Bruno Naspolini Green

Hardware-Based Approach to Support Mixed-Critical Workload Execution in Multicore Processors

Porto Alegre - RS, Brasil

2015

Bruno Naspolini Green

Hardware-Based Approach to Support Mixed-Critical Workload Execution in Multicore Processors

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica do Rio Grande do Sul, como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Sinais, Sistemas e Tecnologia da Informação
Linha de Pesquisa: Sistemas de Computação.

Pontifícia Universidade Católica do Rio Grande do Sul – PUCRS

Faculdade de Engenharia

Programa de Pós-Graduação em Engenharia Elétrica

Advisor: Fabian Luis Vargas

Co-Advisor: Aurélio Tergolina Salton

Porto Alegre - RS, Brasil

2015

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

HARDWARE-BASED APPROACH TO SUPPORT MIXED-CRITICAL WORKLOAD EXECUTION IN MULTICORE PROCESSORS

CANDIDATO: BRUNO NASPOLINI GREEN

Esta Dissertação de Mestrado foi julgada para obtenção do título de MESTRE EM ENGENHARIA ELÉTRICA e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica do Rio Grande do Sul.

DR. FABIAN LUIS VARGAS - ORIENTADOR

DR. AURELIO TERGOLINA SALTON - CO-ORIENTADOR

BANCA EXAMINADORA

DRA. FERNANDA GUSMÃO DE LIMA KASTENSMIDT - DO PROGRAMA DE PÓS GRADUAÇÃO EM MICROELETRÔNICA - PGMICRO - UFRGS

DRA. LETÍCIA MARIA BOLZANI POEHLS - DO PPGEE - FENG - PUCRS

PUCRS
Campus Central
Av. Ipiranga, 6681 - Prédio 30 - Sala 103 - CEP: 90619-900
Telefone: (51) 3320.3540 - Fax: (51) 3320.3625
E-mail: engenharia.pg.eletrica@pucrs.br

Agradecimentos

Inicialmente gostaria de agradecer ao meu orientador, Dr. Fabian Luis Vargas, e ao meu coorientador, Dr. Aurélio Tergolina Salton, pelos ensinamentos e pelo suporte dados durante todo o processo de desenvolvimento desta dissertação. Seus aconselhamentos foram vitais para a realização deste trabalho.

Aos colegas do laboratório EASE pela parceria nas atividades realizadas no mesmo ao longo do mestrado. E aos professores que contribuíram para a minha formação acadêmica.

À Pontifícia Universidade Católica do Rio Grande do Sul pela oportunidade a mim ofertada desde a minha graduação na mesma, onde desde então venho me aprimorando e aprofundando em meu âmbito de estudo na sua ampla e moderna estrutura física disponibilizada a nós estudantes.

À CAPES e à Hewlett-Packard Brasil Ltda. pelas bolsas de estudos a mim proporcionadas para esta oportunidade de aprendizado.

Ao meu amigo Brandon Arp pelos conhecimentos essenciais compartilhados que possibilitaram o desenvolvimento desta dissertação.

E finalizo agradecendo a meus pais e demais familiares pelo apoio incondicional dado a mim todo este tempo no desenvolvimento de meus conhecimentos e habilidades. Muito obrigado a todos!

Resumo

O uso de processadores *multicore* em sistemas embarcados em tempo real de propósito geral tem experimentado um enorme aumento nos últimos anos. Infelizmente, aplicações críticas não se beneficiam deste tipo de processadores como se poderia esperar. O principal obstáculo é que não podemos prever e fornecer qualquer garantia sobre as propriedades em tempo real do software em execução nessas plataformas. O barramento de memória compartilhada está entre os recursos mais críticos, que degrada severamente a previsibilidade temporal do software *multicore* devido à contenção de acesso entre os núcleos. Para combater este problema, apresentamos neste trabalho uma nova abordagem que suporta a execução de carga de trabalho de criticidade mista em um sistema embarcado baseado em processadores *multicore*. Permitindo que qualquer número de núcleos execute tarefas menos críticas concorrentemente com o núcleo crítico que executa a tarefa crítica. A abordagem baseia-se na utilização de um *Hard Deadline Enforcer* (HDE) implementado em hardware, que permite a execução de qualquer número de núcleos (executando cargas de trabalho menos críticas) simultaneamente com o núcleo crítico (executando a carga crítica). A partir do melhor de nosso conhecimento, em comparação com as técnicas existentes, a abordagem proposta permite a exploração do desempenho máximo oferecido por um sistema *multicore*, garantindo a escalonabilidade da tarefa crítica. Além disso, a abordagem proposta apresenta a mesma complexidade de projeto, como qualquer outra abordagem dedicada a análise temporal para processadores de núcleo único, não importando o número de núcleos que são utilizados no sistema incorporado ao design. Caso técnicas atuais fossem utilizadas, a complexidade do projeto para análise temporal de sistemas de múltiplos núcleos aumentaria dramaticamente conforme o aumento do número de núcleos do sistema embarcado. Foi implementado um estudo de caso baseado em uma versão *dual-core* do processador LEON3 para demonstrar a aplicabilidade e assertividade da abordagem. Vários códigos de aplicações críticas foram compilados para este processador, que foi mapeado na FPGA Spartan 3E da Xilinx. Resultados experimentais demonstram que a abordagem proposta é muito eficaz na obtenção da alta performance do sistema respeitando o deadline da tarefa crítica.

Palavras-chaves: Processadores *Multi-core*, Aplicação Crítica, Sistema Embarcado de Alto Desempenho, Escalonamento de Tarefas Críticas, *Hard Deadline Enforcer* (HDE).

Abstract

The use of multicore processors in general-purpose real-time embedded systems has experienced a huge increase in the recent years. Unfortunately, critical applications are not benefiting from this type of processors as one could expect. The major obstacle is that we may not predict and provide any guarantee on real-time properties of software running on such platforms. The shared memory bus is among the most critical resources, which severely degrades the timing predictability of multicore software due to the access contention between cores. To counteract this problem, we present in this work a new approach that supports mixed-criticality workload execution in a multicore processor-based embedded system. It allows any number of cores to run less-critical tasks concurrently with the critical core, which is running the critical task. The approach is based on the use of a dedicated Hard Deadline Enforcer (HDE) implemented in hardware, which allows the execution of any number of cores (running less-critical workloads) concurrently with the critical core (executing the critical workload). From the best of our knowledge, compared to existing techniques, the proposed approach allows the exploitation of the maximum performance offered by a multiprocessing system while guaranteeing critical task schedulability. Additionally, the proposed approach presents the same design complexity as any other approach devoted to perform timing analysis for single core processor, no matter the number of cores are used in the embedded system on the design. If current techniques were used, the design complexity to perform timing analysis would increase dramatically as long as the number of cores in the embedded system increases. A case-study based on a dual-core version of the LEON3 processor was implemented to demonstrate the applicability and assertiveness of the approach. Several critical application codes were compiled to this processor, which was mapped into a Xilinx Spartan 3E FPGA. Experimental results demonstrate that the proposed approach is very effective on combining system high-performance with critical task schedulability within timing deadline.

Key-words: Multicore Processor, Critical Application, High-Performance Embedded System, Critical Task Schedulability, Hard Deadline Enforcer (HDE).

List of Figures

List of Tables

List of abbreviations and acronyms

AHB	Advanced high-performance bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced peripheral bus
BCET	Best-case execution time
CFA	Control-flow analysis
CFG	Control-flow graph
CT	Critical time
CPU	Central processing unit
DSU	Debug support unit
EPC	Executed program counter
FPGA	Field Programmable Gate Array
FCFS	First-come, first-served
GTT	Graph traversal technique
HDE	Hard Deadline Enforcer
IPET	Implicit-path enumeration technique
ILP	Integer linear programming
IF	Instruction fetch
JTAG	Joint Test Action Group
LFSR	Linear-feedback shift register
MPC	Model predictive control
RP	Reference point
RP_{ID}	Reference point time identification
RAM	Random-access memory

SMP	Symmetric multiprocessing
TDMA	Time division multiple access
TA	Timed automaton
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
WCET	Worst-case execution time
$WCET_R$	Remaining worst-case execution time

List of symbols

I_i	Instruction i
A_i	Address of instruction i
TT_i	Time tag of instruction i
L_i	Latency of instruction i
$a \leftarrow b$	Assignment of b to the identifier a
v_i	Vertex i
e_i	Edge i
x_{v_i}	Count variable of vertex i
x_{e_i}	Count variable of edge i
t_{v_i}	Time coefficient of vertex i
p_{e_i}	Penalty coefficient of edge i
$e_{a \to b}$	Directed edge pointing from vertex a (v_a) to vertex b (v_b)
$LB_{a \to b}$	Loop Bound of the cycle with back-edge $e_{a \to b}$
$x_{a \to b}$	Count variable of edge $e_{a \to b}$
$a \setminus b$	Relative complement of set b in set a
T_{v_i}	Maximum estimated CPU clock cycles of vertex i
$S_{e_{t \to h}}$	State of the cycle with back-edge $e_{t \to h}$
$C_{e_{t \to h}}$	Constraint of the cycle with back-edge $e_{t \to h}$
$\mathbb{Z}$	Integer numbers set
T_s	Sampling period
i_a	Armature current
R_a	Armature resistance
L_a	Armature inductance

J	System inertia
b	Viscous friction coefficient
k_t	Torque constant
k_ω	Speed constant
I_f	Field current
w	Angular speed
A^T	Transpose of the A matrix
$\|\|x\|\|$	Euclidean norm of the x vector, that is, $\|\|x\|\|= \sqrt{x^T x}$
$\|A\|$	Cardinality of set A.
$0_{n\times 1}$	Vector of zeros composed by n lines

Contents

1 Introduction

It is of common agreement among designers and users that multicore processors will be increasingly used in future embedded real-time systems for critical applications where, in addition to reliability, high-performance is at a premium. The major obstacle is that we may not predict and provide any guarantee on real-time properties of software on such platforms. As consequence, the timing deadline of critical real-time tasks may be violated. In this context, the shared memory bus is among the most critical resources, which severely degrade the timing predictability of multicore workloads due to access contention among cores. In critical embedded systems (e.g., aeronautical systems), this uncertainty of the non-uniform and concurrent memory accesses prohibits the full utilization of the system performance. In more detail, the system is designed in such a way that the processor runs in the stand-alone mode when a critical task has to be executed, i.e., the bus controller allows only one core to run the critical workload, and inhibits the other cores to execute less-critical workloads till the completion of the critical task. In order to counteract this problem and properly balance reliability against performance as long as possible, we present in this work a new approach that supports mixed-criticality workload execution by fully exploiting processor parallelism. It allows any number of cores to run less-critical tasks concurrently with the critical core, which is running the critical task. The proposed approach is not based on any multicore static timing analysis or any timing model of multicore processor parts such as pipeline, cache memory and interactions of these parts with shared memory bus. Instead, the approach is based on the use of a dedicated Hard Deadline Enforcer (HDE), which works as follows: when a critical task starts running, the HDE allows the execution of any number of cores (running less-critical workloads) concurrently with the critical core (executing the critical workload) till the moment when the HDE predicts that the critical workload deadline will be violated if the processor continues running all cores concurrently. At this moment, the HDE inhibits the non-critical cores to execute less-critical tasks until the completion of the critical task by the critical core. Then, it is said that the system is switched from the “shared mode” (where two or more cores are fighting for shared memory bus access) to the “stand-alone mode”, where a single (the critical) core is running. Given the above, the proposed approach presents the following features and advantages compared to the existing techniques:

a) It minimizes the computational complexity imposed by multicore static timing analysis: it needs only to analyze interactions between pipeline and cache models of a single core when executing a given critical task. All inter-core conflicts generated during the execution of the critical and the various less-critical tasks caused by their

non-uniform and concurrent memory bus accesses are not taken into account by the HDE.

b) From the above (a) statement, it is also concluded that the proposed approach does not need the development and it is not based on timing analysis models of multicore processors. Therefore, the approach is not based on the faithfulness of the multicore processor model to guarantee a precise workload timing prediction.

c) In contrast to the existing approaches, the proposed approach does not require any knowledge about the implementation of the less-critical workloads or even the number of workloads that will run concurrently with the critical task. The HDE is configured according to specific code structure and timing characteristics of the critical task. Then, the HDE monitors online the critical task execution and automatically switches the bus usage from the "shared" mode to the "stand-alone" mode to guarantee the maximum possible processor performance with workload schedulability. Note that if the number of less-critical tasks changes, there is no need to re-compute the timing analysis process for the critical task to guarantee workload schedulability. This condition is ensured because the HDE can switch from the "shared mode" to the "stand-alone mode" automatically, no matters is the number of less-critical tasks are running in parallel with the critical one.

d) The approach can be applied to any type of processor, considered that the designer is able to collect two signals from the processor ("Program Counter" and "Annul"). The latter signal indicates if the current instruction in the pipeline was actually executed or not.

e) Given that the approach can be applied to any type of processor, it allows a large spectrum of real-time operating systems to be used. Thus, traditional and well-stablished real-time operating systems for critical applications such as VxWorks, LynxOS, Integrity or RTEMS and their advanced versions compliant with ARINC-653 (an avionics standard for safe, partitioned systems) (ARINC Specification 653, 2006a; ARINC Specification 653, 2007; ARINC Specification 653, 2006b) could also be considered in the whole system design.

f) The proposed approach does not requires recompilation of the critical task in order to guarantee its timing deadline.

1.1 Objectives

The objectives of this work are as follows:

- The primary goal is the development of a hardware-based approach, namely "Hard Deadline Enforcer (HDE)" to guarantee the timing deadline of a single critical task running concurrently with one or more non-critical tasks in a multicore processor.

- The secondary goal is the automation of the determination process of the remaining worst-case execution time ($WCET_R$) parameter of reference points distributed strategically within the critical task. This automation by adopting an existing and well established technique found in the literature (IPET) to automate the determination of the $WCET_R$ parameter by one side. by the other side, it was developed a new technique to determine this parameter, named GTT. Both techniques presents advantages and drawbacks and can, thus, be used complementary to each other.

2 Preliminaries

2.1 Real-Time Systems

Real-time systems are computing systems that must react in strict timing constraints. Therefore the correct system behavior not only depends in the correctness of the output but also depends in the moment in which the output was produced. A real-time task is a task that posses a timing constraint known as deadline. The deadline is the maximum time in which the system must produce the output. A real-time task is classified by the consequences caused by a missed deadline (BUTTAZZO, 2011) as follows:

- **Hard**: A hard real-time task has catastrophic results when a deadline is missed.
- **Firm**: The result of a firm real-time task loses its purpose after the deadline, but does not cause any damage.
- **Soft**: In a soft real-time task, the results produced after the deadline has some utility, but causing a performance degradation.

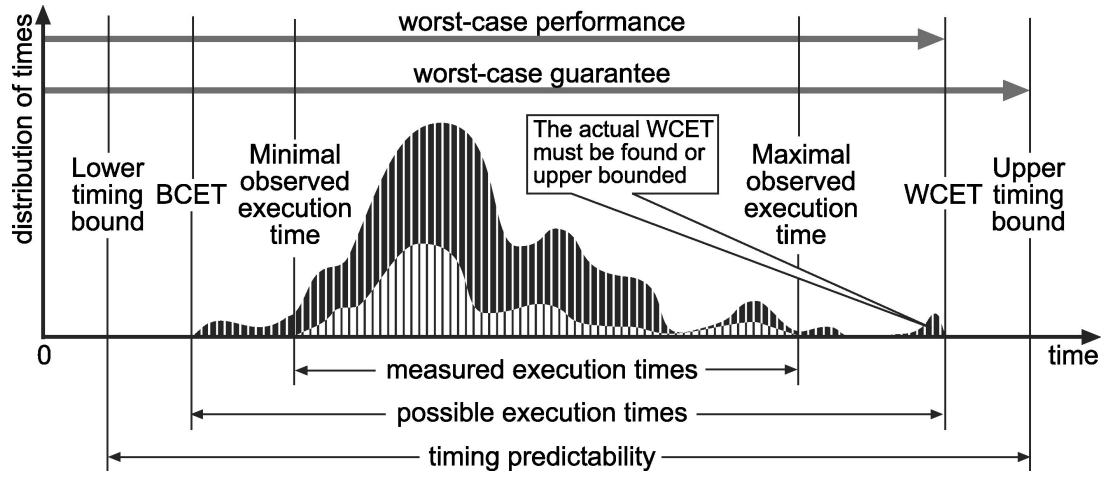
2.2 Timing Analysis

The timing of real-time tasks running in a real-time systems must analyzed to verify the deadline constraints of the tasks. If the constraints cannot be met in the evaluated system, the tasks can be optimized or the system must be replaced with a more powerful system in order to satisfy the constraints. Figure 1 illustrates basic notions concerning the timing analysis of systems. The execution time of a task varies depending on the input data or different behavior of the environment (WILHELM et al., 2008). The upper curve shows the distribution of execution times of the task, and the bottom curve is the distribution of measured times for a given set of inputs and environment behaviors.

A task typically shows a certain variation of execution times depending on the input data or different behavior of the environment. The longest response time is called the worst-case execution time (WCET). In most cases, the state space is too large to exhaustively explore all possible executions and thereby determine the exact WCET. Timing analysis tools tries to determine a timing upper bound that is higher than the WCET but as close to it as possible.

In most parts of industry, the common method to estimate execution time bound is to measure the end-to-end execution time of the task for some set of inputs (test cases) on the target hardware or on a clock cycle-accurate simulator. This determines the maximal

Figure 1 – Basic notions concerning timing analysis of systems.



Reference: Wilhelm et al. (2008).

observed execution time. In general, this will underestimate the WCET and so is not safe for hard real-time systems. This method is often called dynamic timing analysis. In contrast to this method, there is the static timing analysis, which is the preferred method used by academia. This method does not rely on executing code on real hardware or on a simulator. Rather, it takes the task code itself, most often together with some annotations, constructs a control-flow graph (CFG) of the workload and analyzes the set of all possible paths through the CFG. Next, this technique combines control-flow analysis with (abstract) models of the processor architecture (e.g., pipeline, cache memory and bus-access policy models) in order to obtain the WCET bound for the workload.

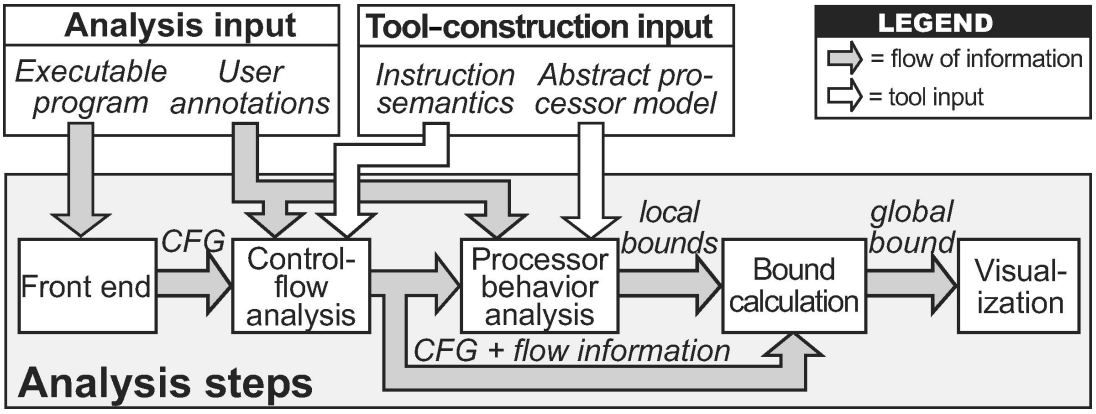
2.2.1 Static Timing Analysis

Figure 2 depicts the core components of a timing-analysis tool. The front end is responsible of generating the Control flow graph (CFG) of the application from the provided executed application. To do so, the instructions are decoded from the executed application based in the instruction set of the target architecture. The vertices in the CFG are known as basic blocks, which contains contiguous instructions. Branches are represented by edges in the CFG.

Value analysis can be used to determine the loops bounds, alternatively these can be provided by the user. Control-flow analysis (CFA) gathers information about possible execution paths in the application. That allows to potentially yield tighter bounds, by removing unfeasible paths in the bound calculation step.

The processor-behavior analysis uses abstract models of the processor to gather information on the processor behavior for the task under analysis. It analysis the behavior of components that affects the execution times, such as memory, caches, pipelines and branch prediction. A common method to analyze the cache is by abstract interpretation

Figure 2 – Core components of a timing-analysis tool.



Reference: Wilhelm et al. (2008).

(COUSOT; COUSOT, 1977; ALT et al., 1996; FERDINAND; WILHELM, 1999), memory accesses are classified (e.g. always hit, always miss, first hit).

The bound calculation computes and upper bound of all execution times considering all possible paths of the task. It is based on the flow and timing information derived in the previous steps. Analytically determined or measured times can be combined by three main methods in the literature: *structure-based*, *path-based* and *implicit-path enumeration technique* (IPET) (WILHELM et al., 2008).

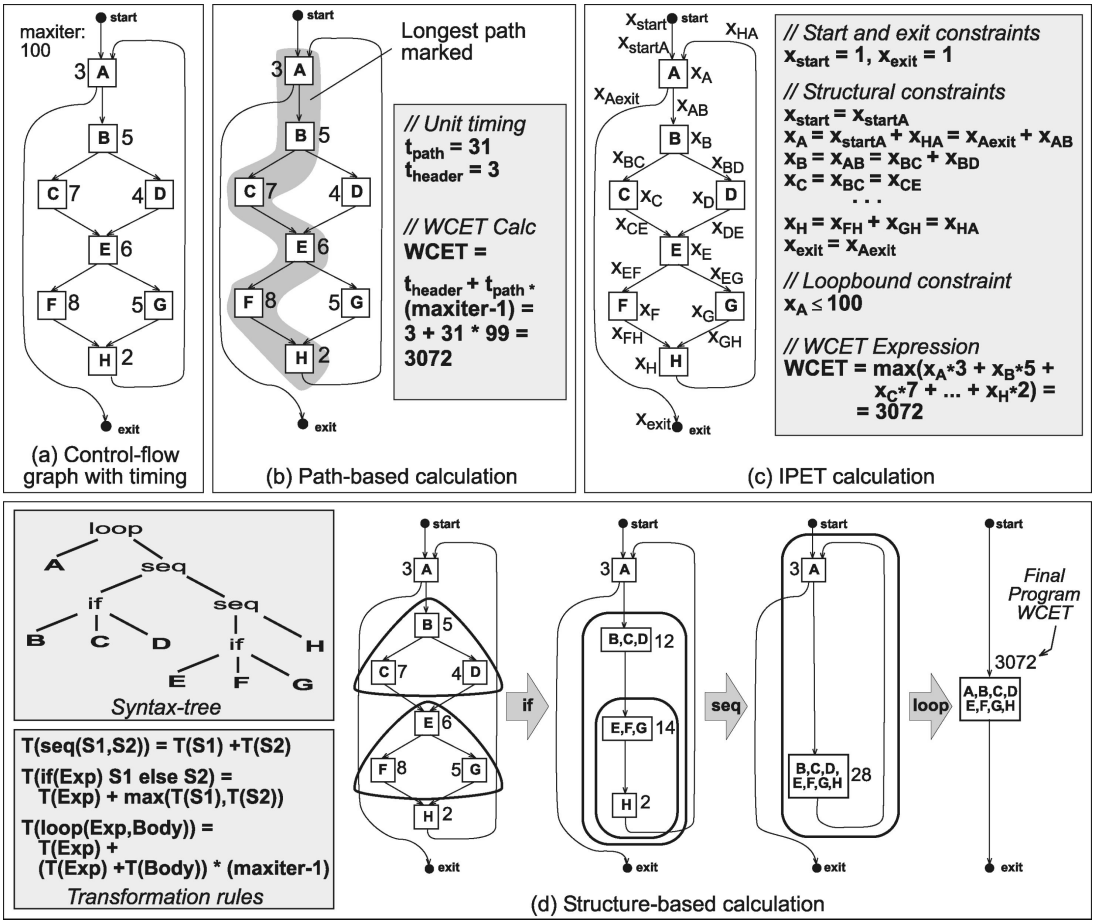
In the structure-based method, the upper bound is computed in a bottom-up traversal of the syntax tree of the task. It combines the computed task bounds for constituents of statements by following combination rules for the type of the statement.

Figure 3d depicts an example of the upper bound computation for the task in Figure 3a. It first determines the bound for the *if* statement composed by "E, F and G", which is $T(E) + \max(T(F), T(G)) = 14$, then it combines this time with the time of H, yielding 16. Then it computes the time of the "B, C, D" if statement, yielding 12, which is then combined with the time of "E, F, G, H", yielding 28. Finally it matches the resulting structure with a loop statement, which time is given by $T(A) + (T(A) + T(B, C, D, E, F, G, H)) \times (maxIteration - 1) = 3 + (3 + 28) \times 99 = 3072$, which is the upper bound of the task.

The Implicit Path Enumeration Technique (IPET), first proposed by (LI; MALIK, 1995), is a technique where the bound calculation is performed by formulating and solving an Integer Linear Programming (ILP) constrained optimization problem. In ILP, the objective function and variables are linear and are constrained to $\mathbb{Z}$. Methods for solving ILP problems are detailed in (GARFINKEL; NEMHAUSER, 1972).

In IPET, a time coefficient (t_{entity}) is given for each basic block in the task, which express the upper bound of the contribution of that basic block of every time it is executed. It is also given a count variable (x_{entity}) for every basic block in the task, which counts

Figure 3 – Methods for calculating the upper bound of a task.



Reference: Wilhelm et al. (2008).

how many times that entity was executed. The objective function of the ILP problem is defined to be the sum of the product of the time coefficient and count variable for all basic blocks in the task ($\sum_{i \in entity} x_i \times t_i$).

The structure of the task is described as a set of constraints in the count variables. The sum of the count variables of the edges that enters an entity is equal to the sum of count variables of the edges that leaves the entity, that is also equal to the count variable of the entity in which the edges entered (exited)., i.e., $x_B = x_{AB} = x_{BC} + x_{BD}$. Loops bounds are defined as constraints in the count variable of the loop header, i.e., $x_A \leq 100$.

The *start* and *exit* constraints specifies that the task was entered and exited once. Once the ILP problem is formulated, a tool such as lp_solve (BERKELAAR, 2010) can be used to obtain the WCET and the count variable of all basic blocks of the task. Figure 4 depicts the solution for the ILP formulation in Figure 3c computed using the lp_solve tool. It shows the WCET (3072) and the count variables of every basic block and edge. From the Figure 4, it can be noticed that the basic blocks D and G, were not executed, once they would not contribute for the WCET of the task.

Figure 4 – IPET results for the ILP in Figure 3c.

Variables	result	Variables	result
	3072	xha	99
xa	100	xab	99
xb	99	xbc	99
xc	99	xbd	0
xd	0	xce	99
xe	99	xde	0
xf	99	xef	99
xg	0	xeg	0
xh	99	xfh	99
xstart	1	xgh	0
xend	1		

An extension of IPET is presented in (OTTOSSON; SJODIN, 1997) to allow the independent modeling of micro-architectural aspects, it shows examples for modeling pipelined execution and set-associative caches. Ballabriga e Cassé (2008) decreases the WCET computation time by partitioning the CFG of the task into smaller parts, where a local WCET can be computed faster.

In path-based methods, the upper bound is given by computing bounds for different paths of the task, searching for the path with the longest execution time. The number of paths is exponential to the number of branch points in the task (WILHELM et al., 2008). Figure 3b depicts an example the path-based method, where the longest path is marked and its time is combine with the loop bound to computed the upper bound for the task. A path search method considering pipeline effects, caches and complex flows is described in (STAPPERT; ERMEDAHL; ENGBLOM, 2001).

2.2.2 Measurement-based Methods

These methods works by executing the task or parts of the task with a set of inputs on the target hardware and sampling the time of every instruction or basic block of the task. A clock-cycle accurate simulator can be used instead of executing it in the target hardware. Measurement can be applied to task snippets as well, and then combining the results using methods of static methods, such as in Figure 3. Although, using measurement, safe bounds can only be guaranteed on simple architectures (WILHELM et al., 2008). Using measurement of parts of the tasks (or with instruction granularity), replaces the “processor-behavior analysis” in Figure 2. Then, using CFA to find all possible paths, and combining the measured times on the bound calculation step, all paths of the task is considered in the timing bound estimation. Although, if the measured times are not safe (e.g. due to cache), the estimated timing bound is not safe.

Burns e Edgar (2000) predicts the WCET of tasks for complex processor architectures using measurement associated with statistical analysis, where the computation time is represented by a probability function and the WCET estimates have a level of confidence.

Bernat, Colin e Petters (2002), Bernat, Colin e Petters (2003) presents a method based in measurement and static analysis. It combines probabilistically the worst case effects observed in individual blocks collected during measurement into the worst-case path of the application.

2.2.3 Timing Analysis Tools

2.2.3.1 AbsInt's WCET Analyzer aiT

aiT is a commercial WCET static analysis tool that provides support for several hardware platforms (Motorola PowerPC MPC 555, 565, and 755, Motorola ColdFire MCF 5307, ARM7 TDMI, HCS12/STAR12, TMS320C33, C166/ST10, Renesas M32C/85 (prototype), Infineon TriCore 1.3, LEON2 and LEON3). aiT first derives safe upper bounds for the execution time of basic blocks and then computes, by integer linear programming, an upper bound on the execution times over all possible paths of the program (FERDINAND; HECKMANN, 2004).

At first, aiT reconstructs the CFG from a binary program, then value analysis computes value ranges for register and address ranges for instructions accessing memory. Loop bound analysis determine the loop bounds for simple loops in the application. Cache analysis is performed using abstract interpretation, classifying memory references as cache misses or hits. Cache analysis is based in the results of value analysis to then predict the behavior of the cache. Pipeline analysis is perform and finally the WCET is computed by path analysis using integer linear programming.

The user must provide annotations if the automatic loop bound analysis fails. Furthermore, if some application does not adheres to the standard calling convention, the user might need to provide additional annotations describing the control-flow properties of the task (WILHELM et al., 2008).

2.2.3.2 Bound-T Tool

The Bound-T tool was originally a commercial tool developed at Space Systems Finland Ltd, an was intended for verification of spacecraft on-board software. Tidorum Ltd extended Bound-T to other application domains. Since January 2014, the Bound-T tool is supplied free of charge in an open-source license. Although some modules that relies on proprietary information from third party vendors remains closed source.

The Bound-T tool is based in static timing analysis, using implicit path enumeration technique to determine the worst-case path and the upper bound. The architecture is designed to be adaptable to different target processors, extensible with new kinds and methods of analysis, and portable to different host platforms (HOLSTI; SAARINEN, 2002).

The task being analyzed must not be recursive, must use standard calling conventions and function pointers are not supported. No cache analysis is yet implemented, and any timing anomalies in the target processor must be accounted for in the execution time of basic blocks of the CFG (WILHELM et al., 2008). The Bount-T tool supports the following target platforms: Intel-8051 series (MCS-51), Analog Devices ADSP-21020, ATMEL ERC32 (SPARC V7), Renesas H8/300, ARM7 (prototype) and ATMEL AVR and ATmega (prototypes) (WILHELM et al., 2008).

2.2.3.3 RapiTime Tool

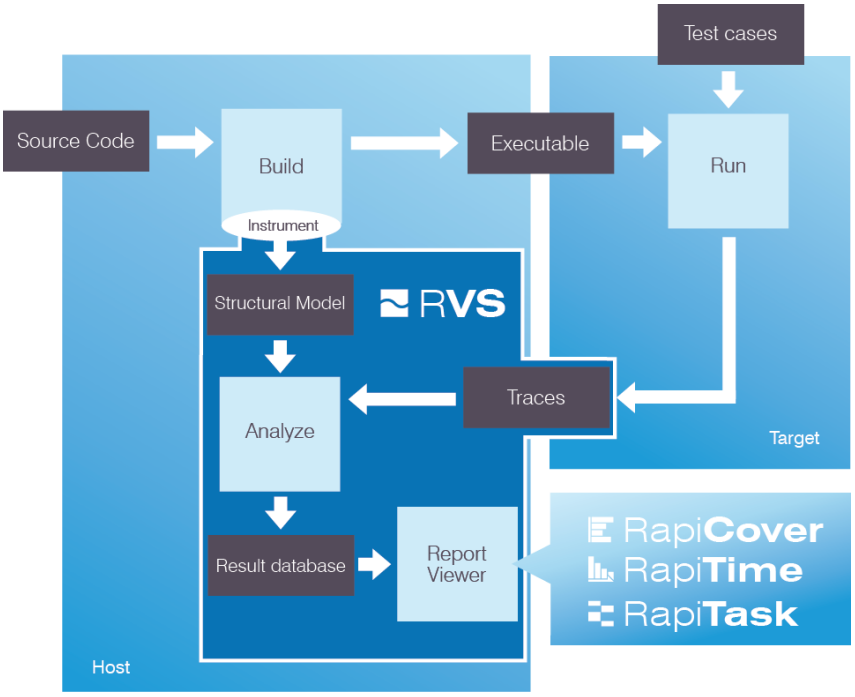
RapiTime aims at medium to large real-time embedded systems on advanced processors. The RapiTime tool targets the automotive electronics, avionics and telecommunications industries (WILHELM et al., 2008).

Figure 5 depicts the block diagram of the RapiTime tool. The user provides a set of source codes of that task under analysis, which is then built, from the executable, a structural model is generated. From test cases provided by the user, alongside with the executable, the application is run in the target hardware, generating timing traces. The application must be instrumented with instrumentation points which indicates that a section of the application was executed (RAPITA SYSTEMS LTD, 2015a). Which can be performed either by a software instrumentation library or by a lightweight software instrumentation with external hardware support. Alternatively the timing information can be captured by purely nonintrusive tracing mechanisms (like Nexus and ETM) or even traces from CPU simulators (WILHELM et al., 2008).

The structural model and the traces are used in the analysis step to compute probability distributions from the measured times, producing performance metrics and predicting the worst-case execution time (RAPITA SYSTEMS LTD, 2015a). Results can be observed for each function and subfunction of the analyzed task.

RapiTime does not relies on processor models, therefore it can work for any processor, as long as traces can be collected from the processor, and that the object code reader is ported for that processor. RapiTime cannot analyze programs with recursion and with nonstatically analyzable function pointers (WILHELM et al., 2008). The location of the jump performed by function pointers can only be determined at runtime (in the case that it is nonstatically analyzable), making it difficult to generate the structural model for these applications.

Figure 5 – RapiTime block diagram.



Reference: RAPITA SYSTEMS LTD (2015a).

2.3 State-of-the-Art on Timing Analysis Techniques for MPSoC

A lot of research has been carried out within the area of WCET analysis (PUSCHNER; BURNS, 2000). However, each task is, traditionally, analyzed in isolation as if it was running on a monoprocessor system. Consequently, it is assumed that memory accesses over the bus take constant amount of time to process. For multiprocessor systems with a shared communication infrastructure, however, transfer times depend on the bus load and are therefore no longer constant, causing the traditional methods to produce incorrect results (ROSÉN et al., 2011). As response to this specific need, several approaches dealing with WCET prediction in multicore platforms have been proposed (ROSÉN et al., 2011; CHATTOPADHYAY et al., 2014; LV et al., 2010; UNGERER et al., 2010).

In (ROSÉN et al., 2011), authors proposed a technique to achieve predictability of tasks running in multiprocessor systems. The approach is based on the simultaneous analysis of the critical task running in a given core with the shared-bus scheduling process, in order to bound the WCET for that task. In order to calculate the whole WCET of such task, the analysis needs to be aware of the TDMA bus, taking into account that cores must only be granted the bus during their assigned time slots.

In (CHATTOPADHYAY et al., 2014), authors proposed a unified WCET static timing analysis approach for multicore processors. This work is based on models of cache and shared bus, which interact with other basic micro-architectural models (e.g. pipeline and branch predictor unit). Each processor core is analyzed at a time by taking care

of the inter-core conflicts generated by all other cores. In this multicore scenario, it is assumed a TDMA shared bus based on round robin arbitration policy, where a fixed length bus slot is assigned to each core. They also assume fully separated caches and buses for instruction and data. Therefore, the data references do not interfere with the instruction references. This work only models the effect of instruction caches. Since it is considered only instruction caches, the cache miss penalty (computed from cache analysis) directly affects the instruction fetch (IF) stage of the pipeline. Finally, authors consider the LRU cache replacement policy.

In (LV et al., 2010), authors proposed a method to bound WCET for workloads running in a multicore architecture where each core has a local L1 cache and all cores use a shared bus to access the off-chip memory. They modeled the local cache behavior of a program running on a dedicated core. Then, based on the cache model, they constructed a Timed Automaton (TA) to model when the programs access the shared bus. Examples for TDMA and FCFS buses were analyzed. The UPPAAL model checker (BENGTSSON et al., 1996) is used to find the WCET of the application.

In contrast to (ROSÉN et al., 2011)(CHATTOPADHYAY et al., 2014)(LV et al., 2010) that are approaches based on static timing analysis, in (UNGERER et al., 2010) authors described a project called “Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability”. This work aimed at developing multicore processor design (described in SystemC) for hard real-time embedded systems and a technique to guarantee the analyzability and timing predictability of every feature provided by the processor. Publications presented results for a quad-core version of this processor, where each core consists of two pipelines and implements the TriCore (Infineon) instruction set. Each core provides up to four thread slots (separate instruction windows and register sets per thread), which allows simultaneous execution of one hard real-time task and three non-hard real-time tasks.

The processor architecture contains one inter-core bus arbiter, which arbitrates requests from different cores, and four intra-core bus arbiters (one per core) that arbitrate among thread requests from the same core. The processor shared memory can suffer from both intra- and inter-core interferences. To avoid these interferences, authors proposed a dynamically partitioned memory, which assigns a private subset of memory banks to each hard real-time task so that no other task has access to it (the Merasa operating system sets the memory partition assigned to each core by modifying special hardware registers). Also, the MERASA processor runs based on a Round Robin bus policy.

The MERASA system-level software represents an abstraction layer between the application software and the embedded hardware. It provides the basic functionalities of a real-time operating system as a foundation for application software running on the MERASA processor. MERASA system-level software guarantees the isolation of memory

accesses of various hard real-time tasks that are running on different cores to avoid mutual and possibly unpredictable interferences. This isolation should also enable a tight WCET analysis of application code. The resulting system software can execute hard real-time tasks in parallel on different cores of the MERASA multicore processor.

The MERASA processor and techniques were validated by means of determining WCET for a given application based on the use of two CAD tools, one academic and one from industry: OTAWA (BALLABRIGA et al., 2010) and RapiTime (RAPITA SYSTEMS LTD, 2015b), respectively. While OTAWA extracts the control flow graph (CFG) from the binary code (thus, performing static timing analysis), RapiTime uses the extracted traces to estimate the WCET by measurements/simulations of the target hardware. Further, the MERASA project was continued on a new action: parMERASA (UNGERER et al., 2013).

Considered all the above mentioned approaches, our proposed approach represents a considerable improvement of the state-of-the-art, since:

a) They are not trivial in such a way that they must not only analyze all interactions between pipeline and cache models of a single core when executing a given critical task. They also analyze all inter-core conflicts generated during the execution of the critical and the various less-critical workloads and their non-uniform and concurrent memory bus accesses. Note that such analysis is even more complex when the number of cores running in parallel increases.

b) If the number of tasks changes, the whole process must be recomputed in order to reschedule the tasks into the TDMA (resp. FCFS or Round-Robin) bus slots.

c) It may happen that after predicting the execution of a critical task in a given multicore platform, the designer concludes that this task is not schedulable when executed in concurrence with other (less-critical) tasks. So, the whole analysis is useless and a new analysis process must restart on the basis of a smaller number of less-critical tasks to running in parallel with the critical one. The final goal is to guarantee schedulability of the critical task. If this is not attained yet, then the whole process is restarted again with an even smaller number of less-critical tasks. This “re-do” work is long and complex. So, time consuming.

d) Unlike the previous methods, in our approach, the WCET computation for the critical task is computed *independently* of the number of less critical tasks running in any number of cores concurrently with the critical task in the critical core. This renders the system design complexity much lower than in the case of the previous approaches. Furthermore if the number of less critical tasks changes (and the number

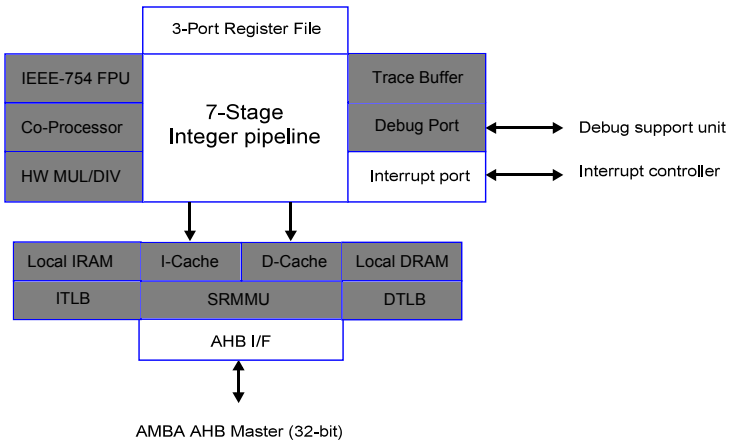
of less critical cores as well), the WCET of the critical task does not need to be recomputed.

2.4 LEON3 Processor

LEON3 is a 32-bit processor core that conforms to the IEEE-1754 (SPARC-V8) (SPARC International INC., 1992) architecture. It is designed for embedded applications and implements a 7-stage pipeline with separate instruction and data caches, memory management unit, hardware multiplier and divider, on-chip debug support and multi-processor extensions (COBHAM GAISLER AB, 2015).

Figure 6 depicts the block diagram of the LEON3 core, the indicated blocks are optional, the "Interrupt port" is required for multi-processor systems, since it is used to start the cores.

Figure 6 – LEON3 processor core block diagram.



Reference: Modified from COBHAM GAISLER AB (2015).

2.4.1 Instruction Pipeline

The LEON3 integer unit is composed by 7 stages, uses a single instruction issue pipeline, and is depicted below (COBHAM GAISLER AB, 2015):

1. FE (Instruction Fetch): The instruction is fetched from the instruction cache if the cache is enabled. Otherwise, the fetch is performed in the AHB bus.

2. DE (Decode): The instructions are decoded and the target addresses of the CALL and Branch instructions are generated.

3. RA (Register access): Operands are read from the register file or from internal data bypasses.

4. EX (Execute): ALU, logical, and shift operations are performed. The address is generated for memory operations (e.g., LD) and for JMPL/RETT instructions.

5. ME (Memory): Data cache is read or written at this stage.

6. XC (Exception) Traps and interrupts are resolved. The data is aligned as appropriate for cache reads.

7. WR (Write): The result of ALU, logical, shift, or cache operations are written back to the register file.

2.4.2 Debug Support Unit (DSU)

The Debug support unit is connected to the LEON3 pipeline and allows non-intrusive debugging of applications running the processor. Four watchpoint registers can be enabled, which allows hardware breakpoints to be placed in the application. The trace buffer monitors and stores executed instructions, which can then be read via the debug interface. It consists of a circular buffer that stores executed instructions information. The following information is stored in real-time without performance degradation:

- Instruction address and opcode
- Instruction result
- Load/store data and address
- Trap information
- 30-bit time tag

2.4.3 AMBA interface

The LEON3 processor uses a single AHB master interface to access data and instructions. Instruction fetches can be configured to be performed by incremental bursts or by single read cycles. For reading cacheable data, burst operations are used, for non-cacheable data, byte, half-word and word accesses are used accordingly. To store data, for 32-bit store a single access is performed, and for a 64-bit store a two-beat incremental burst is used. The bus arbiter supports the fixed-priority and round-robin policies. In round-robin mode if no master requests the bus, the arbiter grants the bus to the last owner.

2.4.4 Cache System

The LEON3 cache system supports a single-way (direct-mapped) or multiway cache. It can be configured with a set associativity of 2 to 4, and way sizes of 1 to 256 kiB, divided into cache lines of 16 or 32 bytes of data. The LEON3 cache system implements the following cache replacement policies:

- Least-recently-used (LRU)
- Least-recently-replaced (LRR)
- Pseudo-random

Data stores uses write-through policy, data is first stored in a double-word write buffer to later be processed in background, therefore, unless the write buffer is full, the pipeline can keep executing while the write is being processed. In the LEON3 multiprocessor system, bus-snooping must be enabled to maintain cache coherency. Whenever a write is performed in a cached area, the cache line is marked invalid and the processor will be forced to fetch the new data from memory when it is read.

2.4.5 Multi-Processor Support

LEON3 can be used in multi-processor systems, each processor has a unique index that allows processor identification. Cache coherency is guaranteed in shared-memory systems by the write-through caches and snooping mechanism. Up to 16 processors can be connected to the same AHB bus in symmetric multiprocessing (SMP) configurations with shared memory.

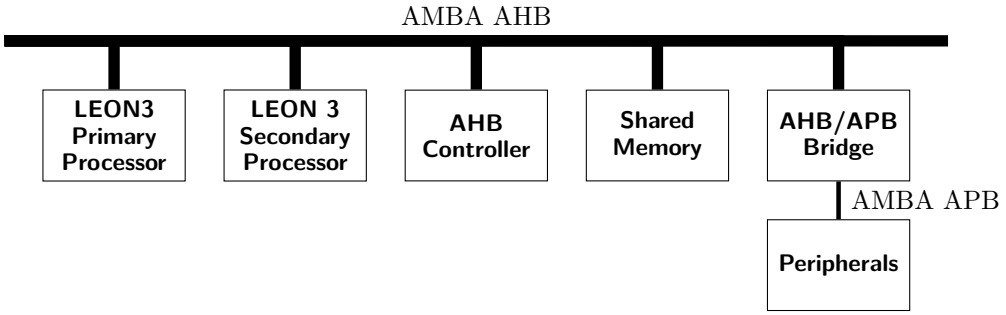
On reset, only the first processor is initialized, the other processors remains in power-down mode. The first processor can start the other processors by writing to the "multiprocessor status register" field in the multiprocessor interrupt controller. To determine which processor an application is running, the "processor index" field can be read in the LEON3 configuration register.

Figure 7 depicts the block diagram of a minimal LEON3 dual-core processor configuration. It is comprised of two LEON3 processors connected to the same bus, sharing the same RAM memory.

2.4.6 Register Windows

The LEON3 processor can be configured to have 2 to 32 register windows. During function calls, the registers does not need to be saved in the stack ("save" and "restore"

Figure 7 – Dual-Core LEON3 processor block diagram.



instructions can be used, that moves between register windows). Although when the register window overflows a trap is generated, executing a code that handles the window overflow by committing the registers to the stack. The "ins" and "outs" registers of adjacent windows overlaps, e.g., the "ins" registers of window 0 are the same as the "outs" registers of window 1.

2.5 Advanced Microcontroller Bus Architecture (AMBA)

The Advanced Microcontroller Bus Architecture (AMBA) is a specification that defines a communications standard for designing high-performance microcontrollers. Three distinct buses are defined in the AMBA specification (ARM Ltd, 1999):

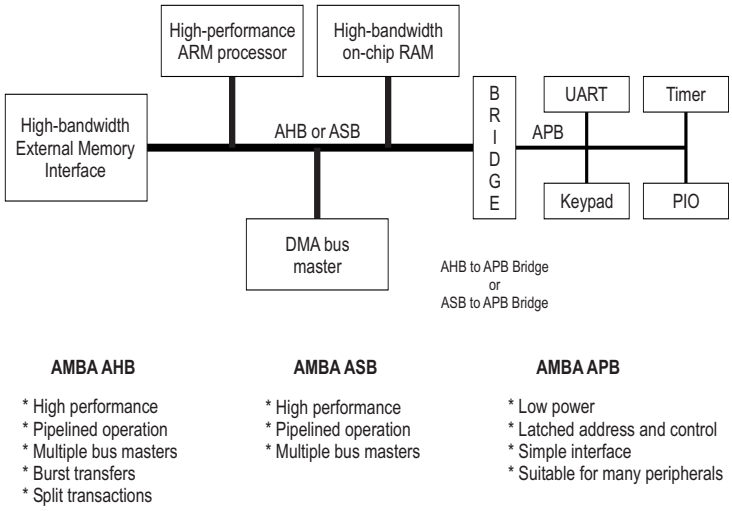
- The Advanced High-performance Bus (AHB)
- The Advanced System Bus (ASB)
- The Advanced Peripheral Bus (APB)

Figure 8 depicts a typical AMBA system, composed by an AHB or APB bus connected to an APB bus via a bridge. In the main bus (AHB or APB), are connected two masters ("High-performance ARM processor" and "DMA bus master") that can initiate transfers to the slaves ("High-bandwidth external memory interface", "high-bandwidth on-chip RAM" and "Bridge"). Due to its simple interface, the APB bus is used to connect devices that does not requires performance, e.g. a Timer and General purpose IOs.

2.5.1 AMBA AHB

The AHB is a high-performance system bus that allows the efficient connection of processors, on-chip memory and off-chip external memory. It is comprised of masters and slaves, where only the masters can start a data transfer from slaves. The bus arbiter ensures that only a master is given the access to the bus at a given time. The master starts by requesting the bus to the arbiter. Once a master acquires the bus, it can commence the

Figure 8 – A typical AMBA system.



Reference: ARM Ltd (1999).

transfer by driving the address and control signals, the control signals provide information on the direction of the transfer (read/write), the width of the transfer, the type of the transfer (single transfer, incrementing burst of unspecified length, and others), among others.

Normally the arbiter does not interrupts a burst transfer. Although if the arbiter determines that the transfer must be terminated early, e.g. if a master with higher priority must access the bus, or if the transfer was taking too long to complete, it can transfer the bus ownership to another bus master before the burst has completed. Once the interrupted burst transfer master acquires the bus, the transfer can be resumed from the point it was interrupted.

3 Proposed Methodology

The proposed methodology is divided in two parts, which are independent to each other. The first part is the implementation of the hardware that is virtually invariant and independent of the target processor architecture. The second part is the computation of the remaining worst case execution time ($WCET_R$) of the critical application which is used to configure the hardware. This part is highly dependent of the target processor architecture.

3.1 Problem Description

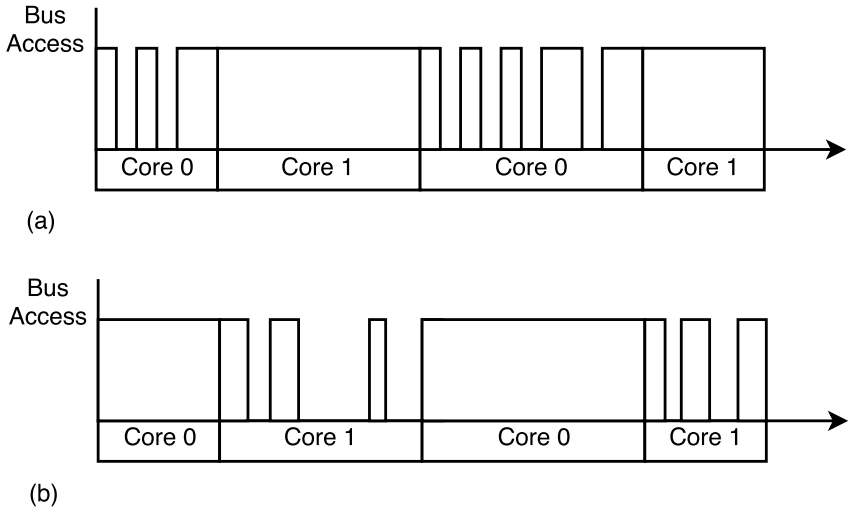
In multi-core processors, the bus sharing between cores causes the time of accessing the bus to vary. Therefore it makes it hard to predict the time of a bus access, and by consequence, making hard to compute the WCET of a task running in a multi-core processor. Tools that computes the WCET of tasks running in multi-core processors must analyze all of the tasks altogether, and if the task set changes, all of the WCET analysis must be performed again. Methods using TDMA tries to find a bus schedule, by assigning fixed slots to each core, that guarantees the deadline of all tasks. Since a core can only access the bus in its own dedicated slot, even when current slot owner is not requesting the bus, the bus might often be idle, causing its throughput to decrease, and by consequence, decreasing the performance of a multi-core system.

Figure 9 depicts an example of a TDMA bus schedule for a dual-core processor. The TDMA slots are shown below the X axis, when the bus is actually accessed is in the Y axis. In Figure 9a, the task running in core 0 had an input in which caused idle times in the bus. Similarly, in Figure 9b, some inputs caused the task in core 1 to sub-utilize the bus. Therefore, in this example, depending on the input of the tasks, the bus can be sub-utilized.

3.2 Proposed Solution

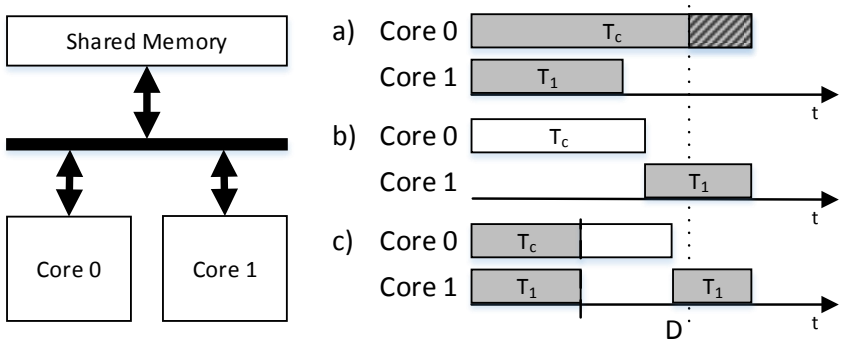
Figure 10 depicts the situation where one critical task (T_c) and one less-critical task (T_1) are running on two cores. When both tasks are executed (in the "shared mode"), the WCET of the T_c violates its deadline D. Thus, the problem is unschedulable (Figure 10a). However, if T_c is executed in the "stand-alone mode", it is schedulable (Figure 10b). In contrast with existing approaches, where only the critical task is executed at a time in the multicore platform until its completion, the proposed methodology is capable of scheduling the T_c by considering the following scenario: initially both tasks (T_c and T_1)

Figure 9 – TDMA bus utilization example.



are executed on the system. Then, reference points (RPs) are used to observe on-line the execution time of the T_c and decide switching the processor from the "shared mode" to the isolated execution of T_c (Figure 10c). The approach can be applied to any type of processor, considered that the designer is able to collect two signals from the processor ("Program Counter" and "Annul"). The latter signal indicates if the current instruction in the pipeline was actually executed.

Figure 10 – Scheduling based on WCET when are considered for execution (a) both tasks, (b) only the critical task and (c) proposed approach.



In this work, we propose an approach to improve core utilization by running several tasks in parallel while guaranteeing the critical task schedulability. The target platform can be a TDMA, First-Come First-Served (FCFS) or Round Robin bus-access policy multicore system. We assume a single core to run a unique critical task in parallel with any number of cores running less-critical tasks. If running in the stand-alone mode the critical task is perfectly schedulable. However, if it is running in parallel with other less-critical tasks, it cannot be guaranteed its schedulability, unless the proposed approach is considered. Our methodology is split in two steps (VARGAS; GREEN, 2015b; VARGAS; GREEN, 2015c; VARGAS; GREEN, 2015a):

i) Off-line, we analyze the control-flow-graph (CFG) of the critical task and safely compute the remaining WCET at several Reference Points (RPs) of the T_c running in the stand-alone mode. The remaining WCET ($WCET_R$) is defined as the WCET between the considered RP until the end of the code.

ii) On-line, for the system running in the shared mode, we use a dedicated Hard Deadline Enforcer (HDE) to monitor the real execution time of T_c and to check whether there is a risk that the critical task misses its deadline due to system overload. If so, the less-critical tasks are temporarily paused so that the critical task continues in the stand-alone mode from that monitored point until its completion. After the critical task is complete, the less-critical tasks can resume execution from the point they were temporarily paused.

Figure 11 shows the computed timeline of a critical task. The reference point zero (RP_0) is hereby defined as the start of the task. The Critical Time (CT) of a given Reference Point (RP) is given by:

$$CT(RP_n) = DeadlineTime - WCET_R(RP_n) - t_{over}, \tag{1}$$

Where,

$$DeadlineTime > WCET + t_{over}, \tag{2}$$

And the $WCET_r(RP_n)$ is the remaining worst-case execution time from the RP_n to the task end instant, which was statically computed for the processor running in the "stand-alone" mode. Finally, t_{over} (turn-over) is a constant cost associated to the bus arbiter to switch the processor between the "shared" and the "stand-alone" modes.

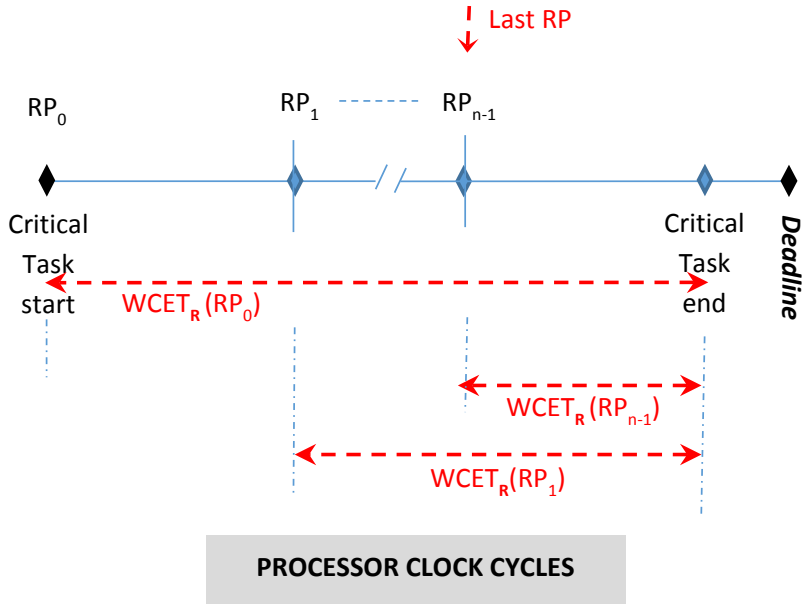
When the critical task starts, the elapsed time is initialized and incremented, clock-by-clock by the HDE and compared against the CT collected at the last RP that it passed by. It should be noted that during the critical task execution, the CPU passes through several code paths with different times (since the path taken along the code is a function of its inputs). Therefore, it is mandatory that the CT be updated at every RP distributed along the code. When the HDE compares the elapsed time against the CT collected at the most recent RP that it passed by, three possible situations may occur:

a) The elapsed time is smaller than the CT; then, the processor continues executing in the "shared" mode.

b) The elapsed time is equal to or greater than the CT; then, the HDE advises the bus arbiter to switch the processor from the "shared" to the "stand-alone" mode.

c) The elapsed time is greater than the "Deadline"; then, the HDE issues an "Error Indication" signal.

Figure 11 – Timeline of a critical task execution.



It is worth noting that during task execution in the "stand-alone" mode, upon arriving in the current RP, if the HDE detects that the execution of the critical task is faster than predicted according to the last RP that it passed by, the HDE signals to the bus arbiter to switch back to the "shared" mode in order to privilege task concurrency in the multicore platform.

Figure 12 depicts an example of the HDE of a fictitious dual-core processor composed by the Critical core C and a non critical core N. The critical core runs the critical task T_c with an WCET (of single-core execution) of 8. Although in this example the execution time of the task running in a single-core processor is 6 clock cycles. The critical task has two reference points:

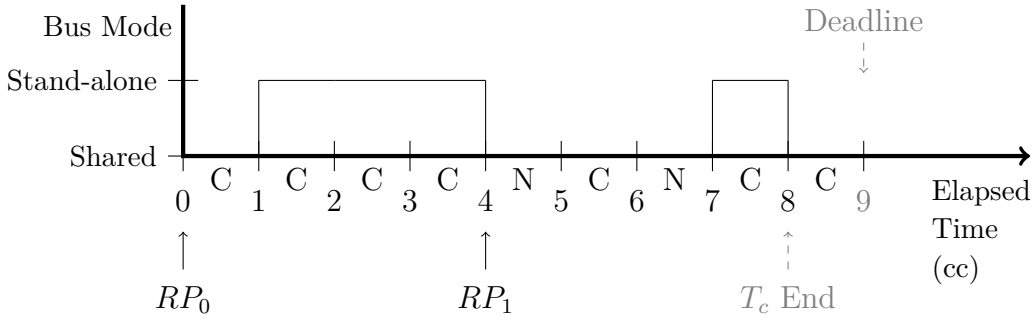
Table 1 – HDE example reference points.

Reference Point	Start Instant (cc)	$WCET_R$ (cc)	CT (cc)
RP_0	0	8	1
RP_1	4	2	7

The Start instant is how many clock cycles the task must be executed for it to reach the given reference point, the critical time of the reference point is computed using equation (1) considering $t_{over} = 0$.

The arbitration sequence while in shared mode is $[C\ N]$ and when entering the stand-alone mode it pauses the sequence and resumes when switching back to the shared mode.

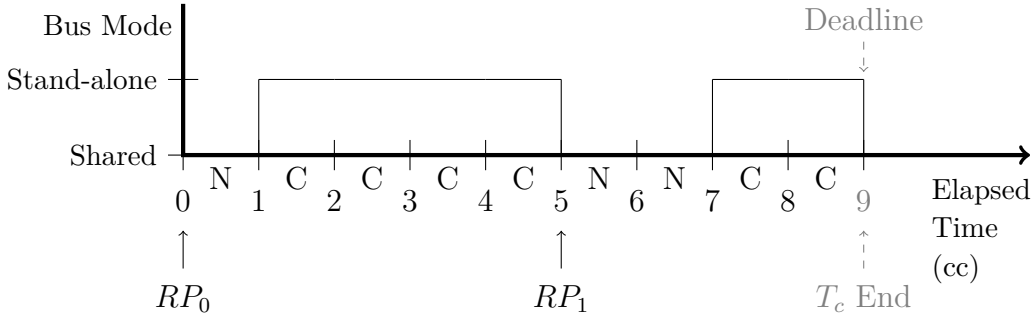
Figure 12 – Hard deadline enforcer example. During the shared mode, the bus is arbitrated to both cores.



At the instant 1 cc the HDE switches the bus to stand-alone mode since the elapsed time was equal to the CT of the current reference point (RP_0), and continued in that mode until the instant 4 cc, where the next reference point was set (RP_1). At the instant 7 cc, the bus switched to stand-alone mode since the elapsed time was equal to the CT of the current reference point. The critical task ends a cycle before the deadline.

Figure 13 depicts an example of the HDE with the same parameters as the previous example but with a different shared mode arbitration sequence ($[N]$), arbitrating only to the non critical core while in the shared mode.

Figure 13 – Hard deadline enforcer example. During the shared mode, the bus is arbitrated only to the non critical core.



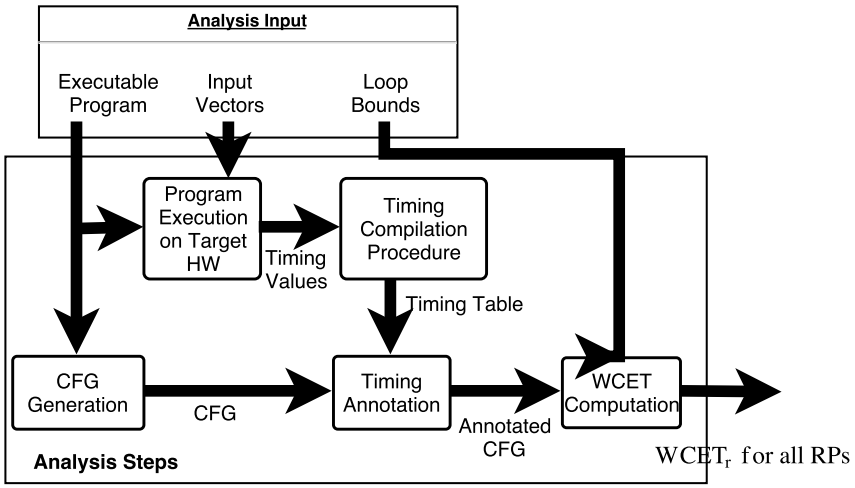
Unlike the previous example, the reference point RP_1 happens at instant 5 cc, and the critical task ends at the deadline.

3.3 WCET Computation Technique: a case-study on the LEON3 single-core processor

The technique is based in measurement techniques that combines the measured times into the WCET and $WCET_R$. By doing so we do not need to exercise all possible paths of the application (that is performed during the time combination), but all vertices and edges in the control flow graph (CFG), which is much simpler and less time consuming.

Figure 14 depicts the block diagram of the $WCET_R$ computation tool, and each block responsibility is summarized below.

Figure 14 – WCET tool block diagram.



1. CFG Generation: Decode the instructions of the compiled application and generate the CFG of the task under analysis.

2. Program Execution on Target HW: Execute the application in the target hardware with the input vectors provided and gather timing information of each instruction using measurement techniques. For LEON3 it is used the debug support unit (DSU) instruction trace that stores the time tag of every executed instruction. The time tag is the time in CPU clock cycles in which a instruction left the exception stage of the pipeline.

3. Timing Compilation Procedure: Combines the timing values measured from the target hardware into the timing table.

4. Timing Annotation: Annotate the CFG vertices and edges with timing information from the timing table.

5. WCET Estimation: Estimate the WCET from the annotated CFG and loop bounds provided using implicit path enumeration technique (IPET) or graph traversal technique (GTT).

3.3.1 Program Execution on Target HW

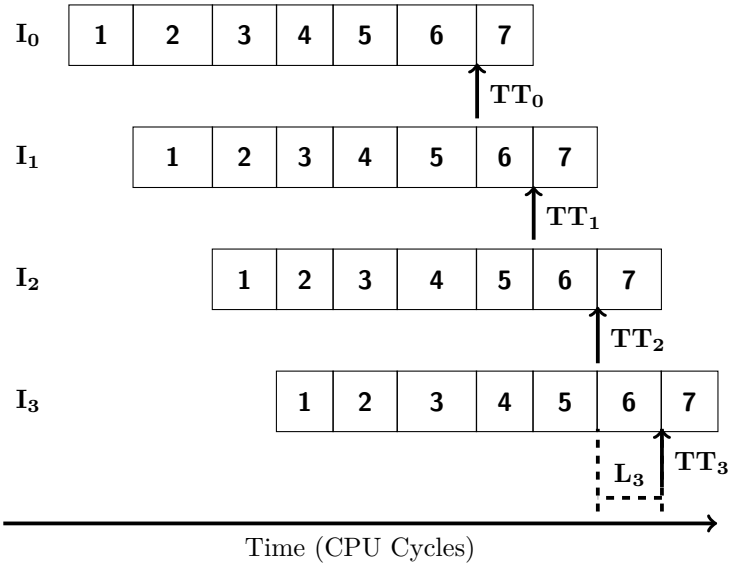
In order to compute the WCET of a task, the task instructions must be sampled in the target hardware. The sampling must cover all basic blocks and edges of the task CFG, for that, the user must generate input vectors to be used during sampling.

Input vectors can be generated either randomly or defined by the user, in this case, the user must have knowledge about inputs leading to every branch of the task, thereby defining input vectors that excites all basic blocks and edges of the task. While a single task run with an input vector does not needs to include every basic block and edge of the task, the combination of all runs w.r.t. the input vectors must cover all the basic blocks and edges of the task.

The sampling is performed by the LEON3 DSU Instruction trace module, that collects information about executed instructions in real-time without performance impact to the processor. The information is collected at the end of the sixth stage of the LEON3 pipeline stage. Within that information, the time tag and instruction address (PC) is used to compute the timing of each instruction.

Figure 15 depicts the execution of four instructions in the LEON3 processor with the time tag TT_i of each instruction I_i being collected at the sixth pipeline stage. The latency of an instruction is the difference of the time tag of the current instruction TT_i to the time tag of the previous instruction TT_{i-1}.

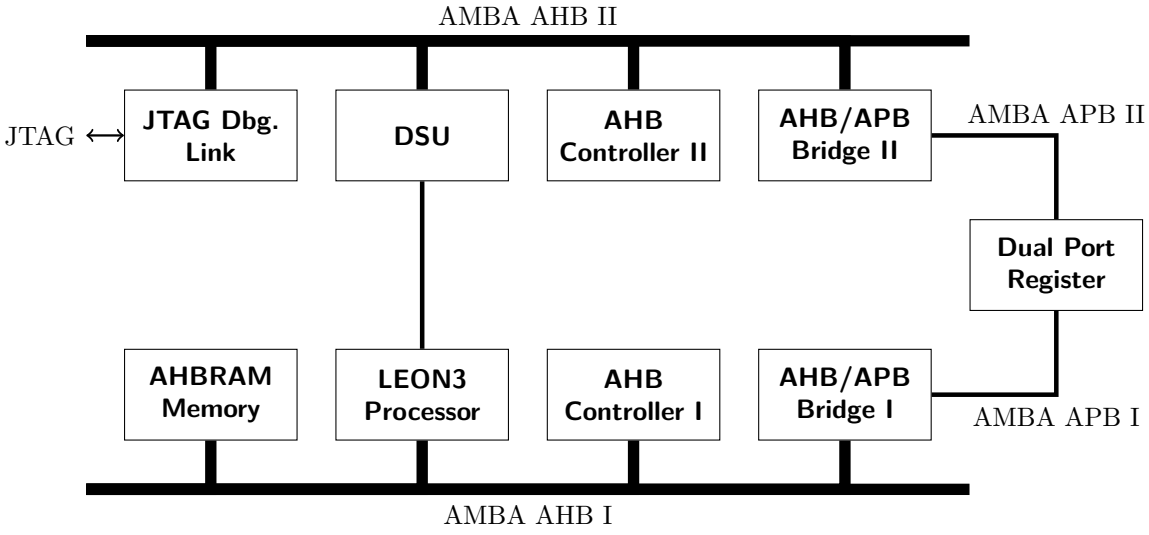
Figure 15 – Measurement procedure of the latency of instructions passing through the LEON3 micropipeline architecture.



In this approach, the latency of pipeline up to the end of the fifth stage of the first instruction is lost, if that time is desired, five *nops* instructions must be prepended to the code section being analyzed and the analysis must start at the first *nop*. Additionally, since the time tag is acquired at the end of the sixth stage, the time of the "write" pipeline stage of the last instruction is lost, and if that time is desired, a *nop* must be inserted at the end of the code section being analyzed.

Figure 16 depicts the block diagram of a LEON3 processor for sampling the timing of instructions of a given application through the DSU. The main components of the

Figure 16 – Single-core LEON3 processor block diagram used to sample instruction times.



processor are connected to the "AMBA AHB I" bus. The other components are connected to the "AMBA AHB II" bus, and are responsible for controlling the processor, retrieving the time (from the DSU), and configuring the input vectors of the application (Dual Port Register). The components are split in two buses to avoid interference in the processor timing caused by access conflicts in the bus. Since the "JTAG Debug Link" is connected to the "AMBA AHB II", and there is no connection between the two AHB buses, it is unable to load the application to the "AHBRAM Memory", therefore the application must be preloaded in the memory.

3.3.2 Timing Compilation Procedure

This procedure is responsible for compiling the values measured in Item 2 into the timing table. The timing table is responsible for mapping the address of an executed instruction I_i together with three future addresses of instructions (relative to I_i) to the latency of the I_i instruction.

The table is generated from a list of a double of instruction address and time tag $[(A_i, TT_i)]$ collected in Item 2. The latency of each entry is calculated by the difference of the current instruction time tag and the former instruction time tag i.e., $L_i = TT_i - TT_{i-1}$, e.g. L_3 in Figure 15. The table is then generated with the addresses and computed latencies.

Figure 17a shows an example trace of a task running in LEON3 processor measured with the DSU instruction trace. The latency is calculated from the trace yielding Figure 17b. For example, the latency of instruction at address 4001284 is computed by $2439-2425$ resulting in 14 CPU cycles. The timing table (Figure 17c) is then generated from the

Figure 17 – Timing compilation procedure example.

Address	Time Tag
40001214	2425
40001284	2439
40001288	2452
4000128C	2457
40001290	2458
40001294	2461
40001218	2472
4000121C	2479

(a) Trace

Address	Latency
40001284	14
40001288	13
4000128C	5
40001290	1
40001294	3
40001218	11
4000121C	7

(b) Latency Table

Instruction Address	Key				Value
	$Address_i$	$Address_{i+1}$	$Address_{i+2}$	$Address_{i+3}$	CPU Cycles
40001284	40001284	40001288	4000128C	40001290	14
40001288	40001288	4000128C	40001290	40001294	13
4000128C	4000128C	40001290	40001294	40001218	5
40001290	40001290	40001294	40001218	4000121C	1

(c) Timing Table

computed latencies and instructions address of each run, and then it is combined in a single timing table.

3.3.3 Timing Annotation

The algorithm to compute and annotate the CFG with timing information of each basic block (vertex) and edge is presented in Figure 18.

The addresses of all instructions belonging to a basic block are retrieved in lines 3 and 13. If the basic block has no out-going edges the addresses variable is appended with three zeros (line 4). The latency of an instruction is found by looking at the timing table for the four consecutive addresses in the addresses variable starting at the instruction address. The latencies of every instruction in the vertex are summed and the sum is assigned to the Time variable in the vertex. This process is depicted in lines 6 to 9 and 17 to 20.

If the basic block does have out-going edges, then the basic block time w.r.t. every out-going edge must be computed. For every edge the next three instructions address (after the basic block traversing through the edge) must be retrieved. Those values are then appended to the *addresses* variable. The time of the basic block w.r.t. that edge is then computed in lines 17 to 20. The penalty of the edge is then set to the computed time (it is later subtracted from the real minimum basic block time) and the minimum basic

Figure 18 – Basic block annotation algorithm.

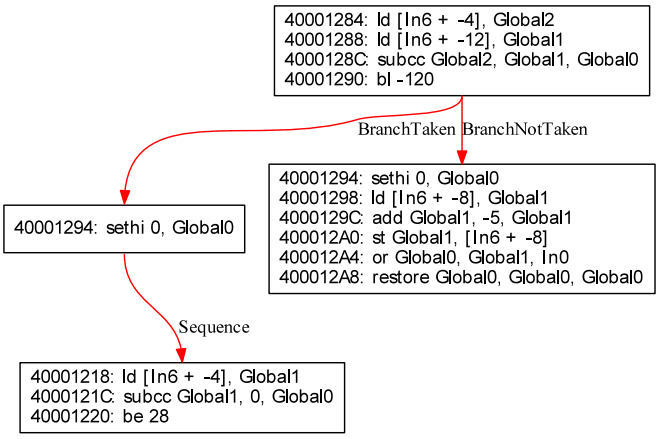
```
1: procedure AnnotateBasicBlock(vertex, timingTable)
2:     if vertex.OutEdges.Count = 0 then
3:         addresses ← vertex.GetVertexInstructionsAddress()
4:         addresses ← addresses.Append([0, 0, 0])
5:         time ← 0
6:         for i ← 0, addresses.Length − 4 do
7:             key ← addresses.Take(i, 4)
8:             time ← time + timingTable[key]
9:         end for
10:        minTime ← time
11:    else
12:        for all edge ∈ vertex.OutEdges do
13:            addresses ← vertex.GetVertexInstructionsAddress()
14:            next3 ← getNextThreeInstructionsAddress(edge)
15:            addresses ← addresses.Append(next3)
16:            time ← 0
17:            for i ← 0, addresses.Length − 4 do
18:                key ← addresses.Take(i, 4)
19:                time ← time + timingTable[key]
20:            end for
21:            edge.Penalty ← time
22:            minTime ← min(minTime, time)
23:        end for
24:    end if
25:    vertex.Time ← minTime
26:    for all edge ∈ vertex.OutEdges do
27:        edge.Penalty ← edge.Penalty − minTime
28:    end for
29: end procedure
```

block time *minTime* is updated with the minimum of the current value of *minTime* and the computed *time*.

Figure 19a depicts a section of a CFG of a task in which the algorithm in Figure 18 will be applied to the topmost vertex. Figure 19b depicts the contents of the *addresses* variable for the Branch Taken and Branch Not Taken edges. In Figure 19c shows the total CPU cycles of the vertex w.r.t. every edge. It also shows the CPU cycles of every instruction and the key used to retrieve the CPU cycles value from the timing table.

After computing the basic block time w.r.t. every edge, the basic block time is then set to the *minTime* variable and the penalty variable of every edge is subtracted from the *minTime* variable. Figure 20a shows a CFG of a task with each instruction and its respective address. The algorithm in Figure 18 was applied to every vertex of the CFG and the result is shown in Figure 20b. The annotated CFG is a composition of the CFGs from Figure 20a and 20b, containing the data of both graphs.

Figure 19 – Timing annotation example.



(a) CFG Section

Branch Taken
40001284
40001288
4000128C
40001290
40001294
40001218
4000121C
Branch Not Taken
40001284
40001288
4000128C
40001290
40001294
40001298
4000129C

(b) *addresses* Variable Contents

Branch Taken					
Inst. Address	Key				CPU Cycles
40001284	40001284	40001288	4000128C	40001290	14
40001288	40001288	4000128C	40001290	40001294	13
4000128C	4000128C	40001290	40001294	40001218	5
40001290	40001290	40001294	40001218	4000121C	1
Total:					33
Branch Not Taken					
Inst. Address	Key				CPU Cycles
40001284	40001284	40001288	4000128C	40001290	14
40001288	40001288	4000128C	40001290	40001294	13
4000128C	4000128C	40001290	40001294	40001298	5
40001290	40001290	40001294	40001298	4000129C	3
Total:					35

(c) Timing Table Entries

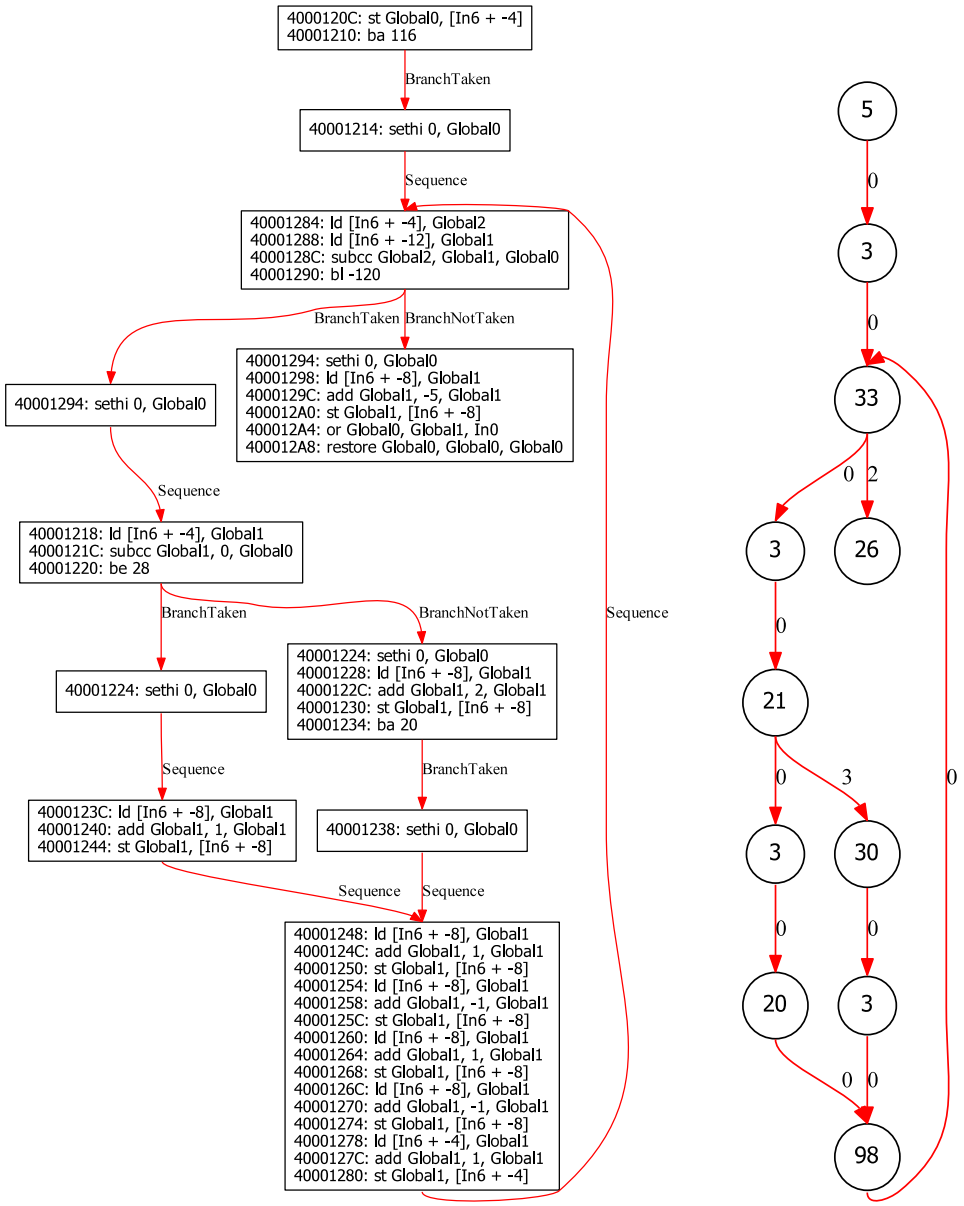
3.3.4 $WCET_R$ Computation

This step is responsible for composing the previously collected and aggregated times into the WCET and $WCET_R$, to do so, we detail here two techniques with their respective trade-offs.

3.3.4.1 Implicit Path Enumeration Technique (IPET)

A count variable x_{v_i} and a time coefficient t_{v_i} are defined for every basic block (vertex) in the CFG. Additionally a count variable x_{e_i} and a penalty coefficient p_{e_i} are defined for all edges in the graph. The WCET can be computed by maximizing the sum of the product of count variables of all vertices (edges) with the time (penalty) coefficients of all vertices (edges) ($\max(\sum_{i \in vertices} t_{v_i}\ x_{v_i} + \sum_{i \in edges} p_{e_i}\ x_{e_i})$) subject to a set of constraints. These constraints reflect the structure of the task and possible flows. The constraint of a vertex v_i is imposed into its count variable x_{v_i} and equals to the sum of the count variables of all of its incoming (out-going) edges ($x_{v_i} = \sum_{o \in in\ \ edges \in i} x_{e_o} = \sum_{o \in out\ \ edges \in i} x_{e_o}$). That is, the number of times a vertex was entered should be the same as the number of times

Figure 20 – Task CFG example.

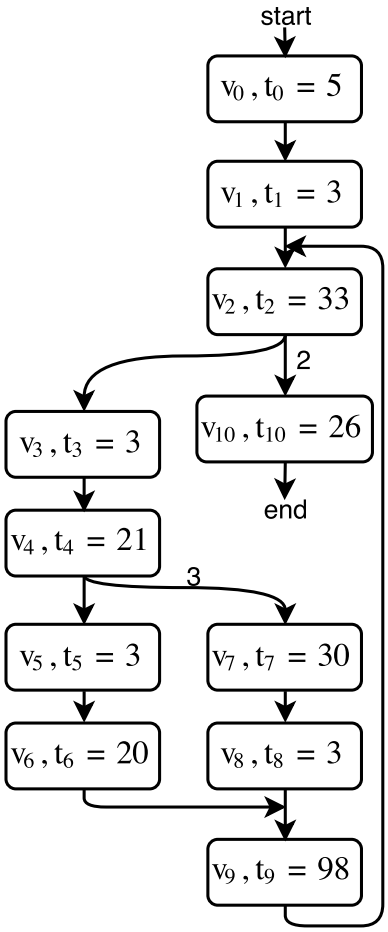


(a) CFG (b) Timed CFG

the vertex was exited. Another constraint is that the count variable should be one for start (end) vertices, that is, a task should only be entered (exited) once. Loop bounds are expressed as constraints in the loop back edge count variable.

Figure 21a depicts a CFG of a task showing the count variable x_i (x_{e_i}) and the time (penalty) coefficient t_i (p_{e_i}) of vertices (edges). Figure 21b show the maximization equation in line 1, depicting both vertices and edges. In line 16 is defined a constraint related to the loop bound $LB_{9\rightarrow 2}$, the loop is bounded to 1000. Lines 3 to 13 depicts the constraints related to the structure of task and its possible flows. The ILP formulation is solved by using the lp_solve (BERKELAAR, 2010) tool yielding the max value of 191069 CPU Cycles.

Figure 21 – IPET example.

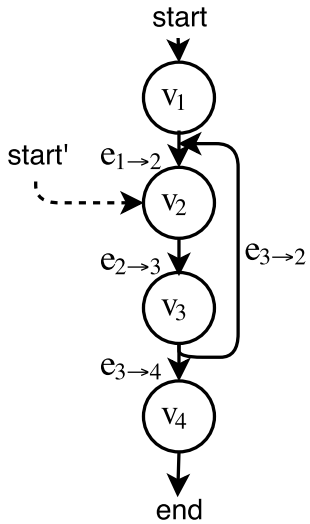


(a) Task CFG

1: $WCET = \max(5 \cdot x_0 + 3 \cdot x_1 + 33 \cdot x_2 + 3 \cdot x_3 + 21 \cdot x_4 + 3 \cdot x_5 + 20 \cdot x_6 + 30 \cdot x_7 + 3 \cdot x_8 + 98 \cdot x_9 + 26 \cdot x_{10} + 3 \cdot x_{4\to7} + 2 \cdot x_{2\to10})$
2: subject to the constraints:
3: $x_0 = x_{start} = x_{0\to1}$
4: $x_1 = x_{0\to1} = x_{1\to2}$
5: $x_2 = x_{1\to2} + x_{9\to2} = x_{2\to3} + x_{2\to10}$
6: $x_3 = x_{2\to3} = x_{3\to4}$
7: $x_4 = x_{3\to4} = x_{4\to5} + x_{4\to7}$
8: $x_5 = x_{4\to5} = x_{5\to6}$
9: $x_6 = x_{5\to6} = x_{6\to9}$
10: $x_7 = x_{4\to7} = x_{7\to8}$
11: $x_8 = x_{7\to8} = x_{8\to9}$
12: $x_9 = x_{6\to9} + x_{8\to9} = x_{9\to2}$
13: $x_{10} = x_{2\to10} = x_{end}$
14: $x_{start} = 1$
15: $x_{end} = 1$
16: $x_{9\to2} \leq LB_{9\to2} \cdot x_{1\to2}$

(b) ILP Formulation

Figure 22 – IPET $WCET_R$ example with a single loop.



(a) Task CFG

1: $WCET_r^{v_2}(n) = \max(t_1 \cdot x_1 + t_2 \cdot x_2 + t_3 \cdot x_3 + t_4 \cdot x_4 + p_{1\to2} \cdot x_{1\to2} + p_{2\to3} \cdot x_{2\to3} + p_{3\to4} \cdot x_{3\to4} + p_{3\to2} \cdot x_{3\to2})$
2: subject to the constraints:
3: $x_1 = x_{1\to2}$
4: $x_2 = \boldsymbol{x_{start'}} + x_{1\to2} + x_{3\to2} = x_{2\to3}$
5: $x_3 = x_{2\to3} = x_{3\to2} + x_{3\to4}$
6: $x_4 = x_{3\to4} = x_{end}$
7: $x_{start'} = 1$
8: $x_{end} = 1$
9: $x_{3\to2} \leq \boldsymbol{LB_{3\to2} - n}$

(b) ILP Formulation

Even though the IPET is a well established technique from the point of view of mathematical formulation and automation through the development of CAD tools, this

technique cannot by directly used to compute the $WCET_R$ parameter. In this scenario, one of the goals is to adapt the existing IPET technique in order to allow the determination of the $WCET_R$ parameter.

The process of computing the $WCET_R$ using IPET consists of defining constraints of edges of cycles depending on which iteration of the loop the $WCET_R$ must be computed. Along with those, the start of the CFG is no longer the start of IPET analysis, but it is set to the vertex in which the $WCET_R$ is desired to be computed on. Figure 22a is example of a CFG containing a single loop, and the ILP formulation for the $WCET_R^{v_2}$ of vertex 2 is depicted in Figure 22b. The presented ILP is a function of n which is the loop iteration of the edge $e_{3\to2}$ in which the $WCET_R$ is computed. The start of the CFG is replaced by a fictitious edge (start') pointing to the $WCET_R^{v_2}$ target (vertex 2), that is shown in line 4 of Figure 22b. The value of $x_{3\to2}$ is constrained to the value of the loop $LB_{3\to2}$ minus the number of iterations already performed n, e.g. if the loop is bounded to 100, and it is desired to compute the $WCET_R^{v_2}$ after 4 iterations, $x_{3\to2}$ must be less than or equal to 96. That is depicted in line 9 of Figure 22b.

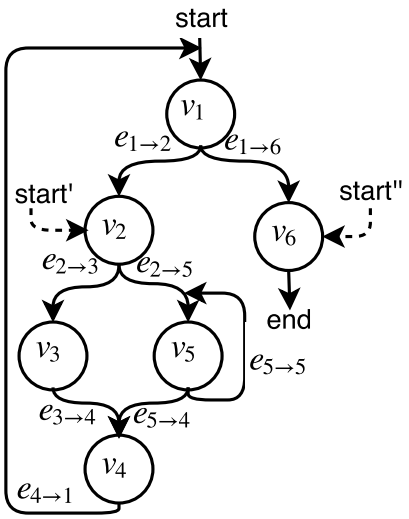
An example of a CFG with a nested loop is given in Figure 23a. In Figures 23b and 23c it is given the ILP formulation for the $WCET_R$ for vertices 2 and 3 respectively. For the vertex 2, the constraint in the count variable of the outer loop back edge $x_{4\to1}$ is simply the loop bound of the outer loop $LB_{4\to1}$ minus the number of iterations already performed of the outer loop n. The constraint in the inner loop count variable back edge $x_{5\to5}$ is the product of the loop bound of the inner loop $LB_{5\to5}$ by the count variable of the in edge of the header of the cycle $x_{2\to5}$. Those constraints are shown in the lines 11 and 12 of Figure 23b.

For the cycles constraint formulation we define the reachable edges (re) of the graph starting at the analysis starting point, and the cycle *back* edge in which the constraint is set $e_c = e_{t\to h}$. We define an auxiliary variable α that is given by the expression $\alpha = (ie_{h_c} \setminus e_{t\to h}) \cap re$, where ie_{h_c} is a set containing the incoming edges of the header (v_h) of the cycle. The cycle constraint is given by:

$$x_{t\to h} \leq \begin{cases} 0 & \text{if } re \cap e_{t\to h} = \varnothing \\ LB_{t\to h} & \text{if } re \cap e_{t\to h} \neq \varnothing \wedge \alpha = \varnothing \\ LB_{t\to h} \cdot \sum_{e\in\alpha} x_e & \text{if } re \cap e_{t\to h} \neq \varnothing \wedge \alpha \neq \varnothing \end{cases} \tag{3}$$

Where the set α contains the incoming edges of the header of the cycle, which is reachable and it is not the back edge of the cycle. The first condition is satisfied if the cycle back edge is not reachable, that is, the cycle cannot be reached from the vertex in which the $WCET_R$ is desired to be computed at. In this case the loop bound of the cycle is actually 0, therefore the loop does not contribute to the $WCET_R$.

Figure 23 – IPET $WCET_R$ example with a nested loop.



(a) Task CFG

1: $WCET_r^{v_2}(n) = \max(t_1 \cdot x_1 + t_2 \cdot x_2 + t_3 \cdot x_3 + t_4 \cdot x_4 + t_5 \cdot x_5 + t_6 \cdot x_6 + p_{1\to2} \cdot x_{1\to2} + p_{1\to6} \cdot x_{1\to6} + p_{2\to3} \cdot x_{2\to3} + p_{2\to5} \cdot x_{2\to5} + p_{5\to5} \cdot x_{5\to5} + p_{3\to4} \cdot x_{3\to4} + p_{5\to4} \cdot x_{5\to4} + p_{4\to1} \cdot x_{4\to1})$
2: subject to the constraints:
3: $x_1 = x_{4\to1} = x_{1\to2} + x_{1\to6}$
4: $x_2 = \boldsymbol{x_{start'}} + x_{1\to2} = x_{2\to3} + x_{2\to5}$
5: $x_3 = x_{2\to3} = x_{3\to4}$
6: $x_4 = x_{3\to4} + x_{5\to4} = x_{4\to1}$
7: $x_5 = x_{2\to5} + x_{5\to5} = x_{5\to4} + x_{5\to5}$
8: $x_6 = x_{1\to6} = x_{end}$
9: $x_{start'} = 1$
10: $x_{end} = 1$
11: $x_{4\to1} \leq \boldsymbol{LB_{4\to1} - n}$
12: $x_{5\to5} \leq \boldsymbol{LB_{5\to5} \cdot x_{2\to5}}$

(b) ILP Formulation

1: $WCET_r^{v_6}(n) = \max(t_1 \cdot x_1 + t_2 \cdot x_2 + t_3 \cdot x_3 + t_4 \cdot x_4 + t_5 \cdot x_5 + t_6 \cdot x_6 + p_{1\to2} \cdot x_{1\to2} + p_{1\to6} \cdot x_{1\to6} + p_{2\to3} \cdot x_{2\to3} + p_{2\to5} \cdot x_{2\to5} + p_{5\to5} \cdot x_{5\to5} + p_{3\to4} \cdot x_{3\to4} + p_{5\to4} \cdot x_{5\to4} + p_{4\to1} \cdot x_{4\to1})$
2: subject to the constraints:
3: $x_1 = x_{4\to1} = x_{1\to2} + x_{1\to6}$
4: $x_2 = x_{1\to2} = x_{2\to3} + x_{2\to5}$
5: $x_3 = x_{2\to3} = x_{3\to4}$
6: $x_4 = x_{3\to4} + x_{5\to4} = x_{4\to1}$
7: $x_5 = x_{2\to5} + x_{5\to5} = x_{5\to4} + x_{5\to5}$
8: $x_6 = \boldsymbol{x_{start''}} + x_{1\to6} = x_{end}$
9: $x_{start''} = 1$
10: $x_{end} = 1$
11: $x_{4\to1} \leq 0$
12: $x_{5\to5} \leq 0$

(c) ILP Formulation

The second condition happens when the cycle back edge is reachable and the set α is empty, that is, the header of the cycle is only reachable through the cycle back edge, therefore the constraint must be set to the actual loop bound provided by the user.

The third condition happens when the cycle back edge is reachable and the set α is not empty, then the constraint must be set to the loop bound provided by the user multiplied to the sum of the count variable of the edges in the α set.

In Figure 22a, the reachable edges considering the vertex 2 are $\{e_{2\to3}, e_{3\to4}, e_{3\to2}\}$, the incoming edges of the header of the cycle $e_{3\to2}$ are $\{e_{1\to2}, e_{3\to2}\}$, therefore the set α

is empty. Since the cycle back edge ($e_{3\to2}$) is in the reachable edges, the constraint of the cycle is given simply by the loop bound ($LB_{e_{3\to2}}$) minus n.

In Figure 23a for $start'$, the reachable edges are $\{e_{2\to3}, e_{2\to5}, e_{3\to4}, e_{5\to4}, e_{4\to1}, e_{1\to2}\}$, the incoming edges of header of the cycle $e_{5\to5}$ are $\{e_{2\to5}, e_{5\to5}\}$, therefore $\alpha = \{e_{2\to5}\}$ and since the cycle is contained in the reachable edges and α is not empty, the cycle back edge constraint is given by the third condition in Equation 3, yielding $x_{5\to5} \leq LB_{5\to5} \cdot x_{2\to5}$.

For $start''$, the reachable edges are $\varnothing$, therefore, since the cycle $e_{5\to5}$ is not contained in the reachable edges, the cycle back edge constraint is given by the first condition, and it is $x_{5\to5} \leq 0$.

3.3.4.2 Graph Traversal Technique (GTT)

The graph traversal consists of a simple depth first (CORMEN et al., 2009) traversal that computes the $WCET_R$ of all vertices in a CFG. The maximum CPU cycles of a vertex (T_{v_i}) is computed by summing the time coefficient of the vertex (t_{v_i}) with the maximum CPU cycles of its descendants considering the penalty of traversal through the descendants edges ($p_{e_{t\to h}}$). That is, $T_{v_i} = t_{v_i} + \max(T_{v_h} + p_{e_{t\to h}})$, where $e_{t\to h}$ is a directed edge pointing from v_t to v_h, and $e_{t\to h} \in oe_{v_i}$ where oe_{v_i} is the vertex v_i out-going edges.

In Figure 24 the maximum CPU clock cycles for vertices v_2 and v_3 was previously computed, and the maximum CPU clock cycles for vertex v_1 is computed by the expression $T_{v_1} = t_{v_1} + \max(T_{v_2} + p_{e_{1\to2}}, T_{v_3} + p_{e_{1\to3}})$ yielding the value 176.

Figure 24 – Vertex max CPU cycles example.

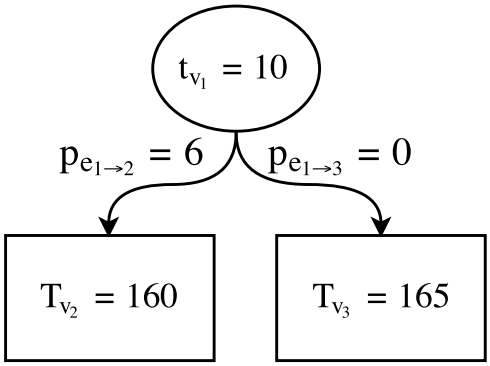
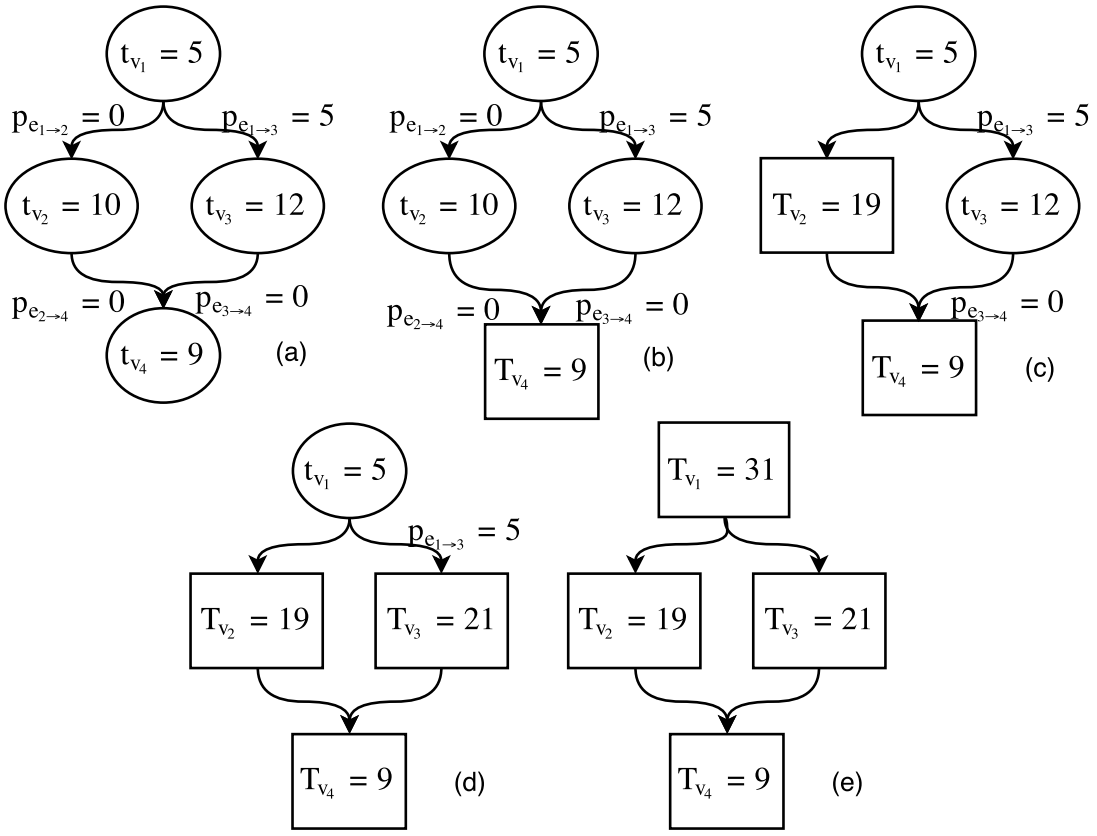


Figure 25 depicts an example of graph traversal in order to compute the maximum CPU cycles of the basic blocks in the graph. The algorithm initially starts at the vertex v_1, traversing through its first out-going edge $e_{1\to2}$ and then to its only out-going edge $e_{2\to4}$. Since the vertex v_4 has no out-going edges, its maximum CPU cycles is set to its own timing coefficient t_{v_4} (Figure 25b). The algorithm then traverses back to v_2, and since it has no other out-going edges, T_{v_2} is then set to $t_{v_2} + T_{v_4} + p_{e_{2\to4}}$ (Figure 25c). Back at vertex v_1, the algorithm must now traverse through edge $e_{1\to3}$ and then through $e_{3\to4}$, retrieving the value of T_{v_4} and traversing back to v_3, computing $T_{v_3} = t_{v_3} + T_{v_4}$ (Figure

25d). Traversing back to vertex v_1, T_{v_1} is computed by $t_{v_1} + max(T_{v_2} + p_{e_{1\to2}}, T_{v_3} + p_{e_{1\to3}})$, yielding the value of $T_{v_1} = 31$ (Figure 25e).

Figure 25 – Graph Traversal example.



The presented algorithm is only capable of computing the WCET for directed acyclic graphs, thus whenever the analyzed task has loops, it cannot be analyzed by the algorithm. To overcome that limitation, it is introduced a cycles state variable ($S_{e_{t\to h}}$) to the traversal algorithm, which stores the number of times that a cycle back edge was traversed. Then the constraint of the cycle state being lower than the loop bound of the respective cycle is added to every cycle back edge of the graph. By doing so, the algorithm traverses through edges in which its constraints is satisfied. Upon traversing through a cycle back edge the state of that edge is incremented, and the state is cleared when traversing through incoming edges of the cycle header (excluding the cycle back edge).

Figure 26a shows a CFG example containing a single cycle, which its back edge denoted by $e_{3\to2}$, the state of that cycle is denoted by $S_{e_{3\to2}}$. When traversing through the edge $e_{1\to2}$ the state $S_{e_{3\to2}}$ is cleared. In the cycle back edge $e_{3\to2}$ the cycle state is incremented, and the edge also posses a constraint $C_{e_{3\to2}}$ that only allows the traversal through the edge when the cycle state is lower than the loop bound $LB_{e_{3\to2}}$. Figure 26b is a tree containing all paths that is analyzed by the graph traversal algorithm, considering $LB_{e_{3\to2}} = 3$. Figure 26c is a tree of all paths traversed through the CFG depicted in Figure

26a by the algorithm, showing the $WCET_R$ of each vertex given the states it depends on, e.g., vertex v_2 depends in the state $S_{e_{3\to2}}$. Therefore, in this example, it posses three $WCET_R$. Since the vertices v_1 and v_4, does not belongs to any cycle, each vertex has only a single $WCET_R$. The algorithm exemplified in Figure 25 can be applied to Figure 26b yielding the $WCET_R$ of all vertices, shown in Figure 26. In the case of inner loops, the vertices belonging to an inner loop depends both on the state of the outer and inner loop, although the outer loop only depends on its own state, this is shown in Figure 27 where both loops have a bound of 2. The previous algorithm is detailed in Figure 28 and is applied to the root of the graph (the first basic block of the task).

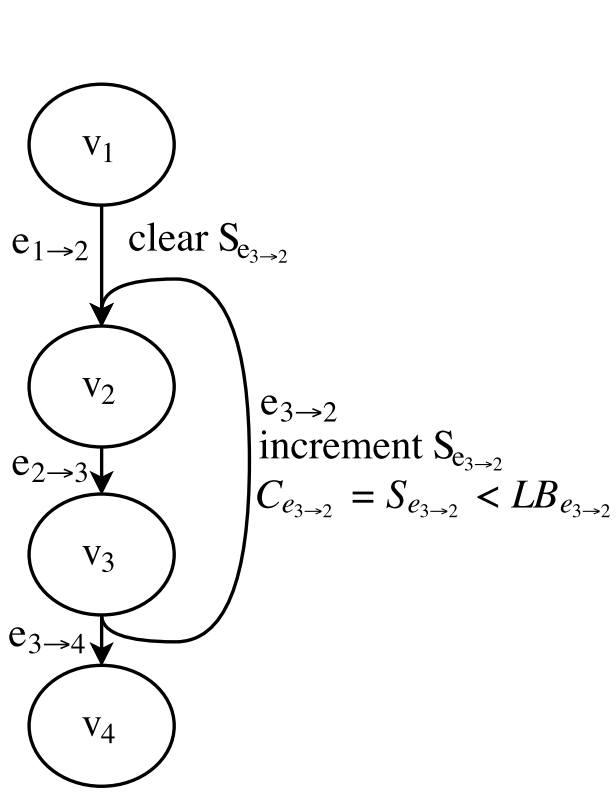
The presented algorithm for computing the timing bound of basic blocks will most likely compute the timing bound for a vertex several times, one time for each path leading to the vertex. That has a drastic effect in the algorithm performance. The algorithm is then improved to perform memoization (MICHIE, 1968) in the computed timing bound estimates of vertices. It works by caching the computed values w.r.t. the vertex and cycles state. Since the timing bound of a vertex inside a loop in the first iteration will be different from any other iteration (and between iterations as well) the memoization should also consider the cycles state variable on the computation. It is important to note that the $WCET_R$ of every vertex is stored in the memoization cache w.r.t. every cycles states variable the vertex depends on. Therefore memoization, in this specific case, serves two purposes, performance gain and storing the $WCET_R$ of the vertices to be read when the computation terminates.

The memoized algorithm is shown in Figure 29. The algorithm performance is compared with and without memoization for a task which its CFG is shown in Figure 20a yielding Figures 30b and 30a respectively, which depicts the WCET computation time for different values of loop bounds. The memoized algorithm shows severals orders of magnitude of performance gain compared to the non-memoized algorithm. Additionally for that specific task, the time complexity of the algorithm changed from exponential to quadratic w.r.t. the loop bound.

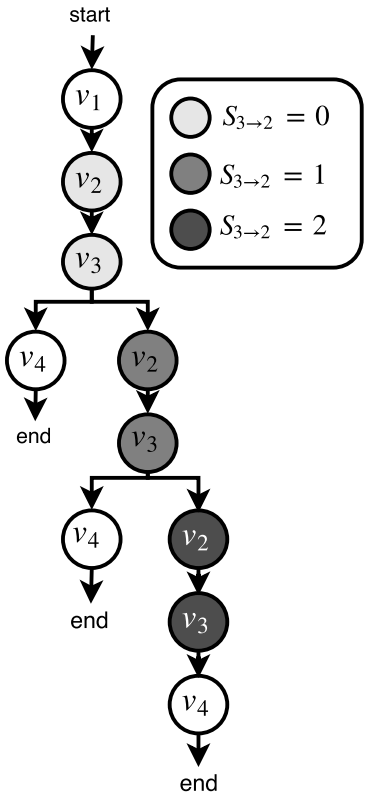
3.3.4.3 WCET Computation Technique Comparison

To compute the $WCET_R$ of N reference points the GTT algorithm is run a single time while for the IPET, N ILP formulations are generated and solved. IPET posses a lower computational complexity and vastly superior performance compared to GTT. Although for GTT, the implementation effort is minimal: the user must only provide the loop bounds of the analyzed task. Whereas for IPET the user must provide the constraints for the cycles considering their loop bound and what iteration of a loop the $WCET_R$ analysis takes place.

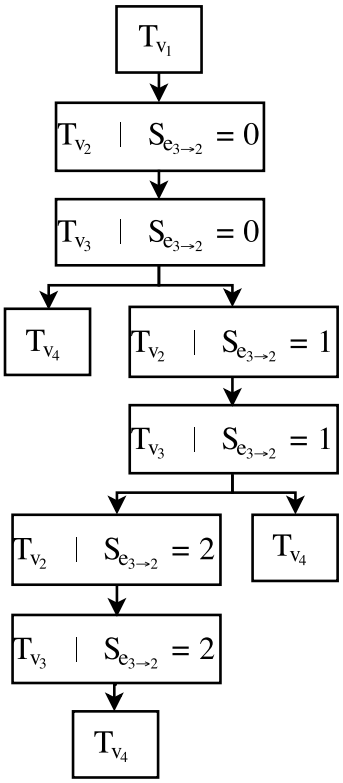
Figure 26 – GTT example with a loop.



(a) CFG

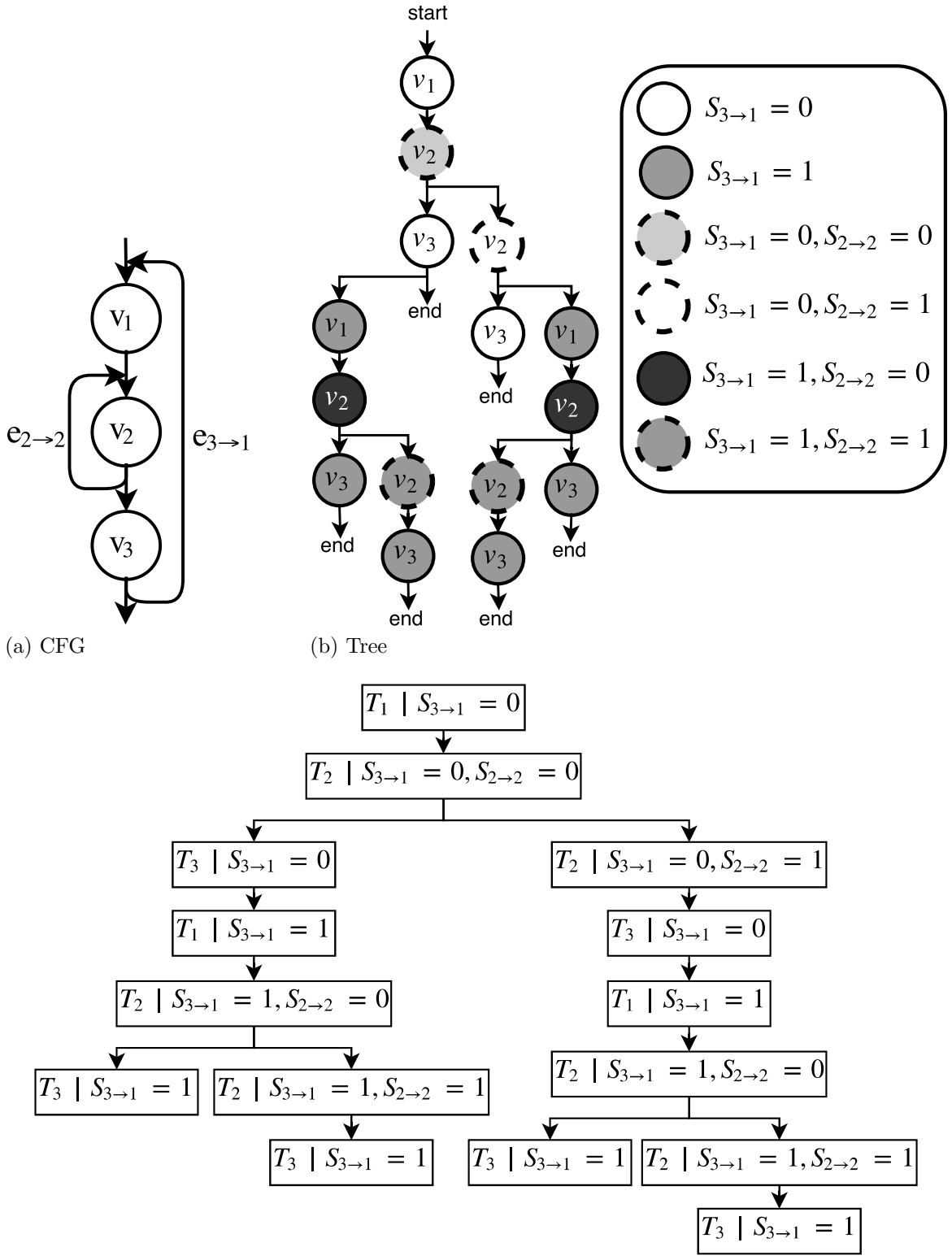


(b) Tree



(c) Tree

Figure 27 – GTT example with a nested loop.



(a) CFG (b) Tree

(c) Tree

The algorithm in Appendix A is used by the memoized GTT to identify the vertices dependence of cycles in the CFG. Those dependencies are used as key for the memoization, identifying these dependencies improperly results in multiple entries in the memoization

Figure 28 – ComputeVertexCycles algorithm.

```
1: procedure ComputeVertexCycles(vertex, cyclesState)
2:     cycles ← 0
3:     hasValidEdges ← false
4:     for all edge ∈ vertex.OutEdges do
5:         if edge constraints validated then
6:             newCyclesState ← copy of cyclesState
7:             if edge is a back edge then
8:                 Increment cycle state of edge S_edge.
9:             end if
10:            if edge points to a cycle header and is not a cycle back edge then
11:                Clear cycle state of the edge cycle.
12:            end if
13:            (eTime, result) ← ComputeVertexCycles(edge.head, newCyclesState)
14:            if result = true then
15:                hasValidEdges ← true
16:                eTime ← eTime + edge.Penalty
17:                if cycles < eTime then
18:                    cycles ← eTime
19:                end if
20:            end if
21:        end if
22:    end for
23:    if ¬hasValidEdges ∧ vertex has OutEdges then
24:        return (0, false)
25:    end if
26:    cycles ← cycles + vertex.Time
27:    return (cycles, true)
28: end procedure
```

Table 2 – GTT and IPET comparison.

Technique	Performance	Parameters Definition
GTT	Lower	Simple
IPET	Higher	Complex

for the same key, resulting in the failure of the GTT algorithm. The CFG in Figure 31a is known to have the dependencies of cycles improperly identified, requiring manual dependency identification by the user for the proper WCET and $WCET_R$ computation. The failure is caused by the interleaved loops, possibly caused by a "goto" statement placed in vertex 3 by the programmer or might have been caused by a compiler optimization, although that possibility was not analyzed.

Two cycles is detected in the CFG of Figure 31b while it is a single loop, it will cause a misscomputation in the WCET. The given CFG was detected to happen in

Figure 29 – MemoizedComputeVertexCycles algorithm.

```
1: procedure MemoizedComputeVertexCycles(vertex, cyclesState)
2:     (cycles, result, found) ← findMemoizedEntry(vertex, cyclesState)
3:     if found = true then
4:         return (cycles, result)
5:     end if
6:     cycles ← 0
7:     hasValidEdges ← false
8:     for all edge ∈ vertex.OutEdges do
9:         if edge constraints validated then
10:            newCyclesState ← copy of cyclesState
11:            if edge is a back edge then
12:                Increment cycle state of edge S_edge.
13:            end if
14:            if edge points to a cycle header and is not a cycle back edge then
15:                Clear cycle state of the edge cycle.
16:            end if
17:            (eTime, result) ← MemoizedComputeVertexCycles(edge.head,
    newCyclesState)
18:            if result = true then
19:                hasValidEdges ← true
20:                eTime ← eTime + edge.Penalty
21:                if cycles < eTime then
22:                    cycles ← eTime
23:                end if
24:            end if
25:        end if
26:    end for
27:    if ¬hasValidEdges ∧ vertex has OutEdges then
28:        addMemoizationEntry(vertex, cyclesState, (0, false))
29:        return (0, false)
30:    end if
31:    cycles ← cycles + vertex.Time
32:    addMemoizationEntry(vertex, cyclesState, (cycles, true))
33:    return (cycles, true)
34: end procedure
```

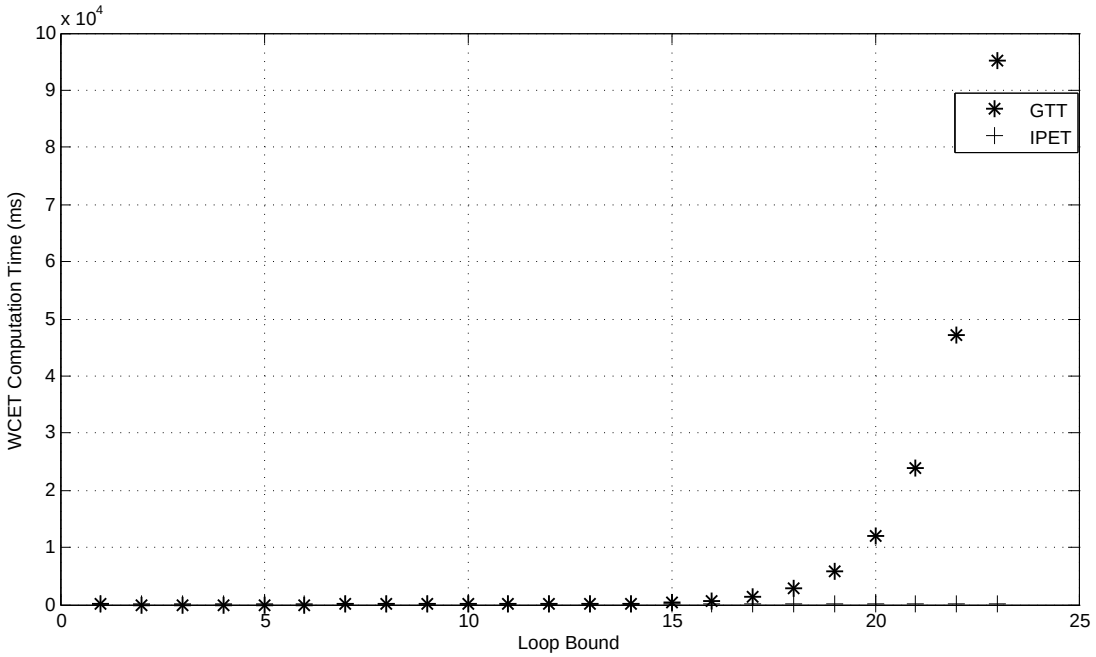
some applications compiled using GCC optimization levels 2 and 3, therefore it is not recommended using GCC optimizations.

3.3.4.4 Limitations of the GTT and IPET tools

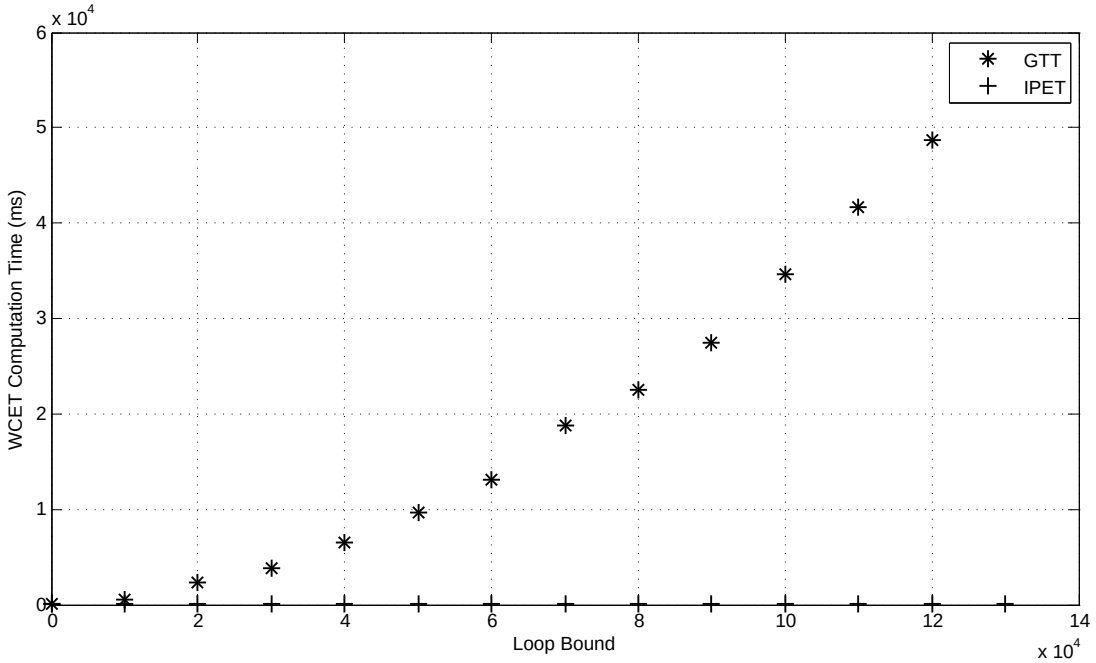
The limitations of the tools are as follows:

- The use of function pointers is forbidden, because the call address is only known at runtime.

Figure 30 – WCET computation time comparison.



(a) Without Memoization

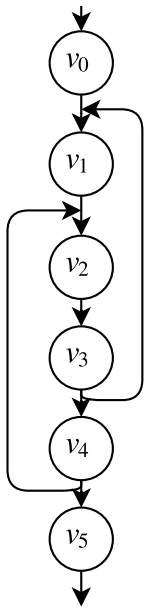


(b) With Memoization

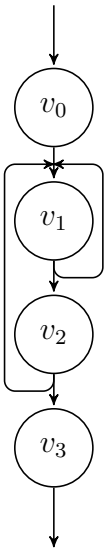
- Traps (i.e., interruptions) are not analyzed in the tool, therefore the programmer must guarantee that the application does not generate traps. Otherwise the estimated WCET and $WCET_R$ are mispredicted.

- When compiling code that uses register windows, the programmer must guarantee that the register window will not overflow. If so, it would generate a trap.

Figure 31 – CFG Structures known to cause failures in the implemented GTT tool.



(a) Failure caused by CFG structure.



(b) Error caused by a loop with two back edges.

- The tools compute the $WCET_R$ with a basic block granularity, that is, the $WCET_R$ is related to the first instruction of the basic block. In other words, it is not possible to compute the $WCET_R$ of instructions embedded in a basic block.

- The use of the "goto" statement is discouraged since it can generate a CFG structure such as the one seen in Figure 31a. That is known to cause failure of the WCET and $WCET_R$ process computation when using the GTT tool.

- The tools are not able to compute the WCET and $WCET_R$ when using instruction and/or data caches, as it would require the development of a processor model or to use probabilistic methods (BERNAT; COLIN; PETTERS, 2002; BERNAT; COLIN; PETTERS, 2003; BERNAT; BURNS; NEWBY, 2005; BURNS; EDGAR, 2000).

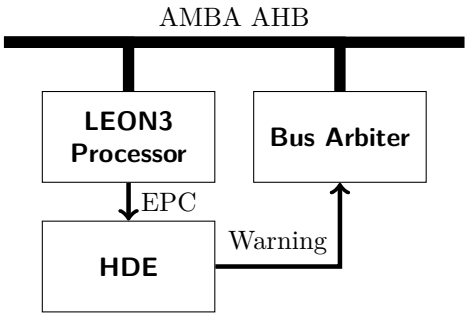
- The tools do not take into account the LEON3 write-buffer to compute the WCET and $WCET_R$ estimations. Therefore, if the buffer is empty during a given measurement sequence, but it is full during runtime, it is our understanding that WCET and $WCET_R$ would be underestimated. A solution to this problem would be to perform exhaustive measurements trying to exercise that condition (write-buffer full), but there is no guarantee that condition would be satisfied. Another (safer) option would be to modify the memory access policy from write-through (which is based on the write-buffer) to the write-back policy (where there is no write-buffer).

3.4 Hard Deadline Enforcer Development (HDE): a case-study on the LEON3 dual-core processor

3.4.1 HDE Interconnection with MPSoC

Figure 32 depicts a block diagram of the interconnection between the LEON3 processor and the HDE. The processor was modified to export the "Executed Program Counter (EPC)", which is set to the value of the PC in the sixth stage of the pipeline whenever the "Annul" bit at the sixth stage is "0". The annul bit informs if the instruction was executed. The modification is shown in Appendix B.

Figure 32 – Block diagram of the interconnection of the LEON3 processor with the HDE.



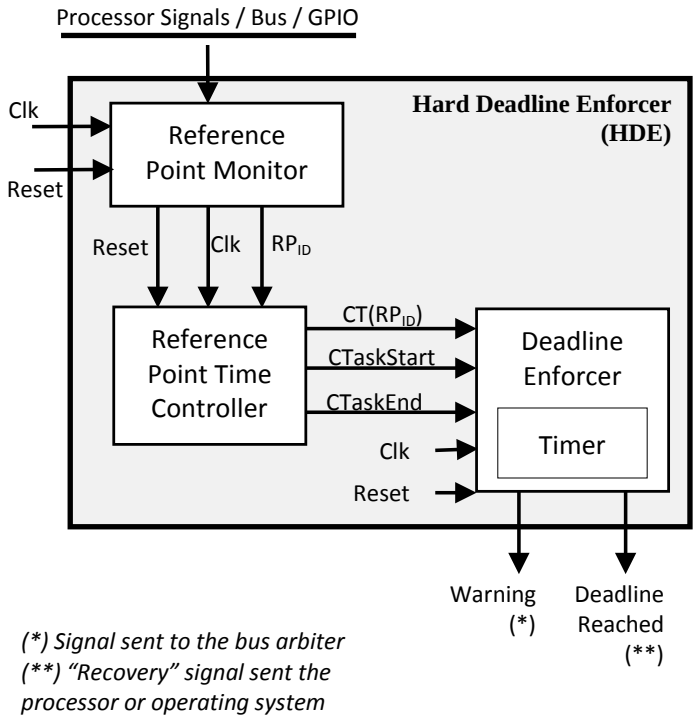
The "Warning" output signal of the HDE is connected to the bus arbiter, and is responsible for informing when the bus must be set to stand-alone mode to guarantee the timing deadline constraint of the critical task. The modification in the bus arbiter to support the stand-alone mode is depicted in Appendix C.

3.4.2 HDE Architecture

Figure 33 depicts a general view of the HDE internal blocks and their respective signals. The first block is the Reference Point Monitor, whose goal is to identify the current RP of the task. This block can accomplish such goal in different ways (depending on which information type is treated by the block) which are briefly described below:

a) The Reference Point Monitor Block observes the instruction address bus for memory reads. By comparing the instruction addresses flowing through the bus, this block detects when monitored core reaches a specific RP. A drawback of such approach is that there is no guarantee that the instruction read is executed by the processor; for example the instructions fetched just after a branch or a speculative execution by the CPU. As consequence, it constrains the locations where a RP can be inserted (since a RP cannot be inserted just after a branch or an instruction that is speculatively executed). Another problem is that if the processor has an instruction cache, then

Figure 33 – General view of the HDE internal blocks and their respective signals.



the address of the executed instruction will not appear in the bus in case of a cache hit.

b) Additionally to observe the instruction address bus, the Reference Point Monitor Block can also inspect the critical core internal signals (such as the program counter - PC). Unlike the previous method depicted in (a), by monitoring the internal signals it is easy to know whether an instruction is executed or not. Therefore relaxing the locations in which RPs can be inserted (this approach enlarges the universe of locations a RP can be inserted and so, it is a better option compared to the previous one). Nevertheless, this method is more intrusive.

c) Finally, the Reference Point Monitor Block can receive explicit information about the RP currently reached by the core, directly from processor general purpose I/O pins (e.g., GPIO port of the processor or any other communication channel). In this case, the Reference Point Monitor Block of the HDE is omitted and the RP Identification (RP_{ID}) is fed directly to the second block (Reference Point Time Controller). However, this method has an important drawback: higher detection latency compared to the methods (a) and (b) described above. Moreover, the critical task must be modified to write the RP_{ID} on the GPIO port of the processor or on any other communication channel. It should be noted that system designers are hesitant about this change in the user code. Although, it has the advantage that it does not need to access any processor internal signal or access to the processor bus.

Therefore, it can potentially work with processors in which system designers do not have access to the bus or internal processor signals (for instance, the processor is a "black-box" third-part intellectual property core).

Among the three methods above, (b) is the one currently implemented in the HDE to demonstrate the validity of the proposed technique. Figure 34 depicts an example of a Reference Point Monitor of a task that its CFG is shown in Figure 34a. The addresses of each instruction in the task is shown in the basic blocks of the task CFG. Two reference points (RP_1 and RP_2) was placed at the address 22 at different iterations of the loop. The reference point RP_0 relates to the start of the task. Table 3 shows the reference points placed in the task and their address and at which iteration of the loop it was placed. Figure 34b depicts the flowchart of the operation of the Reference Point Monitor example, that has as input the Executed Program Counter (EPC) and has as output the Reference Point Identifier (RP_{ID}). The indicated points of this figure are detailed as follows:

a) At this moment, the RP_{ID} is set to the number of reference points (3), which by convention means that the task is not running. This happens either at the start or reset of the reference point monitor, or when the EPC is equal to the address of the instruction before the end of the task (38).

b) At this moment, the EPC is equal to the reference points RP_1 and RP_2 address, although the RP_{ID} is only set to any of these if the cycle state matches 0 or 5 respectively.

c) At this moment, the EPC is equal to the first instruction of the task, therefore the RP_{ID} is set to the RP_0 id (0).

d) At this moment, the EPC is equal to the address of the instruction before entering the loop, therefore the cycle state is cleared.

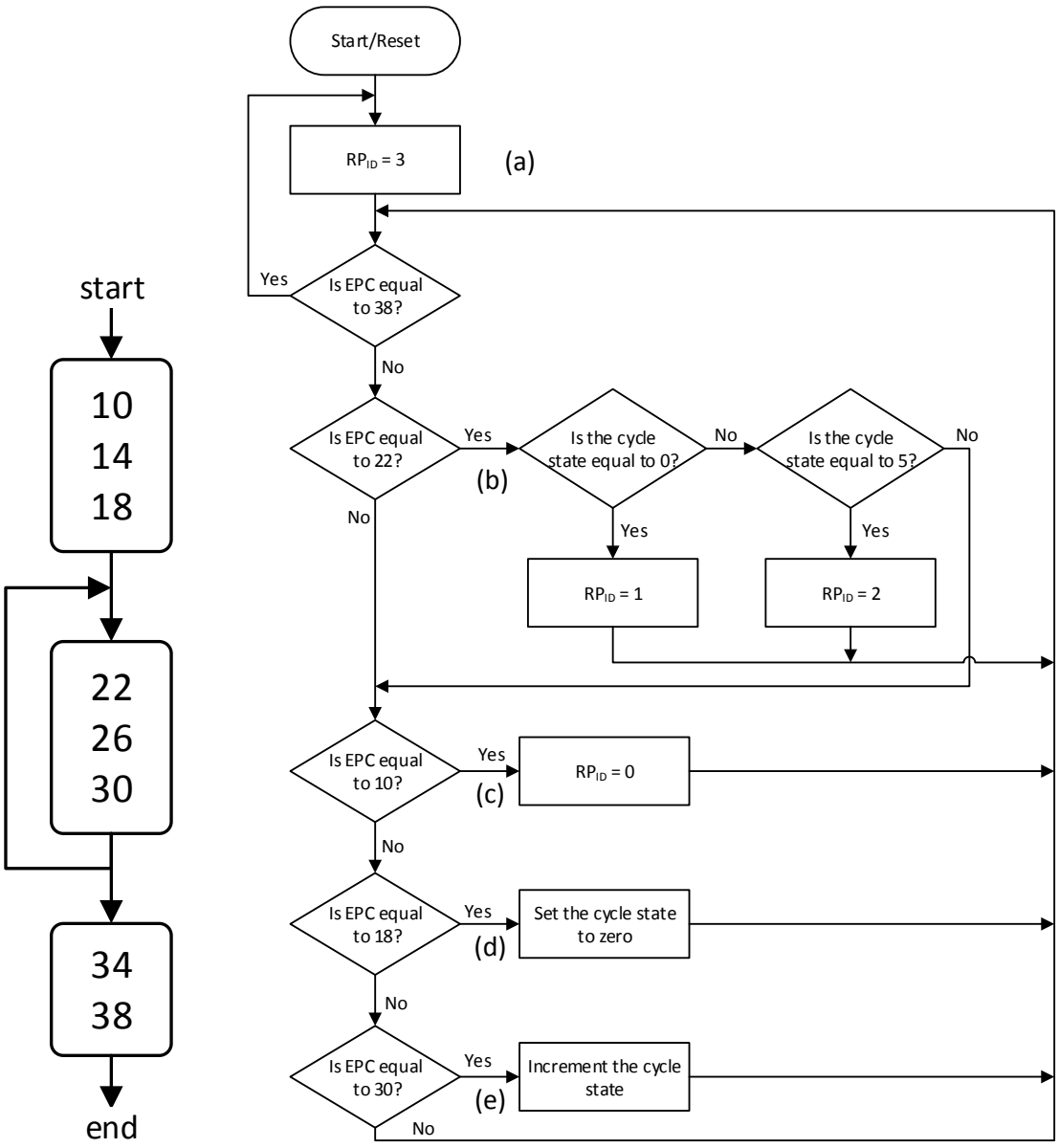
e) At this moment, the EPC is equal to the address of the instruction before to either traversing out of the loop or to the beginning of it, so the cycle state is incremented.

Table 3 – Reference points of the Reference Point Monitor example.

Reference Point	Address	Cycle State
RP_0	10	X
RP_1	22	0
RP_2	22	5

The output of the Reference Point Monitor Block is a Reference Point Identification (RP_{ID}) that is fed to the Reference Point Time Controller, whose responsibility is to

Figure 34 – Reference point monitor example.



(a) Task CFG. (b) Flowchart of the Reference Point Monitor.

correlate a RP_{ID} with its Critical Time CT(RP_{ID}). This mapping process is performed by a decoder: the decoder input is a RP_{ID} and the output is the corresponding CT(RP_{ID}). The Reference Point Time Controller also notifies when the critical task starts running (CTaskStart) and ends (CTaskEnd). The final block (Deadline Enforcer) is responsible for notifying the event "Warning". When this event is yielded, the bus controller forces the processor to switch execution from the "shared-bus" mode into the "stand-alone" one (in which the bus is exclusively allocated to the critical core). This block also signals the "Deadline Reached", which yields an "Error Indication" signal.

Figure 35 depicts the flowchart of the HDE operation. The indicated points in this figure are detailed as follows:

1. By default when a critical task starts running, the processor bus is set on the "shared-bus" mode.

2. The $CT(RP_n)$ constant is compared and one of the three previously mentioned situations may occur: (a) the elapsed time is smaller than the $CT(RP_n)$; (b) the elapsed time is equal to or greater than the $CT(RP_n)$; and (c) the elapsed time is greater than the "Deadline" and then, the HDE issues an "Error Indication" signal.

3. At this moment, the processor is running in the "shared-bus" mode.

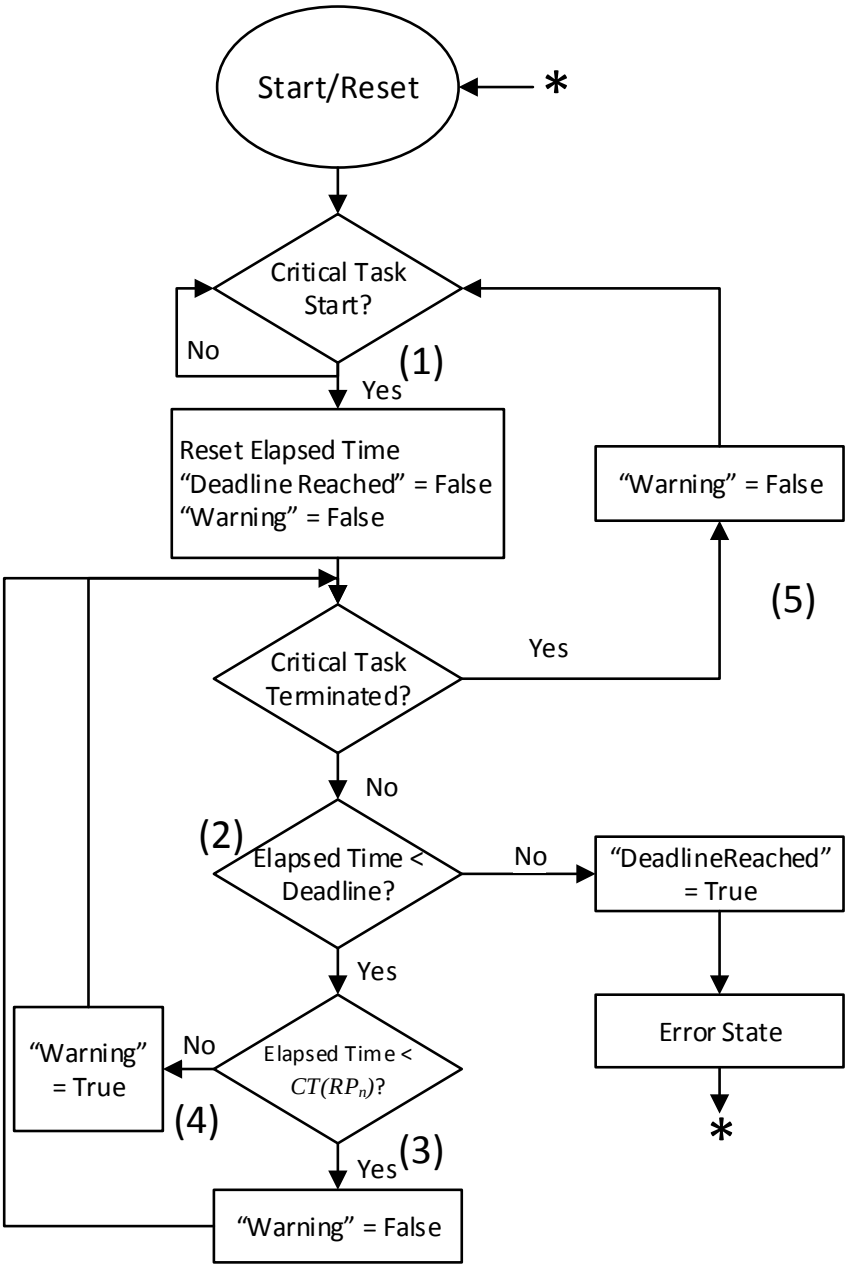
4. At this moment, the bus arbiter switches processor operation from "shared" to "stand-alone" mode.

5. Critical task (being executed in the "shared" or "stand-alone" mode) terminates, conditionally that it does not violates deadline.

3.4.3 Limitations in the Insertion of Reference Points

Reference points cannot be placed in the delay instructions, that is, the instruction right after a branch. Additionally reference points cannot be placed inside functions that are called in more than one location within the critical task. It happens because since the HDE just looks at the address and cycle states, it cannot differentiate between those calls. An workaround is to copy the function with a different name, although it increases the application size.

Figure 35 – Flowchart of the HDE operation.



4 Validation & Evaluation

The hard deadline enforcer (HDE) is validated and evaluated in the LEON3 processor. The $WCET_R$ times are estimated using methods described in section 3.3. Due to the limitations in the $WCET_R$ computation method, the instruction and data caches of the LEON3 processor are disabled. Furthermore, the tasks call depth are guaranteed to be within the limit of the configured number of window registers, therefore the "save" and "restore" instructions used on function calls/return does not yields a trap. The bus arbiter used the round-robin arbitration policy, and the critical processor is the master '0' of the bus. Additionally the processor was synthesized without MMU, and without external memory, the on-chip "AHBRAM" module was used instead, which is mapped to BlockRAMs on Xilinx devices.

4.1 Preliminary Results

The ISE-Xilinx design framework was used to configure and validate the embedded system (i.e., the processor, the HDE and the four application programs to be described later). The whole system was mapped into a Xilinx Spartan 3E Field Programmable Gate Array (FPGA).

The technique is validated against four application programs:

1. "Fibonacci Number Computation": the application computes a Fibonacci number randomly, up to F_{30}, modified from (GUSTAFSSON et al., 2010).
2. "Bubble Sort": a Random list is sorted using the bubble-sort algorithm.
3. "Dummy Application": comprises of an outer loop in which its inner loop might be executed at random.
4. "Heap": consists of pushing N random elements to a binary heap and popping the elements afterwards.

Figure 36 presents the simplified CFGs of the application programs. As observed, the simplest CFG is the Fibonacci code, whereas the most complex is the Heap one. The Bubble Sort and the Dummy Application present an average complexity. Figure 37 depicts a print-screen collected during system run in the ISim HDL simulator. As observed in this figure, bus mode is continuously switching the operating mode (between "shared" and "stand-alone") during critical task execution, in order to meet the best trade-off between "the highest possible task concurrency" against "the guarantee of critical task

schedulability". At the end of this process, the critical task terminates at 3.15ms whereas its deadline is set at 3.45ms. Note that since the critical task terminated before the deadline, there was no "deadline violation" and so, the signal "Deadline Miss", which is the HDE's output, remained at "0" during the whole simulation period ("Deadline Miss" = "1" means "error occurrence due to deadline violation"). It is worth mentioning that the timing latency for the bus controller to switch the processor from the "shared" to the "stand-alone" mode and vice-versa was measured as 1cc (clock cycles). This value plus 3 cc of detection latency of the HDE was used to setup the t_{over} (turn-over) parameter of equation (1).

Figure 36 – Control-flow graphs (CFGs) of the application codes used to validate the proposed approach: (a) Fibonacci Number Computation, (b) Bubble Sort, (c) Dummy Application and (d) Heap.

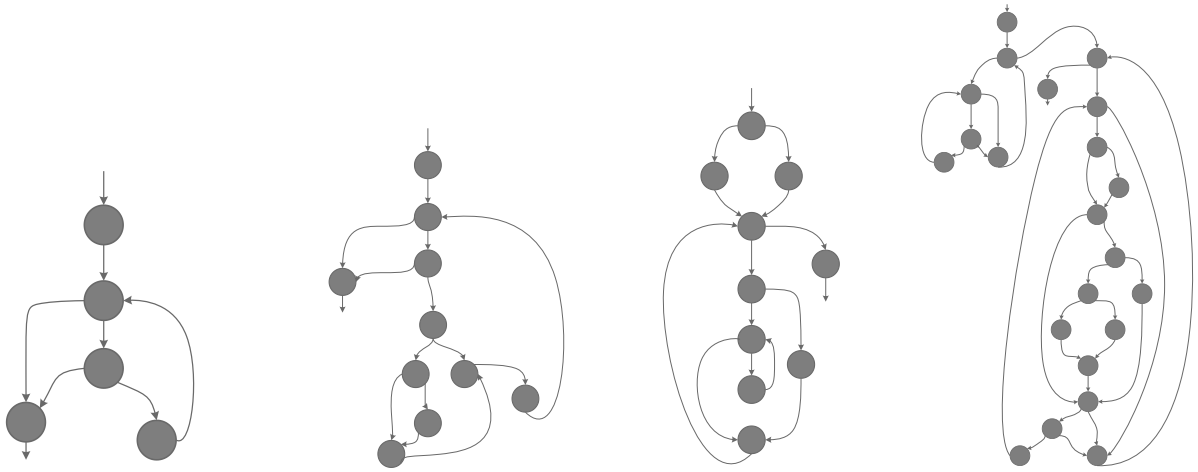
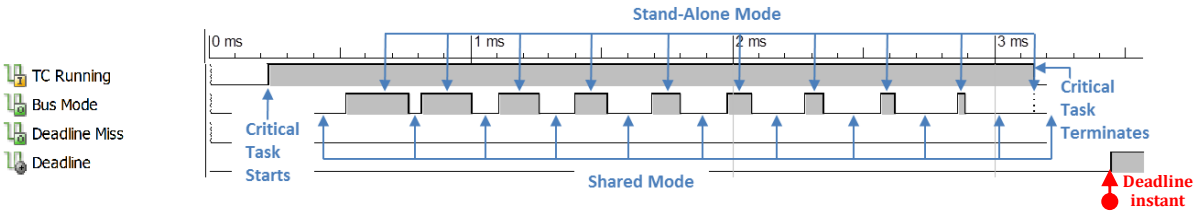


Figure 37 – Timeline of a system run in the ISim HDL simulator showing the following parameters: (a) critical task (TC) running; (b) Bus mode operation between "shared" and "stand-alone" modes; (c) HDE's output (Deadline Miss) and (d) Deadline instant.



Since the deadline was met for all applications, the timelines of the other applications were omitted to avoid redundancy.

Table 4 addresses the area overhead yielded due to the addition of the HDE to monitor the LEON3 processor. For each of the four applications (left-hand side column),

it was determined a given number of RPs along with the code to be monitored by the HDE (central column), which yielded the respective area overhead (right column). The HDE area overhead was evaluated by synthesizing a dual-core version of the LEON3 processor alongside with the HDE for a Xilinx Spartan 3E FPGA. The area overhead is given by the ratio of the number of primitives used by the HDE and the number of primitives used by the processor: $A_{over} = \#P_{HDE}/\#P_{processor} \times 100\%$. These numbers were collected from the Xilinx "PlanAhead" Statistics Report. During the validation, no deadline constraints were missed.

Table 4 – HDE area overhead for the evaluated applications.

Application	Reference Points	Overhead (%)
Bubblesort	10	4,32
Fibonacci	11	4,29
Dummy	21	5,14
Heap	12	4,95

A dual core LEON3 processor connected to the HDE was synthesized for the Xilinx Spartan3E FPGA (XC3S1200EFG320). Reference points were placed on loops and the area overhead of the HDE was analyzed. Table 5 depicts the area overhead of the design by varying the number of loops in which reference points were inserted. The number of reference points per loop was kept constant at value 10.

Table 5 – HDE area overhead by varying the number of loops.

			Overhead (%)			
Loops	RPs/Loop	Total RPs	RP Monitor	RP Time Ctrl.	Deadline Enforcer	Total
1	10	12	1.04	0.33	2.46	3.83
5	10	52	2.56	0.21	2.45	5.22
10	10	102	4.75	0.22	2.49	7.46
30	10	302	13	0.28	2.43	15.7
50	10	502	22.2	0.27	2.48	24.96
70	10	702	27.66	0.32	2.52	30.5
100	10	1002	38.38	0.32	2.48	41.18

Table 6 – Number of primitives by varying the number of loops.

		RP Monitor			RP Time Ctrl.			Deadline Enforcer			LEON3 Dual-Core		
# of Loops	RPs/Loop	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM
1	10	74	38	0	19	16	0	229	36	0	8311	2421	13
5	10	219	56	0	14	8	1	227	36	0	8303	2421	13
10	10	432	77	0	14	9	1	231	36	0	8282	2421	13
30	10	1229	159	0	18	11	1	223	36	0	8239	2421	13
50	10	2130	239	0	17	11	1	229	36	0	8231	2421	13
70	10	2631	320	0	20	12	2	233	36	0	8229	2421	13
100	10	3663	440	0	20	12	2	229	36	0	8250	2421	13

Table 5 shows that the area overhead of the Reference Point Time Controller is roughly constant, that is because the CTs are stored in a BlockRAM by the synthesis tool. The Spartan3E BlockRAM has a capacity of storing 512 32bit words, therefore it

is able of storing 512 CTs. For this reason, two BlockRAM strucutres were used to store 70 and 100 loops, which resulted in a slightly higher area overhead. Table 6 shows that for 1 loop the synthesizer decided not to use a BlockRAM for the Reference Point Time Controller, therefore the area overhead of that block was slightly larger than the others.

The area overhead of the Deadline Enforcer block was also constant, it happens because the number of reference points does not modifies the block logic at all. The only difference of it is the deadline constant.

Table 7 depicts the area overhead of the HDE by keeping the number of loops constant at 1 and varying the number of reference points per loop. The area overhead of the Reference Point Time Controller was significantly small until 6000 reference points per loop, then the synthesizer decided to not use the BlockRAMs causing a much larger area overhead in that block.

It can be noted from Table 8 that up to 6000 RPs per loop, the Reference Point Monitor block was the most significant block for the area overhead of the HDE. After that, when the synthesizer decided not to use BlockRAMs, the Reference Point Time Controller was the block with higher area overhead of the three blocks of the HDE.

Table 7 – HDE area overhead by varying the number of reference points per loop.

			Overhead (%)			
# of Loops	RPs/Loop	Total RPs	RP Monitor	RP Time Ctrl.	Deadline Enforcer	Total
1	10	12	1.04	0.33	2.46	3.83
1	100	102	1.38	0.22	2.48	4.08
1	1000	1002	1.77	0.32	2.47	4.55
1	4000	4002	4.41	0.37	2.47	7.25
1	6000	6002	5.35	6.4	2.46	14.21
1	10000	10002	5.85	7.18	2.45	15.48

Table 8 – Number of primitives by varying the number of reference points per loop.

		RP Monitor			RP Time Ctrl.			Deadline Enforcer			LEON3 Dual-Core		
# of Loops	RPs/Loop	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM
1	10	74	38	0	19	16	0	229	36	0	8311	2421	13
1	100	104	44	0	14	9	1	231	36	0	8311	2421	13
1	1000	140	50	0	20	12	2	229	36	0	8310	2421	13
1	2500	519	54	0	18	14	6	229	36	0	8309	2421	13
1	4000	420	54	0	20	14	6	229	36	0	8309	2421	13
1	6000	519	56	0	654	34	0	229	36	0	8313	2421	13
1	10000	570	58	0	735	36	0	227	36	0	8297	2421	13

In Table 9 the area overhead was analyzed by varying the number of reference points, without any loops. The reference points started at address “40000000h” and incremented by “4h”, that is, each reference point is an instruction apart. Table 10 depicts the number of elements of both the HDE and the LEON3 processor that results in the overhead shown in Table 9.

In Table 11, the reference points are inserted every 100 instructions. The area overhead increased, if compared to Table 9, because the synthesizer could not optimize as

Table 9 – HDE area overhead by varying the number of reference points (for loopless codes). RP added at every instruction in the code.

	Overhead (%)			
# of Reference Points	RP Monitor	RP Time Ctrl.	Deadline Enforcer	Total
10	0.93	0.34	2.43	3.7
100	1	0.22	2.47	3.69
1000	1.12	0.28	2.47	3.86
4000	1.97	0.33	2.47	4.77
8000	2	5.92	2.45	10.36
12000	3.51	6.81	2.45	12.78

Table 10 – Number of primitives by varying the number of reference points (for loopless codes). RP added at every instruction in the code.

	RP Monitor			RP Time Ctrl.			Deadline Enforcer			LEON3 Dual-Core		
# of Reference Points	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM
10	66	34	0	22	15	0	225	36	0	8308	2421	13
100	71	37	0	14	9	1	229	36	0	8308	2421	13
1000	80	40	0	16	12	2	229	36	0	8308	2421	13
4000	170	42	0	16	14	6	229	36	0	8309	2421	13
8000	172	43	0	602	34	0	227	36	0	8308	2421	13
12000	333	44	0	695	36	0	227	36	0	8290	2421	13

well as it did previously. Therefore, the area overhead is tightly dependent on the location where the RPs are inserted. Similarly the same effect can be expected to be observed in reference points insertion inside loops, the address in which they are inserted and at which iterations they are inserted. Table 12 depicts the number of elements of both the HDE and the LEON3 processor that results in the overhead shown in Table 11.

Table 11 – HDE area overhead by varying the number of reference points (for loopless codes). RP added at every bunch of 100 instructions.

	Overhead (%)			
# of Reference Points	RP Monitor	RP Time Ctrl.	Deadline Enforcer	Total
10	1.28	0.34	2.43	4.06
100	3.6	0.22	2.47	6.3
500	13.16	0.27	2.48	15.92
1000	30.55	0.28	2.49	33.32

Table 12 – Number of primitives by varying the number of reference points (for loopless codes). RP added at every bunch of 100 instructions.

	RP Monitor			RP Time Ctrl.			Deadline Enforcer			LEON3 Dual-Core		
# of Reference Points	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM
10	104	34	0	22	15	0	225	36	0	8311	2421	13
100	349	37	0	14	9	1	229	36	0	8276	2421	13
500	1365	39	0	17	11	1	229	36	0	8228	2421	13
1000	3213	40	0	16	12	2	229	36	0	8209	2421	13

4.2 Complex Real-Time Control Application

Model Predictive Control (MPC) is an advanced method of process control that relies in dynamic models of the system. It computes the optimal control signal by predict-

ing the system response in a finite time-horizon, therefore being able to anticipate future events and take control actions accordingly.

Another advantage of MPC is that the control signal can be constrained to a range of values, therefore it is able to account for input saturation. It can also constrain the output of the system, e.g., constraining the output of a system that can only be safely operated within a range of values.

Since the MPC response is highly dependable on the dynamic model of the system, the sampling period must be constant for the proper response of the method. Therefore the deadline of the task that computes the MPC is extremely important and must not be missed. The constrained MPC method is a CPU demanding method whose complexity increases as the system order and the time-horizon increases.

To validate the HDE, the contrained MPC method was used to control a DC motor, the state-space model of the motor is depicted below:

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx \end{cases} \tag{4}$$

where:

$$A = \begin{bmatrix} -\frac{b}{J} & \frac{k_T I_f}{J} \\ -\frac{k_\omega I_f}{L_a} & -\frac{R_a}{L_a} \end{bmatrix},\ B = \begin{bmatrix} 0 \\ \frac{1}{L_a} \end{bmatrix},\ C = \begin{bmatrix} 1 & 0 \end{bmatrix},\ x = [w, i_a]^T. \tag{5}$$

The system can be discretized using the Euler method, resulting in equation (6).

$$\begin{cases} x(k+1) = \underbrace{(I + AT_s)}_{A_d} x(k) + \underbrace{BT_s}_{B_d} u(k) \\ y(k) = \underbrace{C}_{C_d} x(k) \end{cases} \tag{6}$$

It can be observed in equation (6) that the system model depends in the sampling period (T_s). Whenever the sampling period changes, the system model changes, requiring the re-computation of the MPC parameters to account the change in the sampling period, which includes matrix inverses that is usually computed off-line due to its time complexity. Hence the sampling period is usually guaranteed to be constant (or with a very low jitter).

The motor parameters are depicted in Table 13, w is the angular speed of the motor shaft and i_a is the armature current.

Table 13 – DC motor parameters.

Parameter	Symbol	Value
Armature resistance	R_a	8 Ω
Armature inductance	L_a	170 mH
System inertia	J	10×10^{-3} N m s^2
Viscous friction coefficient	b	3×10^{-3} Nm/rad/s
Torque constant	k_t	0.521 N m/A
Speed constant	k_ω	0.521 V s/rad
Field current	I_f	0.5 A

The system was discretized using the Zero-Order Hold method with a sampling period (T_s) of 100ms, and then augmented with an integrator. The discretized and augmented state-space matrices are depicted in equation (7).

$$A_e = \begin{bmatrix} 0.9066 & 0.5091 & 0 \\ -0.0299 & -0.0071 & 0 \\ 0.9066 & 0.5091 & 1.0000 \end{bmatrix}, \; B_e = \begin{bmatrix} 0.2483 \\ 0.1178 \\ 0.2483 \end{bmatrix}, \; C_e = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}. \tag{7}$$

A time-horizon of 4 was implemented, and the MPC objective function is depicted in equation (8) (WANG, 2009).

$$J = \Delta U^T(\Phi^T\Phi + \overline{R})\Delta U - 2\Delta U^T\Phi^T(R_s - Fx(k_i)), \tag{8}$$

where:

$$\begin{aligned} F &= \begin{bmatrix} C_eA_e \\ C_eA_e{}^2 \\ C_eA_e{}^3 \\ C_eA_e{}^4 \end{bmatrix} = \begin{bmatrix} 0.9066 & 0.5091 & 1.0000 \\ 1.7133 & 0.9669 & 1.0000 \\ 2.4309 & 1.3743 & 1.0000 \\ 3.0693 & 1.7367 & 1.0000 \end{bmatrix} \\ \Phi &= \begin{bmatrix} C_eB_e & 0 & 0 & 0 \\ C_eA_eB_e & C_eB_e & 0 & 0 \\ C_eA_e{}^2B_e & C_eA_eB_e & C_eB_e & 0 \\ C_eA_e{}^3B_e & C_eA_e{}^2B_e & C_eA_eB_e & C_eB_e \end{bmatrix} = \begin{bmatrix} 0.2483 & 0 & 0 & 0 \\ 0.5333 & 0.2483 & 0 & 0 \\ 0.7875 & 0.5333 & 0.2483 & 0 \\ 1.0136 & 0.7875 & 0.5333 & 0.2483 \end{bmatrix} \end{aligned} \tag{9}$$

Where $R_s = [r(k_i), r(k_{i+1}), r(k_{i+2}), r(k_{i+3})]^T$ is a data vector containing the set-point information. For the validation of the HDE it was considered $r(k_i) = r(k_{i+1}) = r(k_{i+2}) = r(k_{i+3})$, that is the reference between a time-horizon was considered to be constant within a time-horizon and equal to $r(k_i)$. $\overline{R}$ is a diagonal matrix in the form of $\overline{R} = r_w I_{3\times3}$, where r_w is a tuning parameter for the desired closed-loop performance. It was used $r_w = 3$. By minimizing J, we find the optimal change in the control signal ΔU.

The control signal was constrained in the range $-3 \leq U \leq 3$, and Hildreth's quadratic programming procedure was used in order to minimize the objective function J. Hildreth's quadratic programming procedure, described in (WANG, 2009), can be used to solve constrained optimization problems. The objective function that is minimized by this procedure is:

$$J_H = \frac{1}{2}\eta^T H \eta + \eta^T f, \tag{10}$$

subject to constraints

$$A_{\text{cons}}\eta \leq b, \tag{11}$$

Which relates to (8) by,

$$H = \frac{1}{2}(\Phi^T\Phi + \overline{R}),\ f = -2\Phi^T(R_s + Fx(k_i)),\ \eta = \Delta U. \tag{12}$$

Furthermore, constraints on input saturation are guaranteed by the following,

$$A_{\text{cons}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ -1 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 \\ -1 & -1 & -1 & 0 \\ -1 & -1 & -1 & -1 \end{bmatrix},\ b = \begin{bmatrix} 3 - u(k_{i-1}) \\ 3 - u(k_{i-1}) \\ 3 - u(k_{i-1}) \\ 3 + u(k_{i-1}) \\ 3 + u(k_{i-1}) \\ 3 + u(k_{i-1}) \end{bmatrix}. \tag{13}$$

A fixed-point arithmetic was developed to be used with the implemented Hildreth's algorithm. The fixed point uses 15 bits to represent the integer part, and another 16 bits for the fractional part, the signal is represented by 1 bit.

Figure 38 depicts the Hildreth's Quadratic Programming Procedure pseudocode. The constant N_c is the number of constraints applied in the MPC. The variable λ is a vector where its dimension is determined by the number of constraints (N_c). The loop in line 5 is the iterative part of the algorithm that computes the λ until the variation in λ between the current and past iterations is lower than a chosen threshold, or when the max number of iterations defined by the constant "maxIterations" was reached.

The loop in line 7 computes each component of the λ vector individually, line 9 guarantees that the λ vector can only contain positive elements, otherwise it is set to zero. The condition in line 11 is responsible for interrupting the loop if the variation between the current and the previous λ is below a threshold.

Finally the future control signals (contained in the η variable) is computed in line 15. Whenever no constraints is violated, η is given by the expression $-H^{-1}f$, otherwise it is adjusted by the expression $-H^{-1}A_{cons}^T\lambda$ using the previous computed λ vector.

Figure 38 – Hildreth's quadratic programing procedure.

```
1: procedure HildrethQP(H, f, A_cons, b)
2:     P ← A_cons H^-1 A_cons^T
3:     d ← A_cons H^-1 f + b
4:     λ ← 0_{N_c×1}
5:     for all iter ∈ 1 : maxIterations do
6:         λ_p ← λ
7:         for all i ∈ 1 : N_c do
8:             w ← P(i,:)λ − P(i,i)λ(i,1) + d(i,1)
9:             λ(i,1) ← max(0, −w/P(i,i))
10:        end for
11:        if ||λ − λ_p||^2 < threshold then
12:            break
13:        end if
14:    end for
15:    η ← −H^-1 f − H^-1 A_cons^T λ
16:    return η
17: end procedure
```

Figure 39 depicts the simplified CFG of the critical task that computes the control signal using the MPC method. The quadratic programming optimization problem is solved by the task using the algorithm in Figure 38. The maximum iterations was set to 25. For every run of the task, the state x, the previous input signal ($u(k-1)$) and the reference are set to a pseudo random value computed by a linear-feedback shift register (LFSR). By generating these values randomly, different paths of the task is exercised during the validation.

Figure 40 shows the block diagram of the LEON3 processor used in the validation. Two tasks were used during the validation, the MPC critical task, and a bubblesort task, which is non critical. The MPC task runs in the critical core, while the bubblesort task runs in the secondary core. The Hard Dealine Enforcer collects the Executed Program Counter (EPC) signal from the critical core, and outputs the "Warning" signal to the "AHB Controller I". The "AHB Controller I" arbitrates the bus between the two cores, and when the "Warning" signal is '1', it only arbitrates to the critical core. The LFSR seed, the number of MPC task runs, the initial state value and a flag informing whether or not the secondary core should be started, are set via JTAG on the "Dual Port Register", which is read by the LEON3 critical core.

The "Results Sampler" block collects the maximum and average bus await time of the secondary core, i.e., how long it had to wait to acquire the bus. To do so, it collects the "hgrant" signal from the AMBA AHB, which informs which master has access to the bus at the current instant. To differentiate between runs, the "Results Sampler" block collects the "CTaskEnd" signal from the HDE, which is low whenever the critical task is

Figure 39 – Simplified MPC task CFG.

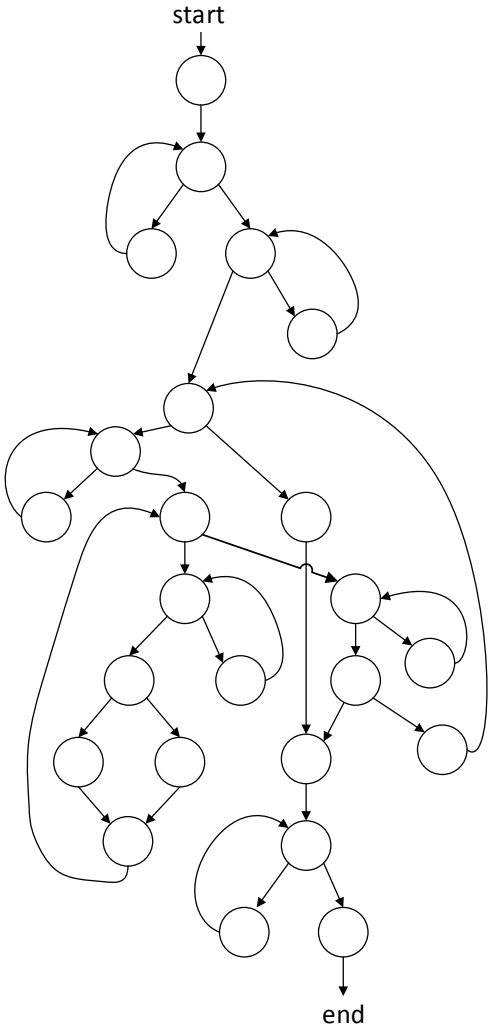
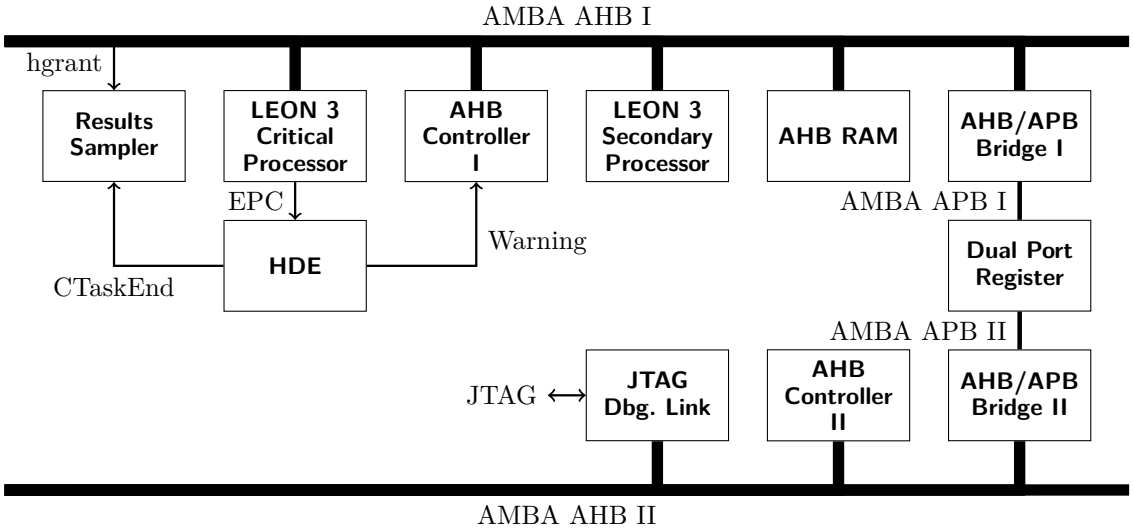


Figure 40 – System used for the validation and evaluation of the HDE comprised of a dual-core LEON3 processor.



running. The results are stored in a BlockRAM with capacity of 512 samples, that is read

back using the Xilinx Impact tool. The design is synthesized for the Spartan3E FPGA (xc3s500epq208).

The single core WCET was estimated to be 3449973 clock cycles using the GTT tool. The MPC task times was measured without the interference of the secondary core. The histogram of the times is depicted in Figure 41. The maximum measured execution time is 3316550 clock cycles, 3.87% smaller than the estimated WCET.

Figure 41 – MPC task measured execution times.

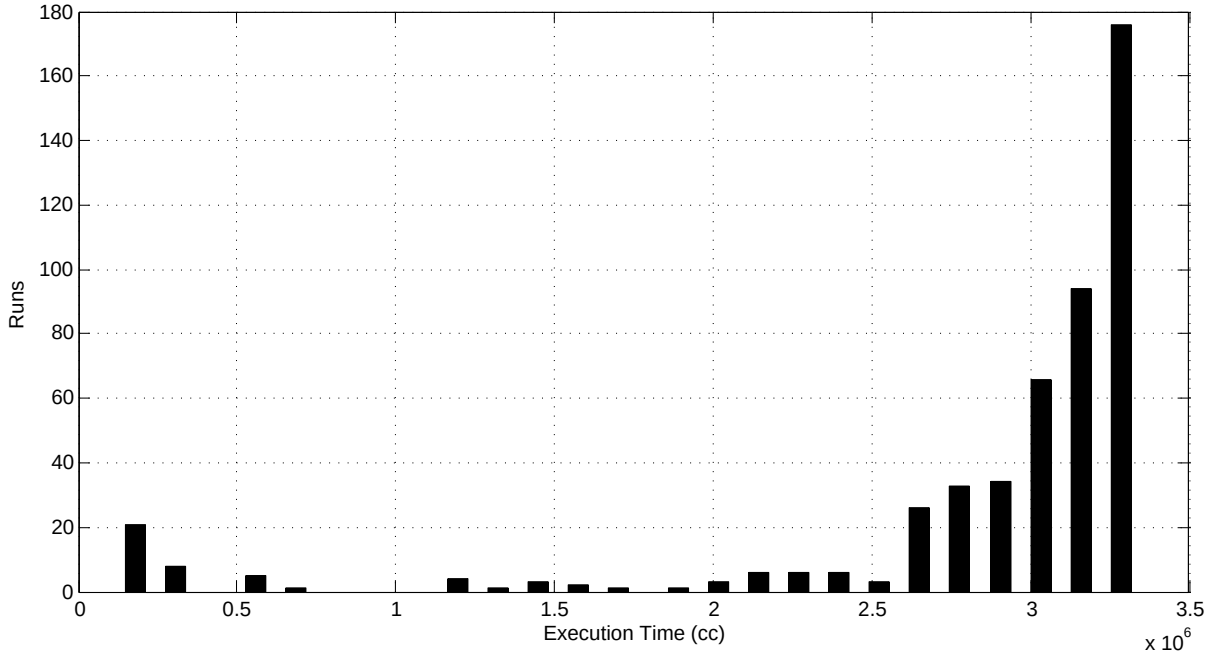


Figure 42 depicts the histogram of the maximum waiting time ratio of the non-critical task(bubblesort) to access the bus with an HDE composed by 15 reference points, and two deadlines was considered: 110% and 150% of the WCET of the MPC task. The ratio is computed by the maximum waiting time that the non-critical task have to wait to access the bus by the execution time of the MPC task in the given run. The longer the deadline, more time the HDE has to dedicate to the non-critical cores, therefore the ratio is smaller for the 150%. The average waiting time is also smaller for the deadline of 150% of the WCET of the MPC task, as is shown in Figure 43.

The bubblesort task is modified to sort an array of length 800, the effect in the maximum and average bus waiting times is negligible when compared with the bubblesort sorting an array of length 20, Figures 44 and 45.

As the number of reference points increases the maximum bus waiting time ratio of the non-critical task decreases, Figure 46. Increasing the number of reference points slightly decreases the average bus waiting time of the non-critical task, Figure 47.

Figure 42 – Maximum waiting times ratio of the non-critical task with an HDE of 15 RPs.

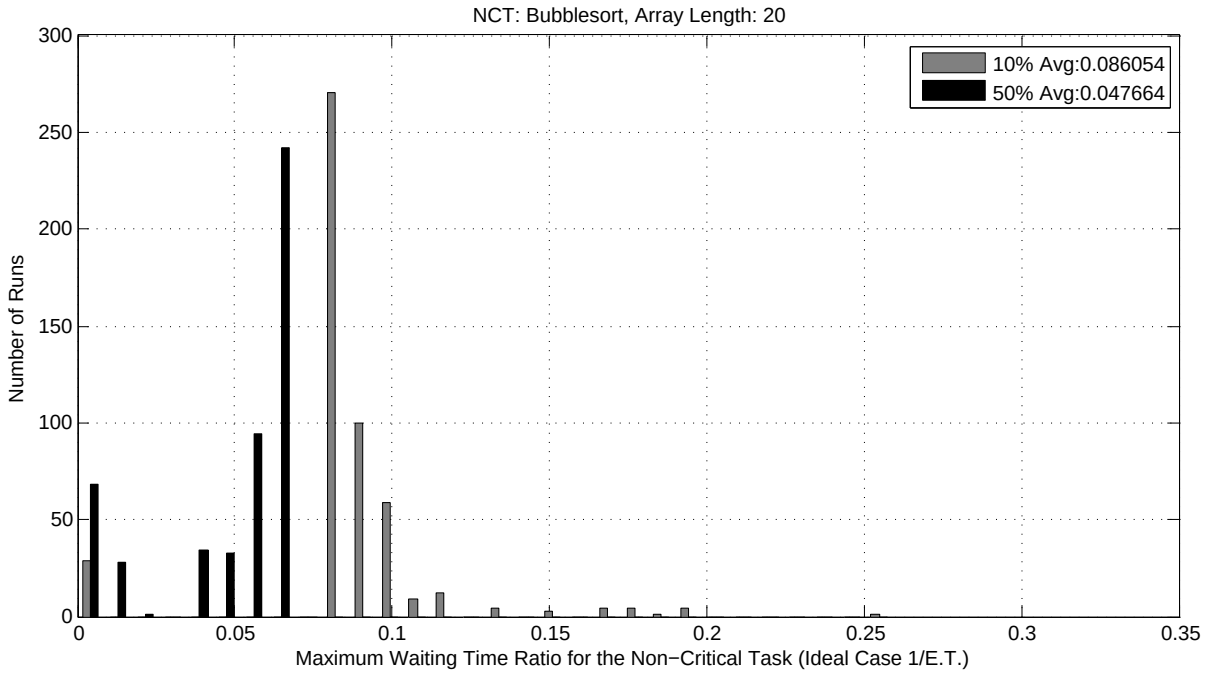
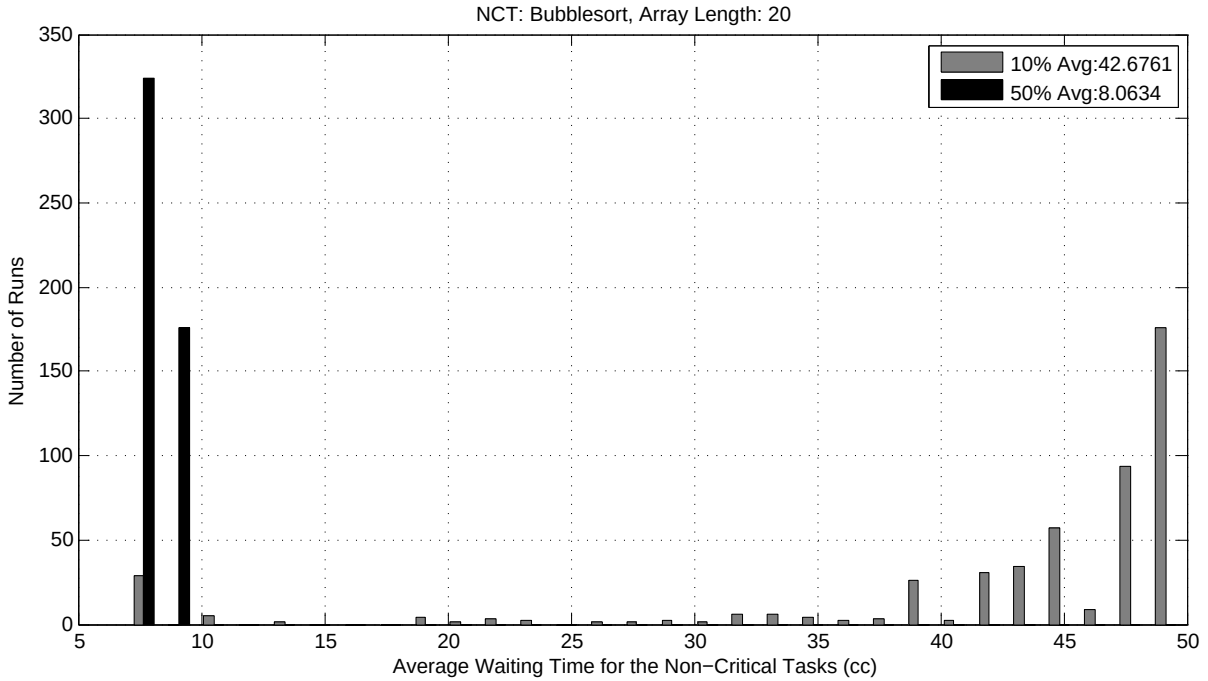


Figure 43 – Average waiting times of the non-critical task with an HDE of 15 RPs.



Figures 48 and 49 depict the measured execution time of the critical task with 15 and 28 reference points respectively, with a deadline of 110% and 150% of the estimated WCET.

Figure 50 depicts the measured execution time of the critical task without the HDE. The deadlines of 110% and 150% of the estimated WCET were missed, that is, without the HDE, the dual-core system cannot be safely used for this critical task. The

Figure 44 – Maximum waiting times ratio of the non-critical task with an HDE of 15 RPs, Bubblesort array length: 800.

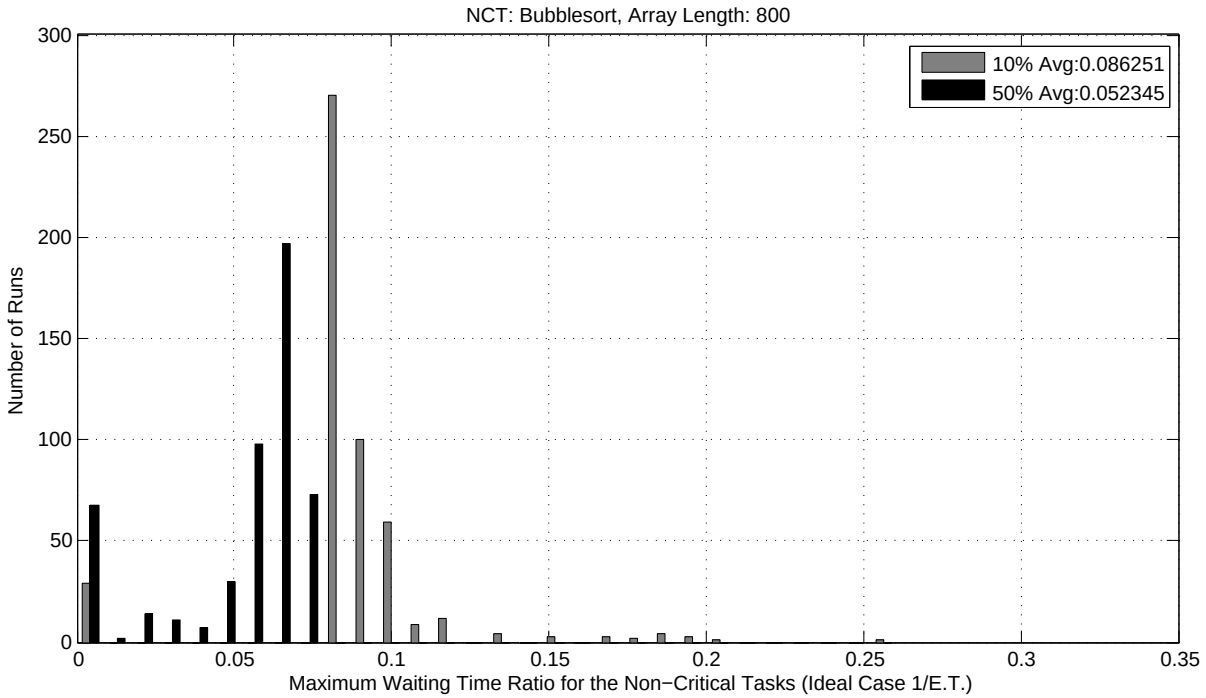
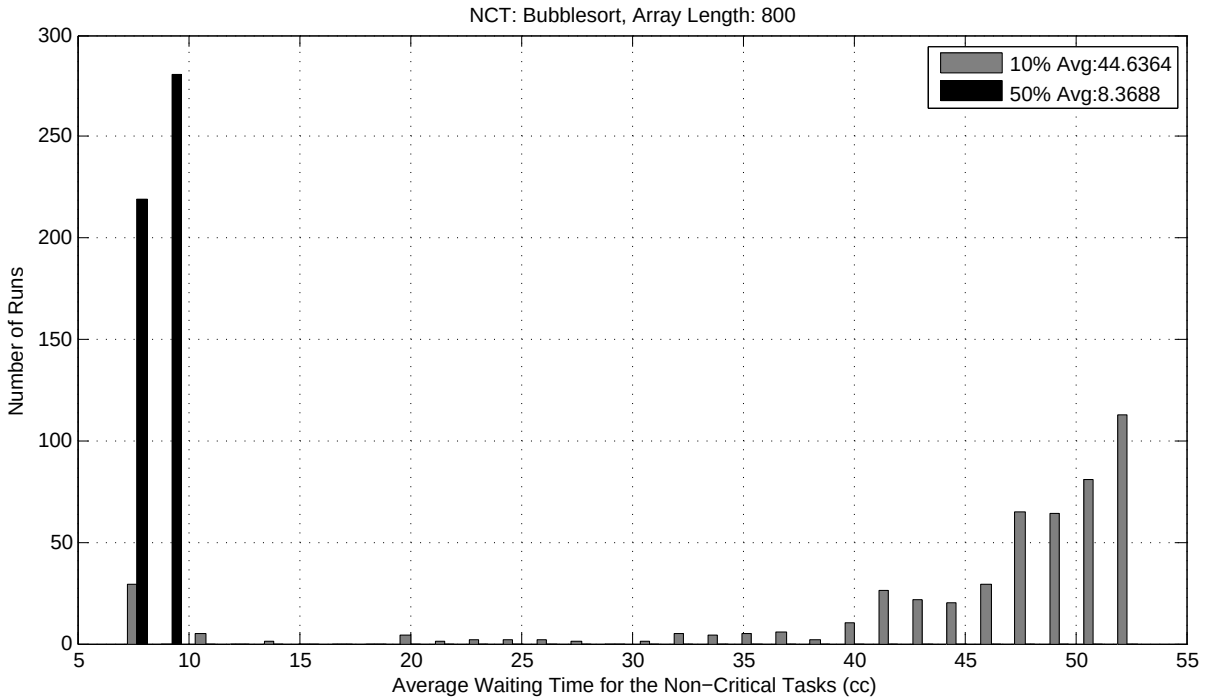


Figure 45 – Average waiting times of the non-critical task with an HDE of 15 RPs, Bubblesort array length: 800.



lines $D_{110\%}$ and $D_{150\%}$ denotes the deadlines given by 110% and 150% of the estimated WCET respectively. The deadlines of 110% and 150% were missed in 438 and 176 runs respectively.

Figure 46 – Maximum waiting times ratio of the non-critical task with an HDE of 28 RPs.

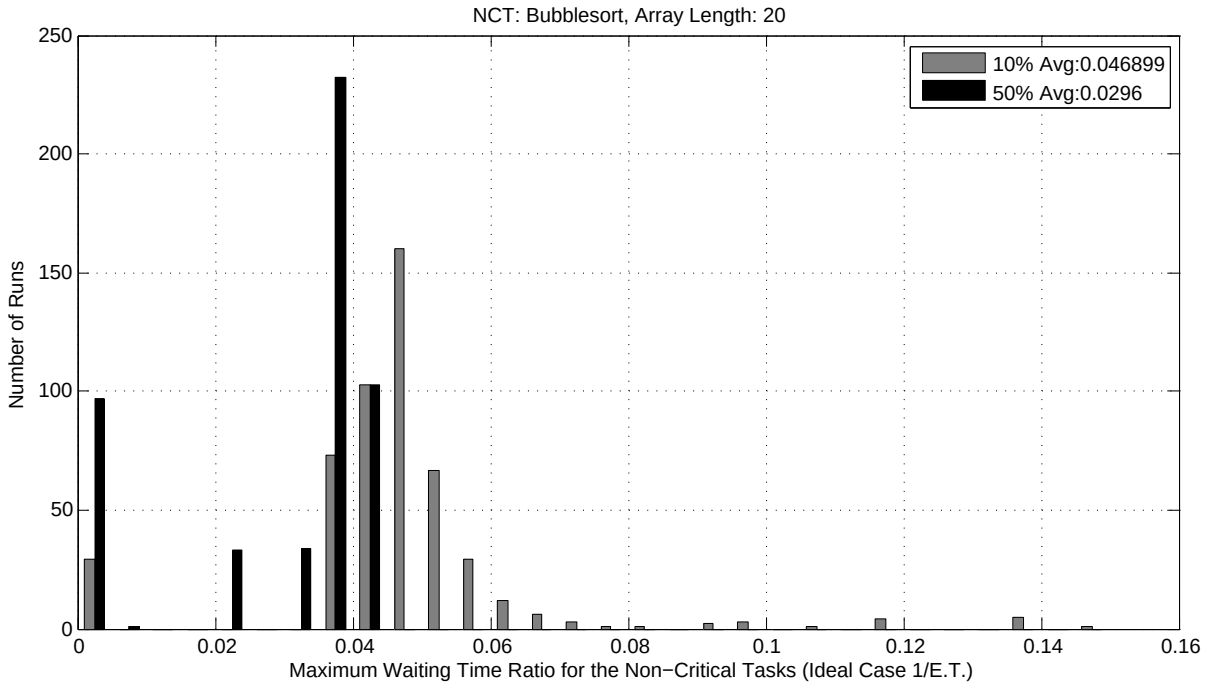


Figure 47 – Average waiting times of the non-critical task with an HDE of 28 RPs.

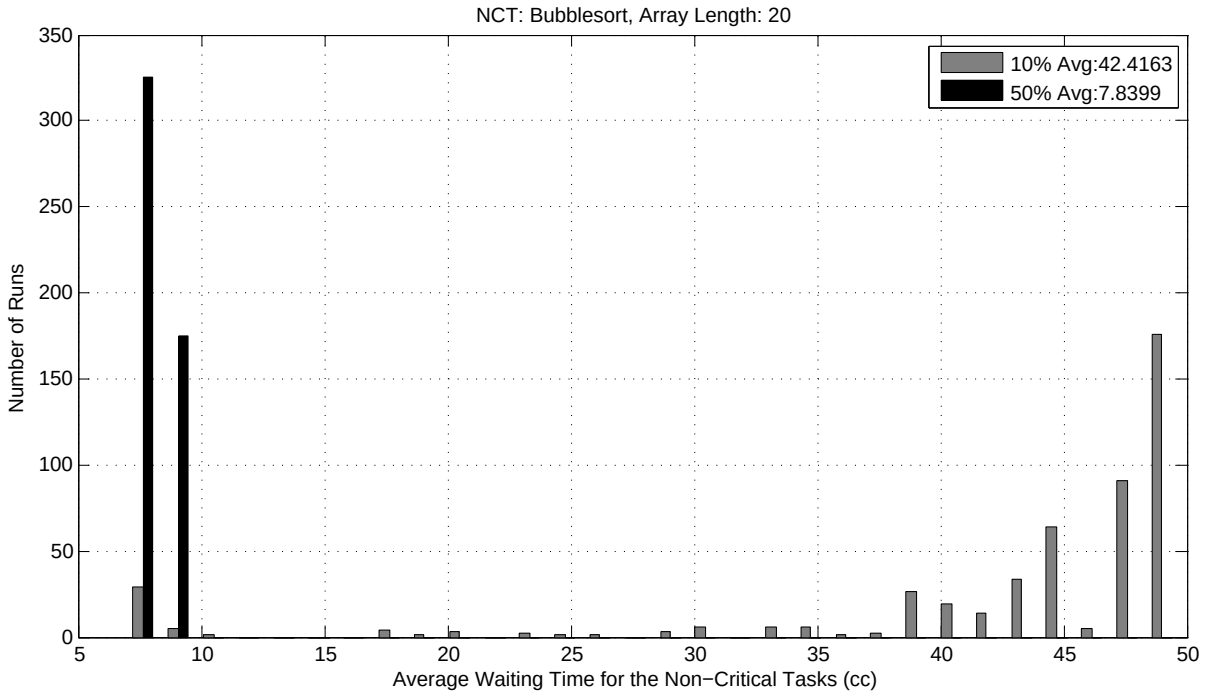


Table 14 depicts the area overhead of the HDE considering different number of reference points and deadlines. For 15 reference points the synthesizer did not used Block-RAMs for storing the times, therefore the area overhead was slightly higher than with 28 reference points.

Table 15 depicts the number of primitives used in both the HDE and the dual-core LEON3 processor.

Figure 48 – Measured execution time of the critical task with an HDE of 15 RPs.

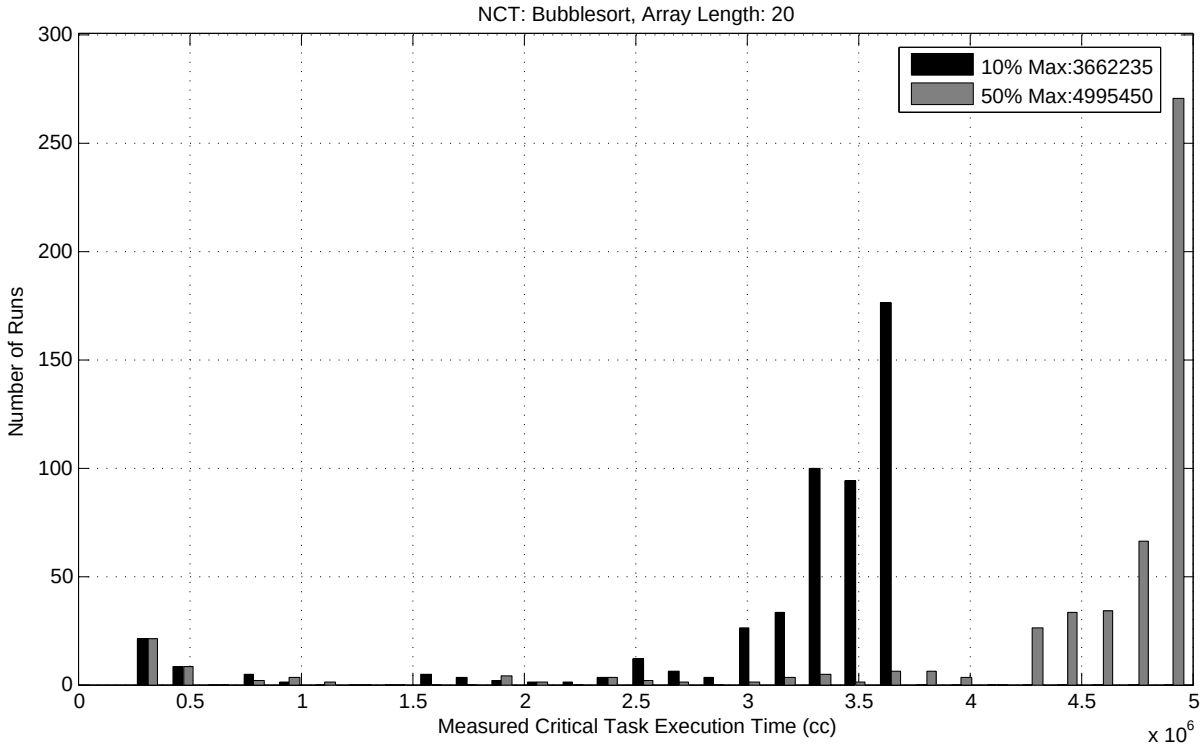


Figure 49 – Measured execution time of the critical task with an HDE of 28 RPs.

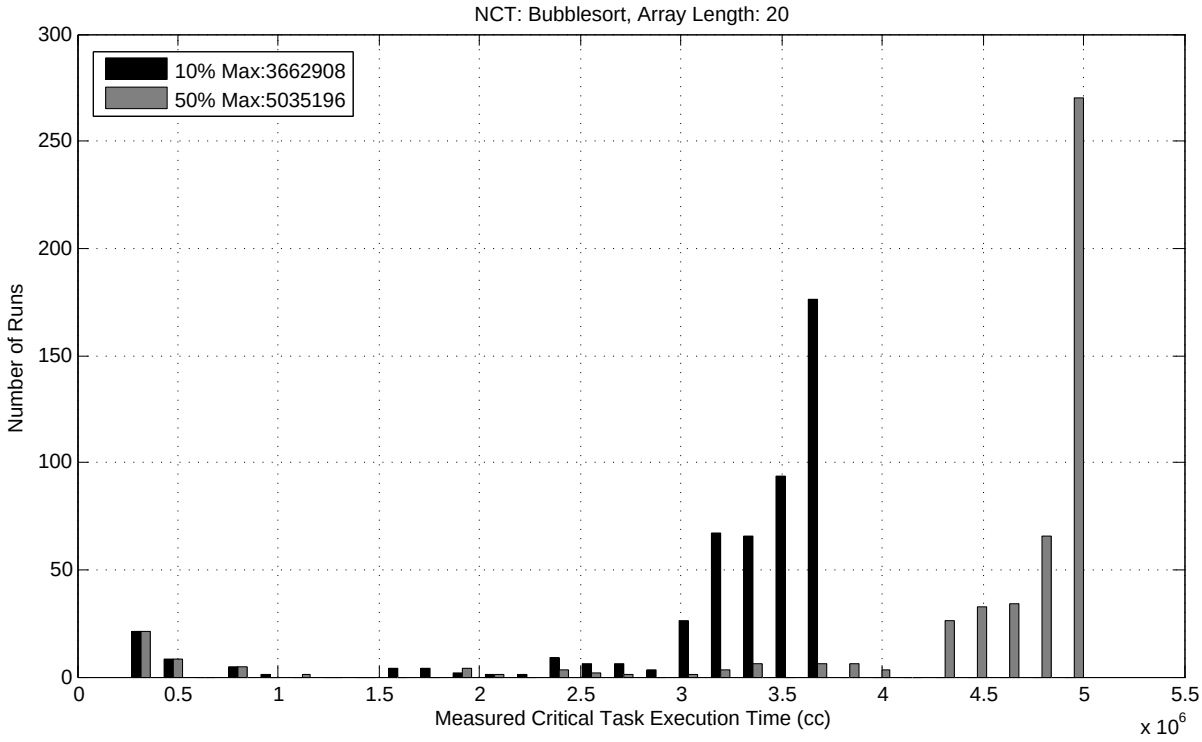


Table 16 depicts the comparison between TDMA and the HDE. The performance between these two techniques was not compared due to the high complexity of implementing tools to compute the TDMA bus schedule for the LEON3 processor.

Figure 50 – Measured execution time of the critical task without the HDE.

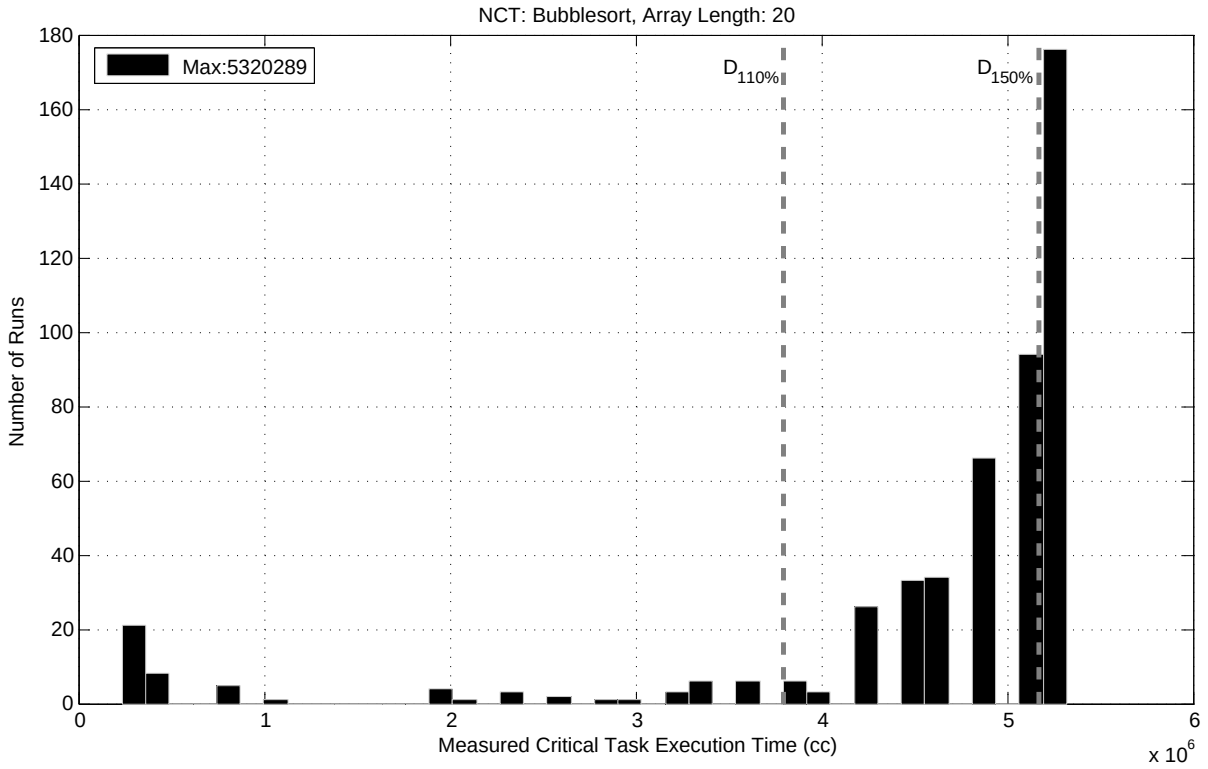


Table 14 – Area overhead of the HDE configured for the MPC task.

		Overhead (%)			
# of Reference Points	Deadline	RP Monitor	RP Time Ctrl.	Deadline Enforcer	Total
15	110%	1.26	0.53	2.47	4.25
15	150%	1.26	0.49	2.49	4.23
28	110%	1.36	0.17	2.47	3.99
28	150%	1.36	0.17	2.49	4.01

Table 15 – Number of primitives used in the design.

		RP Monitor			RP Time Ctrl.			Deadline Enforcer			LEON3 Dual-Core		
# of Reference Points	Deadline	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM
15	110%	102	39	0	51	22	0	229	36	0	8306	2421	13
15	150%	102	39	0	52	22	0	231	36	0	8306	2421	13
28	110%	106	40	0	10	7	1	229	36	0	8304	2421	13
28	150%	106	40	0	10	7	1	231	36	0	8304	2421	13

Table 16 – Comparison between TDMA and the HDE.

Metric	TDMA	HDE
# of critical tasks running concurrently	Many	One
WCET computation	Must consider all cores, tasks and the bus schedule	Same for a single-core processor, disregarding the number and types of the other cores and tasks
Other tasks changes	Computation of a new bus schedule, performing the entire WCET computation again	No change to the HDE

5 Conclusion

We presented a new approach that supports mixed-criticality workload execution in a multicore processor-based embedded system. Given that the proposed approach can be applied to any type of processor, it allows a large spectrum of real-time operating systems to be used as well. Thus, traditional and well-stablished real-time operating systems for critical applications such as VxWorks, LynxOS, Integrity or RTEMS and their advanced versions compliant with ARINC-653 (an avionics standard for safe, partitioned systems) could also be considered in the whole system design.

The approach allows any number of cores to run less-critical tasks concurrently with the critical core, which is running the critical task. The approach is based on the use of a dedicated hardware-based Hard Deadline Enforcer (HDE), which allows the execution of any number of cores (running less-critical workloads) concurrently with the critical core (executing the critical workload). This approach allows the exploitation of the maximum performance offered by a multiprocessing system while guaranteeing critical task schedulability.

A case-study based on a dual-core version of the LEON3 processor was implemented to demonstrate the applicability and assertiveness of the approach. Four critical application codes were compiled to this processor, which was mapped into a Xilinx Spartan 3E FPGA. Experimental results demonstrated that the proposed approach is very effective on combining system highest possible performance with critical task schedulability within timing deadline. Furthermore, area overhead is considerably small: in the order of 4.32% for the dual-core version of the LEON3 processor.

Additionally, the HDE was verified against a complex real-time control application running on the above mentioned LEON3 dual-core version. The maximum and average wait time for the secondary processor to access the bus was evaluated for 15 and 28 reference points, demonstrating that the performance of the HDE increases as the number of reference points increases. The same effect occurs when the deadline is increased: that happens because the HDE has more available time to distribute among the non-critical tasks.

In order to configure the HDE, a measurement based tool was developed to compute the $WCET_R$ of tasks running in a single-core LEON3 processor. Instruction execution times are collected using the LEON3 DSU module. Alternatively, these times can be collected using VHDL simulation of the LEON3 processor. The GTT tool was developed to combine the sampled times into an WCET and $WCET_R$ estimate. Additionally, the well-stablished IPET method was modified to compute the $WCET_R$. GTT is more

user-friendly than IPET, although it has a much lower performance than IPET. Once the $WCET_R$ times are computed, the HDE VHDL is automatically generated by the tool with the indicated reference points.

It is important to mention that the complexity of the WCET computation is the same as for single-core processors, no matter is the number of tasks is running concurrently in the different cores of the processor. In this case, the tasks are computed independently one from each other. This reduces embedded system design complexity and renders the proposed approach a very attractive solution.

During the validation, all execution times were smaller than the deadlines of the tasks. Furthermore, no WCET underestimation was observed for the evaluated tasks.

6 Future Work

We have listed below possible future topics that could be addressed in order to extend the benefits and advantages of the proposed work, they are:

- Development and validation of the HDE for other processors. In particular for the PowerPC 750.
- Inclusion of a real-time operating system to run with the critical application.
- Development of the HDE for multiple critical cores running concurrently.
- The Development or improvement of the current $WCET_R$ tool to support the LEON3 instruction and/or data caches.
- Improve existing commercial WCET tools to compute the $WCET_R$.

Bibliography

ALT, M. et al. Cache behavior prediction by abstract interpretation. In: **Static Analysis**. [S.l.]: Springer, 1996. p. 52–66. Referenced on page 27.

ARINC Specification 653. **Part 1, Avionics Application Software Standard Interface, Required Services (March 7, 2006)**. 2006. ARINC, 2551 Riva Road, Annapolis, MD 21401. Available at: <https://www.arinc.com/cf/store/index.cfm>. Referenced on page 22.

ARINC Specification 653. **Part 3, Avionics Application Software Standard Interface, Conformity Test Specification (October 16, 2006)**. 2006. ARINC, 2551 Riva Road, Annapolis, MD 21401. Available at: <https://www.arinc.com/cf/store/index.cfm>. Referenced on page 22.

ARINC Specification 653. **Part 2, Avionics Application Software Standard Interface, Extended Services (January 22, 2007)**. 2007. ARINC, 2551 Riva Road, Annapolis, MD 21401. Available at: <https://www.arinc.com/cf/store/index.cfm>. Referenced on page 22.

ARM Ltd. **AMBA™ Specification (Rev 2)**. [S.l.], 1999. Available at: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0011a/index.html>. Last Access: December, 2015. Referenced 2 times on pages 38 and 39.

BALLABRIGA, C.; CASSÉ, H. Improving the wcet computation time by ipet using control flow graph partitioning. In: **WCET**. [S.l.: s.n.], 2008. Referenced on page 29.

BALLABRIGA, C. et al. Otawa: An open toolbox for adaptive wcet analysis. In: **Software Technologies for Embedded and Ubiquitous Systems**. [S.l.]: Springer, 2010. p. 35–46. Referenced on page 34.

BENGTSSON, J. et al. **UPPAAL—a tool suite for automatic verification of real-time systems**. [S.l.]: Springer, 1996. Referenced on page 33.

BERKELAAR, M. **"lp_solve, version 5.5"**. 2010. Referenced 2 times on pages 28 and 52.

BERNAT, G.; BURNS, A.; NEWBY, M. Probabilistic timing analysis: An approach using copulas. **Journal of Embedded Computing**, IOS Press, v. 1, n. 2, p. 179–194, 2005. Referenced on page 64.

BERNAT, G.; COLIN, A.; PETTERS, S. pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems. **REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS**, UNIVERSITY OF YORK, 2003. Referenced 2 times on pages 30 and 64.

BERNAT, G.; COLIN, A.; PETTERS, S. M. Wcet analysis of probabilistic hard real-time systems. In: IEEE. **Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE**. [S.l.], 2002. p. 279–288. Referenced 2 times on pages 30 and 64.

BURNS, A.; EDGAR, S. Predicting computation time for advanced processor architectures. In: **Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on**. [S.l.: s.n.], 2000. p. 89–96. ISSN 1068-3070. Referenced 2 times on pages 30 and 64.

BUTTAZZO, G. C. **Hard real-time computing systems: predictable scheduling algorithms and applications**. [S.l.]: Springer, 2011. v. 24. Referenced on page 25.

CHATTOPADHYAY, S. et al. A unified wcet analysis framework for multicore platforms. **ACM Transactions on Embedded Computing Systems (TECS)**, ACM, v. 13, n. 4s, p. 124, 2014. Referenced 2 times on pages 32 and 33.

COBHAM GAISLER AB. **GRLIB IP Core User's Manual**. [S.l.], 2015. Available at: <http://gaisler.com/products/grlib/grip.pdf>. Last Access: December, 2015. Referenced on page 35.

CORMEN, T. H. et al. **Introduction to Algorithms, Third Edition**. 3rd. ed. [S.l.]: The MIT Press, 2009. 603 p. ISBN 0262033844, 9780262033848. Referenced on page 56.

COUSOT, P.; COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM. **Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages**. [S.l.], 1977. p. 238–252. Referenced on page 27.

FERDINAND, C.; HECKMANN, R. ait: Worst-case execution time prediction by static program analysis. In: **Building the Information Society**. [S.l.]: Springer, 2004. p. 377–383. Referenced on page 30.

FERDINAND, C.; WILHELM, R. Efficient and precise cache behavior prediction for real-timesystems. **Real-Time Syst.**, Kluwer Academic Publishers, Norwell, MA, USA, v. 17, n. 2-3, p. 131–181, dez. 1999. ISSN 0922-6443. Available at: <http://dx.doi.org/10.1023/A:1008186323068>. Referenced on page 27.

GARFINKEL, R. S.; NEMHAUSER, G. L. **Integer programming**. [S.l.]: Wiley New York, 1972. v. 4. Referenced on page 27.

GUSTAFSSON, J. et al. The mälardalen WCET benchmarks: Past, present and future. In: **10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium**. [s.n.], 2010. p. 136–146. Available at: <http://dx.doi.org/10.4230/OASIcs.WCET.2010.136>. Referenced on page 71.

HOLSTI, N.; SAARINEN, S. Status of the bound-t wcet tool. **Space Systems Finland Ltd**, 2002. Referenced on page 31.

LI, Y.-T. S.; MALIK, S. Performance analysis of embedded software using implicit path enumeration. In: **Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference**. New York, NY, USA: ACM, 1995. (DAC '95), p. 456–461. ISBN 0-89791-725-1. Available at: <http://doi.acm.org/10.1145/217474.217570>. Referenced on page 27.

LV, M. et al. Combining abstract interpretation with model checking for timing analysis of multicore software. In: IEEE. **Real-Time Systems Symposium (RTSS), 2010 IEEE 31st**. [S.l.], 2010. p. 339–349. Referenced 2 times on pages 32 and 33.

MICHIE, D. “Memo” Functions and Machine Learning. **Nature**, v. 218, p. 19–22, 1968. Referenced on page 58.

OTTOSSON, G.; SJODIN, M. Worst-case execution time analysis for modern hardware architectures. In: CITESEER. **In Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97**. [S.l.], 1997. Referenced on page 29.

PUSCHNER, P.; BURNS, A. A review of worst-case execution-time analysis (editorial). **Real-Time Systems**, v. 18, n. 2/3, p. 115–128, 2000. Referenced on page 32.

RAPITA SYSTEMS LTD. **How does RapiTime work?** 2015. <http://www.rapitasystems.com/products/rapitime/how-does-rapitime-work>. Last Access: December, 2015. Referenced 2 times on pages 31 and 32.

RAPITA SYSTEMS LTD. **RapiTime**. 2015. <https://www.rapitasystems.com/products/rapitime>. Last Access: December, 2015. Referenced on page 34.

ROSÉN, J. et al. Predictable worst-case execution time analysis for multiprocessor systems-on-chip. In: IEEE. **Electronic Design, Test and Application (DELTA), 2011 Sixth IEEE International Symposium on**. [S.l.], 2011. p. 99–104. Referenced 2 times on pages 32 and 33.

SPARC International INC., C. **The SPARC Architecture Manual: Version 8**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. ISBN 0-13-825001-4. Referenced on page 35.

STAPPERT, F.; ERMEDAHL, A.; ENGBLOM, J. Efficient longest executable path search for programs with complex flows and pipeline effects. In: ACM. **Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems**. [S.l.], 2001. p. 132–140. Referenced on page 29.

UNGERER, T. et al. parmerasa–multi-core execution of parallelised hard real-time applications supporting analysability. In: IEEE. **Digital System Design (DSD), 2013 Euromicro Conference on**. [S.l.], 2013. p. 363–370. Referenced on page 34.

UNGERER, T. et al. Merasa: Multicore execution of hard real-time applications supporting analyzability. **IEEE Micro**, IEEE, n. 5, p. 66–75, 2010. Referenced 2 times on pages 32 and 33.

VARGAS, F.; GREEN, B. **Hardware Dedicado para Análise e Controle Temporal de Aplicações Críticas em Sistemas Embarcados Baseados em Processadores Multicore**. 2015. Patente: Privilégio de Inovação. Número do registro: BR1020150191863, data de depósito: 11/08/2015, Instituição de registro: INPI - Instituto Nacional da Propriedade Industrial. Instituição financiadora: Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS. Referenced on page 42.

VARGAS, F.; GREEN, B. Preliminaries on a hardware-based approach to support mixed-critical workload execution in multicore processors. In: **Second International Conference On Advances In Computing, Control And Networking - ACCN 2015**. [S.l.: s.n.], 2015. p. 23 – 27. Referenced on page 42.

VARGAS, F.; GREEN, B. Preliminaries on a hardware-based approach to support mixed-critical workload execution in multicore processors. In: **International Journal of Advancements in Electronics and Electrical Engineering**. [S.l.: s.n.], 2015. p. 142 – 146. Referenced on page 42.

WANG, L. **Model Predictive Control System Design and Implementation Using MATLAB**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2009. ISBN 1848823304, 9781848823303. Referenced 2 times on pages 77 and 78.

WILHELM, R. et al. The worst-case execution-time problem—overview of methods and survey of tools. **ACM Trans. Embed. Comput. Syst.**, ACM, New York, NY, USA, v. 7, n. 3, p. 36:1–36:53, maio 2008. ISSN 1539-9087. Available at: <http://doi.acm.org/10.1145/1347375.1347389>. Referenced 7 times on pages 25, 26, 27, 28, 29, 30, and 31.

APPENDIX A – Graph Search Algorithm

This recursive algorithm is responsible for finding the edges and vertices between a start and an end vertices. It returns true if there is at least one path between the start and the end vertices. The parameters "vertices" and "edges" contains, when the algorithm terminates, all the vertices and edges in the path between the start and end vertices. The parameter "visitedEdges" is a set that keeps track of the already visited edges and is initially empty.

```
1: procedure GraphSearch(start, current, target, visitedEdges, vertices, edges)
2:     if current = target then
3:         return true
4:     end if
5:     found ← false
6:     for all edge ∈ current.OutEdges do
7:         if ¬(edge ∈ visitedEdges) then
8:             visitedEdgesCopy ← visitedEdges.Clone()
9:             visitedEdgesCopy.Add(edge)
10:            if edge.head ≠ start then
11:                result ← GraphSearch(start, edge.head, target, visitedEdgesCopy, vertices, edges)
12:                if result = true then
13:                    vertices.Add(edge.tail)
14:                    vertices.Add(edge.head)
15:                    edges.Add(edge)
16:                    found ← true
17:                end if
18:            end if
19:        end if
20:    end for
21:    return found
22: end procedure
```

APPENDIX B – Modification in the LEON3 to Generate the Executed Program Counter (EPC)

```
------------------------------------------------------------
--  This file is a part of the GRLIB VHDL IP LIBRARY
--  Copyright (C) 2003 - 2008, Gaisler Research
--  Copyright (C) 2008 - 2014, Aeroflex Gaisler
--
--  This program is free software; you can redistribute it and/or
    modify
--  it under the terms of the GNU General Public License as
   published by
--  the Free Software Foundation; either version 2 of the License
   , or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be
   useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty
   of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public
   License
--  along with this program; if not, write to the Free Software
--  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
   02111-1307  USA
------------------------------------------------------------
-- Entity:      iu3mod
-- File:        iu3mod.vhd
-- Author:      Jiri Gaisler, Edvin Catovic, Gaisler Research
-- Modified:    Bruno Green, PUCRS
-- Description: LEON3 7-stage integer pipeline
------------------------------------------------------------
...
```

```
entity iu3mod is
  generic (
...
  );
  port (
    clk   : in  std_ulogic;
    rstn  : in  std_ulogic;
...
    sclk   : in  std_ulogic;
    epc : out std_logic_vector(31 downto 2)
    );
...
end;

architecture rtl of iu3mod is
...
  epcgp : process(rstn, r.x.ctrl.annul, r.x.ctrl.pc)
    variable valid : boolean;
    begin
      valid := (((not r.x.ctrl.annul)) = '1');
      if (rstn = '1') then
        if valid then
          epc <= r.x.ctrl.pc(31 downto 2);
        end if;
      end if;
  end process;
...
end;
```

APPENDIX C – AMBA Bus Controller: support for stand-alone bus mode

The following vhdl snippet is a modification in the AMBA AHB Controller to allow the stand-alone bus mode, when the signal "force" is '1', the bus enters the stand-alone mode, and the bus arbitrations are made only for the master 0. Therefore the critical core must be configured as master 0. It is worth noting that this VHDL snippet code is valid for any number of N cores, where N is between 2 and 16. The upper limit is due to the AMBA limitation in the number of masters in the bus.

```
--  This file is a part of the GRLIB VHDL IP LIBRARY
--  Copyright (C) 2003 - 2008, Gaisler Research
--  Copyright (C) 2008 - 2014, Aeroflex Gaisler
--
--  This program is free software; you can redistribute it and/or
    modify
--  it under the terms of the GNU General Public License as
   published by
--  the Free Software Foundation; either version 2 of the License
   , or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be
   useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty
   of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public
   License
--  along with this program; if not, write to the Free Software
--  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
   02111-1307  USA
-----------------------------
-- Entity:      ahbctrlmod
-- File:        ahbctrlmod.vhd
-- Author:      Jiri Gaisler, Gaisler Research
-- Modified:    Edvin Catovic, Gaisler Research
```

```
-- Modified:    Bruno Green, PUCRS
-- Description: AMBA arbiter, decoder and multiplexer with plug&
   play support
...
entity ahbctrlmod is
  generic (
    ...
  );
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    ...
    testsig : in  std_logic_vector(1+GRLIB_CONFIG_ARRAY(
       grlib_techmap_testin_extra) downto 0) := (others => '0');
    force : in std_logic--Force bus into standalone mode when '1'
  );
end;

architecture rtl of ahbctrlmod is
...
begin
  comb : process(rst, msto, slvo, r, rsplit, testen, testrst,
     scanen, testoen, testsig, force)
...
  begin
...
    if (split /= 0) then
      for i in 0 to nahbmx-1 loop
        tmpv(i) := (msto(i).htrans(1) or (msto(i).hbusreq)) and
           not rsplit(i) and not r.ldefmst;
      end loop;
      if (r.defmst and orv(tmpv))  = '1' then arb := '1'; end if;
    end if;
    -- ~line 423
    --rearbitrate bus with selmast. If not arbitrated one must
    --ensure that the dummy master is selected for locked splits.
    if force = '1' then --stand-alone mode
      nhmaster := 0; --force arbitration to master 0
    else --shared mode
      if (arb = '1') then
        selmast(r, msto, rsplit, nhmaster, defmst); -- select the
            next master to adquire the bus
```

```
      elsif (split /= 0) then
        defmst := r.defmst; -- no masters wants the bus,
           arbitrate to the default master
      end if;
    end if;
...
end;
```

APPENDIX D – Step by Step of a Design Process

Figure 51 depicts the flowchart of the steps involved in a design process from compilation of the application to the generation of the HDE.

Figure 51 – Flowchart of a design process.

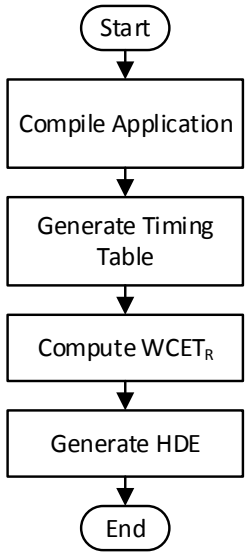


Figure 52 depicts the flowchart of the steps involved in the compilation of the application, having as input the source code (or source codes) and as output the application binary.

Figure 52 – Flowchart of the steps involved in the compilation of the application.

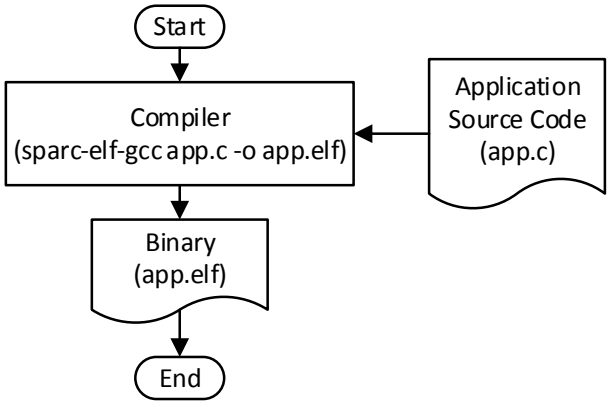
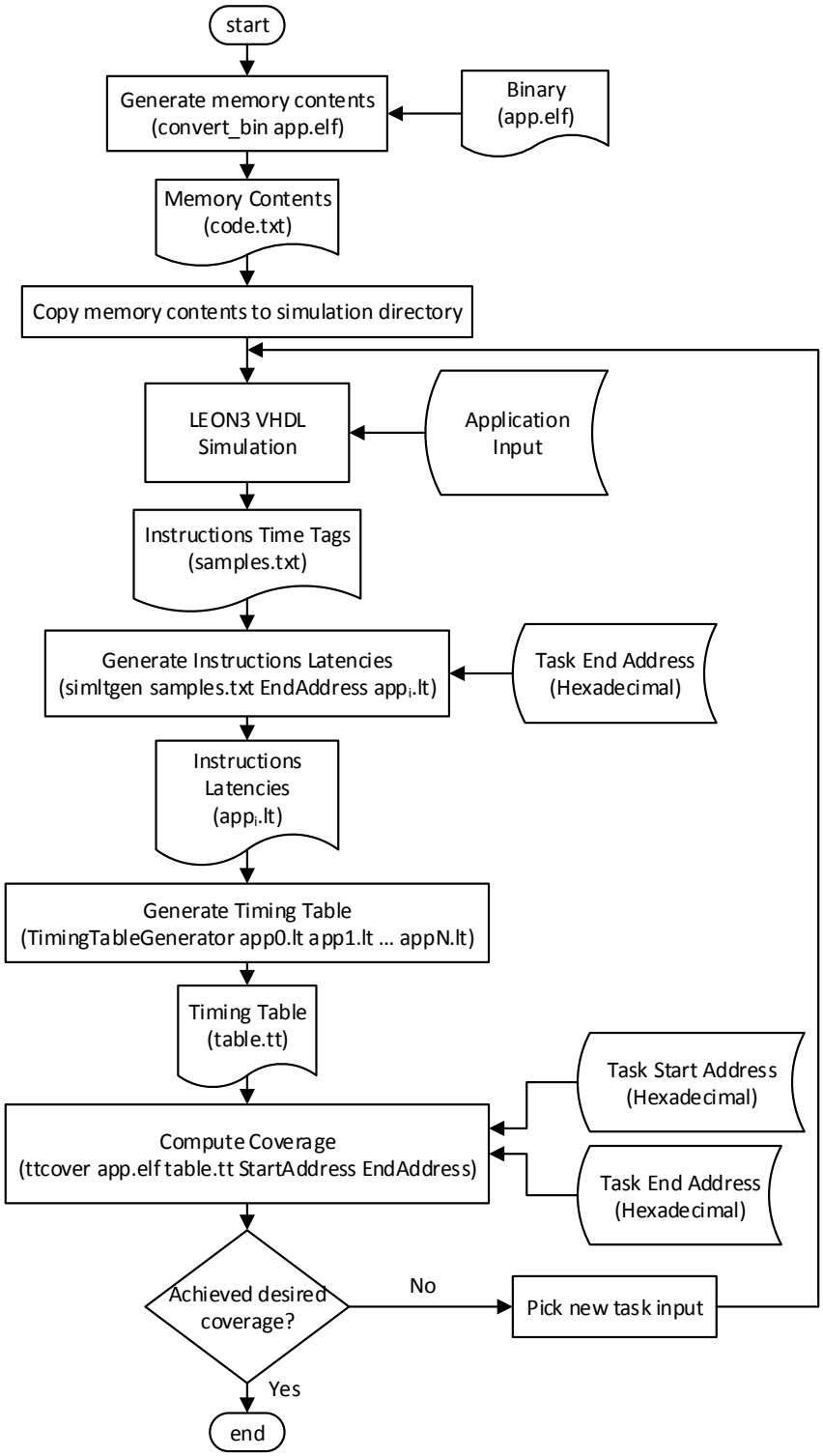


Figure 53 depicts the flowchart of the steps involved in the generation of the timing table. First the application binary is converted to a format that can be read by the VHDL simulator. The user sets the application input and runs the VHDL simulation

of the LEON3 processor. Once the simulation is completed, the instructions time tags are available in the "samples.txt" file. The latencies table is generated from it and from the critical task end address.

Figure 53 – Flowchart of the steps involved in the generation of the Timing Table.



The timing table is generated from the instructions latency table (or tables if several simulations were performed), the coverage[1] is then computed from the timing table and the start and end addresses of the critical task. New simulations with different inputs must be performed as long as the desired coverage is not achieved.

Figure 54 depicts the flowchart of the steps to compute WCET and $WCET_R$ and generate the HDE VHDL files by using GTT. First the user selects the application binary, the timing table, and the task start and end addressses. Then the user defines the bound for each of the loops in the task, and the WCET and $WCET_R$ are computed for every vertex and every possible combination of loop iterations.

The user must then select which vertices should contain reference points, and the reference points are selected by dividing the WCET into a number of segments chosen arbitrarily by the user. A $WCET_R$ of a vertex is selected to be a reference point if it is in a distance shorter than a range value defined (also arbitrarily) by the designer.

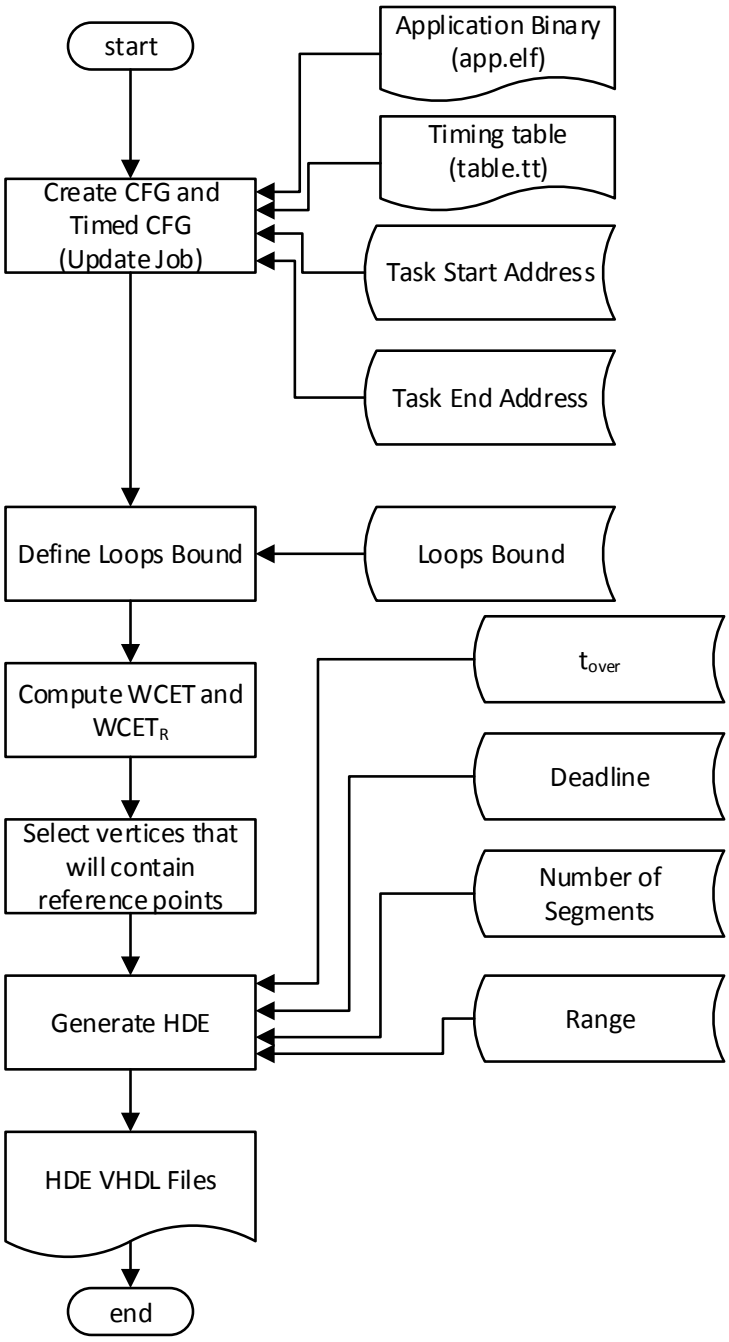
Figure 55 depicts how the process of selecting a $WCET_R$ to be a reference point. To do so, consider that the WCET is split into two equal segments and assume a range of 25% (as alternative, consider also a second range of 50%). This range is computed as a percentage of the segment length. Then, upon analyzing Figure 55, one would select both $WCET_R$s ($WCET_{R1}$ and $WCET_{R2}$) if the considered range is 50%; otherwise (assuming range equal to 25%) it would be selected only $WCET_{R2}$. It should be noted that (for the same vertex) if more than one $WCET_R$ is identified under the umbrella of a given range, only the closest one is selected (in the mentioned example of Figure 55, $WCET_{R2}$). Finally, for a vertex belonging to a loop, for instance, it would be expected to have more than one $WCET_R$ associated to a given vertex.

Once the reference points are determined and the parameters of deadline (a percentage of the estimated WCET) and the turnover t_{over} are informed, the HDE VHDL files are generated.

Figure 56 depicts the flowchart of the steps to compute WCET and $WCET_R$ and generate the HDE VHDL files by using IPET. First the user selects the application binary, the timing table, the task start and end addressses, and then enters the IPET mode in the tool. Then the user defines the constraint for each of the loops in the task for the WCET computation. The user then adds as many reference points as needed for the desired performance of the HDE selecting the vertex and loops constraint of every reference point. The user informs the deadline (a percentage of the estimated WCET)

[1] By "coverage", we are referring to the metrics used to measure the ratio of the entries of the timing table to the required entries of the task to compute the WCET and $WCET_R$. The full coverage (coverage of 100%) is achieved when all the entries required to compute the WCET and $WCET_R$ are found in the timing table. The coverage is computed by $Cov = (1 - \frac{|entries \cap required|}{|required|}) \times 100\%$.

Figure 54 – Flowchart of the steps involved in the computation of the WCET and $WCET_R$ and the generation of the HDE by GTT.



and the turnover t_{over} parameters. Then the WCET and the $WCET_R$ of every reference point are estimated, followed by the generation of the HDE VHDL files.

Figure 55 – Example of selection of reference points with 2 segments and ranges of 25% and 50%.

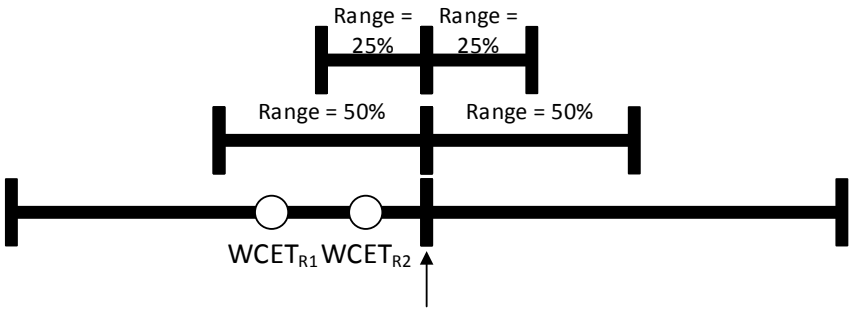


Figure 56 – Flowchart of the steps involved in the computation of the WCET and $WCET_R$ and the generation of the HDE by IPET.