, it is sequenced to the appropriate execution unit.
The link between the Instruction Address Generator and the Execution Units and Register Files area is explained by the fact that this unit uses arithmetic operation from the arithmetic unit and register to compute the effective *jump* address. This is not the case of the BPU who uses special, dedicated registers for its operations (LR, CTR and CR), therefore being able to execute branch instructions separately.
The CC Unit is also involved in the sequence data path and ensures that instructions issued beyond a predicted (therefore potentially incorrect) branch will not get completed (Completion Unit) until the branch is fully resolved. This mechanism is followed by a behavior that is very important for the timing analysis. If the prediction of the BPU is erroneous, all the instructions issued, and therefore executed, on that predicted branch will get flushed (therefore never completed, and therefore the results not written in the result locations) and the new fetch address (of the other branch) will be communicated.
If the branch folding succeeded, meaning that the prediction was correct, the the jump can be done automatically and some instructions are already executed and ready to be completed, which represents a non-negligible temporal gain compared to the case of a miss-prediction. Therefore the processor model and the timing analysis must take into account this behavior.

6.4.4 Execution Units

The MPC555 features independent execution units making it possible to implement advanced features such as look-ahead operations. Each unit decodes the dispatched instruction, by looking at its opcode, in order to immediately start the execution.

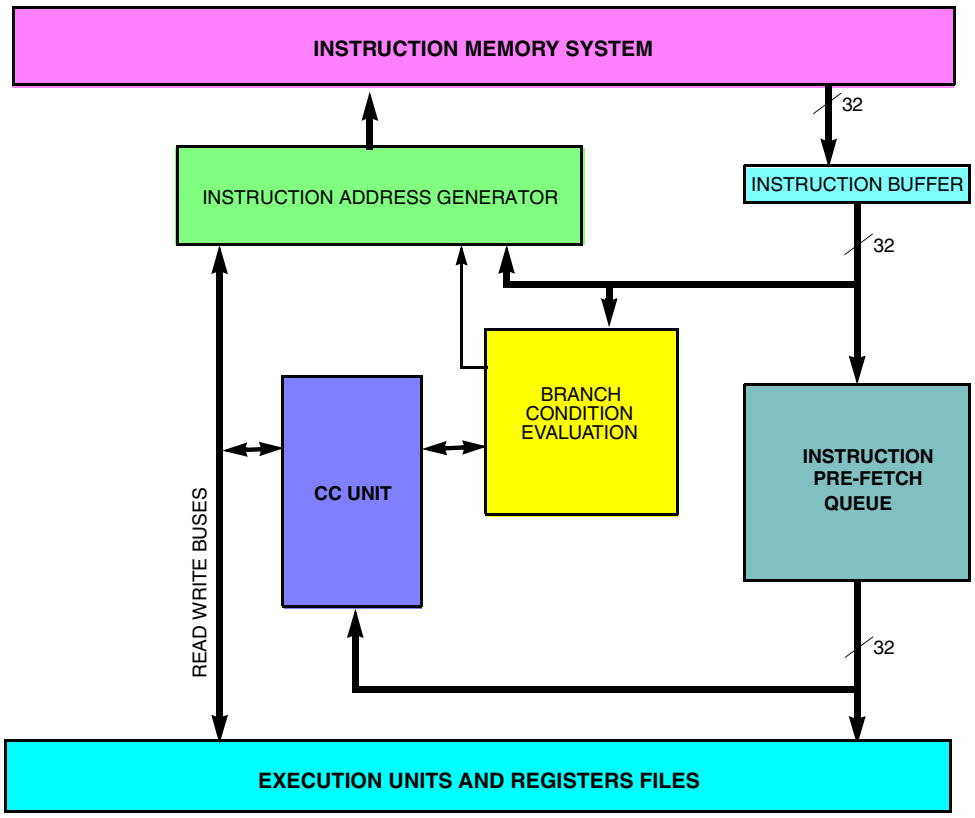


Figure 6.5: Sequencer Data Path from the manual

Branch Processing Unit (BPU) As we already presented in the previous section, the BPU is located in the Instruction Sequencer and looks for branch instructions in the Instruction Buffer. Once it identifies a branch instruction and in the case it is an unresolved conditional branch it informs the fetcher to prefetch instruction from the branch predicted using a bit in the instruction encoding. The goal is to achieve zero-cycle branching, ensuring a tangible speed-up.
The BPU integrates, besides a calculation feature to compute branch target addresses, three special-purpose registers:

- the link register (LR), used to store the calculated return pointer for subroutine calls and also the branch target address for the branch conditional to link register (**bclrx**) instruction,

- the count register (CTR), that contains the branch target address for the branch con-

ditional to count register (**bcctrx**) instruction,

- the condition register (CR).

6.4.5 Integer Unit (IU)

The IU executes the integer instructions except for the load and store instructions which are handled by the dedicated LSU unit (that handles also the floating-point load/store instructions).
The particularity of the unit is that it features two separate units:

- one for the addition/subtraction called ALU-BFU,
- another for the multiplication/division called IMUL-IDIV.

Regarding the timing of the instruction execution, ALU-BFU takes a single clock and IMUL-IDIV instructions require multiple clock cycles. Another key feature is that the ALU-BFU unit is pipelined but only for the multiplication instructions. This means that if a division instruction is intercalated with additions or multiplications, a stall is introduced in the pipeline, like we can see in the example in listing **??**.

Listing 6.1: No stall IU instructions

```
1 addic r1, r2, 1
2 mulli r6, r4, 3
3 subf r3, r5, r3
```

The execution of the first example in listing 6.1 is shown in figure 6.6. As we can see, even though the write back of the multiplication instruction is delayed, there is no bubble in the execution stream.

Listing 6.2: No stall IU instructions

```
1 addic r1, r2, 1
2 mulli r6, r4, 3
3 mulli r1, r2, 4
```

Listing 6.3: Stall IU instructions

```
1 addic r1, r2, 1
2 divw r6, r4, r3
3 mulli r1, r2, 4
```

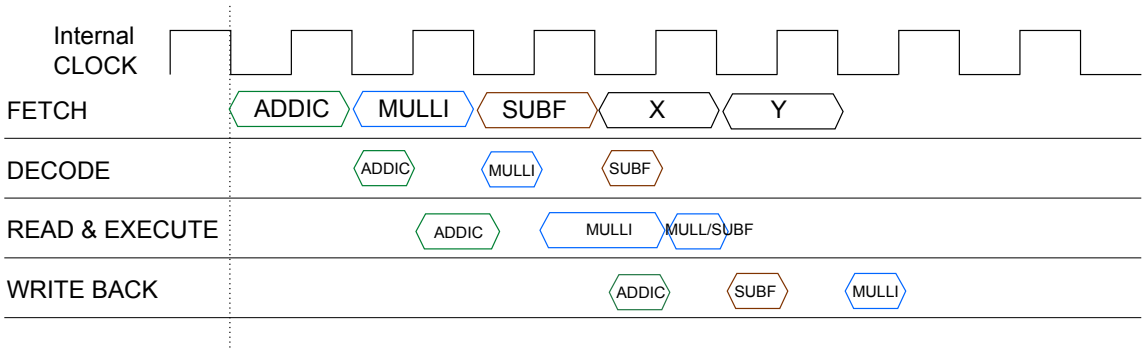


Figure 6.6: Pipelined execution of addition and multiplication intructions

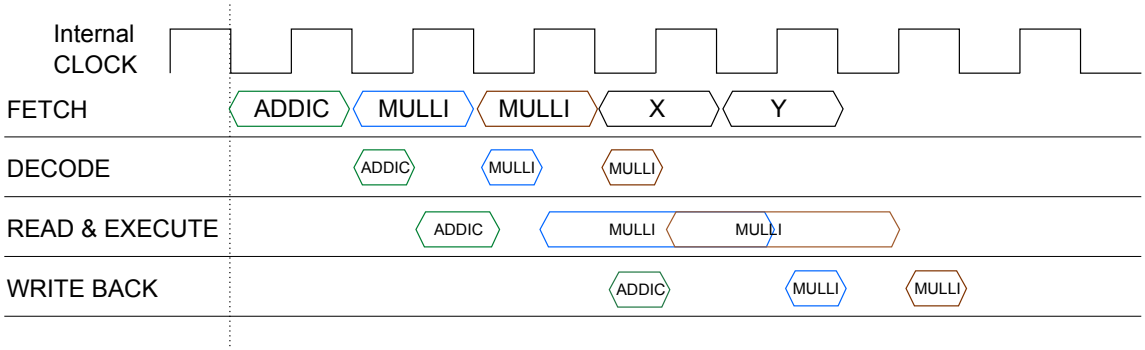


Figure 6.7: Micro-pipelineing of the multiplication instructions by the IMUL-IDIV unit

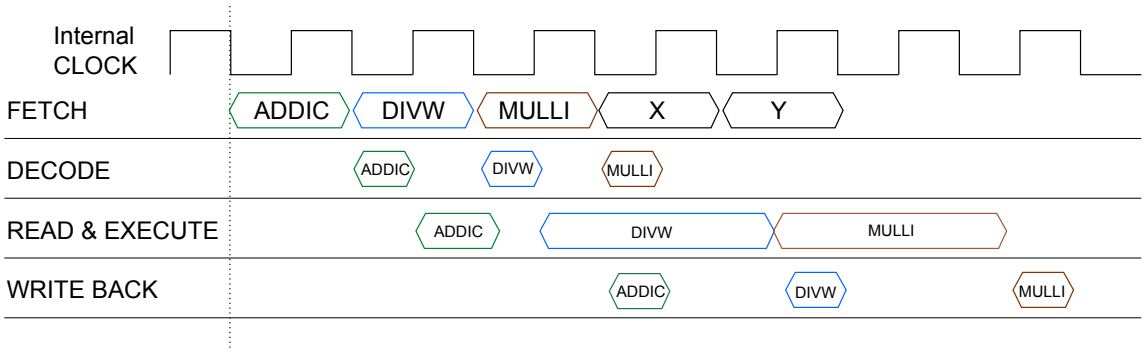


Figure 6.8: IMUL-IDIV micro-pipelineing stall

6.4.6 Load/Store Unit (LSU)

The LSU is in charge with the data transfer between the GPRs and the internal load/store bus (L-bus) through a 32 bits wide datapath. The unit is pipelined therefore stalls in the

memory pipeline do not cause the master instruction pipeline to stall (unless there is a data dependency).

Our model will exploit the timing information in table 6.9 that shows the LSU access latencies.

listing 6.4 shows the computation of the LSU addressing.

Acces size	Clock Cycle Latency	No of accesses per transfer
Single-word	2	1
Double-word	3	2

Figure 6.9: LSU access latencies

Listing 6.4: LSU address format

```
1 if imm(instr) then
2   addr := rA + immAddr(intr)
3 else
4   addr := rA + mem(rB)
```

6.4.7 Floating-Point Unit (FPU)

The Floating-Point Unit is organized in three major parts as follows:

- a double-precision multiply-add array - efficiently implement floating-point operations such as multiply, multiply-add, and divide,
- the floating-point status and control register (FPSCR),
- the FPRs.

One of the most important issues in the determination of the timing behavior of this units is with regards to the implementation of the IEEE floating-point specification. MPC555 depends on a software envelope to ensure the full implementation of the specification. However, this software envelope should never be activated in critical real-time systems because of the poor timing deterministic behavior. Instead, MPC555 can be forced to

deliver results in hardware, treating as legitimate denormalized numbers, NaNs, and IEEE invalid operations instead of raising floating-point assist exceptions.

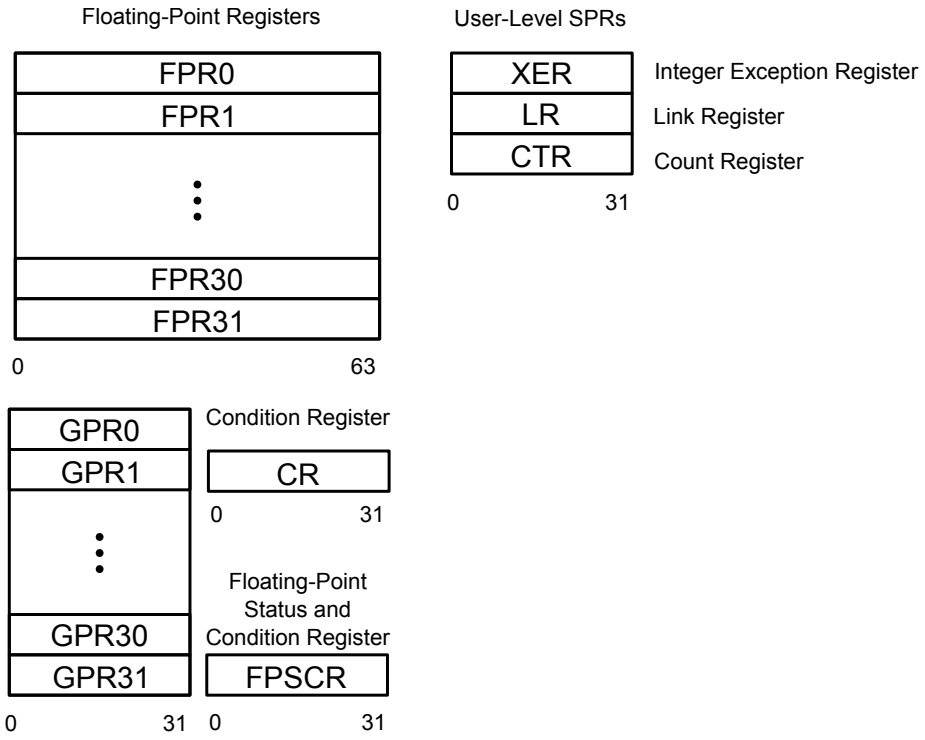


Figure 6.10: RCPU Programming Model - User Model UISA

6.4.8 External Bus Interface

The MPC555 bus is a synchronous, burstable bus. The MPC555 architecture supports byte, half-word, and word operands.

6.4.9 The RCPU HiTAsmL model

In the next sections we will use the following UML-style graphical syntax for the HiTAsmL constructs.

MPC555 HiTAsm main rule Let us now model the MPC555 RCPU. As we already stated, the HiTAsm framework enables incremental design through step-wise refinements.

The first step will be the creation of a new `htasm` and the definition of an abstract `hmodule` name, figure 6.11.

Listing 6.5: HiTAsmL rule

```
1 htasm MPC555
2   hmodules: MPC555module
```

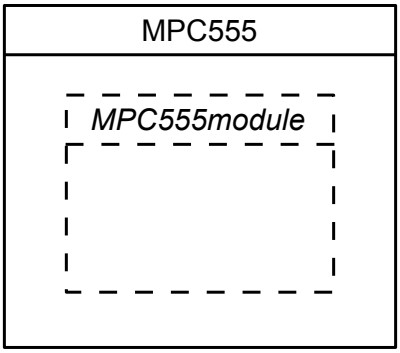


Figure 6.11: MPC555 htasm

The main abstract module is then refined in listing 6.6 and a concrete main module with a main rule is created as we can see in figure 6.12. This corresponds to the highest abstraction level of the processor model.

Listing 6.6: HiTAsmL rule

```
1 htasm MPC555
2   hmodule MPC555module
3     definition:
4       rule mainRule () , T
```

The model can be further refined with the delay information corresponding to the longest execution step of the processor between all the possible configurations. The squared head arrow represents a *concretization* of the definition.

Listing 6.7: HiTAsmL rule

```
1 module MPC555module#1 : MPC555module > MPC555module#1
2   definition:
3     rule mainRule () , maxdelay
```

We further refine the main module with the definition of seven abstract `hmodule` interface that correspond to the five execution units, the prefetch queue and the history buffer.

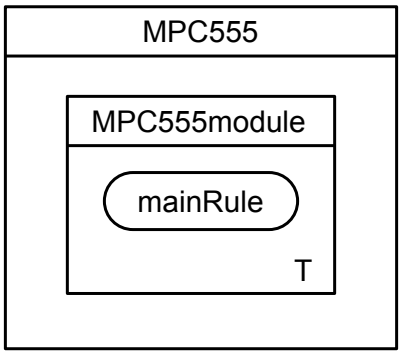


Figure 6.12: MPC555 hmodule

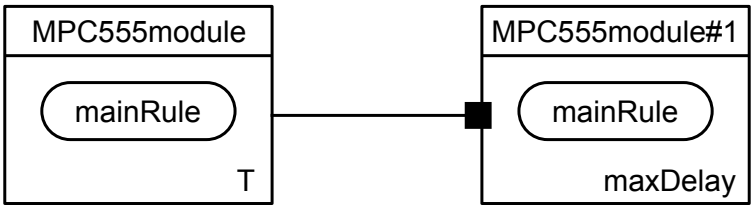


Figure 6.13: Abstract rule definition

Listing 6.8: HiTAsmL RCPU hmodules

```
1 htasm RCPU
2   hmodules: BPU, IPFQ, ALU, FPU, IMDU, LSU, HQ
```

Several refinement steps later, each of the defined interfaces is implemented following the instruction sequencer defined in section 6.4.3 on page 155. At this step each execution unit has a fairly high level design, featuring only generic rule names like *Execute* for each identified functionality. However we can already distinguish the instruction sequencer logic and the data path. Information like the buffer size or the number of history buffer entries are also visible in figure 6.15.

6.4.9.1 Memory model

We can notice in the current HiTAsmL implementation that we do not specify, at any moment, the underlying memory model. In the following we define the memory interpretation based on fundamental HiTAsm framework constructs. Let us consider the RCPU register presented in figure 6.10 on page 162.

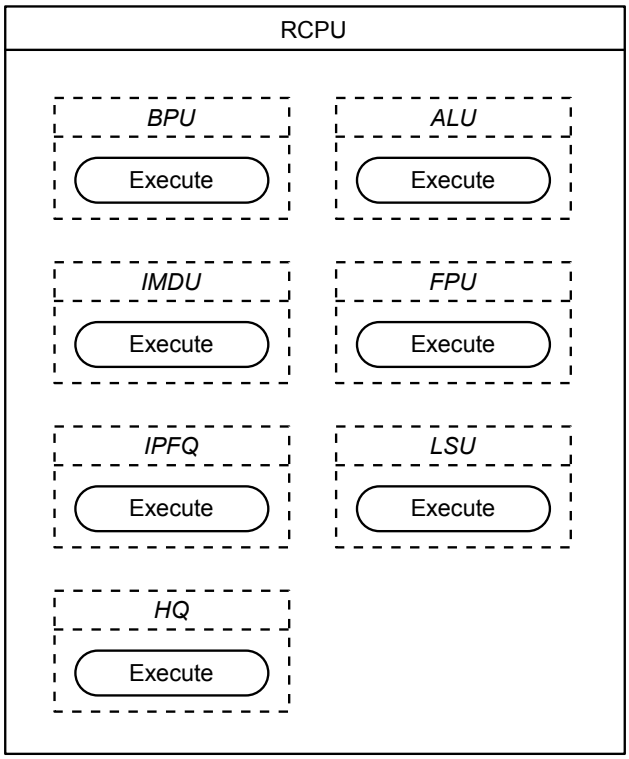


Figure 6.14: HiTAsmL RCPU hmodel

Listing 6.9: HiTAsmL RCPU hmodules

```
1  module GlobalMemory
2    signature :
3      IR -> Int
4      PC -> Int
5      GPR: Int -> Int //32 registers
6      FPR: Int -> Int //32 registers
7      CR -> Int //used by the BPU for look-ahead operations (BPU)
8      FPSCR :-> Int //floating-point status and control register
9      RAM: Int -> Int
10     IPFQ: Int -> Int //can hold up to 4 instr,
11           //provides look-ahead capability
12     //user-level Special Purpose Registers
13     CTR -> Int //count register (BPU - bcctrx instr)
14     LR -> Int //link register (BPU - bclrx instr)
15     XER -> Int //integer exception register (IU)
16     MSR -> Int
17
18     //BBU operation modes
19     enum BPUopMode =
20       {NO, // Normal Operation
21       SO, // Slave Operation
22       RO, // Reset Operation
```

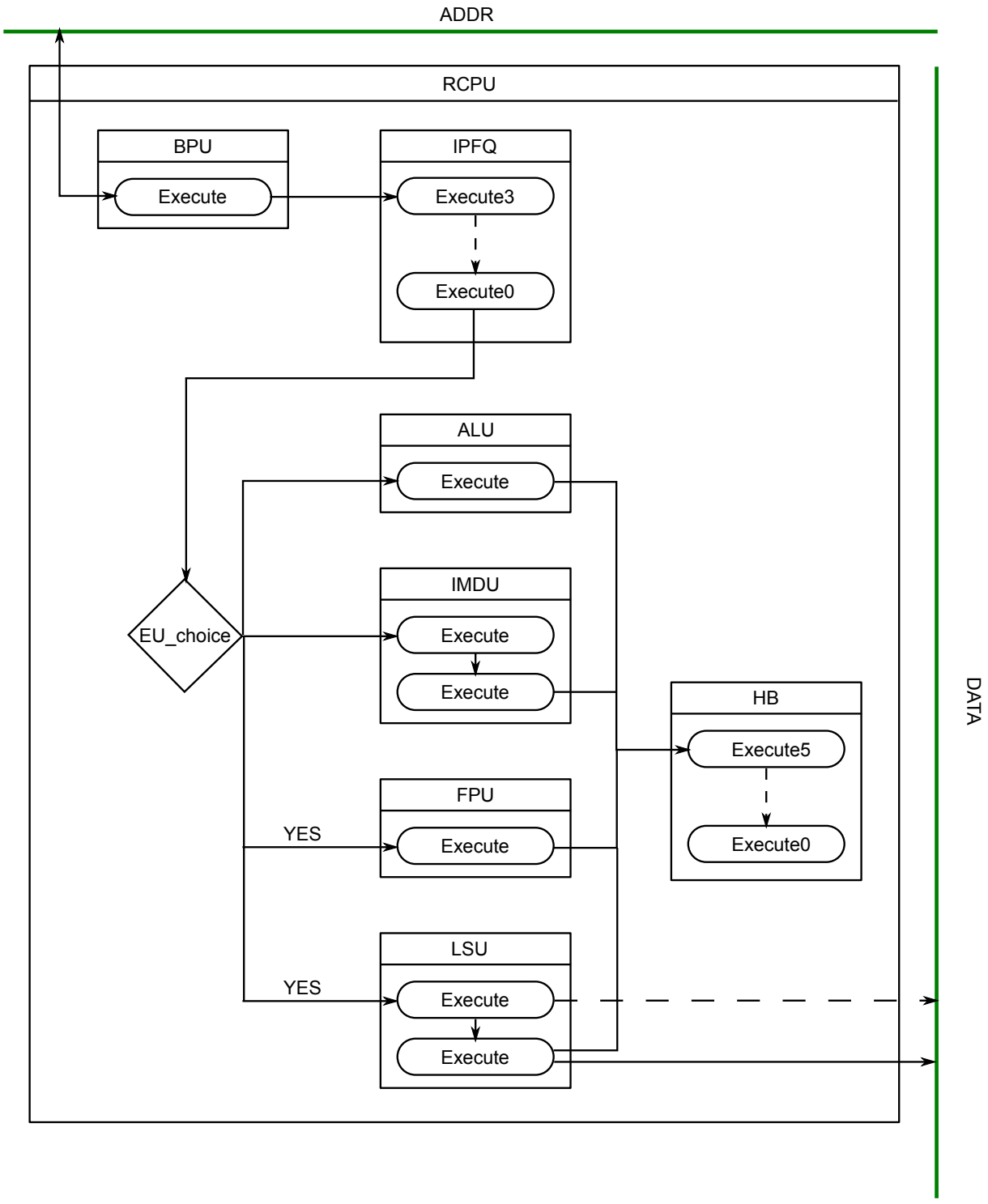


Figure 6.15: HiTAsmL RCPU model

```
23        DMO, //Debug Mode Operation
24        SMO, //Standby Mode Operation
25        Bo, //Burst Operation
26        ErrO //Error Detection
27        }
28      //IMPU variables
29      impuCancelRequest -> Int
30      MI_GRA -> Int // Global Region Attribute Register
31      BBCMCR -> Int //BBC Module Configuration Register
32      BE -> Int //Burst Enabled
33
34      //MPC55 Instruction Set
35      enum  ISA = {add, addi, mulli, subh, ...}
36      //auxiliary functions to get the unit type (handle WB)
37      enum  IUNIT = {ALUBFU, IMULIDIV, FPU, LSID, LSFD,LSADDR}
38      function ALUBFUinstr : Int -> IUNIT
39
40      //maps an opcode to a IS name
41      function RCPU_ISA : Int -> IS
42      //the instruction fields name
43      enum IF = {IF_AA, IF_crbA, IF_crbB, IF_BD,
44        IF_crbD, IF_crfS, IF_BI, IF_BO, ..., IF_rA,
45        IF_rB, IF_Rc, IF_rS, IF_rD, IF_SH, IF_SIMM}
46      //decoded instruction arguments
47      IARG : Int * IF -> Int
48
49      //auxiliary pipeline register
50      ALUexecResReg -> Int
51      FPUexecResReg -> Int
52      MULexecResReg -> Int
53      MEMResReg -> Int
```

Let us examine the GPR and the memory access to those registers whose HiTAsmL implementation is shown in listing 6.10. We use the function names GPR0-GPR31 to distinguish the 32 GPRs and store their particular addresses in the MPC555's memory (UISA area). From a practical point of view we choose to refine this HiTAsmL function to a static unary function `GPR: Int -> Int` that will map the number of the register with its address. This function is static because the addresses of the registers are fixed in the processor's architecture.

In order to implement write and read accesses on memory content, we define a function called `MEM : Int -> Int` that maps memory addresses to their value. This function is obviously dynamic and can also have an initial state. As a memory use example we defined the rule `writeToGPR0(data: Int)` that will write the value *data* to GPR0 in one clock cycle, $\delta = 1$.

Listing 6.10: HiTAsmL RCPU hmodules

```
module GlobalMemory
  signature :
    static GPR: Int -> Int //32 registers
    dynamic MEM : Int -> Int
  definition :
    function GPR(r : Int) =
      initiall{
        [0] -> 0x2F C000
        [1] -> 0x2F C001
        ...
        [31] -> 0x2F C020
      }
    function MEM(gpr : Int) =
      initial{
        [GPR(gpr)] -> 0x0000
        ...
      }
    rule writeToGPR0(data : Int) =
      MEM(GPR(0)) := data, 1
```

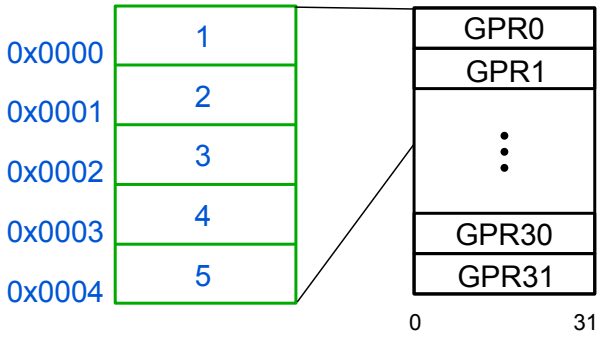


Figure 6.16: HiTAsmL RCPU registers

In figure 6.19 we refine the RCPU model based on the following informations:

1. the IPFQ is a buffer that stores a maximum of four instructions that are accessed in a FIFO manner;

2. the IPFQ dispatches an instruction per cycle to available units;

3. each execution units decodes the dispatched instruction and executes it - therefore each model of the execution units are refined to test the issued instruction;

4. if the instruction is not a valid PPC ISA instruction an exception is raised.

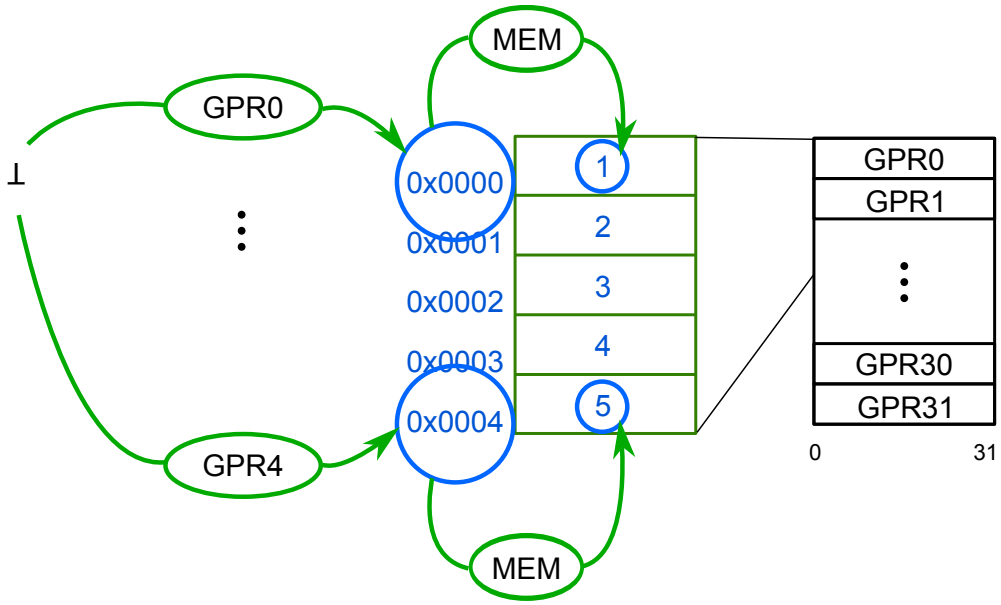


Figure 6.17: HiTAsmL RCPU registers

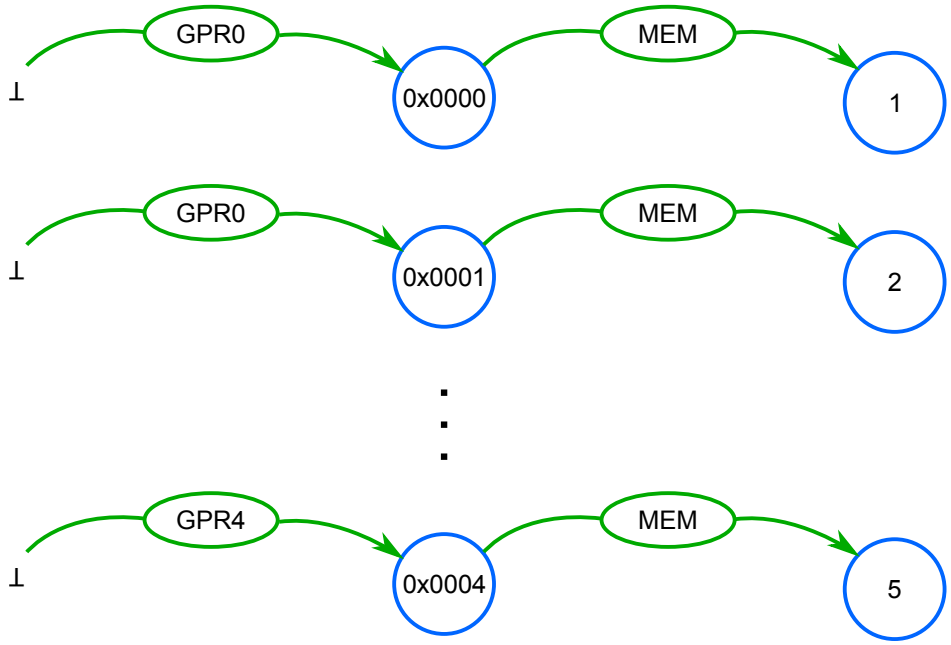


Figure 6.18: HiTAsmL RCPU memory model

In listing 6.11 we give the definition of the refinement with some extra support funcitons.

Listing 6.11: Refined HiTAsmL RCPU hmodules

```
htasm RCPU
  hmodules: BPU, IPFQ, ALU, FPU, IMDU, LSU, HQ
  hmodule BPU
    import FIFOwriteIPFQ
    rule Execute(instr : Int)
      FIFOwriteIPFQ(instr)
  hmodule IPFQ
    signature:
      Q: Int -> Int
    rule Execute(intr : Int)
      FIFOwriteIPFQ(instr) =
    rule FIFOwriteIPFQ(instr) =
      Q[3] : = instr
      Q[0:2] : = Q[1:3]
  hmodule ALU
    rule Execute()
    rule decode(instr: Int) =
      if opcode(instr) then
        Execute()
  hmodule FPU
  ...
```

6.4.9.2 The Fetcher

Listing 6.12: HiTAsmL syntax for abstraction level definition

```
htasm MPC555
    hmodules IPFQ, Fetch, Decode, Execute, WriteBack //...
    import Fetcher1
hmodule Fetch
module Fetcher1 : Fetch > Fetch
module Fetcher2 : Fetch > Fetcher1
module Fetch0 : Fetch < Fetch
```

6.4.9.3 MPC555 pipeline implementation

When implementing a pipelined processor, several design methods are available for the HiTAsmL user. One could divide the code according to the instruction name. Besides specifying the instruction being currently treated, each function could have a *stage* parameter that would specify the current pipeline stage of the instruction and have treatments depending on it.

Listing 6.13: HiTAsmL example of the MPC555 processor

```
enum PipelineStage = {ID, EX, MEM, WB}
```

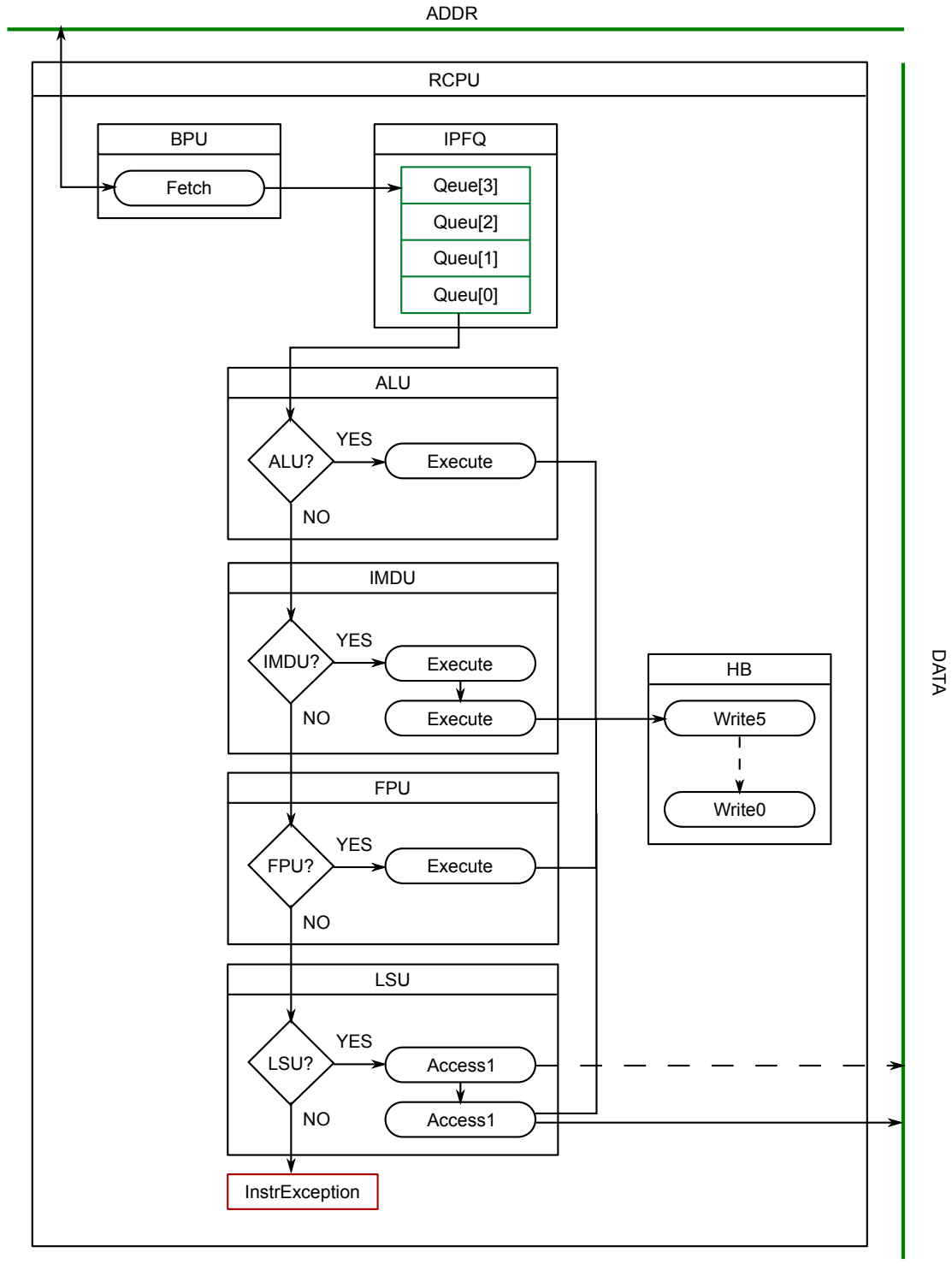


Figure 6.19: Refined HiTAsmL RCPU model

```
2 rule add(instr: int, stage : PipelineStage) =
3   if stage = ID then
4     ...
5   if stage = EX then
6     ...
```

Instead one could group all the treatments related to the instruction pipelineing into different stages rules.

6.4.9.4 Instruction Issue

The RISC Central Processing Unit implemented in the MPC555 processor attempts to issue a sequential instruction on each clock if possible. The conditions necessary for an instruction to be issued specifies that

- the execution unit must be available
- the required source data must be available
- no other instruction still in execution targets the same destination register

The particularity of the processor is that after the sequencer broadcasts the presence of the instruction on the instruction bus, each execution unit decodes the instruction. The execution unit is also responsible for the determining of the aforementioned dependencies.

Listing 6.14: HiTAsmL example of the Intruction Sequencer

```
1  module IntructionSequencer
2    import GlobalMemory
3
4    definition :
5      rule IS() =
6        //fetch instructions from the memory system
7        //fetches instructions from the BBC into the IPFQ
8        //BPU extract from IPFQ
9        //issues instr to available execution units
10       //maintain state history
11       //writeback -
12       //retirement stage - up to 6 instr per clock
13
```

6.4.9.5 Pipeline stalls and forwarding

Instructions hold either the register value for the operand or the immediate value. Some operands are computed using the register value, with indirect addressing for example, however this step occurs in the execution step. Therefore all an instruction needs to know in order to execute is the register address of its operands. The result will be written in the destination register also present in the instruction.
We need to model the eventual stalls or operand forwarding the processor handles. If a result is available before the WB stage (for example the result of an addition that will be written in register *rD* can be forwarded to another instruction, an addition for example, that uses the destination register *rD* as an operand, *rA* or *rB*).
We chose to implement this mechanism using auxiliary local pipeline stage locations that hold the register number of the operands and of the result. Several execution units perform in parallel, therefore before getting the operator indicated by the register number obtained in the decode stage, the unit will first perform a register number search in the destination register number of all the execution units. If the same register number is found, and the result is available, the operand value is replaced with this auxiliary location value.
Once an instruction gets decoded, the value of its fields are stored in a map called *IARG*, instruction arguments, composed of the instruction itself and the *instruction field, IF* as a key and the register address or the immediate value of the instruction field as the value.
The value stored in *IARG* can therefore be either a register address or an immediate value. It is during the execution phase that the distinction will be made as it is intrinsically dependent of the instruction type.

6.4.9.6 Data Hazards

Data hazards can occur between the ALU/BFU and the IMUL/IDIV execution units as they share the same GPR register set. Function must be implemented to check if the register used by a unit is the result register of another. In this case it should wait for the instruction's Write Back stage to complete.

6.4.9.7 Instruction Dispatch/Decode (ID)

The Instruction Decode receives an instruction from the current PC and decodes the instruction fields according to the instruction opcode. The field's values are stored in a map according to the instruction and the field type.
When a decoded instruction that depends on a result register that has not yet been calculated (write back or before), the ID is stalled until the value of that register is known.
Scoreboard information regarding the data dependencies is broadcast to all execution units. The instruction is then dispatched to all the execution units in the same time. Each execution unit decodes the instruction. If the instruction is not implemented, a program exception is taken.
In HiTAsmL code this translates to the fact that for an instruction sequence
opcode1 r1, r2 r3
opcode2 r3, r1, r4,
the **r1** operand of the **opcode2** instruction will only be accessed through `rA = GPR(r1)` only after the value of the destination register from the **opcode1** instruction is calculated.

Listing 6.15: HiTAsmL example of ID

```
1  module ID
2    signature :
3      dependency -> Bool
4      IDstall -> Bool
5    definition:
6      rule dependencyCheckID(reg : Int) =
7        if reg = ALUdestReg then
8          dependency := true
9          ALUOP1 := undef
10         ALUOP2 := undef
11         ALUresReg := undef
12       elif reg = ALUresReg then
13         ALUOP1 := undef
14         ALUOP2 := undef
15         ALUresDest := undef
16
17     rule dependencyCheckID(reg : Int) =
18       if reg = ALUdestReg then
19         dependency := true
20       elif reg = ALUresReg then
21         dependency := ALUres
22       elif reg = LSUdestReg then
23         dependency := true
24       elif reg = EADDRdestReg then
```

```
25          dependency := true
26        if reg = LOADWBdestReg then
27          dependency := true
28        //check all dependencies
29
30      rule ID(instr : Int) =
31        let op = opcode(instr) in
32          case op =
33            add or addc or adde :
34              IARG(instr, IF_rD) := instr[6:10]
35              IARG(instr, IF_rA) := instr[11:15]
36              IARG(instr, IF_rB) := instr[16:20]
37              IARG(instr, IF_OE) := instr[21]
38              IARG(instr, IF_Rc) := instr[31]
39            addi :
40              IARG(instr, IF_rD) := instr[6:10]
41              IARG(instr, IF_rA) := instr[11:15]
42              IARG(instr, IF_SIMM) := instr[16:31]
43          //verify data hazards
44          //
```

6.4.9.8 Execution units

Listing 6.16: HiTAsmL example of the Execution unit

```
1  module EX
2    signature :
3      enum opId = {OP1, OP2, R}
4      ALUop : opId -> Int
5
6      //in
7      ALUOP1 -> Int
8      ALUOP2 -> Int
9      ALUdestReg -> Int
10     ALUres -> Int
11     ALUresReg -> Int
12   definition:
13     rule read_op1(reg : Int, id : opId) =
14       if reg = ALUexecResReg then
15         ALUop(id) := ALUexecResReg //immediate update - just forward
16       elif reg = MULexecResReg
17         ALUop(id) := MULexecResReg
18       elif reg = MEMResReg
19         ALUop(id) := MEMResReg
20       else
21         ALUop(id) := GPR(reg)
22
23     rule read_op2(reg : Int) =
24       if reg = ALUexecResReg then
25         ALUop2 := ALUexecResReg
26
```

```
27      rule EX_IMUL_IDIV(intr : Int) =
28        let op = opcode(instr) in
29          case op =
30            mulli :
31
32      rule EX_BFU(instr :Int) //might be integrated with EX_ALU
33
34      //rule to be defined using a function that stores result
35      rule overflow(val : Int) =
36
37      rule EX_ALU(instr : Int) =
38        let op = opcode(instr) in
39          case RCPU_ISA(op) =
40            add :
41              let rA = IARG(instr, IF_rA) in
42                let rB = IARG(instr, IF_rB) in
43                  let rD = IARG(instr, IF_rD) in
44                    call(read_operand(rA, OP1))
45                    call(read_operand(rB, OP2))
46
47                    let res = ALUop(OP1) + ALUop(OP2) in
48                      if overflow(res) then
49                        XER[0] := 1
50                        XER[1] := 1
51                      if res < 0 then
52                        CR[0] := 1
53                      elif res > 0 then
54                        CR[1] := 1
55                      else
56                        CR[2] := 1
57                      CR[3] := XER[0]
58                      if instr[22:30] = 0x8A then
59                        let CA = 3 in
60                          if carryout(res) then
61                            XER[CA] = 1
62                          else
63                            XER[CA] = 0
64                          ALUres := res + XER[CA]
65                      else
66                        ALUres := res
67                      if instr[22:30] = 0xA then
68                        if carryout(res) then
69                          XER[CA] = 1
70                        else
71                          XER[CA] = 0
72  module Execute1 : Execute > Execute
73  import Decode
74      signature :
75          dynamic rA: -> Int
76          //rA got from the decode stage in final version
77          dynamic rB: -> Int
78          dynamic rD: -> Int
79      definition :
```

```
      rule multiu(rA : Int, rB : Int, rD : Int ) =
         GPR(rD) := GPR(rA) * GPR(rB), 10


module Execute2 : Execute > Execute1
import Decode
  signature :
    dynamic rA: -> Int
    dynamic rB: -> Int
    dynamic rD: -> Int
    //rD got from the decode stage in final version
  definition :
    rule multiu(rA : Int, rB : Int, rD : Int ) =
      call(write_reg)(rt, BVmult_result(32, GPR(rs),
        BVSignExtend(imm, 16, 32)))
    rule exec() =
      if (PMEM(pipeline(Ex)) = MULTIU) then
        call multiu(DecodedRa, DecodedRb, DecodedRd)
```

6.4.9.9 Burst Buffer Unit

Listing 6.17: HiTAsmL example of the BBU

```
module BBC
  import GlobalMemory

  definition :
    rule bbc() =
      BE := BBCMCR[18] ; 1
```

The detailed definition of the Fetcher using the concrete BBU module.

Listing 6.18: HiTAsmL example of the BBU

```
module Fetch
  signature:
    dynamic QueueSize subsetof Integer
    dynamic controlled IPFQsize : QueueSize
    dynamic IPFQfull : Boolean
    static fqueueFull:  -> Boolean

  default init inititalFetchState:
    function IPFQfull = false
    function IPFQsize = 0

  definitions:
    domain IPFQsize = {1, 2, 3, 4}

    rule fqueueNotFull() =
      par
```

```
17          if IPFQsize = 4
18          then
19            IPFQfull := true
20        endpar
21
22      function fqueueFull() =
23        par
24          if IPFQsize = 4
25          then
26            true
27          else
28            false
29        endpar
30
31      rule Fetch() =
32        par
33          fqueueNotFull()
34          if IPFQfull and BusIdle
35          then
36            par
37              if BurstMode="Normal"
38                and ImmediateAcces=true and Mem(BE)=1
39              then
40                par
41                  Mem(BR) = requestBus(Mem(BR))
42                  if receiveBusGrant(BG)
43                    and NoOtherMasterIsDriving then
44                      Mem(BB) := assert(BB)
45                      TS := assert(TS)
46                      ADDRESS := drive(ADDRESS)
47                      ATTRIBUTE := drive(ATTRIBUTE)
48                      BURST := driveAsserted(BURST)
49                      IPFQ := mem(ADDR) \\receiveAddress
50                      returnData
51                        switch (ADDR[28:29] mod 4)
52                          case 0:
53                            par
54                              BDIP <- assert(BDIP)
55                              IPFQsize := IPFQsize + 1
56                            endpar
57                          case 1:
58                            par
59                              IPFQ <- mem(ADDR) \\receiveAddress
60                              IPFQsize := IPFQsize + 2
61                              BDIP <- assert(BDIP)
62                            endpar
63                          case 2:
64                            par
65                              IPFQ <- mem(ADDR) \\receiveAddress
66                              IPFQsize := IPFQsize + 3
67                              BDIP <- assert(BDIP)
68                            endpar
69                          case 3:
70                            IPFQ <- mem(ADDR) \\receiveAddress
```

```
                          IPFQsize := IPFQsize + 4
                      endswitch
                    endif
                  endpar
                endpar
            endpar
        endpar

//Valid is the set of valid PowerPC instructions

module ExecutePC
  import Global *
  signature:
    WritePC : instr -> Bool
  definition:
  //1-ary function that detects if the current i
  //nstruction modifies the normal flow.
    function WritePC($i instr) =
      par

      endpar

    rule ExecutePC =
      if ExecuteOK then
      par
        if not WritePC(instr) then
          PC := NextPC(PC)
      endpar

    rule NextPC =
      if not Branch(instr) and not Jump(instr) then
        if LittleEndian then
          reg(PC) := reg(PC) XOR 0b100
        else
          reg(PC) := reg(PC) + 1[word]

        else
          if Branch(instr) then
            if (LK = 1) then
              reg(PC) := reg(LR)
              \\gets the next instruction from the Link Register
            else
              reg(PC) := BTA
              \\the branch taget address calculatd by the BPU
            endif
          endif
        endif


module InstrOpcode
  import Global *
  signature:
    enum domain IUopcode = { ... } //int instructions opcode
    static IUinstr: Instr  -> Boolean
```

```
125     static Opcode: Instr  -> opcodebites
126     static Truncate: Word -> ?
127
128   definitions:
129     function Truncate($d in Integer,
130       $st in Integer, $end in Integer) =
131       //concatenation of
132
133     function Opcode($i in Instr) =
134       Truncate($i, 0, 5) //type conversion should be made
135
136     function IUinstr($i in Instr) =
137       if opcode($i) in IUopcode then
138         true
139       else
140         false
141
142
```

6.4.9.10 Instruction Memory Protection Unit

Listing 6.19: HiTAsmL example of the IMPU

```
1  module IMPU
2    import GlobalMemory
3    definition :
4      rule detectAccessViolation(addr : Int) =
5        call compareAddr(addr)
6      rule compareAddr(addr : Int) =
7        if (addr = OK) then
8          impuCancelRequest := 0
9        else
10         impuCancelRequest := 1
11       ;1
```

6.4.9.11 Execute PC

Listing 6.20: HiTAsmL example of the WB

```
1  module ExecutePC
2    import Global *
3    signature:
4      WritePC : instr -> Bool
5    definition:
6    //1-ary function that detects if the current i
7    //nstruction modifies the normal flow.
8      function WritePC($i instr) =
9        par
```

```
10
11        endpar
12
13      rule ExecutePC =
14        if ExecuteOK then
15        par
16          if not WritePC(instr) then
17            PC := NextPC(PC)
18        endpar
19
20      rule NextPC =
21        if not Branch(instr) and not Jump(instr) then
22          if LittleEndian then
23            reg(PC) := reg(PC) XOR 0b100
24          else
25            reg(PC) := reg(PC) + 1[word]
26
27          else
28            if Branch(instr) then
29              if (LK = 1) then
30                reg(PC) := reg(LR)
31                \\gets the next instruction from the Link Register
32              else
33                reg(PC) := BTA
34                \\the branch taget address calculatd by the BPU
35              endif
36            endif
37          endif
38
```

6.4.9.12 Write Back

MPC555's Write Back Bus has 2 slots. Separate Execution (ALU/BFU, IMUL/IDIV, FPU) and Load/Store units for integer and floating point instructions allow parallel Write Backs on the two dedicated register sets GPR and FPR. Functions that handle the data hazards between load/store and arithmetic instructions must be implemented for both types. The ALU/BFU, IMUL/IDIV instructions operate on the same register set (GPR) therefore a priority is assigned to write back operations.

For example, the single cycle instruction **subf** has priority on the write back bus over the **mulli**. Therefore even if the **mulli** instruction is issued first, the **mulli** write back is delayed one clock and causes a bubble in the execution stream.

[]

Listing 6.21: HiTAsmL example of the WB

```
module WB
  signatures :
    //the execution unit's results and dest reg
    //are transferred from the exec stage
    WBALUBFUresReg -> Int
    WBALUBFUreg -> Int
    WBIMULIDIVresReg -> Int
    WBIMULIDIVres -> Int
    LOADWBresReg -> Int
  definition:
    //check if it is architecturally correct
    rule WB_arbitration(reg : Int) =
      if

    rule WB(instr : Int) =
      let op = opcode(instr) in
        if ALUBFUinstr(op) = ALUBFU then
          writeReg(WBALUBFUresReg, WBALUBFUres)
        if ALUBFUinstr(op) = IMULIDIV then
          writeReg(WBIMULIDIVresReg, WBIMULIDIVres)
      addToHistoryBuffer(instr)

    rule LOADWB(instr : Int) =
      let op = opcode(instr) in

      addToHistoryBuffer(instr)
```

Listing 6.22: HiTAsmL example of the MPC555 processor

```
htasm MPC555
    hmodules IPFQ, Fetch, Decode, Execute, WriteBack //...
    import Fetcher1
    signature :
        dynamic function GPR : Int -> Int

  definition:
    rule opcode(addr : Int) =
      addr[0:5]
//the evaluation returns the first 6 bits of the address
    function RCPU_ISA(Int opcode) =
      initially {
        31 ->  add,
        37 -> stwu
      }

//transaction with the main memory
//load and store instr will enter this stage
```

The purpose of the multiple implementations for a single module is to allow the definition of multiple abstraction levels for a given component. From the designers point of view

an option to define these levels would be to modify the *transactional* part in order to do more or less modifications of needed locations, depending on the needed level of precision the analyzer needs. As the language enables the assignment of a delay to transactions, the system will detail more or less transitions by variation of the step size (this can be seen as a big step, small step variation).

6.5 Conclusions

In this chapter we presented the construction of the MPC555 processor model using the HiTAsmL language introduced in the previous sections. The pertinence of the choice of this use case is ensured through the vast utilization of this processor in numerous embedded applications, in avionics for example, but also through its architectural features that are complex enough to provide a good exemplification of the modeling and analysis features of our method.

The framework used to describe the processor proved efficient across the refinement paradigm and also able to naturally capture the timing anomalies of the processor. The model of the processor was completely extracted from its manuals and other technical documents depicting the inner functioning of the PowerPC RISC processor family.

During the laborious process of understanding the exact behavior of each component, we could confirm the importance of a clear, explicit and detailed material in the understanding of a processor and the creation of its model.

We can now also reply to an important question regarding the time needed to model the processor (or to take into account a new one in the analysis). The most part of the time was spent understanding precise behaviors and inferring temporal information for instructions or units from the manuals, and once the processor architecture was mastered, the effective coding time was limited. Given the simple syntax and the familiar look of the modeling language, we can state that similar results can be achieved by any adopter of the method.

Chapter 7

The WCET Analysis

7.1 Structure of the method

The two main entries of our method are the processor model and the program binary, as depicted in Figure 7.1. The processor is regarded as the union of its components $\mu P = \bigcup_{i=0}^{n} C_i$ and modeled as a hierarchical timed abstract state machine, that enables multiple definitions for a same component C_i. A supervisor that we call the *Oracle* decides what abstraction level is best suited for the current context in order to optimize the *precision* to *state explosion* ratio. An external value analyzer is used to obtain information regarding the instruction order, their addresses and the control flow graph of the program. Symbolic execution is used to symbolically execute each instruction of the program, meaning that each variable has initially a symbolic value (as we generally do not posses exact information on its value) that gets refined by accumulating all the information and decisions taken during execution. One of the advantages of this method is that it manages to simulate the interactions inside the processor in detail, for example capturing by construction the timing anomalies [RWT+06]. The *SE* generates all reachable states of the processor, meaning that we have to manage a rapidly increasing state space. Our fusion stage consists in merging as much states as possible without affecting too much the precision of the estimation. We achieve this by using the prediction module that will first identify the states that are good candidates for merging and then estimate the impact of the fusion on the global analysis. After browsing and evaluating the processor's states, the time corresponding to the worst path is

selected.

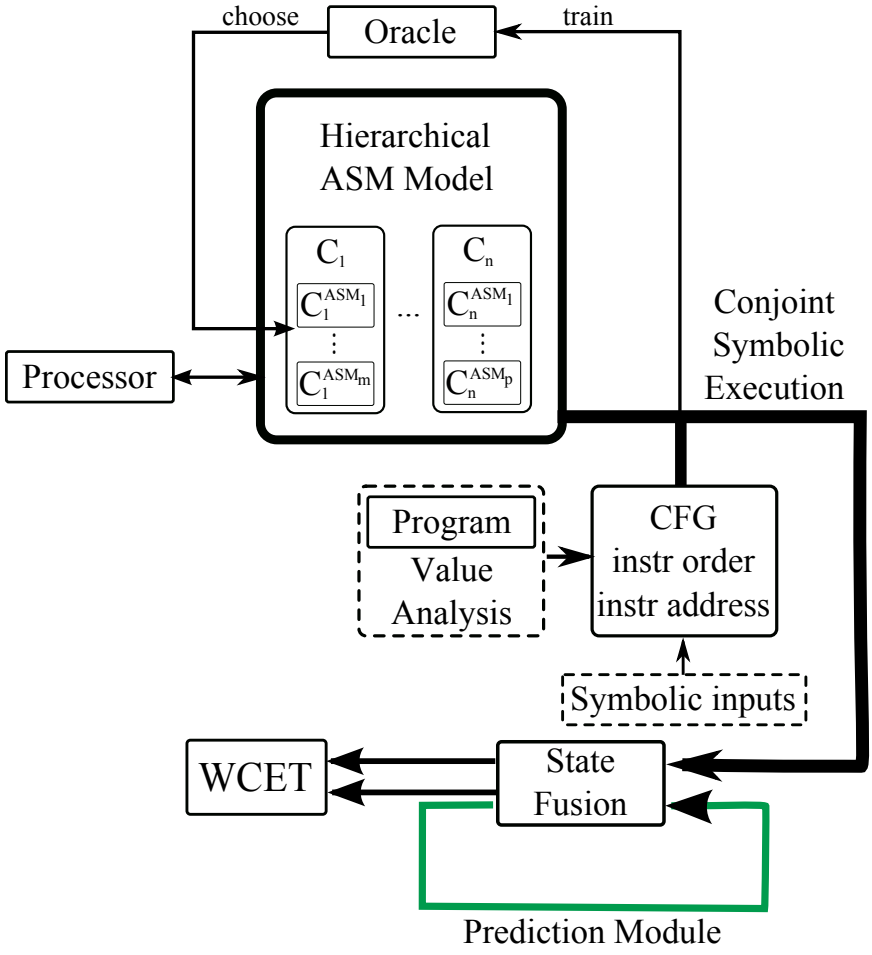


Figure 7.1: Global architecture of the WCET estimation tool

7.2 Value Analysis

In our method we use an abstract interpretation based value analysis that we launch, prior to our analysis, on the program's binary. Once the source code gets compiled in the *.elf* binary file, the value analysis is performed and we obtain a formatted output file, as depicted in figure 7.2.

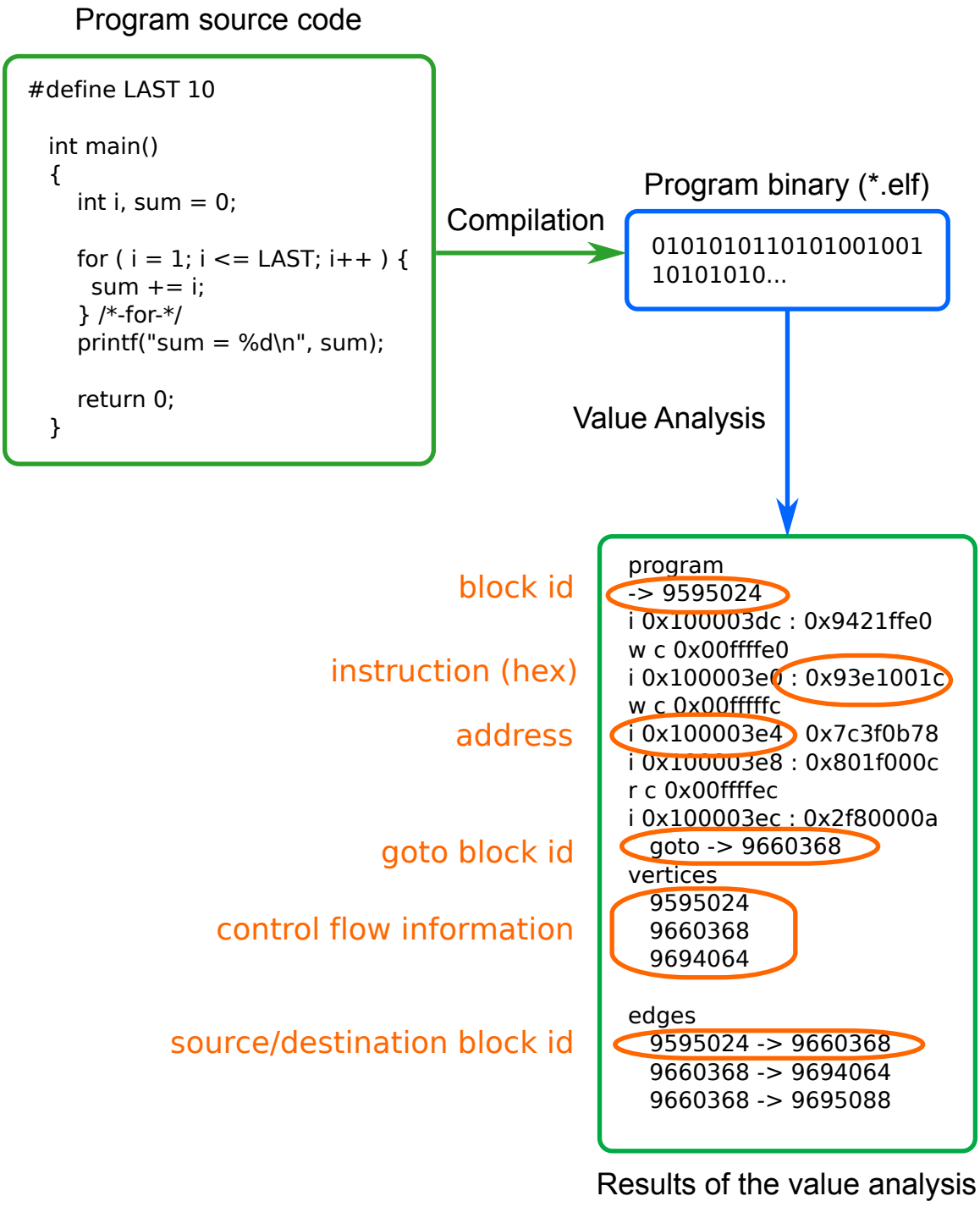


Figure 7.2: Value Analysis

7.2.1 Implementation

The Value Analyzer generates an output with the CFG of the program, the instructions of the program and their values. As we can see in Listing 7.1, the output gives the label of the nodes, `9595024`, their content, `i 0x100003dc : 0x9421ffe0` and the relation between nodes `edges 9595024 -> 9660368`. The `i` means that we have an instruction `0x100003dc` at address `0x9421ffe0` given in hexadecimal. The equivalent of the instruction in binary is `instr = 10010100001000011111111111100000` which means according to the opcode of the instruction, the first 6 bits of the instruction, `100101`, that is a `stwu` instruction, as we can see in Figure 7.3.

	opcode	**S**	**A**	**d**
Binary	100101	0000	10000	111111111111000
Hex	0x25	0	10	7FF8
Decimal	37	0	16	32760
	stwu			

Figure 7.3: Interpretation of the `0x100003dc` instruction

The exact effect of the `stwu` instruction is given in the PowerPC manual and listed in Figure 7.4. The value analyzer receives as input the program in its binary form and outputs a formatted output with the result of the analysis. Listing 7.1 shows the source code of the analysed binary, and listing 7.1 shows the result for that code and the figure 7.5 the CFG of the decompiled binary, labeled with the program's instructions and their addresses.

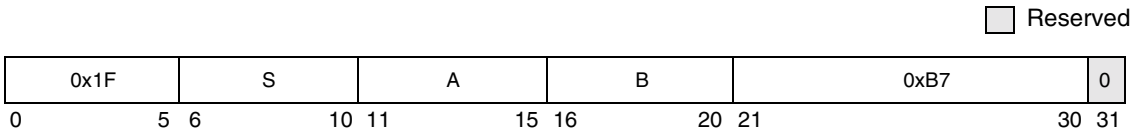
Listing 7.1: Value Analysis result on a C code

```
1
2  program
3  -> 9595024
4  i 0x100003dc : 0x9421ffe0
5  w c 0x00fffe0
6  i 0x100003e0 : 0x93e1001c
7  w c 0x00fffffc
8  i 0x100003e4 : 0x7c3f0b78
9  i 0x100003e8 : 0x801f000c
10 r c 0x00fffec
11 i 0x100003ec : 0x2f80000a
```

stwux **stwux**

Store Word with Update Indexed Load/Store Unit

stwux **r**S,**r**A,**r**B



EA← (**r**A) + (**r**B)
MEM(EA, 4)←**r**S
rA←EA

EA is the sum (**r**A|0)+(**r**B).

The contents of **r**S are stored into the word in memory addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

Figure 7.4: Interpretation of the `0x100003dc` instruction from the RCPU manual

```
12    goto -> 9660368
13 -> 9660368
14 i 0x100003f0 : 0x409d0014
15    else -> 9694064
16    then -> 9695088
17 ...
18
19 vertices
20    9595024
21    9660368
22    ...
23 edges
24    9595024 -> 9660368
25    9660368 -> 9694064
26    ...
```

7.2.2 Syntax

As previously stated the WCET estimation uses an offline value analysis performed on the binary of the program. The analysis provides the CFG of the program, the instruction

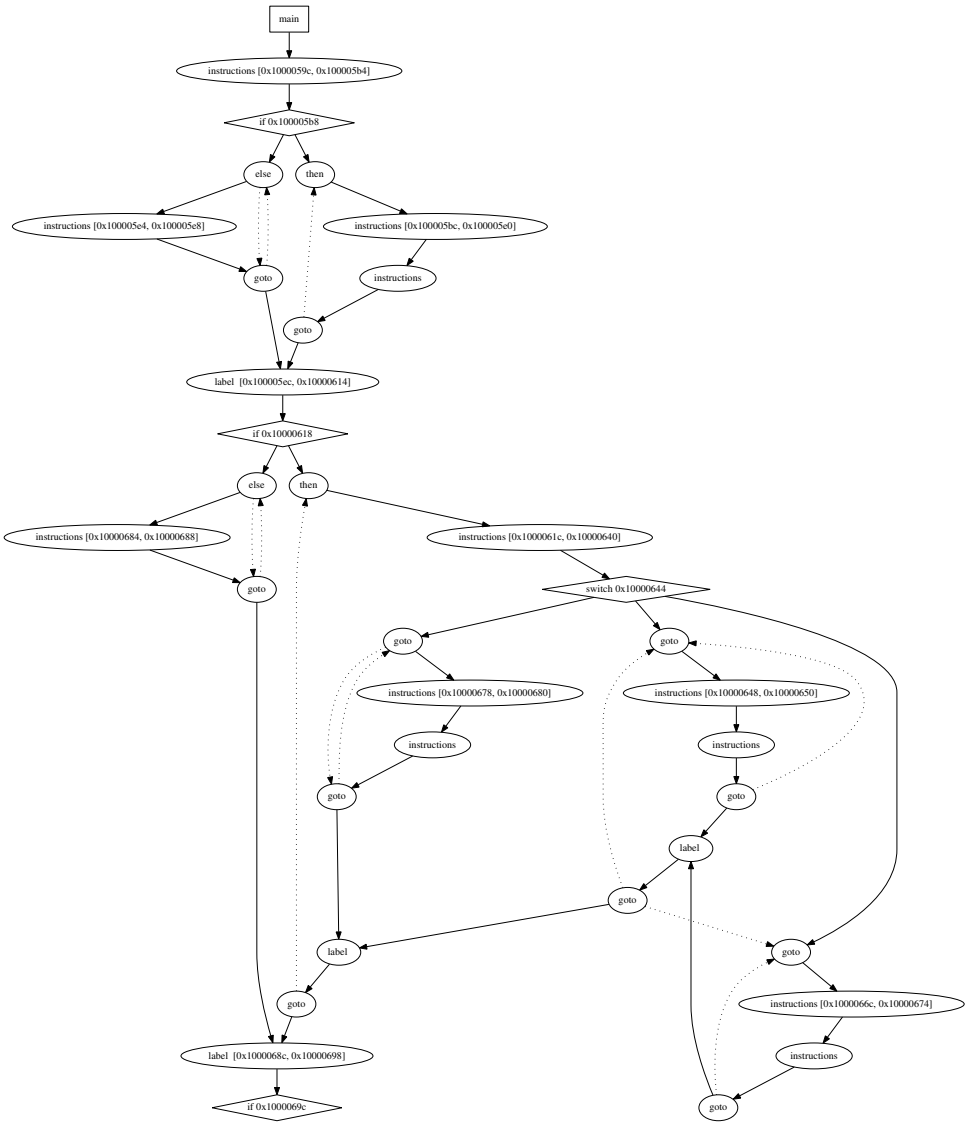


Figure 7.5: The graph of the decompiled binary

and their addresses. We developed a program that parses the output of the value analysis, presented in listing 7.1. In listing 7.2 we present the syntax of the Value Analyser parser in EBNF form and in listing 7.3 its syntax diagram.

Listing 7.2: The EBNF grammar of the value analysis output parser

⟨*program*⟩ ::= 'program' ⟨*codeLine*⟩ 'vertices' { ⟨*number*⟩ } 'edges' { ⟨*number*⟩ '->' ⟨*number*⟩}

⟨*codeLine*⟩ ::= { ⟨*instruction*⟩ | ⟨*label*⟩ | ⟨*branch*⟩ | ⟨*memAccess*⟩ }

⟨*instruction*⟩ ::= 'i' ⟨*number*⟩ ':' ⟨*number*⟩

⟨*label*⟩ ::= '->' ⟨*number*⟩

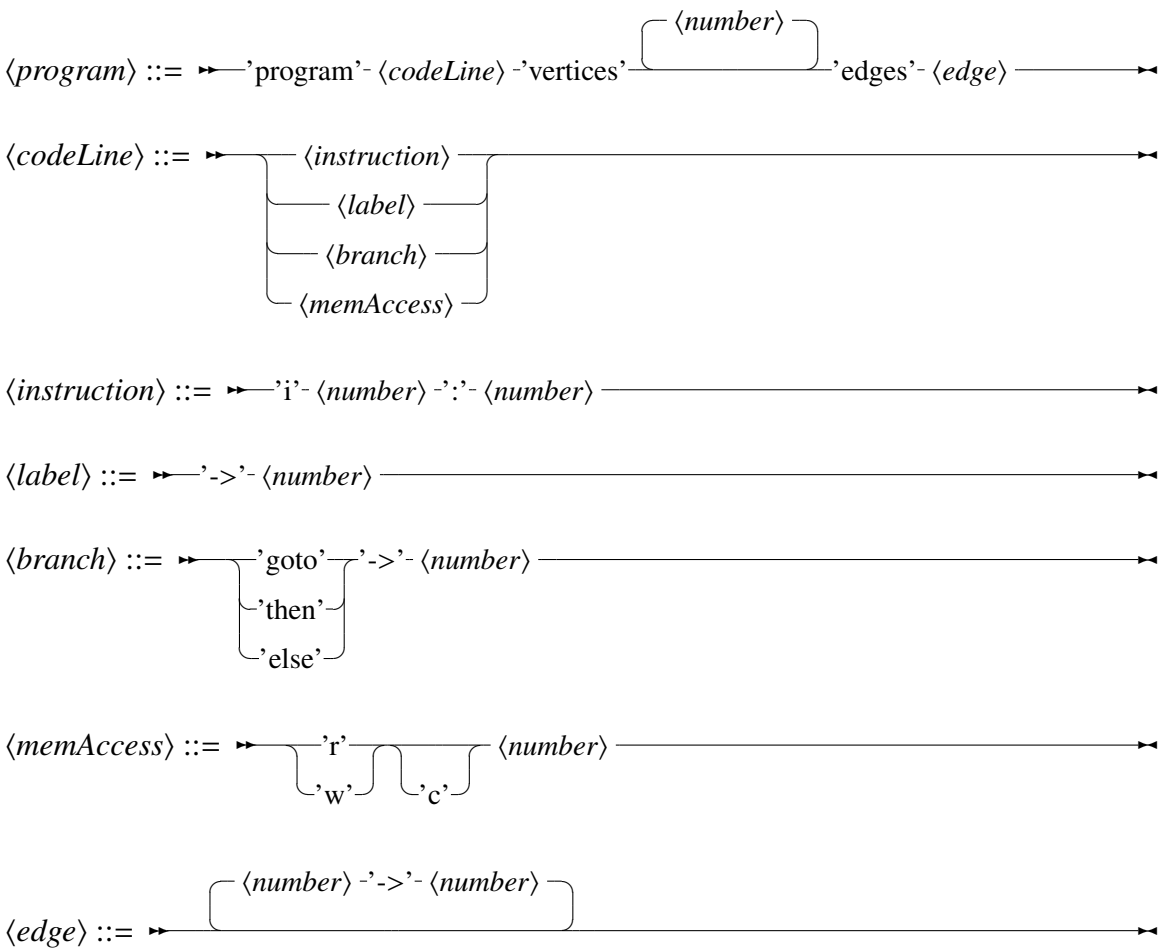
⟨*branch*⟩ ::= ('goto' | 'then' | 'else') '->' ⟨*number*⟩

⟨*memAccess*⟩ ::= ('r' | 'w') ['c'] ⟨*number*⟩

⟨*edge*⟩ ::= { ⟨*number*⟩ '->' ⟨*number*⟩}

Listing 7.3 gives a more visual representation of the syntax used in the value analysis output parser, using rail syntax diagrams.

Listing 7.3: The syntax diagram of the value analysis output parser



7.3 Conjoint Symbolic Execution

The use of *symbolic execution* (SE) in timing analysis as a way to capture the intra-processor interactions has been employed with good results in [Lun02]. However the presented method suffers from the lack of a precise hardware model and inaccurate merging strategies. These factors lead to important overestimations, rendering the method unpractical for industrial large-size applications.
Based on the intuition presented in [BMV08] we further extend the conjoint symbolic execution of a precise hardware model featuring delayed updates and a hierarchy of abstraction levels. We also redefine the *state merging* and present novel merging techniques.

7.3.1 Symbolic Execution

Motivation The symbolic execution can be seen as a generalization of testing. It allows *unknown* symbolic variables in evaluation:

$$\texttt{x=}\alpha\texttt{; assert(f(x) == 2*x*x)}$$

and if the execution depends on *unknown*, the SE forks the program execution.

Insight In a symbolic execution, each execution represents many concrete program runs. The number of included runs in the symbolic execution is given by the set of runs whose concrete values satisfy the *path condition*. Therefore it can cover a greater subset of the execution space than simple testing can.

Remarks Even though SE can cover a great number of concrete runs there are still a lot of possible program paths. In order to decide which paths are feasible and validate assertions, it needs many query solver calls. However, computers are much faster than the time when SE was introduced. Moreover, powerful Satisfiability Modulo Theories (SMT) solvers are available nowadays that can solve very large instances, very quickly (assertion

check, prune infeasible paths). (we have considered Z3, STP, and Yices)

Symbolic execution is a static program analysis approach originally introduced in [Kin76]. Its goal is to provide an analysis of all reachable program states, therefore under all possible inputs. To this end, SE reasons about symbolic values rather than concrete ones, therefore:

- it gives assurance about any execution, prior to deployment;
- it is the root of a number of interesting static analysis ideas and tools.

The main idea behind symbolic execution is to use symbolic values, instead of actual data, as input values. In this way it can reason with multiple values in the same time. Output values will therefore be represented as a symbolic function of the input symbolic values. The symbolic state retains the mapping between variables and their current values, represented as symbolic expressions, together with a path condition. The PC is a quantifier free first order expression that accumulates all the constraints on the symbolic variable values corresponding to a particular path in the program. The interpretation of the assignment rule is straightforward.
A special treatment is applied to conditional statements, *if cond then* $block_1$ *else* $block_2$. If the truth value can not be determined in the current state, using all the constrains in the PC, the execution forks and both branches will be explored. This is equivalent to verifying if either $PC \supset \neg cond$ or $PC \supset \neg cond$. Each PC of the taken branch is updated with the corresponding validated condition: $PC_{true} = PC \cup cond$ and $PC_{false} = PC \cup \neg cond$. The execution continues on a given path until the PC becomes unsatisfiable. The satisfiability is checked with an off-the-shelf constraint solver.
In figure 7.6 on page 194 we show an example of the symbolic execution of the program presented in listing 7.4.

Listing 7.4: SE example program

```
1 int x, y, z;
2 int a = -1;
3 if (x > 0)
4 {
5   y = a*x;
6 }
7 if (y >= 0)
```

```
8  {
9    z = z + y;
10 }
11 else
12 {
13   z = z - y;
14 }
```

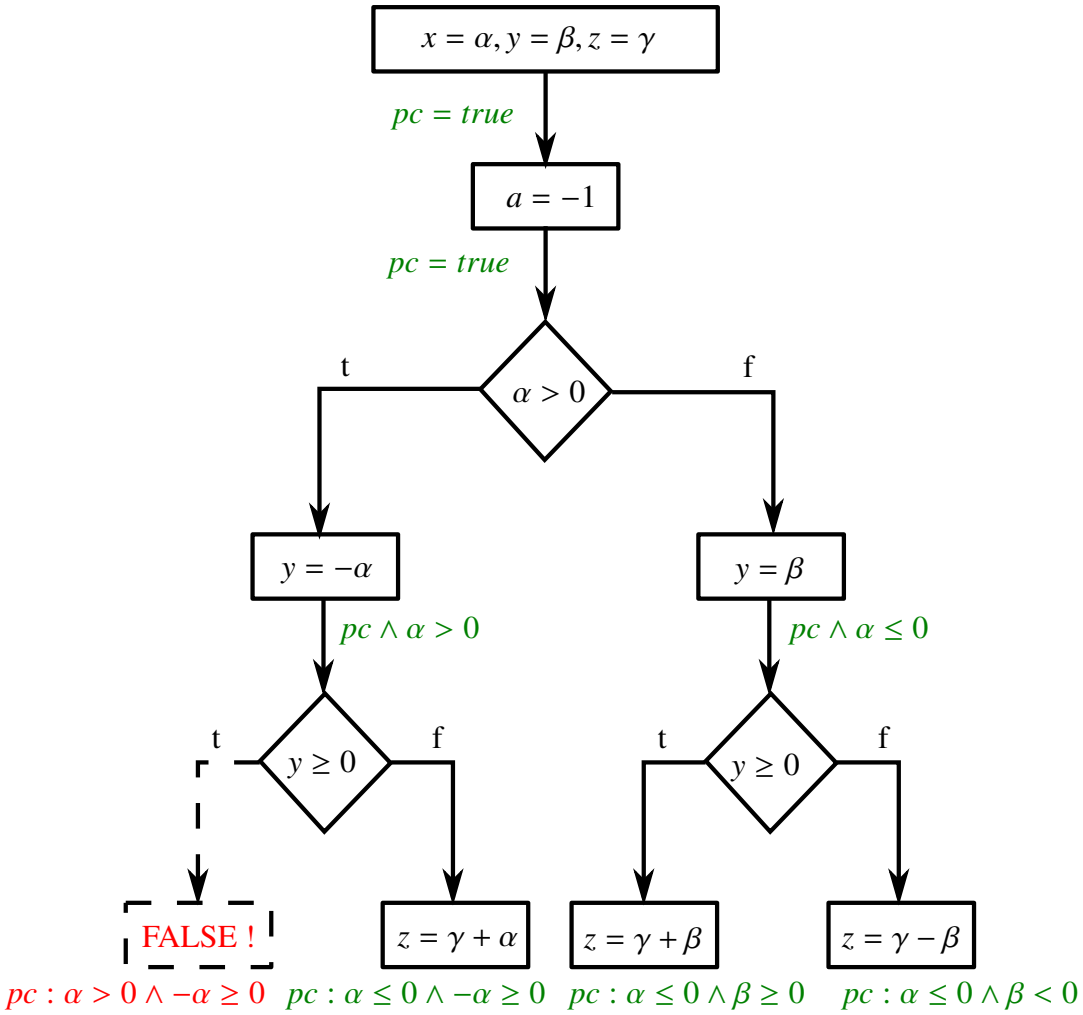


Figure 7.6: Symbolic execution of a program

7.3.2 The global SE implementation

Algorithm 3: Symbolic Execution high level view

input :

$\mathfrak{A}$ - a state

$\mathfrak{A}_{init}$ - the initial state,

Vector $< \mathfrak{A} > \mathcal{ES}$ - execution state vector

```
1  ES.add (A_init);
2  while (ES.size () > 0 ∧ ¬T_timeout) do
3      A_i = selectState ();
4      while (ES.ruleType () ¬ IF) do
           /* normal linear execution                                   */
5          nonForkInterpretation ();
6          A_i = nextState ();
7          if ¬ finalState (A_i) then
8              ES.add (A_i);
9      if (ES.ruleType () = FORK then
           /* if-then-else execution - implemented in the following algorithms */
10         A_T, mathfrakA_F = forkInterpretation ();
11         if ¬ finalState (A_T) ∧¬ finalState (A_F) then
12             ES.add (A_T);
13             ES.add (A_F);
```

7.3.3 SE-HiTAsm

Definition 57 (Symbolic HiTAsm). A *symbolic hierarchical timed abstract state machine* consists of a signature Σ, a set of initial states for Σ , a set of rule declarations and a distinguished rule name of arity zero called *main rule name* of the machine.

The state of the Symbolic HiTAsm contains a sequence of states formed by the set of all the states that satisfy the path condition.

Definition 58 (Symbolic HiTAsm State). A symbolic state $\mathfrak{A}$ for the signature Σ is a non-empty domain X that includes the set of symbols, $X_{\mathcal{SE}}$, $X = X_{HiTAsm} \cup X_{\mathcal{SE}}$, the superuniverse of $\mathfrak{A}$, together with interpretations of the function names of Σ in one of the domains of $\mathfrak{A}$ and the formula Ψ_{pc} called the path condition that accumulates constraints on the symbolic function names, $\mathcal{M}_{\mathcal{SE}} = \left(\mathfrak{A}, \bigcup \mathfrak{A}^{init}, \Sigma, X, R, \mathcal{U}, \Psi_{pc}\right)$.

The symbolic update rule

$$f := \alpha, \delta = \delta_{min} \rightsquigarrow l = (f, \phi), u = (l, \alpha, \delta), \Psi_{pc} = \Psi_{pc} \wedge l = \alpha. \tag{7.1}$$

$$f := \alpha, \delta \rightsquigarrow l = (f, \phi), u = (l, \alpha, \delta). \tag{7.2}$$

7.3.3.1 Symbolic Logic

In this section we extend the mathematical logic associated to basic ASM found in [BS03] to include symbols for the symbolic execution.

Definition 59 (Symbolic Term). Let Σ be a signature of the symbolic state $\mathfrak{A}$, the symbolic terms of Σ are syntactic expressions generated as follows:

1. Variables $x, y, z, \ldots$ are terms.
2. Constant c of Σ are terms.
3. Symbols $\alpha_1, \alpha_2, \ldots$ of Σ and $X_{\mathcal{SE}}$ are terms.
4. If $f \in \Sigma$ is a function names and $t_1, \ldots t_n$ are terms, then $f(t_1, \ldots t_n)$ are terms.

The variable assignment does not change a lot from the definition in [BS03]. We present it anyway for clarity. Please note that even if the definition of the variable assignment map ζ is almost unchanged, the variables can take values from $|\mathfrak{A}|$, therefore symbolic values from $X_{\mathcal{SE}}$.

Definition 60 (Variable assignment). Let $\mathfrak{A}$ be a symbolic state. A variable assignment for $\mathfrak{A}$ is a finite function ζ_α which assigns elements of the super-universe $|\mathfrak{A}| \supseteq X_{\mathcal{SE}}$ to a finite number of variables. We write $\zeta[x \mapsto a]$ for the assignments of element $a \in |\mathfrak{A}|$ to x in ζ such that:

$$\zeta[x \mapsto a](y) = \begin{cases} a & , \ \textit{if } y = x; \\ \zeta(y) & , \ \textit{otherwise}. \end{cases}$$

Definition 61 (Interpretation of symbolic terms). Let Σ be a signature of the symbolic state $\mathfrak{A}$, the symbolic terms of Σ are syntactic expressions generated as follows:

1. $[\![x]\!]^{\mathfrak{A}}_{\zeta} = \zeta(x)$
2. $[\![c]\!]^{\mathfrak{A}}_{\zeta} = c^{\mathfrak{A}}$
3. $[\![\alpha]\!]^{\mathfrak{A}}_{\zeta} = \alpha^{\mathfrak{A}}$
4. $[\![f(t_1, \ldots t_n)]\!]^{\mathfrak{A}}_{\zeta} = \begin{cases} f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\zeta}) & , \ \textit{if } l \in \mathfrak{A}; \\ \mathfrak{A}\left[l_f \mapsto \alpha_f\right] & , \ \textit{otherwise}, \end{cases}$
   where $l_f = (f^{\mathfrak{A}}, [\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\zeta})$.

As it was already stated before, it is notationally convenient and intuitively correct to view state $\mathfrak{A}$ as sets of pairs (l, v) of locations l and their values v. Thus we write $\mathfrak{A}\,[l \mapsto v]$ for the extension of the state $\mathfrak{A}$ by the new location l with the value v.

Implementation note Please note that we apply the same reasoning to free variables and identify them with 0-ary functions symbols so that their interpretation is incorporated into $\mathfrak{A}$ instead of being kept in a distinct environment function ζ.

When a symbolic term is interpreted (for example, when it is used in the right hand side of an update) a function lookup is performed because the function memoization is a standard feature of ASMs. If the location of that function in the current state its not defined, then a new symbol is attributed to that location, through an immediate symbolic update.
The generation of formulas definition 62 and the *range* definition 63 are the same as in [BS03]. We present them for completeness.

Definition 62 (Formula). Let Σ be a signature. The formulas of Σ are generated as follows:

1. $s = t$ is a formula, if s and t are terms.
2. φ is a formula $\implies \neg\varphi$ is a formula.
3. $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ and $(\varphi \implies \psi)$ are formulas if φ and ψ are formulas..
4. $(\forall x\varphi)$ and $(\exists x\varphi)$ are formulas if φ is a formula and x a variable.

Definition 63 (Range of a formula). The range of the formula φ with respect to x is the set of all elements of $\mathfrak{A}$ that make the formula true under ζ:

$$range\,(x, \varphi, \mathfrak{A}, \zeta) = \left\{a \in |\mathfrak{A}| : [\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x\mapsto a]} = true.\right\}$$

Formulas may be used to express properties of functions of an abstract state. The *path condition* is a special formula that expresses properties of the symbolic values of the current state. It is used to refine the value of the symbolic terms by accumulating all the control flow's constraints.

Definition 64 (HiTAsm Path Condition). Let $\mathfrak{A}$ be a state we say that the formula Ψ_{pc} is a *path condition* of the state $\mathfrak{A}$ if it is modeled by the state $\mathfrak{A}$ and write $\mathfrak{A} \models \Psi_{pc}$ meaning that $[\![\Psi_{pc}]\!]^{\mathfrak{A}}_{\zeta} = true$ for all symbolic variable assignments ζ for φ.

$$\mathfrak{A} \models \Psi_{pc} \implies [\![\Psi_{pc}]\!]^{\mathfrak{A}}_{\zeta[x\mapsto a]} = true \mid \forall x \in \Psi_{pc}.$$

We can use definition 64 to express the relation between the Symbolic HiTAsm state and the HiTAsm state:

Definition 65 (Symbolic states).

$$\mathcal{M}_{\mathcal{SE}} = \bigcup \mathcal{M}_i \mid \forall \mathfrak{A} \in \mathcal{M}_i \implies \mathfrak{A} \models \Psi pc.$$

After each update, all the values in the store depending on that updated value are also updated. For example if a symbolic location will be updated to a value, then all other locations will also be updated, and the *path condition* modified.

Boolean-valued expression A formula is either true or false in a state and is also called Boolean-valued expression.

However in the case of symbolic execution, the value of the formula can not always be determined. In order to maintain the above property, the execution is split and two successor states are generated, one in which the formula holds and another in which its negation holds.
This behavior is applied in the case of the **if-then-else** rule if the guard of the rule is a symbolic formula, as we can see in table 7.1:

$$\textbf{if } C_s \textbf{ then } R_t \textbf{ else } R_f.$$

The implementation of the **if-then-else** rule is also given in algorithm 4.
The same reasoning is applied to the **forall-with-do** expression:

$$\textbf{forall } x \textbf{ with } \varphi \textbf{ do } P$$

that has the meaning of executing P in parallel for all the x satisfying φ.

Introducing the analyzer state The work presented in [BM09a] brings the prove that the processor can generate identical states regardless of the execution history. This fact is partially due to the finite and considerably small number of processor states compared to the numbers of states generated by the analysis of the program on that processor.
The conjoint symbolic execution explores all the reachable states of the processor running the analyzed program. Merging identical or similar states is crucial in order to achieve scalability. Similar states can be strongly or weakly similar, meaning that the impact of the fusion will be acceptable or not. This leads to a dynamically approach included in our Prediction Module. Its goal is to evaluate the impact in the future of a fusion by unrolling the execution tree for several steps (generally equal to the pipeline depth), continuing the execution in parallel starting either from the merged states or the un-merged and comparing the result, [BM09a]. The first step of our conjoint *SE* deals with the program's CFG that is regarded as an input for the processor's model *SE*, as shown in figure 7.7.

Algorithm 4: Implementation of the symbolic **if-then-else** rule interpretation

input : C_S - the condition of the guard, R_t, R_f - blocks of the true and false branch

output: nextStates

1 *a symbolic term is detected in the guard of an if rule*;
2 **if** *symbolic*(C_S) = True **then**
3 **if** $\Psi_{PC} \supset C_S$ **then**
  /* the condition is True so continue with the True branch */
4 $\mathcal{U} = \mathcal{U} + U^{R_t}$;
5 $\mathfrak{A}_t = yields\left(R_t, \mathfrak{A}, \zeta, \mathcal{U}, \Psi_{pc}\right)$;
6 nextStates $\longleftarrow \{\mathfrak{A}_t\}$;
7 **else if** $\Psi_{PC} \supset \neg C_S$ **then**
  /* the condition is False so continue with the False branch */
8 $\mathcal{U} = \mathcal{U} + U^{R_f}$;
9 $\mathfrak{A}_t = yields\left(R_f, \mathfrak{A}, \zeta, \mathcal{U}, \Psi_{pc}\right)$;
10 nextStates $\longleftarrow \{\mathfrak{A}_f\}$;
11 **else**
  /* neither $\Psi_{PC} \supset C_S$ or $\Psi_{PC} \supset \neg C_S$ are theorems so fork the execution */
12 $\mathcal{U} = \mathcal{U} + U^{R_t}$;
13 $\mathcal{V} = \mathcal{U} + U^{R_f}$;
14 $\mathfrak{A}_t = yields\left(R_t, \mathfrak{A}, con\zeta, \mathcal{U}, \Psi_{pc} \wedge C_S\right)$;
15 $\mathfrak{A}_f = yields\left(R_F, \mathfrak{A}, \zeta, \mathcal{V}, \Psi_{pc} \wedge \neg C_S\right)$;
16 nextStates $\longleftarrow \{\mathfrak{A}_t, \mathfrak{A}_f\}$;
17 **else**
  /* normal execution - does not imply the Ψ_{pc} lookup of the condition */
18 **if** C_S = True **then**
19 $\mathfrak{A}_{next} = yields\left(R_t, \mathfrak{A}, \zeta, \mathcal{U} + U^{R_t}, \Psi_{pc}\right)$;
20 **else**
21 $\mathfrak{A}_{next} = yields\left(R_f, \mathfrak{A}, \zeta, \mathcal{U} + U^{R_f}, \Psi_{pc}\right)$;
22 nextStates $\longleftarrow \{\mathfrak{A}_{next}\}$;

Table 7.1: Inductive deduction of the semantics of Symbolic HiTAsm

$$\frac{}{yields\left(\textbf{skip}, \mathfrak{A}, \zeta, \phi, \Psi_{pc}\right)}$$

$$\frac{yields\left(\textbf{skip}, \mathfrak{A} + \{(l, v, \delta)\}, \zeta, \phi, \Psi_{pc}\right)}{yields\left(\textbf{skip}, \mathfrak{A}, \zeta, \{(l, v, \delta)\}, \Psi_{pc}\right)}$$

$$\frac{}{yields\left(f(s_1, \ldots, s_n) := t.\delta, \mathfrak{A}, \zeta, \mathcal{U} \oplus \{(l, \alpha, d)\}, \Psi_{pc}\right)} \quad l = (f, (\llbracket s_1 \rrbracket^{\mathfrak{A}}_{\zeta}, \ldots, \llbracket s_n \rrbracket^{\mathfrak{A}}_{\zeta})), \; \alpha = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta} \text{ and } d = \llbracket \delta \rrbracket^{\mathfrak{A}}_{\zeta}$$

$$\frac{yields\left(P, \mathfrak{A}, \zeta, \mathcal{U}, \Psi_{pc} \wedge \varphi\right) yields\left(Q, \mathfrak{A}, \zeta, \mathcal{V}, \Psi_{pc} \wedge \neg\varphi\right)}{yields\left(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, \zeta, \mathcal{U} \cup \mathcal{V}, \Psi_{pc}\right)} \quad \text{if } \exists x\varphi : a \in X_{\mathcal{SE}}, \llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta[x \mapsto a]}.$$

$$\frac{yields\left(P, \mathfrak{A}, \zeta, \mathcal{U}, \Psi_{pc}\right)}{yields\left(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, \zeta, \mathcal{U}, \Psi_{pc}\right)} \quad \text{if } \forall x \in \varphi : a \notin X_{\mathcal{SE}}, \; \llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta[x \mapsto a]} = true.$$

$$\frac{yields\left(Q, \mathfrak{A}, \zeta, \mathcal{V}, \Psi_{pc}\right)}{yields\left(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, \zeta, \mathcal{V}, \Psi_{pc}\right)} \quad \text{if } \forall x \in \varphi : a \notin X_{\mathcal{SE}}, \; \llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta[x \mapsto a]} = false.$$

$$\frac{yields\left(P, \mathfrak{A}, \zeta[x \mapsto a], U_a, \Psi_{pc} \wedge \varphi\right) yields\left(P, \mathfrak{A}, \zeta[x \mapsto b], U_b, \Psi_{pc} \wedge \varphi\right)}{yields\left(\textbf{forall } x \textbf{ with } \varphi \textbf{ do } P, \mathfrak{A}, \zeta, \mathcal{U} = \bigcup_{a \in I} U_a \cup \mathcal{U} = \bigcup_{b \in J} U_b \Psi_{pc}\right)} \quad \text{if } \exists x\varphi : a \in X_{\mathcal{SE}}, \llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta[x \mapsto a]}.$$

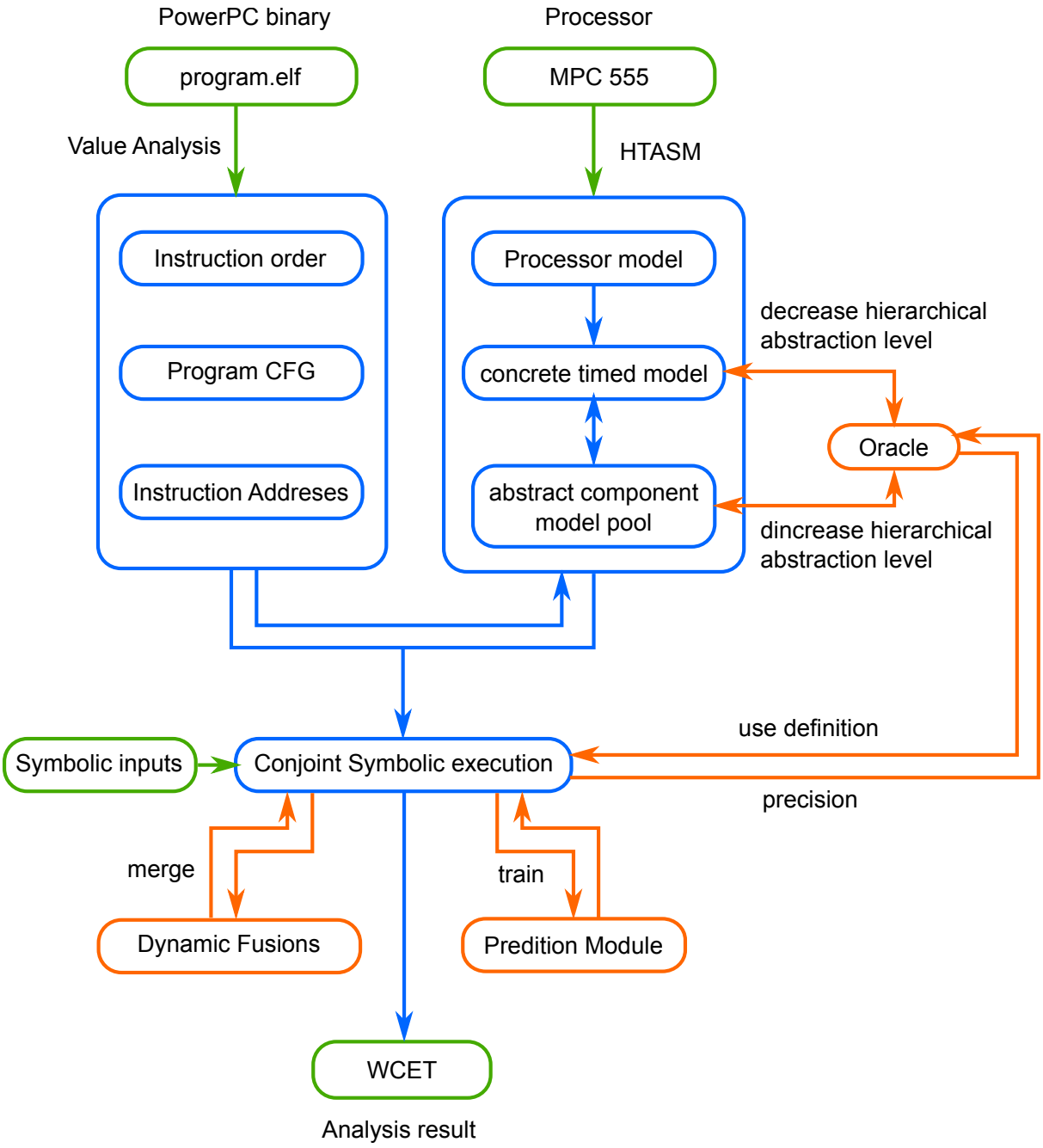


Figure 7.7: WCET analysis overview

7.4 States and HiTAsms

A HiTAsm Machine state can be defined by a store

$$\mathcal{M}_i \overset{\text{not}}{=} \left(\omega_i^{(n)}, u_i^{(n)}\right) \tag{7.3}$$

where $\omega_i^{(n)}$ maps the content of HiTAsm locations at the execution step n for the trace i and $u_i^{(n)}$ is the update set, such that

$$\omega_i^{(n)} + u_i^{(n)} = \omega_i^{(n+1)}, \tag{7.4}$$

where + is the operator that applies an update state to a HiTAsm state introduced in definition 23.

We say that a machine has reached its final state for the path i iff:

$$u_i^{(n)} = u_i^f = \phi \tag{7.5}$$

meaning that no new updates were produced in that state and all the previous delayed updates were fired.

7.4.1 Similar states identification

The next step is the identification of these similar or identical states in a computational efficient way, with the help of the similarity relation $\sim_t$,

$$\left(\omega^{(i)}, u^{(i)}\right) \sim_t \left(\omega^{(j)}, u^{(j)}\right) \tag{7.6}$$

without looking into the whole state space and comparing all the $\mathcal{P}(S)$ states in store set S. The t stands for similarity with regards to the execution time, meaning that two states are similar even if they have a different execution current time.

We take into account the fact that $\omega^{(n)}$ gives the photo of the pipeline's content at step n followed by the identification of the cost function $sim(\omega)$ that will be used to create a hash map

$$\left\{\left(sim(\omega^{(i)}, u^{(i)}) : \bigcup_i \left(\omega^{(i)}, u^{(i)}\right)\right)\right\}, \tag{7.7}$$

with the similarity value as the key and the different $\sim_t -similar$ stores as the value.

A hint on how to construct this function lays in the evaluation of $\left(\omega^{(i)}, u^{(i)}\right)$ through the HiTAsm relations.

Intuition We consider, for example, the fact that some instructions are quite similar like the MPC555 *add* instruction for example, *addi*, *addic*, *addic.*, *adis*, which have the same operand syntax, `rD`, `rA`, `SIMM`, but most of all that differences between certain $\left(\omega^{(i)}, u^{(i)}\right)$ pipeline stages are not very important. Moreover, we can identify the importance of the pipeline stages difference through the HiTAsm code itself by looking at the locations that the different pipeline states have in common.

Example Let us consider a pipelined processor that has no dependency between the *READ AND EXECUTE* stage and the *L ADDRESS DRIVE*, like the MPC555, or more general two stages, s_1 and s_2, that for certain instruction group types, $\mathfrak{T}_{i_1}$ and $\mathfrak{T}_{i_2}$, s_2 has to wait for s_1 completion in order to access a certain operand value and for other instruction families $\mathfrak{T}_{i_3}$ and $\mathfrak{T}_{i_4}$ does not have to. The HiTAsm of the processor will help us identify which are the dependencies between instruction families $\mathfrak{T}_{i_i}$ just by applying the functions and seeing which guards are validated.
We can therefore create groups of pipeline instructions, ordered by their degree of similarity. The chosen cost criteria will be given by the *distance* defined through HiTAsm relations. Other criteria can be found dynamically, for example, by systematically testing the merging of certain stores and keeping track of the features that lead to arriving in identical or similar states. This can be achieved with the use of the *prediction module*.

7.4.2 Order on HiTAsm states

In [Ben11] a proof of existence of equivalent states is sketched.

Lemma 2 *In a processor, different execution histories can lead to equivalent processor states.*

Proof 3 *(sketch) Let $\mathcal{M}$ be the HiTAsm processor model. We note by $\mathcal{M}(\mathcal{I})$ the model executing the instruction $\mathcal{M}$:*

$$\mathcal{M}(\mathcal{I}) \implies P \in \mathcal{S}_{\mathcal{I}}$$

where P is a processor state and $\mathcal{S}_{\mathcal{I}}$ the set of all execution environments allowing the execution of instruction $\mathcal{I}$.
Let $C_{\mathcal{I}}$ be the set of contrains associated to the execution environment $\mathcal{S}_{\mathcal{I}}$, that allow the execution of intruction $\mathcal{I}$. We extend this notion to the set of contraints $\mathscr{C}$ allowing the execution of an instruction flow $\{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_k\}$ we can thus define $\mathscr{C} = C_{\mathcal{I}_1} \wedge C_{\mathcal{I}_2} \wedge \ldots C_{\mathcal{I}_k}$ as the conjonction of the contraint sets associated to each instruction.
The processor's state P must therefore be in the restricted execution environment:

$$P \in \mathcal{S}_{\mathcal{I}_1} \cap \mathcal{S}_{\mathcal{I}_2} \cdots \cap \mathcal{S}_{\mathcal{I}_k}.$$

Execution constraints reduce the total processor states, therefore possible equivalent processor behaviors are possible for similar classes of instruction during the execution.

We extend this intuition with regard to abstraction and the nature of our framework.

Lemma 3 *A processors abstraction hierarchical level is proportional with the number of possible equivalent processor states.*

Proof 4 *(sketch) Abstractions limit the total number of system's state by grouping a greater number of possible states into a single abstract state.*
The extreme abstraction of the processor, using our ASM abstraction techniques is a single module with a single function in the vocabulary named Processor. Having a single function means a single abstract state (the most abstract) therefore, all these states can be merged.

Definition 66 (Identical states) Let $\mathfrak{A}^i$ and $\mathfrak{A}^j$ be two HiTAsm states, we say that state i is identical to the state j, and we note with:

$$\mathfrak{A}^i =_\delta \mathfrak{A}^j \Leftrightarrow \forall l_i \in \mathfrak{A}^i \wedge \forall l_j \in \mathfrak{A}^j . l_i \neq l_j \implies l_i = CT_i \wedge l_j = CT_j.$$

In other words two states are identical if and only if the only locations from those states that are not equal are the current time location, which obviously translate into the following property:

Property 11 *All locations from two identical states are equal except for the current time.*

$$\forall l_i \in \mathfrak{A}^i \setminus CT_i \wedge \forall l_j \in \mathfrak{A}^j \setminus CT_i \implies l_i = l_j.$$

Identical states are very useful in order to reduce the analysis state space. If two identical states from two different paths can be identified, this means that in the execution history there was a point where two states were similar and lead to those particular identical states. Starting from the identical state definition we can generalize the concept of *similarity*. Lemma 2 states that different histories may lead to identical states, which also means that starting from two different states we can arrive in identical states. We would like to identify those states. To limit the search space, we use the HiTAsm framework to extract some properties that states should respect making them more likely to generate identical state.

Statement 4 *The number of different locations between two states is inversely proportional with their chances of becoming identical in the near future.*

Or more formally:

Lemma 4 *Let $\mathfrak{A}^i$ and $\mathfrak{A}^j$ be two HiTAsm states and let $\mathfrak{D}_{ij}$ be their difference:*

$$\mathfrak{D}_{ij} = \mathfrak{A}^i \setminus \mathfrak{A}^j,$$

we introduce the similarity probability, based on the evolution of states, $P(\mathcal{S})$ which is inversely proportional with the number of different locations between the two states:

$$P(\mathcal{S}) = \frac{1}{|\mathfrak{D}|}$$

therefore

$$P\left(\mathcal{S}_{ij}\right) \cdot \left(\mathfrak{A}^i \setminus \mathfrak{A}^j\right) = ct.$$

Statement 4 and lemma 4 simply acknowledge that if we want to identify states that are likely to become identical, we will have a higher probability of success if we start by looking into the less different states. These assumptions serve as a starting point for the similar HiTAsm machine state search criteria. In the following section, the identification will also take into account the update sets of each machine state. If timed updates are set for those same locations that are different, the probability of similarity will increase.

Definition 67 (Delta-similar states) Let $\mathfrak{A}^i$ and $\mathfrak{A}^j$ be two HiTAsm states, we say that state i is similar or delta-similar to the state j, and we note by using the *delta-similarity* operator $\sim_\delta$:

$$\mathfrak{A}^i \sim_\Delta \mathfrak{A}^j \Leftrightarrow \exists l_i \in \mathfrak{A}^i \setminus l_{CT} \mid \forall i \in \{1 \dots \Delta\}.l_i \neq l_j \wedge \forall i \notin \{1 \dots \Delta\}.l_i = l_j$$

where $\Delta = |\mathcal{D}|$.

In other words

$$\mathfrak{A}^i \sim_\Delta \mathfrak{A}^j \Leftrightarrow \Delta = \left|\mathfrak{A}^i \setminus \mathfrak{A}^j\right|. \tag{7.8}$$

Example The $\sim_1, \sim_2, \sim_3, \dots$ delta-similarity relations will form the equivalence classes of the HiTAsm states having $1, 2, 3, \dots$ locations of different values.

Theorem 4 *The delta-similarity is not an R-equivalence relation.*

The notions of *R-equivalence relation* and *equivalence relation* in general are formally defined in section 7.6 on page 216.

Intuition The fact introduce by theorem 4 is obvious because the delta-similarity is defined using an inequality, therefore it is not reflexive, a necessary property of equivalence relations as stated in definition 72 on page 217. In practice this means that the classes of elements from $\mathfrak{A}$, created with the delta-similarity relation, would not form a partition of $\mathfrak{A}$. The idea behind the definition of this relation is however useful. If we want to identify similar states, we have better chances starting with the states that are the *less different* which lead to the concept of minimal state difference and then set difference.

Relaxation For our analysis, the fact that the relation is not reflexive and therefore unable to partition the state space might only be translated in performance loss. In order to perform the merges we need good candidates. If the classes created by a certain relation are good candidates, it will enable to apply more state merges. We therefore do not only search to define equivalence relations.

Plan In the following we extend the delta-similarity to equivalence relations.

Proof 5 *(delta-similarity is not an R-equivalence relation). The proof is very simple, as listed below. In order to prove that the delta-similarity is not an R-equivalence relation, we must prove that it fails to respects the reflexivity, symmetry or transitivity properties.*

- *reflexivity: We must prove that* $\forall \mathfrak{A}_i \in \mathfrak{A} \implies \mathfrak{A}_i \nsim_\Delta \mathfrak{A}_i$.
  Let us suppose that

  $$\exists \mathfrak{A}_i \in \mathfrak{A} \mid \mathfrak{A}_i \sim_\Delta \mathfrak{A}_i \Leftrightarrow \exists \mathfrak{A}_i \in \mathfrak{A} \mid \Delta = \left| \mathfrak{A}^i \setminus \mathfrak{A}^i \right|.$$

  implicitly $\forall \Delta$ *which is obviously false as* $\left| \mathfrak{A}^i \setminus \mathfrak{A}^i \right| = 0$*, therefore we have a contradiction* $\implies \Leftarrow$*, therefore*

  $$\forall \mathfrak{A}_i \in \mathfrak{A} \implies \mathfrak{A}_i \nsim_\Delta \mathfrak{A}_i. \blacksquare$$

- *symmetry: However, the relation is symmetric. We must prove that* $\forall \mathfrak{A}_i, \mathfrak{A}_j \in \mathfrak{A}.\mathfrak{A}_i \sim_\Delta \mathfrak{A}_j \implies \mathfrak{A}_j \sim_\Delta \mathfrak{A}_i$.
  Let us suppose that:

  $$\exists \mathfrak{A}_i, \mathfrak{A}_j \in \mathfrak{A}.\mathfrak{A}_i \sim_\Delta \mathfrak{A}_j \implies \mathfrak{A}_j \nsim_\Delta \mathfrak{A}_i \Leftrightarrow \exists \mathfrak{A}_i, \mathfrak{A}_j \in \mathfrak{A}.\mathfrak{A}_i \setminus \mathfrak{A}_j \neq \mathfrak{A}_j \setminus \mathfrak{A}_i$$

  which is a contradiction $\implies \Leftarrow$ *on behalf of the set difference symmetry, therefore*

  $$\forall \mathfrak{A}_i, \mathfrak{A}_j \in \mathfrak{A}.\mathfrak{A}_i \sim_\Delta \mathfrak{A}_j \implies \mathfrak{A}_j \sim_\Delta \mathfrak{A}_i. \blacksquare$$

- *transitivity: obvious, following the same reasoning as above.*

We use the concept of *delta-similarity* to further distinguish state relations of $\mathcal{M}$, using the delay δ into the following sub-categories:

- *weak similarity*,
- *strong similarity*.

Statement 5 *Besides the quantitative definition that we are about to formulate, the distinction between weak and strong similarity can also be made by model-dependant qualitative approach. The discrimination will be made amongst locations that will have a bigger impact on the state difference (for example locations that store the pipeline content and locations storing the data memory content).*

Definition 68 (Strong similarity.) Let $\mathfrak{A}^i$ and $\mathfrak{A}^j$ be two delta-similar states, $\mathfrak{A}^i \sim_\delta \mathfrak{A}^j$, we say that they are strongly similar and we note:

$$\mathfrak{A}^i \simeq_\delta \mathfrak{A}^j \Leftrightarrow \Delta \leq \epsilon_\Delta,$$

where ϵ_Δ is the tolerated difference coefficient that might be platform dependant.

Theorem 5 *The strong similarity is an R-equivalence relation.*

Proof 6 *(Strong similarity is an R-equivalence relation). The proof follows the same reasoning as above. In order to prove that the strong similarity is an R-equivalence relation, we must prove that it respects the reflexivity, symmetry and transitivity properties.*

- *reflexivity: We must prove that* $\forall \mathfrak{A}_i \in \mathfrak{A} \implies \mathfrak{A}_i \simeq_\Delta \mathfrak{A}_i$.
  Let us suppose that

  $$\exists \mathfrak{A}_i \in \mathfrak{A} \mid \mathfrak{A}_i \not\simeq_\Delta \mathfrak{A}_i \Leftrightarrow \exists \mathfrak{A}_i \in \mathfrak{A} \mid \Delta > \left|\mathfrak{A}^i \setminus \mathfrak{A}^j\right|.$$

  implicitly $\forall \Delta$ *which is obviously false as* $\left|\mathfrak{A}^i \setminus \mathfrak{A}^j\right| = 0$*, therefore we have a contradiction* $\implies \Leftarrow$*, therefore*

  $$\forall \mathfrak{A}_i \in \mathfrak{A} \implies \mathfrak{A}_i \simeq_\Delta \mathfrak{A}_i.\blacksquare$$

- *symmetry: We must prove that* $\forall \mathfrak{A}_i, \mathfrak{A}_j \in \mathfrak{A}.\mathfrak{A}_i \simeq_\Delta \mathfrak{A}_j \implies \mathfrak{A}_j \simeq_\Delta \mathfrak{A}_i$.
  Let us suppose that:

  $$\exists \mathfrak{A}_i, \mathfrak{A}_j \in \mathfrak{A}.\mathfrak{A}_i \simeq_\Delta \mathfrak{A}_j \implies \mathfrak{A}_j \not\simeq_\Delta \mathfrak{A}_i \Leftrightarrow \exists \mathfrak{A}_i, \mathfrak{A}_j \in \mathfrak{A}.\mathfrak{A}_i \setminus \mathfrak{A}_j \neq \mathfrak{A}_j \setminus \mathfrak{A}_i$$

  which is a contradiction $\implies \Leftarrow$ *on behalf of the set difference symmetry, therefore*

  $$\forall \mathfrak{A}_i, \mathfrak{A}_j \in \mathfrak{A}.\mathfrak{A}_i \simeq_\Delta \mathfrak{A}_j \implies \mathfrak{A}_j \simeq_\Delta \mathfrak{A}_i.\blacksquare$$

- *transitivity: obvious, following the same reasoning as above.*

Symmetrically, the week similarity definition is:

Definition 69 (Weak similarity.) Let $\mathfrak{A}^i$ and $\mathfrak{A}^j$ be two delta-similar states, $\mathfrak{A}^i \sim_\delta \mathfrak{A}^j$, we say that they are weakly similar and we note:

$$\mathfrak{A}^i \approx_\delta \mathfrak{A}^j \Leftrightarrow \Delta > \epsilon_\Delta,$$

where ϵ_Δ is the tolerated difference coefficient that might be platform dependant.

Statement 6 *In practice we will use the reflexive closure of the weak similarity relation*

$$\mathfrak{A}^i \approx_\delta \mathfrak{A}^j \Leftrightarrow \Delta \geq \epsilon_\Delta + 1, \tag{7.9}$$

Theorem 6 *The strong similarity is an R-equivalence relation.*

Proof 7 *(The reflexive closure of the weak similarity relation is an R-equivalence relation). See proof 6.*

In the following we will always refer to the reflexive closure of the weak similarity as the weak similarity.

Theorem 7 *(Congruence on the HiTAsm algebras) The δ-similarity R-equivalence relation is a congruence on the algebra $\mathfrak{A}$ of the HiTAsm.*

Statement 7 *(Existence of identical states) At a certain moment of the analysis, regardless of the execution history, $\bigcup_i(\omega^{(i)}, u^{(i)})$, we can arrive in an identical or similar context $(\omega^{(j)}, u^{(j)})$*

Statement 8 *(Existence of similar states) The existence of identical states implies that their has been a common point in the execution time where non-identical state, called similar states, lead to an identical context $(\omega^{(j)}, u^{(j)})$, regardless of the execution history, $\bigcup_i(\omega^{(i)}, u^{(i)})$.*

Statement 9 *(Identification of similar states) A timed HiTAsm machine holds not only a* (*location* $\Longrightarrow$ *value*) *mapping set but also the set of delayed updates* $\bigcup_{i}^{n} u_i$ *that are programmed to be fired after the delay* δ_i*. Therefore it is obvious that some timed updates that have not yet been fired might bring two different paths of the machine in an identical or similar state.*

The importance of statement 9 is crucial for the convergence of the analysis in the context of the state space explosion. It not only says that we can find similar states to merge but it implies that we can identify similar states even before they are actual computed *i.e*. before the timed update sets would be applied to the state in order to generate the new, and therefore similar, state. The advantages are twofold, less states to analyze in the future, as paths will be merged and also less computations of the next states.

7.4.3 State Merging

The major advantage of the symbolic execution is that it naturally generates every feasible path. This can also be a drawback for an industrial real-life program as it generates a combinatorial explosion. Handling this explosion with sufficient accuracy still remains challenging today. The existing techniques range from efficient representation to state merging algorithms. We choose to revisit the merging techniques, developing and extend them to our processor model.
State merging refers to the actual fusion of two execution states and eliminates all the successors of one state therefore whole execution paths. The state merging building blocks fall into the following categories:

- state merging definition;
- merge candidates identification;
- merge candidates validation.

In order to perform merges, a notion of state must be defined, a certain metric associated and an order or pre-order defined. The order on states will enable their comparison. Based

on this comparison criteria for states categorization can be defined.
We can compare the state merging with the execution of an `if-then-else` statement:

- the generated traces lead to different states. The merging is forbidden and the execution continues along the two paths.

- the generated traces lead to strongly similar states. The merging is authorised without affecting the accuracy. This case arises when states from the two generated branches are not dependent of the state split and are similar.

- the generated traces lead to weakly similar states. In order to validate the merge, a parallel execution is started (in our case, the Prediction Module is activated) for a few steps and results thereafter compared. If only a location or a short number of location values are different and their impact is judged insignificant for the global execution, then the merge is authorised.

Our merging method is implemented in such a nature to identify, during execution, the states on which the generated tree is likely to fold (highly similar statements). This implies that at some point, the analysis must stop so that the merger method traverses the symbolic tree. The algorithms backtracks starting with the most recent states in the history and searches for similar states that would be validated for the merge.
In the following we show that our model provides some advantages that are beneficial to state merging identification and validation.
The merging of two similar states can be informally defined as the regroupment of state elements into a single state. Multiple mergings can be defined according to the different merging categories.

Definition 70 (State merging.) Let m_i and m_j be two states of the HiTAsm machine $\mathcal{M}$ we define the merge state:

$$m_{ij} \stackrel{\text{def}}{=} m_i \uplus m_j.$$

7.5 Prediction Module (PM)

Analyzing all the reachable states of the processor through the conjoint symbolic execution of the hardware model and the program leads to a state space explosion. The *prediction module*, depicted in figure 7.8, is a feature aimed at reducing the explored paths by means of *state merging*.

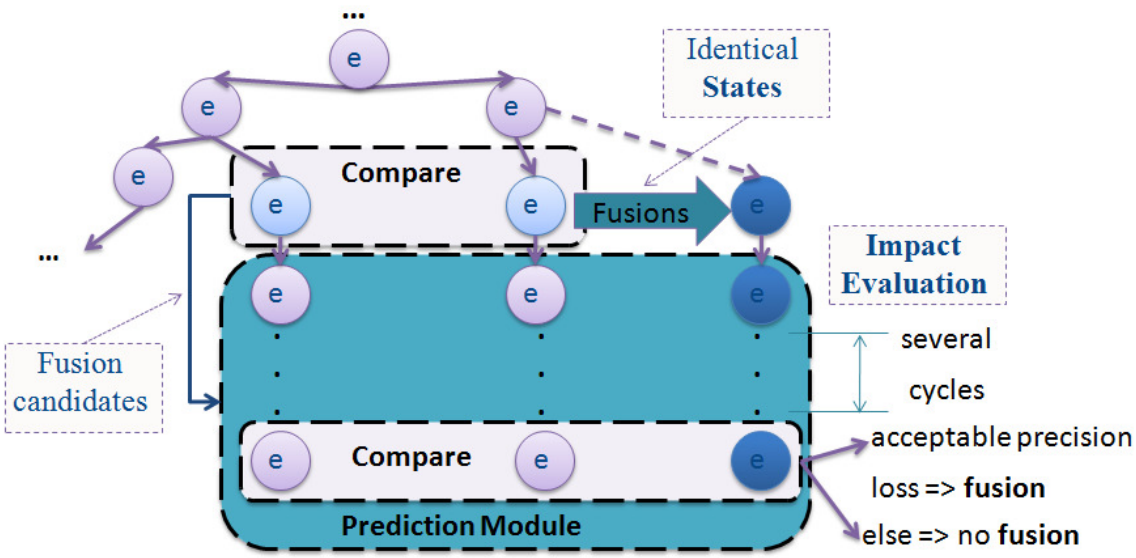


Figure 7.8: The Dynamic Fusion - snapshot of the Prediction Module

This module dynamically searches through the execution traces either the states to be merged or good candidates that will be validated through a specific process. In the following we present the main steps of the PM:

1. Forward lookup - consists in doing merge prediction through the execution of a few steps starting from the current states.

    - The starting point is two state candidates for merging. The algorithm analyzes the impact of a merge on the future execution. The impact is quantified through a parallel execution. After a few parallel steps the difference of the timed states

prior to the merge and after the merge is evaluated. Given the result is satisfactory, the merge prediction is validated and the states merged, otherwise it is discarded. *Please note that this decision will also be used to train state comparison patterns by raising or lowering their grades based on the prediction result*.

2. Backward lookup - consists in searching the analysis history in order to identify the earliest possible merging point starting from a state merge. This strategy requires to keep track of a few previous execution steps.

   - One of the starting points of the PM is the merging point of two identical states $\mathfrak{A}^i =_\delta \mathfrak{A}^j$. These two identical states have a different execution history, which means that they were generted by two non-identical states. The PM will try to identify two common ancestors, as high in the history as possible, that are similar and merge them.
   - Another starting point of the PM is the merging point of two strongly similar states $\mathfrak{A}^i \simeq_\delta \mathfrak{A}^j$. Examining their execution history might lead to identifying strongly similar states and an earlier merge. The prediction can further continue by applying the previous algorithm.

Merge identification goal Early identification of merge points is of great importance for the state space reduction. States split potentially at every analysis step because of unknown values or conditional constraints, generating new analysis paths. If the states that generated them were identical to begin with, the path will be too, and the analyzer has no way of knowing it. Therefore an early merge reduces exponentially many states compared to a later merge.

7.5.1 Prediction Module search strategies

The PM deals with the merging of states. In order to proceed to the merge, states must respect a certain criteria defined in section 7.4.2. The identification of states that respect that criteria is another important step in the analysis.

7.5.1.1 Product domain search

One of the simplest search routine is to start by assuming all states can potentially be merged. Based on this assumption, we therefore proceed to merging any two states $\mathfrak{U}^i$ and $\mathfrak{U}^j$ of the machine $\mathcal{M}$. The idea is to determine, after a few execution steps, if the states were really good merge candidates. The number of execution steps is dependent on the pipeline depth and other processor characteristics. In order to validate the prediction, we simulate a parallel execution of the machine without the merged states, $\mathcal{M}$ and with the merged states, $\mathcal{M}^{\#}$ starting from the current execution point i. After a bounded number of execution steps, p, the prediction is verified by comparing the similarity of the two machine states. If the states are strongly-similar, the prediction is validated, the states merged and the execution resumed from the validation point $i + p$ with the machine $\mathcal{M}^{\#}$ otherwise with the machine $\mathcal{M}$ whose evolutions were already computed.

We acknowledge that the nature of this strategy is quite costly, however the intuition is that more and more machine states will be similar as the execution advances and creates similar state constraints.
If the prediction was successful, we can further backtrack the execution history in search of the most general common similar states that lead to those strongly similar states. Early identification and merge of states has a great interest because execution is rarely linear but branch depending on various conditions. Therefore all those paths can be eliminated by the merge prior to their creation.

Theorem 8 *(State comparisons complexity) The complexity of the state merging algorithm of a HiTAsm machine $\mathcal{M}$ with n states, where $n = |\mathcal{S}|$ and $\mathcal{S}$ is the number of states is of the order of $O(n^2)$.*

Proof 8 *Let $\mathscr{S}$ be the set of states of the HiTAsm machine $\mathcal{M}$ to be explored, $\mathscr{S} = \{s \in \mathcal{M}\}$, m_{test} be the merge decision function:*

$$m_{test} : \mathcal{M} \times \mathcal{M} \longrightarrow \{0, 1\}$$

and R_m be the set with the all results of m_{test} on $\mathcal{M}$:

$$R_m = \left\{ m_{test}(s_i, s_j) \mid \forall s_i, s_j \in \mathscr{S} \right\}$$

then we have $|R_m| = |\mathcal{S}|^2$.

Product domain search - improvement The Prediction Module can decide to keep execution snippets leading to the merging of (strongly similar) states. This execution history will be used to generate state or path patterns having a great prediction success likelihood. During the execution states will be matched against those patterns in order to directly choose more suitable candidates. This actually translates in promoting the similarity concept form states to traces. In other words, combination of instruction and their order can lead to identical states which is very plausible.

7.5.1.2 Relational domain search

The HiTAsm functions and relations can be used to systematically partition state spaces into similarity categories called *equivalence classes*. A distance is defined that will guide the search only between elements of two different equivalence classes.
The set of equivalence classes $\bigcup_i \mathscr{E}_i$ holds all the state of the machine $\mathcal{M}$ and group in the same set, states at a distance $\mathcal{D}$.

7.6 Equivalence Classes

The conjoint symbolic execution generates all the reachable states of a processor running a certain program for all possible inputs. This means it generates a great number of state. When analyzing the outcome of all possible states, computational resources will come short very fast. We previously introduced the concept of state merging. This implies a search through all the state space.
Based on the definition of similarity we can identify if two states can be merged or not:

$$s_1 \simeq_\delta s_2 \implies s_1 \uplus s_2$$

which means that we must first verify if the states are similar, $s_1 \simeq_\delta s_2$ prior to the merge operation $s_1 \uplus s_2$.

7.6.1 Mathematical foundation

Definition 71 (Equivalence class) Let X be a set and $\sim$ an equivalence relation on X, the equivalence class of an element e in X is the subset of all elements in X which are equivalent to e with regard to the equivalence relation $\sim$.

Let us now formally define the equivalence class through its properties and introduce the equivalence class notation:

Definition 72 (Equivalence class) The equivalence class is defined by the binary relation $\sim$, called the equivalence relation, that satisfies the following properties:

- *reflexivity*: $\forall e \in X \implies e \sim e$;
- *symmetry*: $\forall e_i, e_j \in X . e_i \sim e_j \implies e_j \sim e_i$;
- *transitivity*: $\forall e_i, e_j, e_k \in X e_i \sim e_j, e_j \sim e_k \implies e_i \sim e_k$.

We denote the equivalence class of an element e by $[e]$ defined as the set:

$$[e] \stackrel{\text{not}}{=} \{x \in X \mid e \sim x\}$$

of elements that are related to e by the binary relation $\sim$.

Definition 73 (R-equivalence class) Let us define the R-equivalence class, as en equivalence class defined by the properties in definition 72 and by the R-equivalence relation $\sim_R$. We denoted it by $[e]_R$ defined as the set:

$$[e]_R \stackrel{\text{not}}{=} \{x \in X \mid e \sim_R x\}\,.$$

7.6.1.1 Complexity study

In order to scale and speed up our algorithm, highly discriminating equivalence relations will have a plus in determining the state partitionment using the equivalence classes. If we manage to partition the space in k equivalence classes, we can write the total number of states as $|\mathcal{S}| = |\mathcal{S}_1| \ldots |\mathcal{S}_k|$ and the total number of state merge identification operations decreases from n^2 to $n_1^2 + \ldots n_k^2$, where n_i is the cardinal of the set $\mathcal{S}_i$, $n_i = |\mathcal{S}_i|$ and $n = \sum_{i=1}^{k} n_i$. However, they are many possible equivalence classes as they form the power set of the state set $\mathcal{S}$.

Theorem 9 *(Equivalence class determination). The complexity of the equivalence classes determination of a HiTAsm machine $\mathcal{M}$ with n states, where $n = |\mathcal{S}|$ and $\mathcal{S}$ is the number of states is of the order of $O(2^n)$.*

In order to limit this huge number of comparisons, we introduce a way, based on the HiTAsm definition, to pinpoint comparisons that are more likely to result in a merge.
A state i of a HiTAsm machine $\mathcal{M}$ for the HiTAsm $\mathfrak{A}$ is defined by the couple $(\mathfrak{A}^{(i)}, U^{\mathfrak{A}^{(i)}})$, where $U^{\mathfrak{A}^{(i)}}$ is the timed update set of $\mathfrak{A}$ at state i. As it was defined in the previous chapters, the timed update sets accumulate updates that are programmed later in the execution. Our idea is to compare the update sets and identify if two independent states have the same or similar update set. This means that in the update set, the same locations are programmed to change at a certain point, potentially giving a similar state. This means that we no longer treat states equally with regard to the state merging search, but we categorize states from their generation in different comparable sets. The degree of difference gives as the distance $\mathcal{D}$ used to partition the state space with minimum computational cost.

7.6.2 A Formal View on State Partitioning

The state partitioning is based on the notion of distance previously introduced that we will formally define in the following. This section is an extension of the HiTAsm state similarity to machine state similarity by comparing the updates sets additionally to the HiTAsm states.

Definition 74 (Merge candidates) Let i, j be two states of the machine $\mathcal{M} = (\mathfrak{A}, U^{\mathfrak{A}})$, $\mathfrak{A}^{(i)} \neq \mathfrak{A}^{(j)}$, we say that the two states are merge candidates if for :

$$\forall l_k \in U^{\mathfrak{A}^{(i)}}, l_l \in U^{\mathfrak{A}^{(j)}} \implies l_k = l_l.$$

Based on this definition we can define the first class of states consisting in the most likely merge candidates, ass the sets having the same locations programmed for update at the same time:

Definition 75 (Level 0 merge candidates) Let i, j be two states of the machine $\mathcal{M} = (\mathfrak{A}, U^{\mathfrak{A}})$, $\mathfrak{A}^{(i)} \neq \mathfrak{A}^{(j)}$we say that the two states are merge candidates if for :

$$\forall l_k \in U^{\mathfrak{A}^{(i)}}, l_l \in U^{\mathfrak{A}^{(j)}} \mid \delta_k^{(i)} = \delta_l^{(j)} \implies l_k = l_l.$$

We define the distance between two states with regard to the merging likeliness as a function dependent on the number of different locations in their associated timed update set at a certain time:

Definition 76 (Distance Set) Let i, j be two states of the machine $\mathcal{M} = (\mathfrak{A}, U^{\mathfrak{A}})$, $\mathfrak{A}^{(i)} \neq \mathfrak{A}^{(j)}$, we define the distance set of the two states:

$$D_{ij} \stackrel{\text{def}}{=} \bigcup_{k} l_k,$$

where

$$l_k \in U^{\mathfrak{A}^{(i)}}, l_l \in U^{\mathfrak{A}^{(j)}} \mid l_k \neq l_l, \delta_k^{(i)} = \delta_l^{(j)}.$$

In other words, D_{ij} is the set of locations that will be updated in a state with a certain delay δ and not in the other state.If the locations were equal in the two states prior to the update, then they might become different.
Given the fact that the difference is commutative, it is trivial to demonstrate the following property:

Property 12 *(Distance Set Computativity)* $D_{ij} = D_{ji}$.

The distance between two sets is based on definition 76.
Let us now define a measure for updates performed on the same locations in the same time. Statement 4 introduces the probability of similarity of two states based on the number of their different locations later defined in lemma 4.

Statement 10 *Two states that will get updates in the same time of all the locations that are different have the greatest probability of becoming identical or similar.*

Definition 77 (Equivalence Set) Let i, j be two states of the machine $\mathcal{M} = (\mathfrak{A}, U^{\mathfrak{A}})$, $\mathfrak{A}^{(i)} \neq \mathfrak{A}^{(j)}$, we define the equivalence set of the two states:

$$E_{ij}(\delta_{kl}) \stackrel{\text{def}}{=} \bigcup_{k} l_k,$$

where

$$l_k \in U^{\mathfrak{A}^{(i)}}, l_l \in U^{\mathfrak{A}^{(j)}} \mid l_k = l_l, \delta_k^{(i)} = \delta_l^{(j)}.$$

In other words $E_{ij}(\delta_{kl})$ is the set with all the common locations from state i and j that will be updated in the same time, after the delay (δ_{kl}).
We can now revisit the definition 75:

Definition 78 (Best merge candidates) Let i, j be two states of the machine $\mathcal{M} = (\mathfrak{A}, U^{\mathfrak{A}})$, $\mathfrak{A}^{(i)} \neq \mathfrak{A}^{(j)}$ and $\mathfrak{D}_{ij}$ be their difference, we define the most likely merge candidates as:

$$E_{ij}(\delta_{kl}) = \mathfrak{D}_{ij},$$

Definition 79 (Merge State Distance) Let $\mathcal{D}$ be the distance function between two machine states of the machine $\mathcal{M}$:

$$\mathcal{D} : \mathcal{M} \times \mathcal{M} \longrightarrow \mathbb{N}$$

and $\mathcal{M}_i, \mathcal{M}_j$ two states of the machine $\mathcal{M}$, we define the distance as the cardinal of the distance set, D_{ij} between the two states:

$$\mathcal{D}(\mathcal{M}_i, \mathcal{M}_j) = \left|D_{ij}\right|.$$

Finally, we can define the equivalence classes of the machine $\mathcal{M}$:

Definition 80 (Merge Equivalence Class) An equivalence class is a set of states that are equally distanced with regard to the state merging distance $\mathcal{D}$:

$$C_d(\mathcal{M}) = \left\{ \bigcup_k m_k \mid \forall i, j \in \{1, \ldots, k\}.\mathcal{D}(m_i, m_j) = d \right\},$$

where $C_d(\mathcal{M})$ is the merge equivalence class of states that are d-distanced.

In practice The nature of the state merging equivalence classes creates a hierarchical search strategy based on the distance level. The search starts among the least distanced states and proceed in the direction of the distance growth.

7.6.2.1 Processor dependent equivalence relation

As previously proved, the importance state partitioning through the identification of good equivalence classes is very important for the scalability and performances of the algorithm. Their success is tributary to the equivalence relation $\sim_R$.

To this end we previously introduced the global δ-similarity relation $\sim_\delta$ that partitions all the state space depending on their location value differences and programmed δ-update sets.
This is a very useful relation as it can be applied to any processor type and used as a general and adaptable equivalence relation.
The same formalism can however allow us to define ad-hoc distances and R-equivalence relations dependent of the processor's particular architecture.

Functional Congruence We had a look into the *functional congruence* to create equivalence classes for the Burst Buffer unit. The functional congruence is defined as a congruence of the form

$$f(x) \equiv g(x)(\textbf{mod } n),$$

where $f(x)$ and $g(x)$ are both integer polynomials. The functioning of the Burst Buffer implies that it can retrieve as many as four instructions and as less as one instruction per cycle. The behaviour can be explained through frame retrieval composed of four consecutive locations. The unit can retrieve a frame, however if the specified address is at the end of the frame (*i.e*. on the forth position of the frame) it will only retrieve a singe data. If the address corresponds to the second position, it will retrieve three, etc.
Therefore the Burst Buffer retrieves modulo 4 instructions depending on the value of the specified fetch address. Processor units are implemented as functions, therefore the translation of the functional congruence is quite straightforward.
Imprecisions on value are common in static analysis. Let us take the example of an imprecise burst address. In this case our analysis will either split in four cases or use the definition corresponding to a higher level of Burst Buffer component abstraction. The unit's behaviour affects the entire state.
In the case the analysis splits, we can use this functional congruence as the equivalence relation to partition the space.

Function image Another intuition is to use the equivalence relation:
Has the same image under a function on the elements of the domains of a function.

For example we could use a time function that applies to states and creates classes of states taking the same amount of time. Of even greater benefit would be to identify a time function that identifies states that took the same time to execute, δ_n, after n execution steps, when n is greater than the pipeline depth. Adding a certain order would enable us to fusion or disregard certain states without the downsides of the non-compositionality of the analysis.

7.7 Implementation

In this section we provide an algorithm corresponding to a high-level pseudocode implementation of the timing analyzer. Once again, the global architecture is depicted in figure 7.9.

7.7.1 Global algorithm

1. Start from the initial state: where all the components have the unknown value and *pc* is set to *true*
2. For every variable that we encounter and that we do not have the exact value, assign a symbolic value
3. Activate the first ASM model and then add the guard condition *g* to the *pc*
4. Choose from the *oracle* the appropriate version of the ASM modules
5. Compute the update set of the current step
6. Apply the update set (taking into account that some terms will have symbolic values)
7. Add the result of the update set to the global system state
8. Add the generated states to the collection of next states to be executed
9. Add the duration of the transition to the global time
10. Repeat from point 2. until the collection of next states is empty

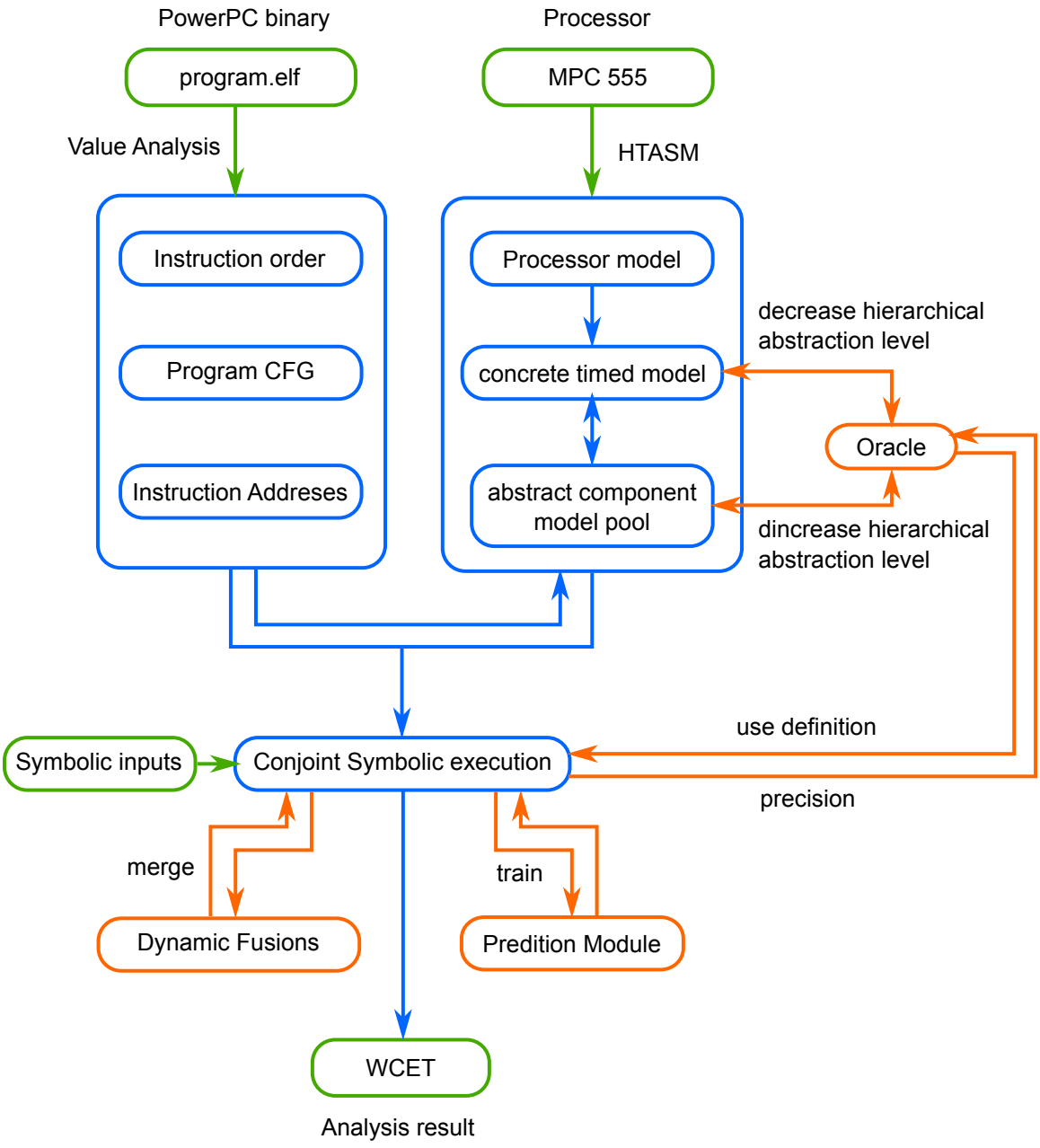


Figure 7.9: Global architecture of the WCET estimation tool

In algorithm 5 we present a high level view of the global algorithm that starts in the initial symbolic state and unfolds the execution path. At every new step, symbolic states will be assigned in a certain equivalence class where merge candidates will be identified.

7.7.2 Analysis termination

The state merging is crucial in the analysis state space reduction. In the ideal case, only identical states are merged which means that no accuracy loss is introduced. In most of the cases, the activation of only these types of merges is not sufficient and merges with accuracy loss must be introduced. Therefore strongly similar states will also be authorised for merging. Unfortunately pathological cases may arrive limiting the identification performance of such states. As the search proceeds, comparing additional states and exploring the history deeper, the computing resources may show their limitations.
In order to ensure that the analysis terminates in all cases with an estimation of the WCET, we can increase the weekly similar state merges threshold. This is achieved by gradually authorising the introduction of more and more inaccuracy or directly merging states that have a greater distance.
The algorithm evolves by relaxing at each step a similarity condition constraint until the available memory proves sufficient for the analysis.

7.8 Conclusions

Software verification and quality assurance process of embedded systems with hard real-time constraints are of great importance. Non-functional properties, such as timing, are highly dependent on the underlying hardware platform. Nevertheless, there is a rising demand to integrate more complex processors, such as the multi-cores, even though many problems are yet to be solved in single-cores. Powerful industrial WCET estimation tools available today can do nothing against the lack of information regarding the exact behavior of the platform or the nondeterministic behavior of certain units. Therefore the choice of the processor is crucial in ensuring the success of the system verification.

Algorithm 5: Dynamic Symbolic Execution Algorithm

input : initial location

1 $\mathcal{M}_{init} = (\mathfrak{A}_{init}, \phi, \text{True})$; Push (WorkList, $\mathcal{M}_{init}$);
2 **while** WorkList $\neq \phi$ **do**
3 Push (WorkList, Succ (Pop (WorkList)));
4 $(\mathfrak{A}_i, \mathcal{U}_i, \Psi_{PC}) \longleftarrow$ Pop (WorkList);
  /* The symbolic execution per se */
5 **if** *(*Rule *(*$\mathfrak{A}_i$*)* = SkipRule*)* $\vee$ *(*Rule *(*$\mathfrak{A}_i$*)* = UpdateRule*)* **then**
6 **if** *(*$\mathcal{U}_i \neq \phi$*)* **then**
7 $CT^{\mathfrak{A}_i} \longleftarrow CT^{\mathfrak{A}_i} = \delta^{\mathfrak{A}_i}_{min}$;
8 **foreach** *(*$(l, v, \delta) \in \mathcal{U}_i | \delta = \delta min$*)* **do**
9 $\mathcal{U}_{i+1} \longleftarrow \mathcal{U}_i \setminus (l, v, \delta)$;
10 $l^{\mathfrak{A}_{i+1}} = eval(v, \mathfrak{A}_i)$;
11 **if** *(*Rule *(*$\mathfrak{A}_i$*)* = UpdateRule*)* **then**
12 $\mathcal{U}_{i+1} \longleftarrow \mathcal{U}_i \bigcup_k$(Rule ($\mathfrak{A}_i$) (l_k, v_k, δ_k));
13 Push (WorkList, $(\mathfrak{A}_{i+1}, \mathcal{U}_{i+1}, \Psi_{PC})$);
14 **else if** *(*Rule *(*$\mathfrak{A}_i$*)* = ConditionalRule*)* **then**
15 *guard* = eval(cond(Rule ($\mathfrak{A}_i$)), $\mathfrak{A}_i$);
  /* *guard* is not a theorem */
16 **if** satisfiable *(*$\Psi_{PC} \wedge guard$*)* $\wedge$ satisfiable *(*$\Psi_{PC} \wedge \neg guard$*)* **then**
17 $\mathfrak{A}'_{i+1}$ = generate ($\mathfrak{A}_i$, *guard*);
18 $\mathfrak{A}''_{i+1}$ = generate ($\mathfrak{A}_i$, $\neg$*guard*);
19 Push (WorkList,
  $\{(\mathfrak{A}'_{i+1}, \mathcal{U}'_{i+1}, \Psi_{PC} \wedge guard), (\mathfrak{A}''_{i+1}, \mathcal{U}''_{i+1}, \Psi_{PC} \wedge \neg guard)\}$);
20 *evaluate the* True *and* False *case and the other rule types*
21 **foreach** *s in* Succ *(*cSE *(*$\mathfrak{A}$*)))* **do**
22 $C_d \longleftarrow$ Categorieze (s)
23 **foreach** *c in* $\bigcup C_d$ **do**
24 PredictionModule;
25 Merge *symbolic states*;

Chapter 8

Conclusions

The presented work establishes fundamental techniques for integrating ASMs and conjoint symbolic execution efficiently into an adaptable and modular static WCET analysis. We have presented the structure of a complete method for timing analysis with a controllable accuracy. Thanks to the conjoint symbolic execution, all the reachable states of the processor, running the binary, are generated. The possibility to make delayed transition is presented as a support for abstracting the processor components in order to achieve a more compact simulation. We have also introduced a formal framework and a language implementation that can be used to dynamically refine the components of the processor. The framework enables, during the analysis, dynamic hierarchical abstraction that can be made in both ways between the concrete and the more abstract definition. The use of novel state merging techniques and the Prediction Model solves some important scalability problems. The tool is currently in the implementation phases, however prototypes of a previous version, using state merging and a classical ASM processor model had given good results regarding the precision of the estimation on benchmarking code examples.

8.1 Original research (at a glance)

We present a novel and complete approach for timing analysis that exploits an original extension of the abstract state machines. The approach uses conjoint symbolic execution using the HiTAsm framework to precisely model the processor.

8.2 Industrial Applications and Future Research

The timing analysis method presented in this thesis makes the object of an ongoing implementation that will respond to a real industrial need and integrates into a product certification workflow. The choice of the case study presented in chapter 6 is also related to this industrial application, the Motorola MPC555 being used on several projects.
Through this choice we aim at providing a direct comparison with the results obtained not only through the in-house ad-hoc dynamic methods but also with a licensed commercial WCET tool that targets this platform.

The HiTAsm framework and its integration into the timing analysis provides modularized ways of identifying state merge candidates and more fundamentally guide the hierarchical level of abstraction. The first part can be achieved by implementing new techniques into the Prediction Model. The second extension could be made in the implementation of new Oracle strategies to better adapt dynamically to the symbolic execution context.

8.3 Outlook

In the WCET estimation of modern processors, taking into account the hardware in the analysis and its interactions with the program code is crucial in order to safely and precisely estimate the worst-case. The choice of the way the processor is modeled is also very important. To this end we have studied another method before arriving to the HiTAsm framework, [PHM11], approach based on timed SystemC waiting state automata (TWSA) that are similarly, symbolically executed in conjunction with the binary of the program.

The TWSA model has the advantage of being able to serve in validating both functional and non-functional properties of the hard real time systems as it guarantees both critical functional properties about the interactions between concurrent processes and non-functional properties especially the time constraints. Based on this model we also developped a method which starts from a SystemC code and provides a tight execution time upper bound.

The fact that the same analysis method could be used with another, fundamentally differ-

ent, processor modeling formalism proves that our approach is not only adaptable to other processors to analyze but also to the use of different ways and languages to simulate the processor's behavior.

Appendix A

Code listing

Listing A.1: Value Analysis result on a C code

```
1
2  program
3  -> 9595024
4  i 0x100003dc : 0x9421ffe0
5  w c 0x00ffffe0
6  i 0x100003e0 : 0x93e1001c
7  w c 0x00fffffc
8  i 0x100003e4 : 0x7c3f0b78
9  i 0x100003e8 : 0x801f000c
10 r c 0x00ffffec
11 i 0x100003ec : 0x2f80000a
12    goto -> 9660368
13 -> 9660368
14 i 0x100003f0 : 0x409d0014
15    else -> 9694064
16    then -> 9695088
17 -> 9694064
18 i 0x100003f4 : 0x813f000c
19 r c 0x00ffffec
20 i 0x100003f8 : 0x38090001
21 i 0x100003fc : 0x901f000c
22 w 0x00ffffec
23 i 0x10000400 : 0x48000010
24    goto -> 9699712
25 -> 9695088
26 i 0x10000404 : 0x813f000c
27 r c 0x00ffffec
28 i 0x10000408 : 0x3809ffff
29 i 0x1000040c : 0x901f000c
30 w 0x00ffffec
31    goto -> 9699712
32 -> 9699712
```

```
i 0x10000410 : 0x801f0008
r c 0x00ffffe8
i 0x10000414 : 0x2f800005
i 0x10000418 : 0x409d0014
   else -> 9693568
   then -> 9693248
-> 9693568
i 0x1000041c : 0x813f0008
r c 0x00ffffe8
i 0x10000420 : 0x38090001
i 0x10000424 : 0x901f0008
w c 0x00ffffe8
i 0x10000428 : 0x48000010
   goto -> 9701984
-> 9693248
i 0x1000042c : 0x813f0008
r c 0x00ffffe8
i 0x10000430 : 0x3809ffff
i 0x10000434 : 0x901f0008
w c 0x00ffffe8
   goto -> 9701984
-> 9701984
i 0x10000438 : 0x81610000
r c 0x00ffffe0
i 0x1000043c : 0x83ebfffc
r c 0x00fffffc
i 0x10000440 : 0x7d615b78
i 0x10000444 : 0x4e800020
   goto -> 9706912
-> 9706912

vertices
   9595024
   9660368
   9694064
   9695088
   9699712
   9693568
   9693248
   9701984
   9706912

edges
   9595024 -> 9660368
   9660368 -> 9694064
   9660368 -> 9695088
   9694064 -> 9699712
   9695088 -> 9699712
   9699712 -> 9693568
   9699712 -> 9693248
   9693568 -> 9701984
   9693248 -> 9701984
   9701984 -> 9706912
```

Bibliography

[a3it] Absint advance analyzer, http://www.absint.com/ait/.

[ABD+95] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed priority pre- emptive scheduling: an historical perspective. *Journal of Real-Time Systems*, 8:129–154, 1995.

[Aer11] Aeroflex Gaisler AB. *GR712RC - Dual-Core LEON3FT SPARC V8 Processor, User's Manual*, 2011.

[Aer12] Aeroflex. *UT699 LEON 3FT/SPARCTM V8 MicroProcessor, Functional Manual*, 2012.

[AMMS10] Sergei N Artëmov, Yuri Matiyasevich, Grigori Mints, and Anatol Slissenko. Simulation of Timed Abstract State Machines with Predicate Logic Model-Checking. *Ann. Pure Appl. Logic*, 162(3):173–174, 2010.

[Anl00] Matthias Anlauff. XASM- An Extensible, Component-Based Abstract State Machines Language. In Yuri Gurevich, PhilippW. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 69–90. Springer Berlin Heidelberg, 2000.

[ARM99] ARM. *AMBA Specification (Rev 2)*, 1999.

[BCP03] Guillem Bernat, Antoine Colin, and Stefan Petters. pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems. Technical report, 2003.

[BCS00] Danièle Beauquier, Tristan Crolard, and Anatol Slissenko. A Predicate Logic Framework for Mechanical Verification of Real-Time Abstract State State Machines: A Case Study with PVS, 2000.

[Ben11] Bilel Benhamamouch. *Calcul du pire temps d'execution - Methode formelle s'adaptant a la sophistication croissante des architectures materielles*. PhD thesis, ENSTA PArisTEch, 2011.

[BG97] Andreas Blass and Yuri Gurevich. The linear time hierarchy theorems for Abstract State Machines. *J. Universal Computer Science*, 3:247–278, 1997.

[BG11] A. Burns and D. Griffin. Predictability as an emergent behaviour. In Robert I. Davis and Linh T.X. Phan, editors, *4th Workshop on Compositional Theory and Technology for Real?Time Embedded Systems*, pages 27–29, 2011.

[Bil95] Patrick Billingsley. *Probability and Measure*. NY John Wiley and Sons, 1995.

[Bil99] Patrick Billingsley. *Convergence of Probability Measures*. New York John Wiley and Sons, Inc, 1999.

[BK08] Sven Bünte and Raimund Kirner. The acquaintance of hardware timing effects: A sine qua non to validate temporal requirements in embedded real time systems. *Junior Scientist Conference*, Nov. 2008.

[BM09a] B. Benhamamouch and B. Monsuez. Computing worst-case execution time (wcet) by symbolically executing a time-accurate hardware model. *International Journal of Design, Analysis and Tools for Circuits and Systems*, 1(1), 2009.

[BM09b] Bilel Benhamamouch and Bruno Monsuez. Computing worst case execution time (WCET) by Symbolically Executing a time-accurate Hardware Model. In *International MultiConference of Engineers and Computer Scientists*, volume II, pages 3–8, 2009.

[BMV08] Bilel Benhamamouch, Bruno Monsuez, and Franck Védrine. Computing wcet using symbolic execution. In *Proceedings of the Second International Conference on Verification and Evaluation of Computer and Communication Systems*, VECoS'08, pages 128–139, Swinton, UK, UK, 2008. British Computer Society.

[Bor03] E. Borger. The asm refinement method. *Formal Asp. of Comput.*, 15(2-3):237–257., 2003.

[bou] Bound-t time and stack analyser.

[BPPS00] Valerie Bertin, Michel Poize, Jacques Pulou, and Joseph Sifakis. Towards validated real-time software. In *In Proc. 12th Euromicro Conference of Real-Time Systems*. IEEE Computer Society Press, 2000.

[BS02] Danièle Beauquier and Anatol Slissenko. A First Order Logic for Specification of Timed Algorithms: Basic Properties and a Decidable Class. *Annals of Pure and Applied Logic*, 113(1–3):13–52, 2002.

[BS03] E. Borger and R. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[BT08] Sven Bünte and Michael Tautschnig. A benchmarking suite for measurement-based WCET analysis tools. In *First International Conference on Software Testing, Verification and Validation (ICST)*, Lillehammer, Norway, April 2008. IEEE Computer Society Press.

[BZK11] Sven Bünte, Michael Zolda, and Raimund Kirner. Let's get less optimistic in measurement-based timing analysis. In *Proc. 6th International Symposium on Industrial Embedded Systems (SIES'11)*, Västerås, Sweden, June 2011. IEEE.

[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*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CFG+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, pages 36–42, May 2010.

[CGSH+12] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F.J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 91–101, July 2012.

[cod11] Embedded software test, 2011.

[CP01] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *In In Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44., 2001.

[CR09] Sudipta Chattopadhyay and Abhik Roychoudhury. Unified cache modeling for WCET analysis and layout optimizations. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 47–56, 2009.

[CR11] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *Proceedings of the*

2011 IEEE 32nd Real-Time Systems Symposium, RTSS '11, pages 193–203, Washington, DC, USA, 2011. IEEE Computer Society.

[CRM10] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software 38; Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, New York, NY, USA, 2010. ACM.

[CRTM98] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – a revolution in on-board communications. Technical Report 1:9–19, Volvo Technology Report, 1998.

[CS08] Joëlle Cohen and Anatol Slissenko. Implementation of Sturdy Real-Time Abstract State Machines by Machines with Delays. In *Proc. of the 6th Intern. Conf. on Computer Science and Information Technology (CSIT'2007), September 24–28, 2007, Yerevan, Armenia. Organized by National Academy of Science of Armenia in cooperation with Test Technology Technical Council of IEEE Computer Society*. National Academy of Science of Armenia, 2008.

[Cul06] C. Cullmann. *Statische Berechnung sicherer Schleifengrenzen auf Maschinencode*. PhD thesis, Universität des Saarlandes, Saarbrücken, 2006.

[DSW95] Nathalie Drach, André Seznec, and Daniel Windheiser. Direct-mapped versus set-associative pipelined caches. In *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, PACT '95, pages 79–88, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[EPB+06] Jochen Eisinger, Ilia Polian, Bernd Becker, Alexander Metzner, Stephan Thesing, and Reinhard Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *DDECS*, pages 15–20. IEEE Computer Society, 2006.

[ERT06] ERTS. *Otawa, A framework for experimenting WCET computations*, 2006.

[FH08] C. Ferdinand and R. Heckmann. Worst-case execution time – a tool provider's perspective. In *11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing ISORC*, 2008.

[Fre00] Freescale Semiconductors. *MPC555 - MPC556 User's Manual*, 2000.

[Gan01] Jack Ganssle. Really real-time systems. In *In Proc. Embedded Systems Conference San Fransisco (ESC SF)*, 2001.

[Gau95] T. Gaul. An Abstract State Machine specification of the DEC-Alpha Processor Family. Technical report, University of Karlsruhe, 1995.

[GH96] Yuri Gurevich and JamesK. Huggins. The railroad crossing problem: An experiment with instantaneous actions and immediate reactions. In Hans Kleine Bning, editor, *Computer Science Logic*, volume 1092 of *Lecture Notes in Computer Science*, pages 266–290. Springer Berlin Heidelberg, 1996.

[GP07] Susanne Graf and Andreas Prinz. Time in State Machines. *Fundamenta Informaticae*, 77(1-2):143–174, May 2007.

[Gur95a] Y. Gurevich. *Evolving Algebras 1993: Lipari Guide, Specification and Validation Methods*. Oxford University Press, Inc., 1995.

[Gur95b] Yuri Gurevich. Evolving algebras 1993: Lipari guide. pages 9–36, September 1995.

[Gur00] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. Technical report, Microsoft Research, 2000.

[GZ00] Gerhard Goos and Wolf Zimmermann. Verifying compilers and asms or asms for uniform description of multistep transformations, 2000.

[HC97] James K Huggins and David Van Campenhout. Specification and Verification of Pipelining in the ARM2 RISC Microprocessor. *ACM Transactions on Design Automation of Electronic Systems*, 3:563–580, 1997.

[HJK+11] Andreas Holzer, Visar Januzaj, Stefan Kugele, Boris Langer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Seamless testing for models and code. In *FASE 2011*, 2011.

[HLTW03] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003.

[HP08] Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS '08, pages 456–466, Washington, DC, USA, 2008. IEEE Computer Society.

[HP09] Damien Hardy and Isabelle Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In Laurent George and Maryline Chetto andMikael Sjodin, editors, *17th International Conference on Real-Time and Network Systems*, pages 45–54, Paris, France, 2009.

[HPP09] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 68–77, Washington, DC, USA, 2009. IEEE Computer Society.

[HSTV08] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, Lecture Notes in Computer Science, Princeton, NJ, USA, July 2008. Springer.

[HSTV09] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Query-driven program testing. In Neil D. Jones and Markus Müller-Olm, editors, *Proceedings of the Tenth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, volume 5403 of *Lecture Notes in Computer Science*, pages 151–166, Savannah, GA, USA, January 2009. Springer.

[HSTV10] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. How did you specify your test suite ? In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, pages 407–416, Antwerp, Belgium, September 2010. ACM.

[HSTV11] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. An introduction to test specification in FQL. In Sharon Barner, Daniel Kroening, and Orna Raz, editors, *Proceedings of Haifa Verification Conference (HVC 2010)*, volume 6504 of *Lecture Notes in Computer Science*, pages 9–22. Springer, 2011.

[IBM05] IBM. *PowerPC User Instruction Set Architecture - Book I*, 2.02 edition, 2005.

[Int10] International Electrotechnical Commission. *IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems*, 2010.

[Iro] Irony - .net language implementation kit.

[Ive98] Anders Ive. Runtime performance evaluation of embedded software. In *In Presented at the Eighth Nordic Workshop on Programming Environment Research*, 1998.

[JHE04] M. Jersak, R. Henia, and R. Ernst. Context-aware performance analysis of efficient embedded system design. In *Automation and Test in Europe Conference (DATE)*, 2004.

[KF12] Daniel Kästner and Christian Ferdinand. Static verification of non-functional software requirements in the ISO-26262. In *Automotive - Safety & Security 2012, Sicherheit und Zuverlässigkeit für automobile Informationstechnik, 14.-15. November 2012, Karlsruhe, Proceedings*, pages 39–53, 2012.

[KFM+11] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. In *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems*, ECRTS '11, pages 3–12, Washington, DC, USA, 2011. IEEE Computer Society.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[KKP10] Albrecht Kadlec, Raimund Kirner, and Peter Puschner. Avoiding timing anomalies using code transformations. In *Proc. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 123–132, May. 2010.

[KKPly] R. Kirner, A. Kadlec, and P. Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 119–128, July.

[KZ11] Raimund Kirner and Michael Zolda. Compiler support for measurement-based timing analysis. In *Proc. 11th International Workshop on Worst-Case Execution Time Analysis*, Porto, Portugal, July 2011. OCG.

[LDM+12] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Syst.*, 48(6):638–680, November 2012.

[Lis14] Bjorn Lisper. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, volume 8803 of *Lecture Notes in Computer Science*, pages 482–485. Springer Berlin Heidelberg, 2014.

[LL73a] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[LL73b] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard- real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. In *Science of Computer Programming*, 2007.

[LNBCG11] Yue Lu, T. Nolte, I. Bate, and L. Cucu-Grosjean. A trace-based statistical worst-case execution time analysis of component-based real-time embedded systems. In *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–4, Sept 2011.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LS99a] T. Lundqvist and P. Stenstrom. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(183-207), 1999.

[LS99b] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 12–, Washington, DC, USA, 1999. IEEE Computer Society.

[Lun02] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Goteborg, 2002.

[Mic] University of michigan, asm homepage.

[MNS13] A. Melani, E. Noulard, and L. Santinelli. Learning from probabilities: Dependences within real-time systems. In *18th IEEE International Conference on Emerging Technologies and Factory Automation*, 2013.

[MRR04] Wolfgang Muller, Jurgen Ruf, and Wolfgang Rosenstiel. An asm based systemc simulation semantics. In Wolfgang Muller, Wolfgang Rosenstiel, and Jurgen Ruf, editors, *SystemC*, pages 97–126. Springer US, 2004.

[Mue94] F. Mueller. *Static cache simulation and its applications*. PhD thesis, Departement of computer Computer Sciences, Florida State University, 1994.

[OL07] M. Ouimet and K. Lundqvist. The timed abstract state machine language: Abstract state machines for real-time system engineering. *JUCS*, 2007.

[OL08] Martin Ouimet and Kristina Lundqvist. The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering. *Journal of Universal Computer Science*, 14(12):2007–2033, June 2008.

[PB12] V. A. Paun and Monsuez B. Adaptable and Precise Worst Case Execution Time Estimation Tool. In *LCTES WiP Session*, page 4, 2012.

[PHM11] V. A. Paun, N. Harrath, and B. Monsuez. A WCET Estimation Workflow Based on the TWSA Model of SystemC Designs. In *The 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, November 2011.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[PMB13a] V. A. Paun, B. Monsuez, and P. Baufreton. Hierarchical Timed Abstract State Machines for Hard Real-Time Embedded Processors. In *VECoS*, page 12, 2013.

[PMB13b] V. A. Paun, B. Monsuez, and P. Baufreton. Hierarchical Timed Symbolic Abstract State Machines for precise WCET estimation. In *RTCSA WiP*, page 2, 2013.

[PMB13c] V. A. Paun, B. Monsuez, and P. Baufreton. On the Determinism of Multicore Processors. In Christine Choppy and Jun Sun, editors, *1st French Singaporean Workshop on Formal Methods and Applications (FSFMA 2013)*, volume 31 of *OpenAccess Series in Informatics (OASIcs)*, pages 32–46, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[PP07] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[PQnC+09] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. *SIGARCH Comput. Archit. News*, 37(3):57–68, June 2009.

[Pug91] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[pWC] Probabilistic worst case execution time analysis.

[Rad] Radio Technical Commission for Aeronautics. *DO-178B Software Considerations in Airborne Systems and Equipment Certification*.

[Rei09] Jan Reineke. *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity*. PhD thesis, Saarland University, 2009.

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2):99–122, November 2007.

[RS09] Jan Reineke and Rathijit Sen. Sound and Efficient WCET Analysis in the Presence of Timing Anomalies. In Niklas Holsti, editor, *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–11, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6.

[RWT+06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, 2006.

[Sch09] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16. IEEE Computer Society, 2009.

[SHP12] Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, DOI: 10.1007/s11241-012-9159-8:1–28, 2012. doi: 10.1007/s11241-012-9159-8.

[SPA92] SPARC International Inc. *SPARC V8 architecture manual, Revision SAV080SI9308*, 1992.

[SV07] A. Slissenko and P. Vasilyev. Simulation of timed abstract state machines with predicate logic model-checking. *JUCS*, 2007.

[The04] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universitat des Saarlandes, Postfach 151141, 66041 Saarbrucken, 2004.

[Tim] Timesys embedded linux.

[U.S11] U.S. Department of Transportation Federal Aviation Administration. *Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 5 Report, DOT/FAA/AR-11/5*, 2011.

[vHHL+11] Reinhard von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Ploedereder, Reinhard Wilhelm, Armelle Bonenfant, Hugues Casse, Sven Bünte, Wolfgang Fellger, Sebastian Gepperth, Jan Gustafsson, Benedikt Huber, Nazrul Mohammad Islam, Daniel Kästner, Raimund Kirner, Laura Kovacs, Felix Krause, Marianne de Michiel, Mads Christian Olesen, Adrian Prantl, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Simon Wegener, Michael Zolda, and Jakob Zwirchmayr. WCET tool challenge 2011: Report.

In *Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Porto, Portugal, July 2011.

[VLX03] Xavier Vera, Björn Lisper, and Jingling Xue. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, RTSS '03, pages 154–, Washington, DC, USA, 2003. IEEE Computer Society.

[WD95] Chip Weems and Steve Dropsho. Real-time risc processing, 1995.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, D. Whalley S. Thesing, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller T. Mitra, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7, 2008.

[Win] Embedded timing test.

[WKPR05a] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*, pages 295 – 303, sept. 2005.

[WKPR05b] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Proceedings of the Fifth International Conference on Quality Software*, QSIC '05, pages 295–306, Washington, DC, USA, 2005. IEEE Computer Society.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[WSE02] Fabian Wolf, Jan Staschulat, and Rolf Ernst. Hybrid cache analysis in running time verification of embedded software. *Design Automation for Embedded Systems*, 7(3):271–295, 2002.

[WW08] Reinhard Wilhelm and Björn Wachter. Abstract interpretation with applications to timing validation. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 22–36, Berlin, Heidelberg, 2008. Springer-Verlag.

[You82] S. Young. *Real Time Languages: Design and Development*. Elis Herwood, 1982.

[YZ07] Jun Yan and Wei Zhang. Hybrid multi-core architecture for boosting single-threaded performance. *SIGARCH Comput. Archit. News*, 35(1):141–148, March 2007.

[YZ08] Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 80 –89, april 2008.

[ZBK10] Michael Zolda, Sven Bünte, and Raimund Kirner. Context-sensitivity in ipet for measurement-based timing analysis. *4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'10)*, October 2010.

[ZBK11] Michael Zolda, Sven Bünte, and Raimund Kirner. Context-sensitive measurement-based worst-case execution time estimation. In *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11)*, Toyama, Japan, August 2011. IEEE. Accepted.

[ZK08] Michael Zolda and Raimund Kirner. Divide and measure: Cfg segmentation for the measurement-based analysis of resource consumption. In *Junior Scientist Conference 2008*, pages 117–118, Vienna, Austria, November 2008. Technische Universität Wien.

[Zol08] Michael Zolda. INFER: Interactive timing profiles based on bayesian networks. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis*, Oct. 2008.

[ZY09] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 455 –463, aug. 2009.