UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

FELIPE ROCHA DA ROSA

Fast and Accurate Evaluation of Embedded Applications for Many-core Systems

Bachelor Thesis presented in partial fulfillment of the requirements for the degree of Computer Engineer.

Advisor: Prof. Dr. Ricardo Reis
Co-advisor: Prof. Dr. Luciano Ost

Porto Alegre
2014

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do Curso de Engenharia de Computação: Prof. Marcelo Götz
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"Scientists investigate that which already is;
Engineers create that which has never been."
Albert Einstein

“Scientists dream about doing great things.
Engineers do them.”
James A. Michener

RESUMO

Avaliação rápida e precisa de aplicações embarcadas para sistemas de muitos núcleos

Sistemas embarcados multiprocessados (*Many-cores*) são apontados com a solução mais viável para abordar as emergentes restrições de design em custo, potência e performance [Borkar and Chien 2011]. Todavia, conceber estes sistema impõe novos desafios para engenheiros de software, compreendendo entre outros: (i) definição de protocolos de comunicação entre processos, (ii) analisar e portar sistemas operacionais, (iii) possiblidade de melhor explorar modelos de programação para tratar de questões para processamento paralelo. [Marongiu and Benini 2012], (*iv*) desenvovilmento de *drivers* [Gray and Audsley 2012], (v) traduzir aplicações entre sistemas multiprocessados.

Esta crescente complexidade de software faz com que a verificação funcional torne-se mais difícil, como resultado, engenheiros de software estão dedicando-se para escalar a performance. Tornado a simulação critica durante o desenvolvimento de software, principalmente nas fases inicias, durante a exploração do espaço de design.

Plataforma virtuais *Event-driven* e *quasi-cycle accurate* como GEM5 tem como objetivo o desenvolvimento micro arquitetural uma vez que modelos detalhados são fornecidos (e.g. protocolo de coerência de cache) [Binkert et al. 2011 p. 5]. Este tipo de simuladores não é escalável a um grande número de processadores, especialmente quando se trata de usabilidade, facilidade de criação de modelos e tempo de simulação.

O contexto resultante leva a adoção de plataformas virtuais que são capazes de simular sistemas embarcadas executando código de aplicações reais a velocidade de centenas de milhões de instruções por segundo [Sanchez and Kozyrakis 2013]. Nestes cenários o *Open Virtual Plataforms (OVP)* [Imperas 2014] está emergindo com um poderoso framework de simulação provendo dezenas de arquiteturas (e.g. ARM, MIPS, MicroBraze, etc.) e vários periféricos (e.g. memória cache). No entanto, o simulador OVP semelhantemente a outros simuladores JIT-based não provem modelos precisos mas, em modelos com precisão em nível de instruções, o qual provem estimativas incorretas de software (e.g. estimativas de energia e tempo de execução).

Este trabalho objetiva começar a tratar os desafios impostos na construção de simuladores *JIT-Based* adequados para estimativas de software, aprimorando a capacidade de engenheiros de software melhor explorar o espaço de design em estágio inicial de desenvolvimento de sistemas. Nesta tese, será apresentado um modelo chamado *Watchdog* visando fornecer estimativas de energia e tempo de execução em uma metodologia integrável em qualquer simulador baseado em *JIT*

A metodologia proposta foca em um modelo baseado em eventos, baseada nas instruções individualmente executadas, simplificando a construção entorno do simulador. Adicionalmente, a abordagem proposta nesta tese é puramente *run-time*, isto significa que toda a computação necessária para implementar o modelo é realizada concomitante com a simulação, evitando enorme quantidade de uso de memória necessária para abordagens baseadas em *trace-driven* ao mesmo tempo que mantém a escalabilidade para sistemas *many-core*.

A fim de demonstrar a validade do modelo proposto, várias conjuntos de aplicações populares foram selecionados, dentre elas MiBench [Guthaus et al. 2001], Mälardalen WCET [Jan Gustafsson 2010], SPLASH-2 [Woo et al. 1995].

Os resultados mostram que a precisão do nosso modelo de tempo de execução varia de 0,6% a 11,5%, com 4,35% em de erro média. O modelo de energia atinge 0,01% a 8,6% de precisão dependendo do perfil de referência com um erro médio de 4,33%. Além disso, o modelo foi submetido a cenários de 1000 processadores mantendo um desempenho estável de 1,8 MIPS.

Palavras-chave: Sistemas multiprocessados. Estimativas de Energia. Estimativa de tempo de execução. Sistemas Embarcados.

ABSTRACT

Many-core embedded systems are pointed to be the most viable solution to addressing emerging design constraints on cost, power and performance scalability [Borkar and Chien 2011]. Nevertheless, conceive and design many-core systems impose new challenges to software engineers, comprising among other: (*i*) inter- processor communication protocol stacks definition, (*ii*) operating system (OS) porting and analysis, (*iii*) exploration of better programming model facilities to address parallel programming [Marongiu and Benini 2012], (*iv*) drivers development [Gray and Audsley 2012], (*v*) application software portability for heterogeneous multiprocessing hardware.

This increasing software complexity makes the software functional verification more difficult, as result, software engineers are struggling to scale up the system performance. Simulation becomes critical to software development, principally in early stage during space design exploration where many design decisions must be taken.

This work address the challenge of making JIT-Based simulator as OVP suitable for software performance estimation, providing to software engineers better means to explore the design space at early stage of system development. This Bachelor Thesis proposes two instruction-driven performance models, which can be used for early software performance evaluation, which were integrated into a JIT-based simulator. The proposed approach is a purely run-time based, i.e. the entire computation necessary to implement the model is concomitant with the simulation, avoiding huge amount of memory usage.

The proposed models were validated by using several benchmarks suits MiBench [Guthaus et al. 2001], Mälardalen WCET [Jan Gustafsson 2010], SPLASH-2 [Woo et al. 1995]. Results show that the accuracy of our timing model varies from 0.6% to 11.5% with 4.35% in average. In turn, the energy model provides an accuracy of 0.01% to 8.6% depending on the benchmark profile with a mean error of 4.33%. Additionally, the model was submitted to 1000 CPU’s scenarios maintaining a stable performance of 1.8 MIPS.

Keywords: Many-Core Systems. Energy Estimation. Timing Estimation Embedded Systems.

LIST OF FIGURES

Figure 1.1 - Software and architectural design costs for embedded systems at advanced process technologies. Figure extracted from IBS 2013 [IBS 2013] 12
Figure 3.1 - OVP Virtual Platform Simulation Interfaces. Figure extracted from [Davidmann and Graham 2014]. 19
Figure 3.2- Scalability of Time Slice. 20
Figure 3.3 - Block diagram and main flow of the run-time based approach. 22
Figure 3.4 - Callback example code. 24
Figure 4.1 - Proposed instruction-driven evaluation flow 26
Figure 4.2 - Block diagram of the energy approach 28
Figure 5.1 - Block diagram of developed watchdog module thread extension. 31
Figure 6.1 - Adopted reference board platform. Proposed illustration integrates figures captured from their owner's websites. 32
Figure 6.2 - Sample code necessary to access the DWT registers. 33
Figure 6.3 - Execution time comparison between real board and proposed timing model. 35
Figure 6.4 - Benchmark execution time comparison between real board and timing CPU model (OVP), varying the number of loops. Left a FFT (a) and in right a Harmonic (b) application. 36
Figure 6.5 - Simulation performance as the simulated MPSoC scales from 1 to 1000 CPUs. 37
Figure 6.6 - Execution time comparison between real board and proposed thread extension. 37
Figure 6.7 - Comparison between the timing model mismatch and the thread extension mismatch 38
Figure 6.8 - Speedup comparison between timing model in sequential and thread extension versions. 39
Figure 6.9 - Application benchmark energy consumption: gate-level simulation versus proposed instruction-driven energy model in OVP. 41
Figure 6.10 - Gain in terms of speedup: gate-level simulation versus proposed OVP energy model. 41
Figure 6.11 – Platform with four clusters used in the evaluation. 42
Figure 6.12 – Energy cost of the three mapping heuristics. 42

LIST OF TABLES

TABLE 2.1 - RELATED WORKS ON FULL-SYSTEM SIMULATION. 14
TABLE 2.2 - STATE-OF-ART IN INSTRUCTION-DRIVEN ENERGY MODELS. 17
TABLE 4.1 - ENERGY GROUPS PROFILED. 27
TABLE 6.1 - LIST OF USED BENCHMARKS TO CYCLE ESTIMATION. 34
TABLE 6.2 - LIST OF USED BENCHMARKS IN ENERGY ESTIMATION.......... 40

LIST OF ABBREVIANTIONS AND ACRONYMS

API	Application programming interface
BHM	Behavioral Models
CPU	Central processing unit
DBT	Dynamic Binary Translation
GCC	GNU Compiler Collection
GDB	GNU Debugger
HPC	High-performance computing
IBS	International business strategy
ICM	Innovative CPU Manager
ILP	Instruction-level Parallelism
ISA	Instruction Set Architecture
ISS	Instruction set simulator
ITRS	International Technology Roadmap for Semiconductors
JIT	Just In Time
KIPS	Thousand Instructions per Second
MIPS	Million Instructions per Second
MPSoC	Multiprocessor System-on-Chip
NoC	Network on a chip
NOP	No Operation
OS	Operating System
OVP	Open Virtual Platforms
OVPsim	Open Virtual Platforms Simulator
PE	Processing element
RTL	Register-transfer level
TRM	Technical Reference Model
UFRGS	Universidade Federal do Rio Grande do Sul
WCET	Worst-case execution time

CONTENT

RESUMO 4
ABSTRACT 6
LIST OF FIGURES 7
LIST OF TABLES 8
LIST OF ABBREVIANTIONS AND ACRONYMS 9
1 INTRODUCTION 11
1.1 Outline of this thesis 13
2 RELATED WORK 14
2.1 Related work in Timing CPU models in JIT-based Simulators 15
2.2 Timing CPU model in JIT-based Simulators - Closing Remarks 16
2.3 Related work in Instruction-driven Energy Models in JIT-based Simulators 16
2.4 Instruction-driven Energy Models - Closing Remarks 18
3 INSTRUCTION DRIVEN TIMING CPU MODEL 19
3.1 Open Virtual platforms (OVP) 19
3.1.1 Simulation Capability 20
3.2 Instruction-driven Timing CPU Model 21
3.3 Timing Calibration Process 21
3.4 Run-time based Approach 21
3.4.1 Disassembly and Parser (*i*) 22
3.4.2 Hash table (*ii*) 23
3.4.3 Timing information computation (*iii*) 23
3.4.4 Callback 23
3.5 Simulation behavior 24
4 INSTRUCTION-DRIVEN ENERGY MODEL 25
4.1 Characterization 25
4.1.1 Benchmark conception 25
4.1.2 Activity measurement 26
4.1.3 Power acquisition 27
4.1.4 Energy per group 27
4.2 Application Estimation 27
4.3 Simulation Behavior 28
5 THREAD EXTENSION 29
5.1 Modules modification 29
5.2 Buffer Management 30
5.3 Thread Management 30
5.4 Simulation behavior 31
6 EXPERIMENTAL SETUP AND RESULTS 32
6.1 Timing CPU model results 32
6.1.1 Timing CPU model - Experimental Setup 33
6.1.2 Accuracy results and comparisons 35
6.1.3 Scalability 36
6.1.4 Thread extension accuracy 37
6.1.5 Thread extension speedup 38
6.2 ENERGY EXPERIMENTAL SETUP AND RESULTS 39
6.2.1 Test Planning 39
6.2.2 Accuracy results and comparisons 40
6.2.3 Relative speedup gain 41
6.2.4 Application to Large Scale Systems 41
7 CONCLUSION 44
REFERENCES 45
APPENDIX A- TRABALHO DE CONCLUSÃO I 49

1 INTRODUCTION

Many-core embedded systems are pointed to be the most viable solution to addressing emerging design constraints on cost, power and performance scalability [Borkar and Chien 2011]. Such embedded systems increase performance by scaling the number of cores, which vary in terms of structure, performance and energy-efficiency, to execute system application tasks. However, employing a large number of cores will be restricted by the so-called *power wall* [Bose 2013; Esmaeilzadeh et al. 2011; Zhang et al. 2013].

Under this scenario, a significant number of cores must remain inactive or in low-consumption state at some point in time, in order to preserve the system activity within the available energy budget. Further, important challenges inherent to the design of such systems are:

- *reliability*: the transistors reach the physical limits of operation, thus becomes increasingly difficult for the hardware components to achieve reliable operation [Papanikolaou et al. 2008];
- *energy efficiency*: in battery-driven devices, it is becoming more critical than high-speed operation, and dark silicon era is imposing more power-oriented constraints to the design of such systems [Miura et al. 2013];
- *programmability*: ease of programming is a feature of paramount importance in large-scale systems composed of different processors, resulting in different platform libraries (e.g. APIs), compilers, instruction set architecture (ISAs) [Marongiu and Benini 2012];
- *simulation*: To achieve efficient exploration of emerging many-core systems, the use of flexible and scalable simulators becomes mandatory. Such simulators should combine efficient modeling, debugging and simulation capabilities for verifying the both software and hardware development,

In addition to such challenges, software development becomes one of the major challenges in many-core system design. Software development comprises, among other: (*i*) inter- processor communication protocol stacks definition, (*ii*) operating system (OS) porting and analysis, (*iii*) exploration of better programming model facilities to address parallel programming [Marongiu and Benini 2012], (*iv*) drivers development [Gray and Audsley 2012], (*v*) application software portability for heterogeneous multiprocessing hardware.

Such challenges make the software functional verification more difficult, resulting into increased development cost [Borkar 2007; Ceng et al. 2009]. IBS [IBS 2013] projects that software development consumes at least 50% of the system's design cost, and that percentage is rising, as illustrated in Figure 1.1. Developing and evaluating complex software stacks (OS, drivers, etc.), require fast and effective means for assessment of the performance-oriented and energy-efficiency practices. For instance, to assess the energy impact of software stacks, several software and hardware parameters must be tuned and evaluated properly, considering a large design space. With 200-core chips available in the market [MPPA 2014], the use of analytical models and prototyping boards is inadequate, especially for many-core architectures.

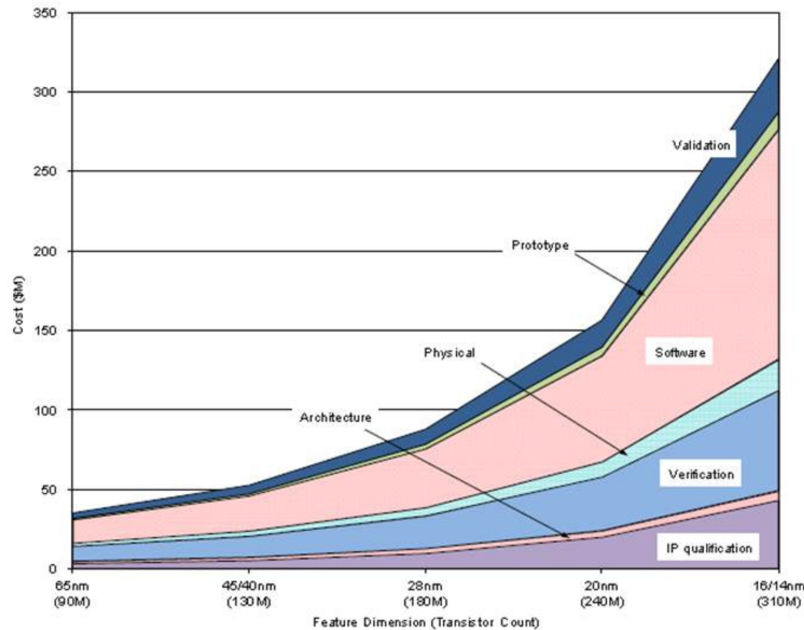


Figure 1.1 - Software and architectural design costs for embedded systems at advanced process technologies. Figure extracted from IBS 2013 [IBS 2013].

Analytical models undergo substantial development effort to identify behaviors that can be estimated by employing equations without compromising the model purpose. In turn, while specialized board designs produce accurate results, they require a substantial development effort to setup/port the software stacks. Further, physical boards can be expensive, with limited resources (e.g. number of CPUs, memory), as well as poor debuggability due the lack of internal observability and controllability of its components.

The resulting context leads to the adopting of virtual platform frameworks that are capable of simulating embedded systems running real application code at the speed of hundreds of MIPS [Imperas 2014]. While accelerating the software development, such simulators usually offer a set of CPU models and memory system models, allowing the analyses of executing different application/OSs onto multiprocessor architectures without modifications, which gives flexibility to explore more features at earlier design phase. Examples of such simulators are Simics [Magnusson et al. 2002], PTLsim [Yourst 2007], SimpleScalar [Austin et al. 2002], GEM5 [Binkert et al. 2011] and OVPSim [Imperas 2014]. Such simulators differ in terms of accuracy, simulation speed, as well as modeling and debugging support (e.g. GDB).

For example, event-driven and quasi-cycle accurate virtual platform frameworks like GEM5 target microarchitecture exploration since specific modeling details are provided (e.g. instruction pipeline details, cache coherence protocols, etc.). Such simulators are not scalable to a large number of CPUs, specifically when it comes to usability, ease-of-modeling and simulation time (around 200 KIPS [Sanchez and Kozyrakis 2013]). In contrast, simulators such as the Open Virtual Platforms (OVP) OVPsim that rely on just-in-time (JIT) dynamic binary translation can achieve simulation speeds of up to 100 MIPS. However, such simulation speed comes at the expense of accuracy; OVPsim provides instruction accuracy only, which results in inaccurate software performance estimation (e.g. application execution time).

The foregoing context provides the *motivation* for this Bachelor Thesis, which aims at making JIT-based simulators suitable for software performance and energy evaluation. The original *contribution* of this thesis is enhancing OVPSim capability by including energy and timing models, making it suitable for software performance and

energy analysis. With the underlying *contribution*, this Thesis advocates that software engineers can validate the functional behavior of the entire software stack executing it onto a given CPU architecture, using the original OVPSim. Then, software engineers may use the proposed OVPSim extension in a still reasonable simulation speed to investigate if target software stack can be executed according to the performance and energy requirements.

The contributions of the paper are summarized as follows:

- the implementation and integration of a quasi-cycle accurate timing model into OVPSim simulator;
- the extensive timing model evaluation by using several benchmarks, while comparing it to a real hardware platform;
- the development of a fast and accurate instruction-driven energy model;
- integration of proposed energy model into a NoC-based MPSoC platform;

1.1 Outline of this thesis

This work is organized in 7 chapters. Chapter 2 describes the state-of-the-art in timing and energy estimation models, taking into account approaches that are developed into JIT-based simulators. Additionally, a survey considering the most popular virtual platforms is also presented. Chapter 3 introduces the proposed run-time instruction-driven timing CPU model. After, Chapter 4 presents the development of a fast and accurate energy model. Chapter 5 contains a thread extension for the model in order to improve the simulation speed. In 6 the experimental setups and related results to timing and energy models were presented. Finishing with conclusion in the chapter 7.

2 RELATED WORK

Due the different simulation speed/accuracy tradeoffs, it's difficult to cover all modeling (e.g. flexibility, debuggability) and simulation (e.g. accuracy, scalability) requirements into one single simulator. This Chapter starts by providing an extension of the survey proposed in [Butko et al. 2012], considering the most popular virtual platforms. Such virtual platform simulators, also called *full-system simulators*, are compared according to different criteria: (*i*) accuracy, (*ii*) flexibility in terms of supported processor architectures, (*iii*) licensing, and (*iv*) support activity. Table 2.1 summarizes the reviewed work according to the four criteria mentioned.

Table 2.1 - Related works on full-system simulation.

Reference	*Simulator*	*Accuracy*	*Supported processor architectures*	*License*	*Active support*
[Magnusson et al. 2002]	Simics	Functionally-accurate	Alpha, ARM, MIPS, PowerPC, SPARC, and x86	Private	Yes
[Yourst 2007]	PTLsim	Cycle-accurate	X86	Open	Yes
[Austin et al. 2002]	SimpleScalar	Cycle-accurate	Alpha, ARM, PowerPC, and x86	Open	No
[Binkert et al. 2011]	GEM5	Cycle-accurate	Alpha, ARM, MIPS, PowerPC, SPARC, and x86	Open	Yes
[Bellard 2005]	QEMU	Instruction-accurate	ARM, MicroBraze ,MIPS, PowerPC, SPARC, x86, and others	Open	yes
[Imperas 2014]	OVPsim	Instruction-accurate	Alpha, ARC, ARM, MIPS, PowerPC, MicroBraze, and others	Open and Private	Yes

The Simulation Software Engineer (Simics) is simulator that enables unmodified target software (e.g. operating system, applications) to run onto a platform model similar to a real physical implementation. A wide range of processor architectures (e.g. ARM, MIPS, PowerPC), as well as operating systems (e.g. Linux, VxWorks, Solaris, FreeBSD, QNX, RTEMS), can be adopted to model the desired systems. This simulator includes SystemC interoperability, debuggers, software and hardware analysis views, as well. Simics has one main disadvantage, it is not claimed to be open source, and thus, commercial license is required by Wind River Systems.

In turn, PTLsim also supports simulation of different processor architectures [Yourst 2007]. PTLsim is a cycle accurate microprocessor simulator, thus the complete cache hierarchy, memory subsystem and supporting hardware devices are offered. PTLsim presents two main drawbacks; only x86 architectures are supported and the tool suite is not actively maintained anymore.

SimpleScalar [Austin et al. 2002] is an open source infrastructure for simulation and architectural modeling. As previous simulator, software engineers can use SimpleScalar to develop applications and execute them onto a range of processor architectures, which varies from simple unpipelined processors to detailed microarchitectures with multiple-level memory hierarchies. However, SimpleScalar is not actively maintained anymore (last update was in March 2011), and other faster solutions, like GEM5 are available.

GEM5 is a modular discrete event simulator, which is open-source and supports

a rich set of ISAs [Binkert et al. 2011 p. 5]. Moreover, this simulator has an active development and support team. As mentioned before, GEM5 target microarchitecture exploration, which incurs in huge simulation overheads due the number of modeled aspects. Further, the amount of memory required by these approaches is too high, making their use infeasible when exploring a large design space exploration.

QEMU [Bellard 2005] is an open source and a functional simulator that relies on dynamic binary translation. QEMU can be used to simulate several CPUs (e.g. x86, PowerPC, ARM, and Sparc). Nevertheless, QEMU is designed to single-processor platforms and virtualization purposes. Thus the lack of documentation on the APIs or standardized methodology for creating many-core platform models limits its use.

Excluding PTLSim that only supports x86, reviewed simulators are composed of several processor architectures. Quasi clock accurate simulators such as SimpleScalar and GEM5 entail high-simulation time, thereby limiting its applicability to the exploration of large many-core systems. Further, while Simics has a private license. Further, SimpleScalar does not provide support or development anymore.

OVPsim supports the larger number of processor architectures (ISAs) among reviewed simulators. OVP supports dozens of architectures (e.g. MIPS, ARM, x86, PowerPC) ramifying in several model variants (e.g. arm cortex-A5, cortex-A9, cortex-M4F, etc.), as well peripherals (e.g. DMAs, TIMERs), and integration with System-C modules. Besides, of supplied models, the user is able to create customized models easily integrated with the platform, justifying our choice.

As mentioned before, the lack of accuracy inherent to JIT-based simulators is motivating research in alternatives performance / accuracy tradeoffs. The next Section presents approaches that are instrumenting JIT-based simulators with timing models. In this context, instructions, basic architecture block models and their inter-operations (e.g. read and write) are calibrated according to a reference platform. Thus, software performance evaluation can be estimated by, for instance, summing up the annotated timing numbers along execution of given application.

2.1 Related work in Timing CPU models in JIT-based Simulators

Chiang et al. [Chiang et al. 2011] utilize the integration of QEMU and SystemC in order to allow faster clock-accurate evaluation when compared to RTL-based. Attaching a SystemC co-processor in the simulator framework, using the information extracted from the DBT interface. This approach reduces the simulation speed of QEMU, capable to reached approximate 38 MIPS, to approximate near 0.46 MIPS using the full simulation with SystemC. A pipeline model was included into QEMU in [Thach et al. 2012], where the authors proposed a two-phase approach an offline and an online phase to estimate the application performance. In the offline phase, a cycle pre-estimation of the application execution time is performed. Using the computed information at dynamic adaption phase when CPU status and execution time of critical instructions are also taken in account, improving the approach accuracy presenting a mismatch around 10%.

A similar approach is presented in [Stattelmann et al. 2012], where worst-case execution time (WCET) analysis and QEMU are combined for a LEON3 processor. In this work, the offline phase is composed by four steps, which produce a timing database that is used during the QEMU simulation. The proposed work in [Bohm et al. 2010] modified an ARC instruction set simulator based on JIT DBT to improve the simulation accuracy. A complex pipeline and execution state models were constructed direct in the DBT framework, taking advantage from the direct access of JIT Translation block.

2.2 Timing CPU model in JIT-based Simulators - Closing Remarks

Making an overview of the reviewed researches, it is noticeable that QEMU is largely employed due its free GNU licensing. It is possible to verify that almost all related approaches are based on a reconstruct pipeline model that update its internal state according to executed instructions during the simulation. There are different abstraction levels of implementations and consequently, the necessary amount of data information considered before and during the simulation. Complex models may implement several abstract queues, internal pipeline state, etc.

The drawback of the approach proposed in [Stattelmann et al. 2012], is the prior application profiling phase, which restricts its use when exploring large scenarios composed of diverse applications. Another disadvantage of this work is that any software modification (e.g. changing the OS scheduling algorithm) implies in re-running offline phases.

Different from the reviewed work, the proposed approach (described in section 3.4) relies on OVP and run-time basis, eliminating huge trace files, as well as pre- or post-processing software/application profiling. Despite the low-memory usage, the proposed approach can be easily configured to observe as many CPUs as desired. Another contribution of the proposed timing CPU model, when compared to the reviewed works, is the easy portability to other CPU architectures.

2.3 Related work in Instruction-driven Energy Models in JIT-based Simulators

In the case of prototyping boards, the power information is captured from a precision resistor positioned between the power supply and the power input pin [Bazzaz et al. 2013; Konstantakos et al. 2008; Lee et al. 2001; Nikolaidis et al. 2003]. The use of physical information can aggregate precision to the high-level models (error varying from 2.5% to 7% as presented in the third column of Table I). However, to measure the power of each instruction, additional and expensive hardware (e.g. high-performance oscilloscopes) are required. Another drawback of this approach is the difficulty of accessing/isolating individual modules inside the processor due to internal structure and connections (e.g. Flash, Rom, SPI, AD, and DC).

In simulated-based techniques, the required information is extracted from low-level simulators (e.g. SPICE, gate-level), in which a hardware description is used to execute input benchmark applications and to profile the power of each instruction. For example, in [Abril Garcia et al. 2002] an instruction set simulator (ISS) is enriched with an energy model based on the mean switching activity of the processor, which is modeled by two states, *active* and *NOP*. A similar approach is presented in [Sultan and Masud 2009], which considers the average switching activity of an LEON3 processor simulated at RTL level. In this work, the power is computed according to the number of transitions generated in response to a certain instruction that is fetched from BootROM of LEON3.

Authors in [Castillo et al. 2007] propose obtaining energy values directly from the analysis of the source-code without requiring simulation or even compilation. A further higher-level approach is proposed in [Callou et al. 2011], in which the source-code is converted in a Colored Petri net model, which is used to estimate the energy cost of a given application.

Table 2.2 - State-of-Art in instruction-driven energy models.

Reference	Reference model	Claimed Accuracy	Benchmark suite	Description
[Lee et al. 2001]	ARM7TDMI	Average 2,5% and worst case 6,33%	36 randomly-generated instructions	Model based in a linear regression analysis.
[Abril Garcia et al. 2002]	Gate-level estimation using an ARM920	Not available	MPEG-4 video decoder	Inclusion of a power model calibrated in gate-level activity in a System-level Cycle-Accurate simulator
[Kalla et al. 2003]	Synthesizable RTL of a SPARC	Energy less than 5% and per-cycle power inside 15% of error	Bubble Sort, Heap Sort, Insertion Sort, Key 3 and 3D image processing	Model based in Active and Stall consumption for each module of the architecture, refined with inter-instruction effect. Additionally provides the maximum and the minimum of power.
[Nikolaidis et al. 2003]	ARM7TDMI	5%	A few instructions	Abstract model for pipeline with static, inter-instruction, and pipeline power.
[Konstantakos et al. 2008]	Motorola HC908GP32	Not available	Not available	The instructions are divided into groups by the cycle's length.
[Lee et al. 2006]	Gate-level estimation for M32R-II and SH3-DSP	Average 3% and worst case 16%	JPEG and MPEG2 encoders, compress, FFT and DCT	Training benchmarks are used in conjunction with a gate-level simulator and linear optimization to generate several parameters to describe frames of instructions. Afterward this parameters are utilize together with ISS.
[Castillo et al. 2007]	Arm ISS, arm-elf-gdb, for a ARM9TDMI and ARM TRM	Less than 11%	Bubble Sort, FIR, Array, Fibonacci and Quicksort	An online analysis of the source-code without requiring simulation or even compilation. Based in the mean energy per instruction calculated from values provide by ARM Manual. Detailed study about the operators in C e.g. + = >> and their costs in meter of instructions.
[Sultan and Masud 2009]	Synthesizable RTL of a LEON3	Not available	Not available	Propose of an instruction level power model profiling each instruction in different stages of a pipelined processor. The aim is to measure the activity generate in the processor and taking in count the capacitance to calculate the power.
[Callou et al. 2011]	NXP LPC2106 with an ARM7TDMI-S	7% in average	5 applications	Stochastic approach based on Coloured Petri nets and source code analysis
[Bazzaz et al. 2013]	AT91	Less than 6%	8 MiBench benchmark	ISS Model calibrated from real measures. Complete model with static, inter-instruction, and pipeline power.
Proposed approach 2014	Gate-level estimation	Between 0,06% and 8%	19 benchmarks from WCET and in-house applications	Instruction-driven model calibrated from the switching activity of the processor internal components. Run-time model developed on the basis of OVP API that monitors the instructions executed by a given CPU.

2.4 Instruction-driven Energy Models - Closing Remarks

Reviewed approaches focus on creating instruction-driven models, which compute energy/power values by observing the sequence of executed instructions. One difference between such approaches is the calibration process. For instance, in [Bazzaz et al. 2013] authors evaluate instructions individually to feed the instruction-driven model, while in [Lee et al. 2001] fixed length instruction groups are used. Authors argue that different transition scenarios may significantly affect the energy estimation. For that reason, some works such as [Bazzaz et al. 2013; Kalla et al. 2003; Nikolaidis et al. 2003] also calculate the inter-instruction energy, i.e. the energy required to switch from one to another.

Another distinction lies in the energy evaluation process. For instance, the approach proposed in [Castillo et al. 2007], differs from the other works in the sense that it translates the source code in an intermediary code representation, which is used to estimate the application energy consumption. This approach does not require simulation that may decrease the energy evaluation effort. However, to predict the behavior of loop and branches only by code inspection is not a trivial task that may pose other design/evaluation challenges.

Our contribution distinguishes from previous works by enhancing the OVPSim (JIT-based simulator) with energy evaluation capability allowing faster and accurate exploration of energy-efficiency software development. Contrary to the most of reviewed approaches, our approaches cover both timing and energy evaluations. Another advantage of our approach is that once calibrated whatever OS/application can be ported, modified, and its timing and energy-efficiency can be evaluated without any code modification or re-calibration phase. To accomplish these features, a common foundation for monitoring at run-time the instructions executed by each CPU, while presenting the system functionality was developed. This foundation is described in the next Chapter.

3 INSTRUCTION DRIVEN TIMING CPU MODEL

This Chapter describes the proposed timing CPU model in OVP. The following Section presents the basic concepts and features related to OVP.

3.1 Open Virtual platforms (OVP)

Open Virtual platforms is a simulation framework marketed Imperas [Imperas 2014]. OVP is composed of three main components: (*i*) APIs that enable modeling in C/C++ hardware components, (*ii*) library with a large number of CPU architectures, peripheral, memory, and sub-system models, and (*iii*) the OVPsim simulator.

As illustrated in Figure 3.1, OVPsim employs a Just-In-Time Code Morphing binary translation simulator engine that dynamically translates target instructions to the host machine instructions. In this context, OVPSim is capable to handle virtual memory simulation with minimal performance penalty. OVPsim also supports non-intrusive semihosting using dynamically loaded libraries that are completely separate to processor models. As defined in [Davidmann and Graham 2014], there is no need to compile application code using special flags to support semihosting: given an appropriate semihost library, OVPsim can run unmodified binary. OVPsim semihosting works by allowing semihosting libraries to take special actions either when particular functions are executed in the simulated application (e.g. write) or when particular instructions are executed (e.g. break instructions).

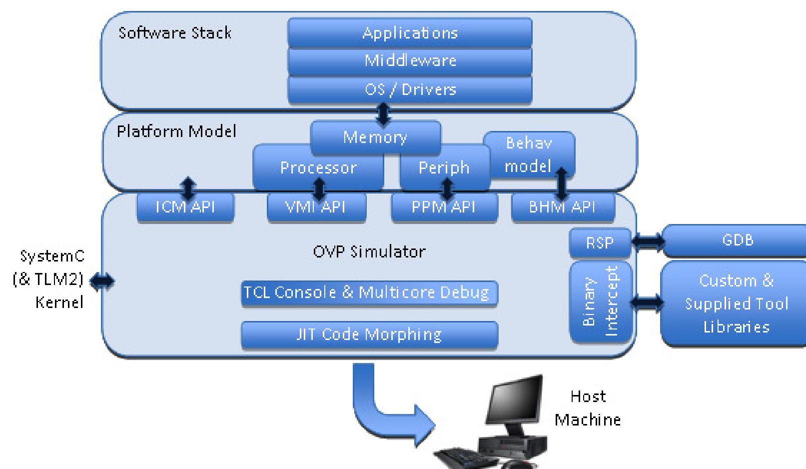


Figure 3.1 - OVP Virtual Platform Simulation Interfaces. Figure extracted from [Davidmann and Graham 2014].

OVPsim has been developed for the maximum simulation throughput and includes several optimizations enabling simulation of platforms utilizing many homogeneous and heterogeneous processors with many complex memory hierarchies. Also includes several models of MMUs, caches, and TLBs.OVP framework features four API: Innovative CPU Manager (ICM), Virtual Machine Interface (VMI), Behavioral Hardware Modeling (BHM), and Peripheral Programming Mode (PPM). Each of which has a specific purpose, for instance the ICM is intended to create and to simulate the target platform, including any number of processor, busses, memories and peripherals models. Busses, memories and processors can be interconnected in arbitrary topologies and arbitrary multiprocessor shared memory configurations. Further, ICM functions also encapsulate OVPsim models in SystemC or TLM 2.0 simulations. OVP models also include processors and peripherals wrappers for use with SystemC TLM2.0. The ICM is responsible for merging all four APIs in a single environment

providing interoperability between then. OVPSim platform is composed of one C file containing few lines of code compiled with Imperas libraries to create an executable file.

3.1.1 Simulation Capability

In order to support multi–core simulations, OVPsim implements a Round-Robin scheduling algorithm similar to a typical used in OS schedulers. Thus, each processor entity (PE) has a time slice variable, typically 0.001 seconds. Note that it is possible to define a common time slice for all processors of a given scenario. Such variable is converted into a number of instructions that should be executed by each processor in the defined time slice. The number of simulated instructions is obtained multiplying the time slice by the processor nominal MIPS, which is defined 100 per default. OVPSim works in sequential way (i.e. simulating a unique processor at time, even if the simulator is hosted in a multi-core host machine). Nevertheless, this algorithm inserts an issue related to the synchronization between simultaneous events related to different processors. For instance, if a processor sends a message to another in the middle of their time slices, the receiver only will be aware of the message at the begin of his time slice. In simulation scenarios tightly based in intercommunications between processor (e.g. NoC-based MPSoC), the precision of results may be affected.

A possible solution could be resizing the time slice, decreasing the number of instruction executed each time by the each processor. In order to observe this behavior a series of experiments was conducted, using the same application (FFT), varying the time slice from 1 to 0.000001 and the instructions per window from 1.000.000 to 1. Figure 3.2 shows the degradation in terms of simulation speed vs. time slice. It can be observed that the simulation slows down dramatically when less than 5 thousand instructions per round are executed. Notwithstanding, the modification impacts in the performance obtain by the simulator due the cost in the context switch between the processors.

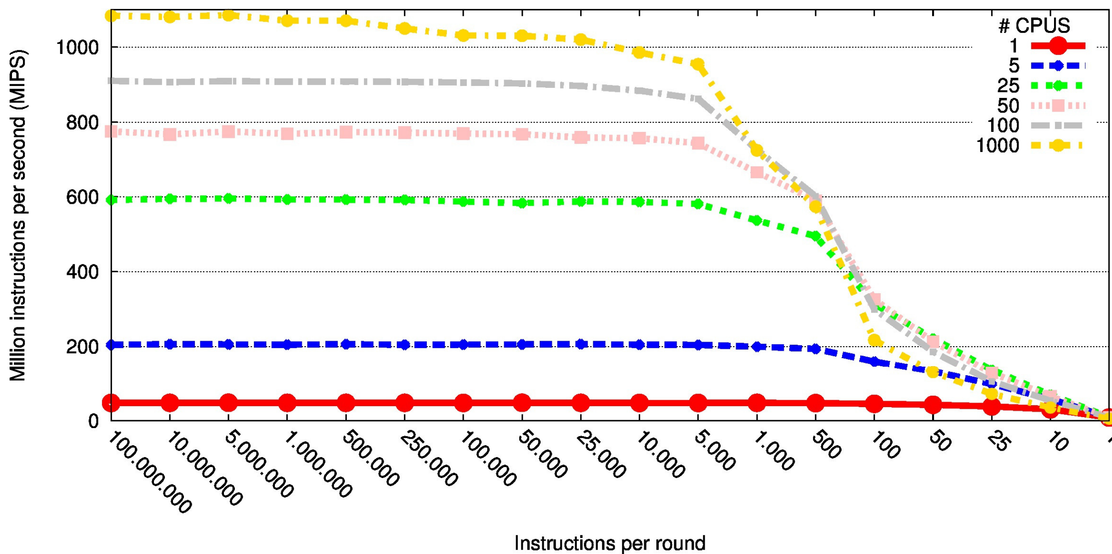


Figure 3.2- Scalability of Time Slice.

3.2 Instruction-driven Timing CPU Model

This Section describes the development and the integration of the proposed timing CPU model into OVPSim. As mentioned before, our timing CPU model relies on monitoring at run-time the instructions executed by a target CPU. The proposed approach requires an instruction set profile as mean to determine cycle count (timing information), which can be captured from datasheet, physical boards, low-level simulation, etc. In order to diminish development complexity, we propose to combine instructions in groups according timing costs similarity. Separating ISA in classes and groups according different timing behaviors and other representing constant one-cycle instructions. While simulating, unless the model identifies an instruction as belonging to one of defined groups, it is considered as one-cycle instruction. In order to demonstrate the timing calibration process, let's consider the Cortex-M4F ISA as study case.

3.3 Timing Calibration Process

Timing Behavior of ARMv7-M Thumb instruction set implantation in Cortex-M4F can be grouped according their similarities, for instance, almost all logical and arithmetic instructions are single cycle. Notwithstanding, division has a dependable cycle count imposed by the early termination accelerator based on the number of leading ones and zeroes in the input operands. In this case, one division can take from 2 to 12 cycles.

While not taken branch instruction execution requires a single cycle, taken breaches lead to 3 cycles as result of pipeline flushing. Semaphore cycle count is usually two, and for *push* and *pop* the exact cycle count relies on the number of registers in the register list, increasing in one cycle per register. Additionally, instructions that use PC as destination register have three cycle's penalty.

In the case of load and store, the timing analysis is the most complex one. Load and store are normally two cycle's instructions, as result of neighboring load and store. Single instructions may pipeline their address and data phases affecting cycle's count, leading to one cycle's instructions. The instructions are pipelined when the next instruction is an *LDR* or *STR*, and the destination of the first is not used to compute the address for the next instruction, then one cycle is removed from the cost of the next instruction (e.g. *LDR R0,[R1,R5]; LDR R1,[R2]; LDR R2,[R3,#4]* - normally four cycles total instead six). Further, other environmental factors that could modify the cycle count can be found in the TRM [ARM 2013]. This document has important timing information, which are fundamental to the proposed timing model. Nevertheless, the document not exposes detailed information concerning timing behavior. In order to verify the collected timing information, several experiments were conducted using the platform [STM32F4 2014], considering the interleaving between instructions, as load and stores.

3.4 Run-time based Approach

The monitoring process was developed on the basis of OVP APIs and integrated in a component called Watchdog, comprising three main modules: (*i*) disassembly and parser, (*ii*) a hash table containing pre-characterized groups of instruction, and (*iii*) timing information computation, in the Figure 3.3 and descripted in sections 3.4.1, 3.4.2, and 3.4.3.

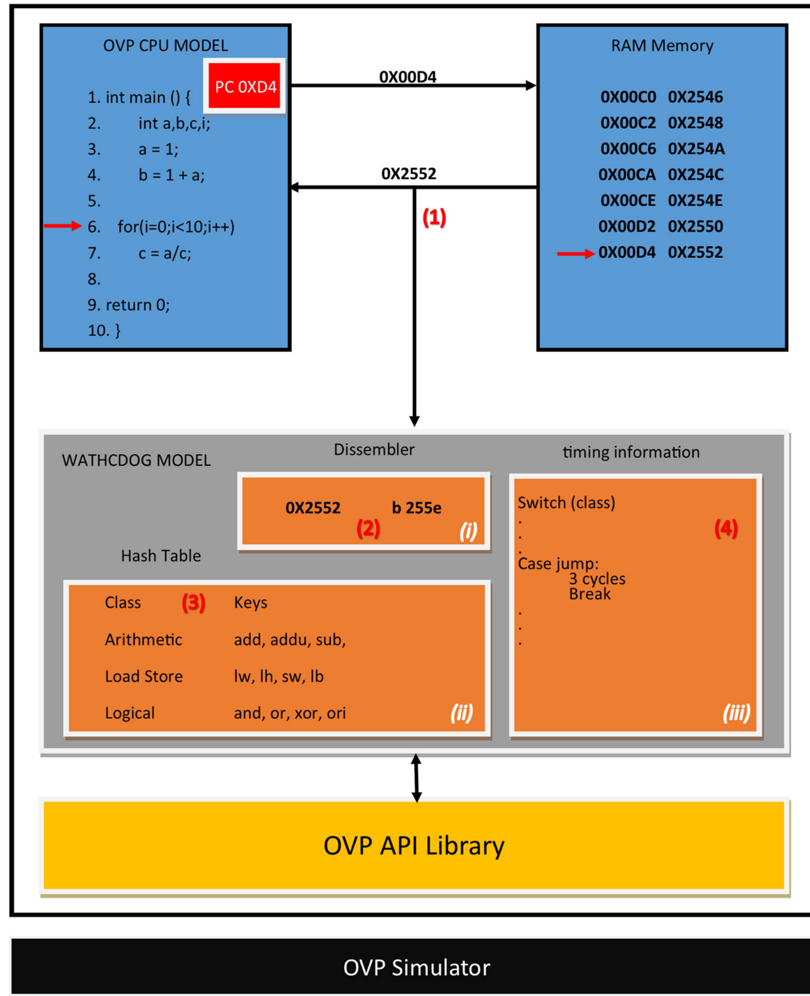


Figure 3.3 - Block diagram and the main flow of the run-time based approach.

3.4.1 Disassembly and Parser (*i*)

In this module, an instruction binary code is disassembled into a string and afterwards subdivided in other substrings. The purpose is to isolate the instruction mnemonic from the instruction registers arguments to feed both hash table (*ii*) and timing computation module (*iii*). As a means to disassemble binary code instructions, our implementation employs a function provide in the ICM API called *icmDisassemble*. This function call disassembles an arbitrary memory position for an arbitrary CPU instance at any moment of the simulation. The function arguments are the CPU model object and the target memory address. This function returns a string containing the address, the mnemonic, and the arguments of the disassembled instruction (e.g. *0X2550 STR r3, [r7, #20]*).

In *icmDisassemble* function is important to note issues related to concurrent access, consequently this calls are necessary exclusive, cause by internal buffers when storing partial results. The model describes in this chapter is sequential, i.e. the instructions are guaranteed to be exclusively resulting in mutual exclusion for all access. Nevertheless, with the purpose of increase simulation speed was developed a POSIX Thread version. As a consequence, concurrent access is possible, necessitating assure mutual exclusion, subsequently, is necessary to parse the string, separating in different sub-strings in order to correctly update internal structures and supply the next module.

3.4.2 Hash table (*ii*)

With the purpose of efficiently store the timing information, it was employed a hash table, instructions mnemonic are the key for the data, which relates the instruction group and the key. Thus, in a single access, using the mnemonic as key is possible retrieve the information for the timing computation. The hash table is created and initialized at simulation begin, inserting all mnemonic in the hash.

In the early model version was not used a hash table, instead a linked list due the implementation simplicity. Occasionally, was noticeable this storage access as performance bottleneck during simulation. As result of our methodology, every executed instruction is processed sequentially after the fetch event; consequently an access is performed after each fetch. Therefore, the search in linked list is not scalable and do not have a fixed time cost, leading to performance loss. In order to overcome this situation, a hash table replaces the linked list as storage solution in newer version.

3.4.3 Timing information computation (*iii*)

This module is responsible for processing information provide by previous modules together with processor state information. Structured as switch, using returned value from the hash table as a selector variable. A null pointer signifies a single-cycle instruction, accordingly to our methodology when assume one cycle per default. Estimate cycle's count requires several chained tasks, separate in two groups: Those performed to every group and those restricted to individual group. For instance, all instructions possessing registers list requires several verifications: Instructions modifying the program counter need to be identified, as well inline shifters.

Each profile class requires a specific treatment after been identified. For instance, at any time a load occurs it is necessary verify if the precedent instruction was a store or load, the same for stores. To implement this verification, a buffer of one instruction is maintained during the simulation and verify when is necessary. In instructs as *pop* or *push* requires count the number of registers contained in the register list of the instruction, each one adds one cycle to cycle count. To correct estimate division cycles is necessary acquire the two operands, verifying the number of leading zeros in the operands. According to the number of zeros, the cycle count is estimated.

3.4.4 Callback

The monitoring process bases in a special function supply by the ICM API called Callback. Triggering when a predefined particular event occurs, and subsequently the simulator call a handler function provided by the designer. Trigger configuration events, as memory access or as bus access, take place at compilation-time. Additionally allowing restrict this event a memory range instead entire address space. Restricting address range impacts in the simulation speed as a result of the algorithm employed by the simulator.

Callaback instantiation requires two steps, shows in Figure 3.4. First, an ICM function (i.e. icmAddFetchCallback) (a) located in the platform creator inserts callback trigger. This specific function adds a fetch event trigger. Handler function construct as ICM specific wrapper (b) perform any task desire. Is important note that simulation stops during callback treatment and handler execution.

```
icmAddFetchCallback(processor_objetct,memory_base_address,SIZE, fetch,(void*) arguments);
                                    (a)
ICM_MEM_WATCH_FN(fetch) //Memory callback wrapper
{

   //Handler function

}
                                  (b)
```

Figure 3.4 - Callback example code.

3.5 Simulation behavior

As propriety of OVPsim all processor, busses, and memory are created at run-time using linked libraries. The same process is applied to our Watchdog module and its internal components. Also is possible define different constraint to the model, as an CPU instance that will be monitored. Figure 3.3 shows a block diagram of the watchdog associated with the platform. Numbers from 1 to 4 are used to describe the model behavior.

After the platform simulation begins, whenever an instruction is fetched from the memory (1) is triggered a callback, thus starting the Watchdog. Inside the first module, the binary code of the instruction is acquired using the program counter (PC) register. Thus, the binary code is disassembled, divided into sub-strings, and identifies the instruction that must be executed (2). The identified instruction is employed as a hash table key to discovering which class (e.g. arithmetic, load, store) such instruction belongs (3). Thus, computing the necessary number of cycles to perform this instruction, considering the predefined timing information. Once, computed the cycle count, each instruction is executed in the CPU (4). At simulation end is possible retrieve the cycle estimation separated by CPUs, additionally, the statistics concerning the number of executed instruction per mnemonic.

Our methodology can be extended and applied to other simulator frameworks. Nevertheless, to achieve better simulation performance, the development demonstrated in section 3.4 takes full advantage of OVP functions and methods. Additionally, as result of the OVP APIs extensive use diminishes developing time and cost, improving productivity.

The proposed methodology lies in a single-event model, based on the individually executed instruction, simplifying construction around the simulator. Capturing as an event, identifying the instruction, processing properly, and storing the computation. The proposed approach in this thesis is a purely run-time based, i.e., full computation is executed concomitantly with the simulation, avoiding a huge amount of memory usages needed by trace-driven based approaches and maintaining scalability, as well as pre- or post-processing software/application profiling.

4 INSTRUCTION-DRIVEN ENERGY MODEL

Proven the flexibility of previous timing model, in this chapter, is introduced an instruction-level energy estimation model. It is important to highlight that the proposed profiling method is simple and transparent, applicable in any other processor cores with no considerable rework.

The Proposed energy model maintains a methodology presented in chapter 3 to timing estimation. Focusing in a single-event model, based in the individually executed instruction, simplifying construction around the simulator. Capturing as an event, identifying the instruction, processing properly, and storing the computation.

However, in the first case, almost information about cycle duration was available in the documentation, for this energy approach, the manufacturer does not offer the information about energy consumption. In order to overcome this problem is necessary acquire information through a calibration phase. As a result, dividing the process flow in two: a Characterization phase and the simulation phase.

4.1 Characterization

The first and most important phase is the characterization, which profiles the energy spent by each instruction belong with the target ISA. The proposed characterization flow is validated taking as reference the Plasma processor [Plasma 2014], a 32-bit RISC processor based in the MIPS architecture with a 3-stage pipeline. The characterization flow is executed once per ISA architecture, and it comprises four main steps: (*i*) benchmark development; (*ii*) activity measurement; (*iii*) power acquisition; (*iv*) energy computation. In Figure 4.1, the numbers from 1 to 10 are used to describe intermediary files, while the letters from A to D represent the adopted tools.

4.1.1 Benchmark conception

The first step of the characterization flow encompasses developing the benchmarks that will be used to profile the energy consumption for each instruction (1 in Figure 4.1). To reduce the computation cost of our model, we classified the instructions in seven groups due their behavior in the processor data-path: (*i*) arithmetic, (*ii*) logical, (*iii*) move, (*iv*) branches, (*v*) load/store, (*vi*) nops, and (*vii*) shifts. One practical example is the close relationship between instructions such as *add* and *addiu* or between *lw* and *sw*. Note that the mnemonic *move* is considered in this work as arithmetic instruction through a pseudo-instruction implementation (performed by a *lui* and *ori*).

To profile the energy consumption of each instruction group (1 in Figure 4.1), an application was carefully developed, in a way, that at least 90% of the executed instructions would belong with the target group, including the possible variations of the same instruction (e.g. *add, addi, addiu*, etc.). Previous experiments using higher percentages (e.g. 95%), showed a negligible difference. Note that multiplication and division instructions are modeled as 12 arithmetic instructions each, since our Plasma version takes 12 cycles to execute them. Further, an application benchmark was created to characterize the pipeline stall as a *nop* instruction.

Each application is executed in the OVPSim simulator (C in Figure 4.1) to verify its correctness and to extract the exact number of executed instructions (8 in Figure 4.1). Each application executes, in average, 35 thousand instructions, which requires less than one 1 second of simulation. The Plasma is synthesized with Cadence RTL Compiler tool (2 in Figure 4.1) targeting a 65nm low-power library from ST Microelectronics. Then, each application is simulated using Cadence Incisive simulation tool (B), taken as inputs: the Plasma netlist (2), the application object code (3), a tcl script (4), and the sdc file containing the timing constraints (5). The simulating is

executed until the end of the application.

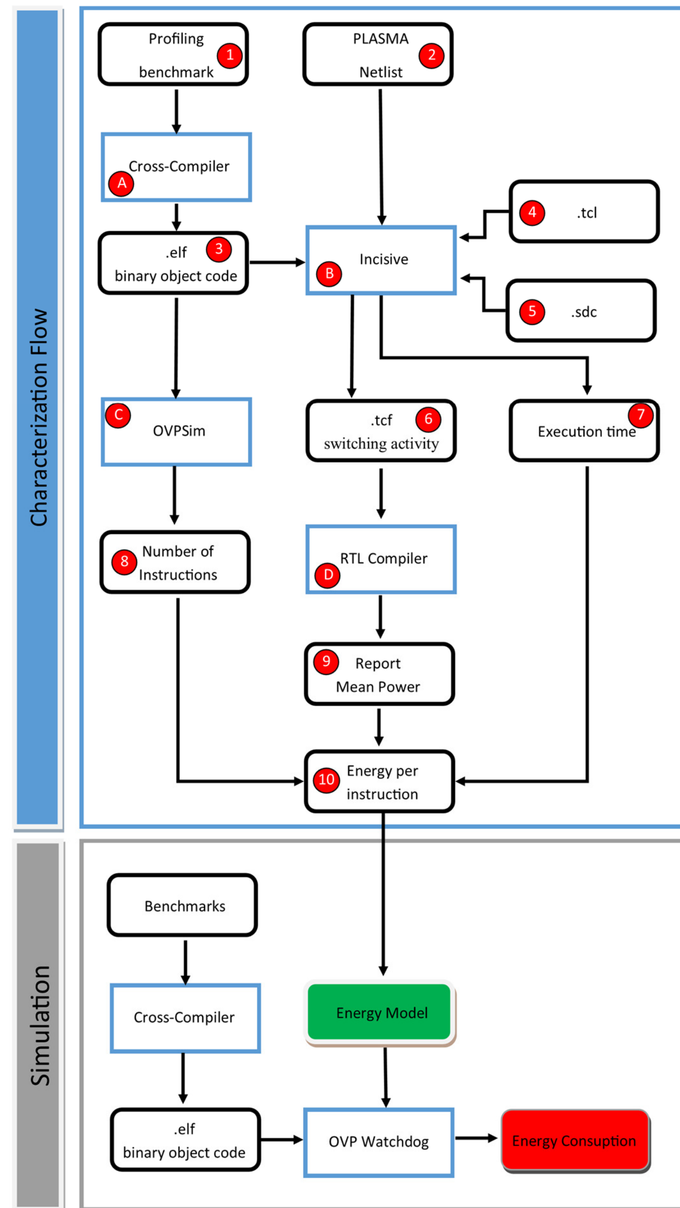


Figure 4.1 - Proposed instruction-driven evaluation flow

4.1.2 Activity measurement

Each application is simulated using Cadence Incisive simulation tool (B in Figure 4.1), taken as inputs: the Plasma netlist (2 in Figure 4.1), the application object code (3), a *tcl* script (4), and the *sdc* file containing the timing constraints (5). The simulating is executed until the end of the application. As a result, a *tcf* file (6) is generated. This file contains statistic information about the switching activity of each cell and wire in the netlist. In addition, the exact execution time of each application is collected (7).

4.1.3 Power acquisition

Finally, the power evaluation is executed. Cadence RTL Compiler (D Figure 4.1) also performs this task; the tool reads the netlist (2 Figure 4.1) and computes de average power consumed by each cell based in their switching activity information in the *tcf* (6) file. Subsequently, the tool produces a report containing the average power consumption (9) for the application.

4.1.4 Energy per group

The final step computes the average energy spent by each characterized group (10 in Figure 4.1). Associating the average power (9) and execution time (7) collected in the previous step with the number of instructions (8), the energy consumed per instruction group is obtained using Equation 4.1. This flow is repeated for each instruction class.

Equation 4.1 - Formula to calculate the energy spend by each group.

$$Average\,energy = \frac{execution\ time(\mu s)\ x\ power\ (mw)}{executed\ instructions}\ (nJ)$$

Table 4.1. Summarizes the energy results for each instruction group. Results in the column "number of instructions" are obtained through the OVPSim simulation. Results in the columns "power" and "execution time" are obtained through the gate level simulation. The total energy consumption ("energy" column) is obtained by multiplying the number of instructions by the total execution time. Then, with the number of executed instructions and the total energy consumed, it is possible to compute the energy consumed by each instruction ("Energy per Inst." column).

Table 4.1 - Energy groups profiled.

Groups	Power (mW)	Exec Time (us)	Energy (nJ)	# of inst	Energy per Inst (nJ)
Arithmetic	6,456	342,755	2212,826	34764	0,0636528098
Jump	6,046	102,600	620,320	10224	0,0606728873
Load-Store	4,094	1042,800	4269,223	48561	0,0879146476
Logical	4,469	349,735	1562,966	35462	0,0440743815
Move	3,129	480,725	1504,189	39363	0,0382132593
NOP	2,141	257,155	550,569	26130	0,0210703733
Shift	3,824	298,735	1142,363	30362	0,0376247494

4.2 Application Estimation

After the characterization phase is finished, the next phase comprises building the energy model in the instruction-set simulator. Developing the monitoring process by modifying the previous cycle estimation model presented in the section 3.4, extending and enriching with energy models. Figure 4.2 shows the three main Watchdog modules: (*i*) disassembler, (*ii*) a hash table with pre-characterized groups of instructions, and (*iii*) internal data structures. In energy model, the hash-table contains the belonging class (e.g. arithmetic, load, store) to each instruction mnemonic (e.g. *add, or, lw, etc.*).

4.3 Simulation Behavior

During the simulation, whenever an instruction is fetched from the memory (1) is triggered a callback, thus activating the Watchdog. Inside the first module, the binary code of the instructions is acquired using the program counter (PC) register, thus the binary code is disassembled, divided into sub-strings, and identifies the instruction that must be executed (2). The identified instruction is employed as a hash table key to discovery which class (e.g. arithmetic, load, logical) such instruction belongs (3). Thus, computing the necessary energy to perform this instruction, considering the predefined energy information. Once, storage the information, each instruction is executed in the CPU (4). At simulation ends is possible retrieve the cycle estimation separated by CPUs, additionally, the statics concerning the number of executed instruction per mnemonic or grouped by class.

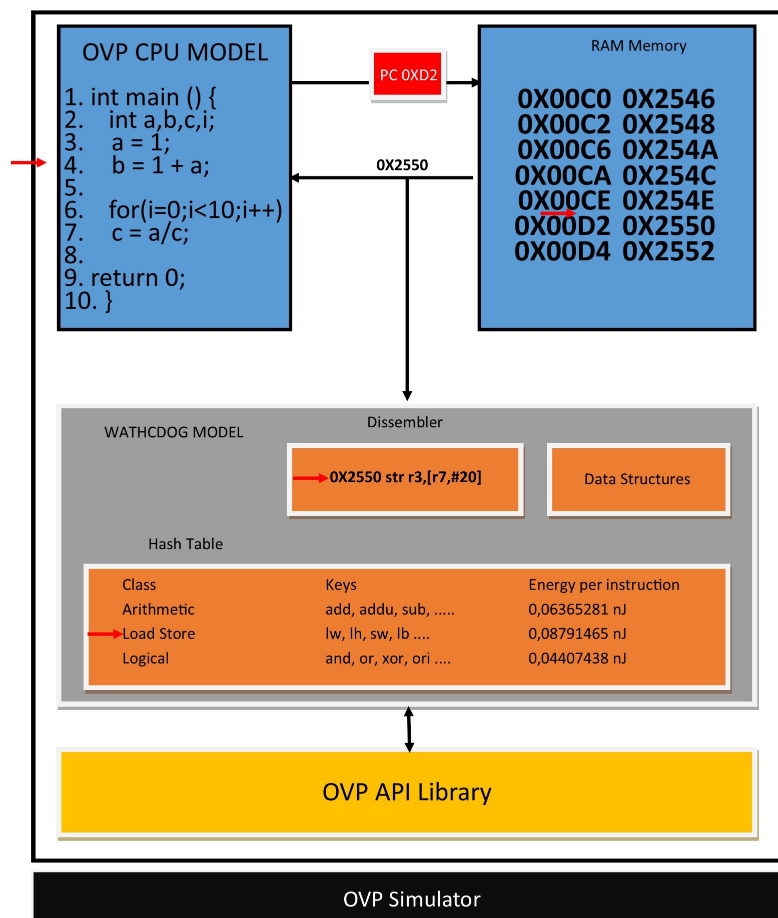


Figure 4.2 - Block diagram of the energy approach.

5 THREAD EXTENSION

In order to improve the simulation speed of proposed timing and energy models, a thread-based implementation (i.e. POSIX Thread library) is proposed. We select POSIX Thread due its portability to multiple host machines in different operation systems (i.e. Windows and Linux), its availability in standard compilers as GCC, and its low footprint. The model describes in chapter 3 will be further referred as *sequential* version while be named *thread-based*.

Proposed extension maintains a methodology presented in the previous chapter, based in individually executed instruction, every fetched instruction is disassembled, parsed, identified, and processed. The sequential version accomplishes all tasks in the interval between two fetches producing speed degradation. In this context, every task inserted in the model causes significant computation, preventing the addition of more complex features. However, is impossible to parallelize the entire model to achieve better workload distribution. This extension proposes divide cycle estimation in two phases: a concomitant and parallel. The first occurs in the fetch event treatment (i.e. the callback), and the other is asynchrony using the OVPsim natural flow as referential. Figure 5.1 shows the thread-based version block diagram. The propose extension aim transfer as much as possible workload to parallel phase, introducing a buffer between concomitant and parallel phases in order to accumulate information before processing. Note, this buffer differs of other works exist a post-processing moment, in this proposed model the buffer consumption is concurrent with the simulation. Described in section 5.2.

The concomitant phase is responsible to intercept instructions, binary code disassemble, buffer management, and thread creation. Instruction trace is mandatory belong to this phase, is impossible create a satisfactory trace using functions asynchronous to OVP CPU model in the adopted framework. Consequentially is necessary store the trace in a temporary buffer, adding buffer management cost to this phase. Binary code disassemble theoretically is compulsory in the concomitant phase, nevertheless, while development was discovered a problems related to icmDisassemble function presented in section 3.4.1. Thus, was necessary maintain disassembly and consequently the buffer was redesign to contain a string (32 bytes) instead 32 bits operation code increasing memory consumption. However, the proposed approach has a smart buffer management in order to reduce memory footprint during execution.

5.1 Modules modification

Insertion of explicit parallelism certainly increases model complexity, as so, creating other and modifying previous modules presented in sections 3.4.1, 3.4.2, and 3.4.3. Figure 5.1 shows the module in roman algorisms (from i to vi). Disassembly and Parser module are organized in two modulus, (i) and (vii) respectively. Comprising sequential phase, disassembly module (i) utilize icmDisassemble function, generating a string. Buffer module (ii) is responsible to storage the generate strings to posterior processing. Buffer implementation follows a queue data type, i.e. making bottom insertions and top removals. Module (iii) manages both buffer and thread. Buffers creation is on demand (i.e. they are dynamic created when certain conditions are satisfied and detailed in the next section). In thread side, the buffer (v) module sources the string in the same order they are stored. Timing computation (vi) is almost identical with the sequential version. Differing only by the division cycle counting due unavailability of operands, required to calculate early termination algorithm. Hash table (iv) was not modified. Parser module (vii) process the string arising from the buffer, supplying sub-string to module (iv) and (vi).

5.2 Buffer Management

The proposed extension requires a buffer as provisory trace storage as means to separate timing estimation in two phases. In order to preserve scalability, focusing many-core systems, is impossible maintain a static buffer as previous discussed. We propose take benefit of compartmentalized OVPsim nature (see section 3.1), using time-slice switching as size referential and as trigger event. A buffer receives instructions arriving from one processor until the OVP scheduler switch its model, per default at each 100K instructions. At this moment, an event creates a thread and passes buffer pointer as an argument. Therefore, through one model is simulated other threads can profit from host processor parallelism.

The time-slice event is triggered even when the platform simulates only one CPU. Nevertheless, this event is configurable, enabling insert an index trigger, besides time-slice transitions is possible provoke thread creation during the time-slice of a processor. For instance, if the CPU is configured with a 10K instructions index in a 100K time-slice, will be generated 10-work threads during each simulate time-slice for this CPU with the processing beginning immediately after thread creation. As a result, workload is separate in more threads although several tests not demonstrate a measurable gain.

Buffer allocation occurs when a processor try store an instruction and discover a null pointer in the buffer handler, and the size is by default a time-slice (100K). Additionally, buffers are independent of others, i.e. a processor in a determined instant may have several buffers been consumed, however, just one buffer is receiving data at any moment. Notwithstanding the number of simultaneous buffers is unpredictable due thread scheduling, experimental data show not more than four buffer existing simultaneously. As a result of host parallelism, for example, in a quad-core host OVPSim use only one leaving the other three to work threads. This behavior is due the duration of timing estimation of a chunk of instructions, been faster than simulate these instructions in OVPsim with the concomitant phase attached due the necessity of simulation interrupt at every fetch.

5.3 Thread Management

This extension also requires thread management, and in order to benefit better from host parallelism threads are independent of simulation main flow, i.e. there will be not a join calls after. The use of detached threads improves performance, while decreases the memory usage, since resources are immediately released as soon as a thread execution is completed.

When simulations come to end is not guarantee to all worker threads already finished, as result is necessary synchronize all threads. The synchronization is performed through a barrier like mechanism insert at simulation main flow end. As previous mentioned, threads are detached; consequently, thread join function is not an option and is not truly imperative since is necessary wait all threads finishes, not a specific one.

We use a global thread count variable, responsible by control the active threads number. Whenever a thread is created, the counter is incremented. Just after a work thread updates internal data, counter is decremented. The barrier is developed as a loop statement until thread counters reach zero, to avoid busy waiting and release resources a sleep call is realized inside the loop.

5.4 Simulation behavior

This extension also requires thread management, and in order to benefit better from host parallelism threads are independent of simulation main flow, i.e. there will be not a join calls after. The use of detached threads improves performance, while decreases the memory usage, since resources are immediately released as soon as a thread execution is completed.

When simulations come to end is not guarantee to all worker threads already finished, as result is necessary synchronize all threads. The synchronization is performed through a barrier like mechanism insert at simulation main flow end. As previous mentioned, threads are detached; consequently, thread join function is not an option and is not truly imperative since is necessary wait all threads finishes, not a specific one.

We use a global thread count variable, responsible by control the active threads number. Whenever a thread is created, the counter is incremented. Just after a work thread updates internal data, counter is decremented. The barrier is developed as a loop statement until thread counters reach zero, to avoid busy waiting and release resources a sleep call is realized inside the loop.

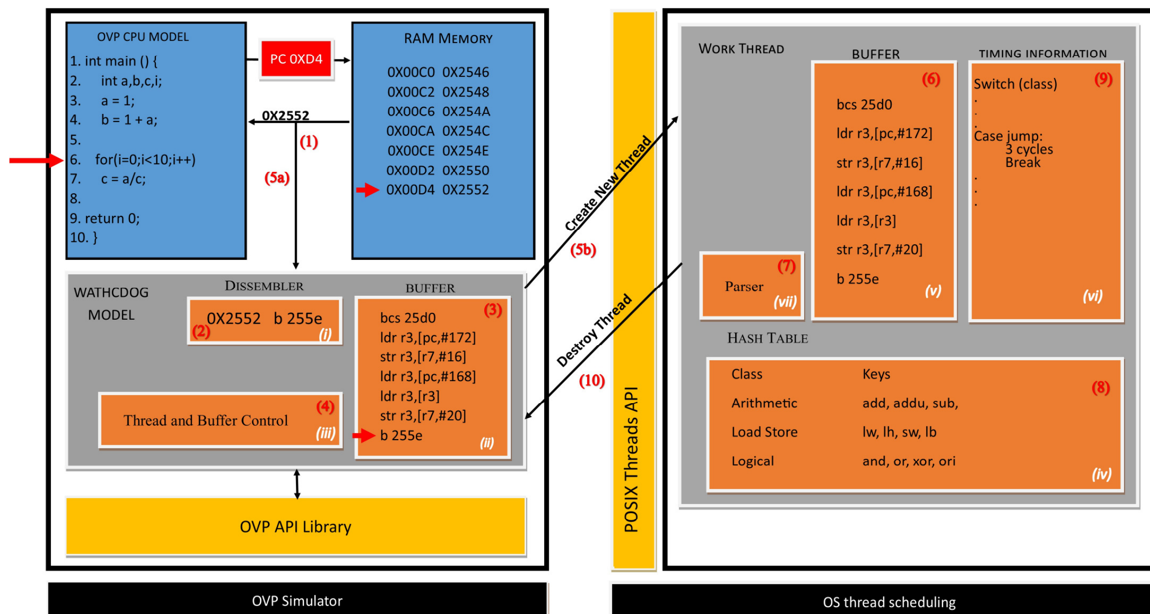


Figure 5.1 - Block diagram of developed watchdog module thread extension.

6 EXPERIMENTAL SETUP AND RESULTS

6.1 Timing CPU model results

In order to demonstrate the effectiveness of our approach, a 32-bit ARM Cortex-M4F processor, which is based on the ARMv7M architecture, was used as a case study. In the study case are used benchmarks from different domains, demonstrating the benefits towards the software evaluation facilities inherent to the proposed approach.

For our experiments, the STM32F4-Discovery board was used as reference platform, as illustrated in Figure 6.1. The reference STM32F4 Discovery board is built around a 32-bit ARM Cortex-M4F core running a FreeRTOS kernel version V.7.4.21 at 168 MHz. Among other features, ARM Cortex-M4F supports single precision floating-point unit (FPU) and power saving modes, which can be used for the development of energy-efficient embedded systems. Both Cortex-M4F and FreeRTOS are highly used in high-performance embedded system design, justifying the choice.

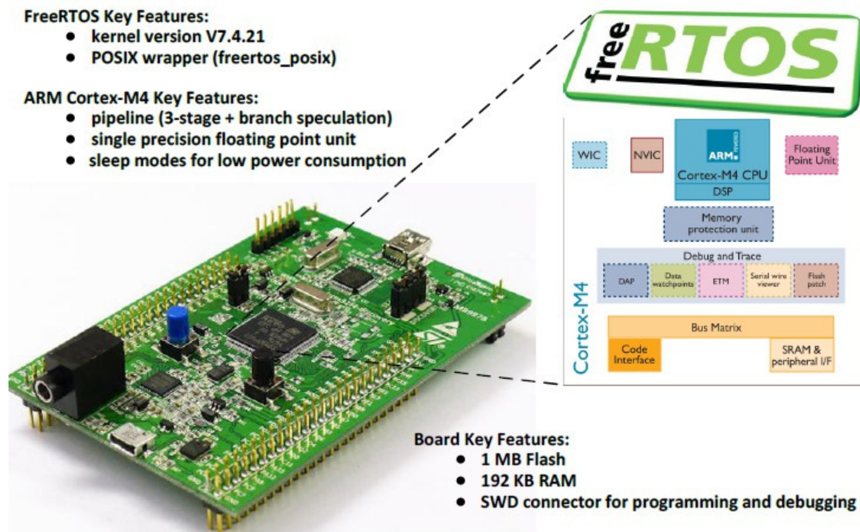


Figure 6.1 - Adopted reference board platform. Proposed illustration integrates figures captured from their owner's websites.

As means to verify model accuracy is fundamental expose the watchdog to real scenarios, diversifying possible instructions patterns encountered during simulation. Benchmark execution is composed by three phases. First, port chosen benchmarks to our framework, real board and OVP. Committed to ensuring both platform runs most similar code as possible, although is not feasible execute exactly the same binary duo platform related initialization. However, this discrepancy is less than 500 instructions. Assuring a compilation using same cross-compile, libraries, compilation flags is possible create almost identical binaries.

Using the Mentor Graphics Sourcery Tools version 4.8.1 and flags -mcpu=cortex-m4; -mfpu=fpv4-sp-d16; -mfloat-abi=softfp; -mthumb; -Wall; -ffunction-sections; -g; -O0; -w; -lm; -DSTM32F407VG; -DSTM32F4XX. Second, execute in the adopted board to acquire cycles count, use as referential. Finally, run all applications in the Timing CPU mode and variants. Realizing both execution, in the board and OVP several times.

To provide relevant metrics, selecting application benchmarks that permit exploiting and assessing performance of embedded CPUs from different research domains. Such diversity allows observing the accuracy of the proposed approach under different conditions. Applications from several suites were used, such as MiBench

[Guthaus et al. 2001], Mälardalen WCET [Jan Gustafsson 2010], SPLASH-2 [Woo et al. 1995], and other benchmarks created in house. Table 6.1 presents, the benchmarks number, the benchmarks name, origin, and a brief description.

Among the benchmarks, some are not appropriate to execution in this microarchitecture. Factors as memory usage and execution time may require code modifications, for instance, matrix sizes are dimensioned according available memory. Other had loop smoothed or remove not crucial functions as debug to diminish execution time.

6.1.1 Timing CPU model - Experimental Setup

As means to capture execution time (i.e. cycles) directed in the reference board, was necessary create an infrastructure. In order to development, compile, and test the applications in the host machine and transfer the binary code generated to the chip was employed the CooCox CoIDE Version 1.7.6. A free software development environment for ARM Cortex MCU based microcontrollers. Adopt the CoIDE instead of more popular platforms, as instance Keil uVision5, as driven by CoIDE liberty to selected and configure a third-party compiler. As a result, the same cross-compiler was used in the OVP platform as in the board platform. Ensuring maximum fidelity between OVP and real board binary code.

Subsequently, acquire the number of cycles for each application. The ARMv7 provides an internal logic that provides information about execution time called Data Watchpoint and Trace (DWT) Unit. Among them are three registers of special interest, the Control Register DWT_CTRL located at the address 0xE0001000, the Debug Exception and Monitor Control Register SCB_DEMCR at the address 0xE000EDFC, and the Cycle Count Register DWT_CYCCNT located at the address 0xE0001004. With this information, it is possible to manage and acquire the number of cycles of the application using the three registers. Figure 6.2 shows a sample code to acquire the clock count.

```
1  volatile unsigned int *DWT_CYCCNT   =(volatile unsigned int *)0xE0001004;
2  volatile unsigned int *DWT_CONTROL  =(volatile unsigned int *)0xE0001000;
3  volatile unsigned int *SCB_DEMCR    =(volatile unsigned int *)0xE000EDFC;
4
5  int counter;
6
7  int main(void)
8  {
9
10     *SCB_DEMCR = *SCB_DEMCR | 0x01000000;   //Trace system enable
11     *DWT_CONTROL = *DWT_CONTROL | 1 ;       // Enable the counter
12     *DWT_CYCCNT   =0;                       // reset the counter
13
14     //Some instructions
15
16     counter = *DWT_CYCCNT; //Acquire the result
17
18 }
19
```

Figure 6.2 - Sample code necessary to access the DWT registers.

Table 6.1 - List of used benchmarks to cycle estimation.

#	Name	Suite	Description
1	Adpcm	Mälardalen WCET	Adaptive pulse modulation algorithm.
2	Barnes	SPLASH 2	Performing an n-body simulation.
3	BasicMath	MiBench	Series of sums, divisions, and multiplications
4	Bfsh	House production	Blowfish is a symmetric-key block cipher
5	BinarySearch	Mälardalen WCET	Binary search
6	BitManipulation	Mälardalen WCET	(ndes) Bit manipulation, shifts, array, and matrix calculations.
7	Bubble	Mälardalen WCET	Bubble sort program
8	Compress	Mälardalen WCET	Data compression program (Adopted from SPEC95 for WCET)
9	Counts	Mälardalen WCET	Counts non-negative numbers in a matrix.
10	Crc	House production	Cyclic redundancy check computation
11	Dijkstra	MiBench	Dijkstra's algorithm
12	Edn	Mälardalen WCET	Integer Finite Impulse Response (FIR) filter calculations
13	Expint	Mälardalen WCET	Series expansion for computing an exponential integral function.
14	Factorial	House production	Factorial calculation
15	Fdct	Mälardalen WCET	Fast Discrete Cosine Transform
16	Fft	Mälardalen WCET	Fast Fourier Transform using the Cooly-Turkey algorithm.
17	Fib	House production	Fibonacci algorithm
18	Fir	Mälardalen WCET	Finite impulse response filter
19	Hanoi	House production	Tower of Hanoi solver
20	Harm	House production	Harmonics calculations
21	InsertSort	Mälardalen WCET	Insertion sort algorithm
22	Jfdctint	Mälardalen WCET	Discrete-cosine transformation on a 8x8 pixel block
23	Lms	Mälardalen WCET	LMS adaptive signal enhancement. The input signal is a sine wave with added white noise.
24	Lu	SPLASH 2	LU decomposition
25	MatrixInver	Mälardalen WCET	Inversion of floating point matrix
26	Mdc	House production	minimum common divisor
27	Patricia	MiBench	PATRICIA tree insert and search
28	Peakspeed	Impearas	Imperas development
29	Petri	Mälardalen WCET	(nsichneu) Simulate an extended Petri Net
30	Prime	Mälardalen WCET	Calculates whether numbers are prime.
31	Qsolver	Mälardalen WCET	Quadratic equation solver
32	Qsort	MiBench	Non-recursive version of quick sort algorithm.
33	Sha	MiBench	Secure Hash Algorithm
34	Statistic	Mälardalen WCET	Computes for two arrays of numbers the sum, the mean, the variance, and standard deviation, and the correlation coefficient between the two arrays.
35	Stringsearch	MiBench	Sub-string search
36	Sw	SPLASH 2	Smith-Waterman algorithm
37	Tree	House production	Binary tree insert and search
38	Ud	Mälardalen WCET	Integer Calculation of matrixes
39	Usqrt	Mälardalen WCET	Integer Square root function

6.1.2 Accuracy results and comparisons

After collect all data from the board and OVP models, Figure 6.3 shows a comparison between referential board time execution, timing estimation from the sequential version, and thread version. Left y-axis presents the number of cycles to each execution in a logarithmic scale. Background bars expose perceptual mismatch between each model variant and referential board in right y-axis. The x-axis show the number of each benchmark associated with Table 6.1.

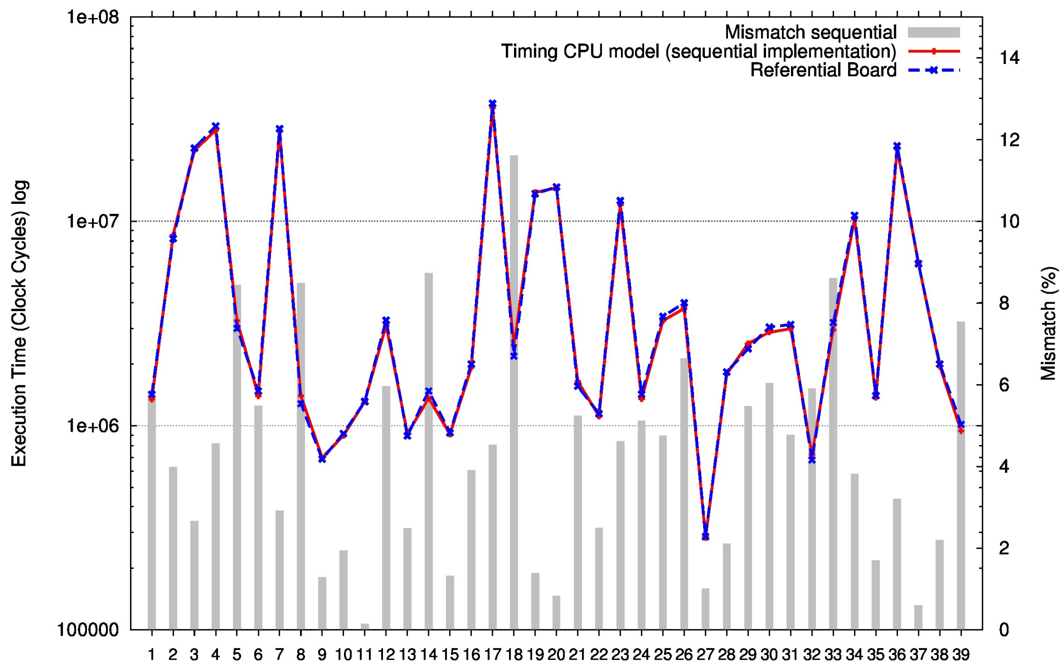


Figure 6.3 - Execution time comparison between real board and proposed timing model.

The major error contribution is due miss prediction over load/store execution. As mentioned in section 3.3, the timing behavior related to these instructions is complex and has several possibilities during hardware execution. In benchmarks as FIR heavily IO bounded that presents an intricate load/store patterns can insert incorrect cycles. For instance, our FIR implementation has 753.994 loads, 329.835 stores in a total of 1.624.604 representing almost 67% of executed instructions

Division instructions is another possible error source, as result of the variable number of cycles, 2 to 12, due the early termination algorithm implemented. The model covers the cases describe by the ARM Cortex M4 technical reference manual, although the exact algorithm is not provided, the document does supply some information into each instruction via a couple footnotes. Divide instructions use an early termination based on the number of leading zeros and ones in the input arguments the high accuracy achieved is an important contribution, hence flexible and accurate system modeling becomes imperative to the software development of today's MPSoCs.

To further investigate the accuracy of timing model, we evaluate the effect of increasing application execution time by having successive iterations of a loop. The chosen applications for such experiments were: (*i*) FFT (Fast Fourier Transform) and (*ii*) harmonic. The graphs in Figure 6.4 (a) and (b) shows that even changing the number of loops, the mismatch between the real board and the simulated OVP remains negligible (less than 0.6% in the worst case). Such small variations could be explained by the incidence of a given instruction that was not well characterized.

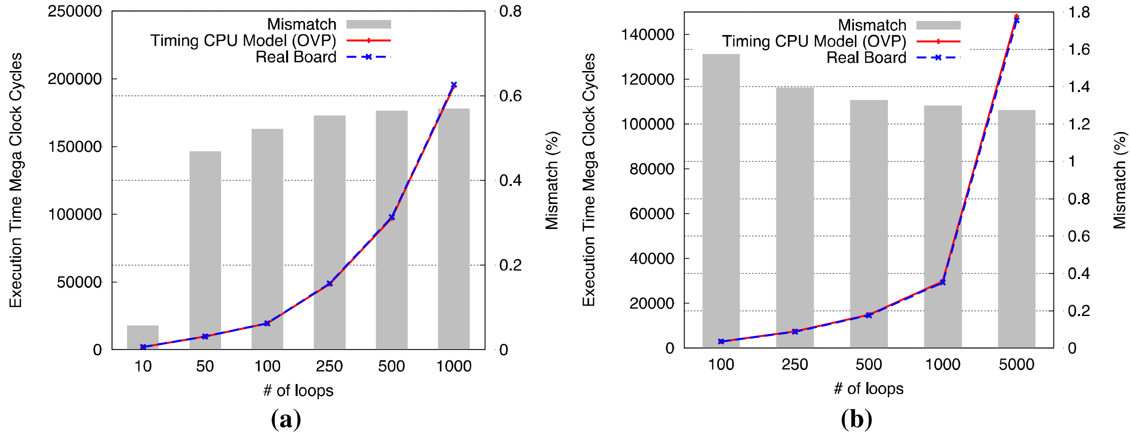


Figure 6.4 - Benchmark execution time comparison between real board and timing CPU model (OVP), varying the number of loops. Left a FFT (a) and in right a Harmonic (b) application.

6.1.3 Scalability

As demonstrated in section 3.1.1, OVPSim preserves the performance when exposed to scenarios composed of hundreds of cores. As the proposed model alters the traditional execution flow, the performance decrease is expected. As mentioned before, OVP relies on in Dynamic Binary Translation to active the claimed high speed by the direct relationship between host and target ISA. Subsequently, in the quasi-cycle accurate model is mandatory to stop the simulation at every fetched instruction. At this point in time, several instructions related to the Watchdog must be executed in the host machine.

The Watchdog model has two main variants, each one divided in four variants. First presented was the sequential, following the flow of OVPSim as describe in the section 3.1. Two other variants were evaluated, the sequential with memory count and the sequential with the full report. The memory count deploys callbacks to count the number of memory access, including fetched instructions and read and write transactions. When full report is enabled, every executed instruction is stored into an internal data struct, which requires more processing time during its execution.

The fourth variant is the union these two variants to create the sequential with the full report and memory count. In order to characterize the scalability of the parallel branch, chapter 5, was create three subtypes, Thread with the full report, Thread with memory count and the union of these two. The full report is performed parallel. As a result, and its impact is negligible when compares with a Thread version without the full report, totaling seven variants.

In order to demonstrate the scalability of all proposed model variants, the number of CPUs was varied from one to a thousand. In each CPU executes an instance of FFT. Noting the average MIPS remains constant to all models and variants, around 1.8 to threads extension, 1.2 to the sequential, and 1 to the sequential with the full report. Is notifiable also the great improving in thread extension when passing from 1 to 100 PE, when the individual PE becomes less active compared with total simulation time given more time to worker thread finishes. Proving the capacity of the model to simulate large many-core systems without loss performance.

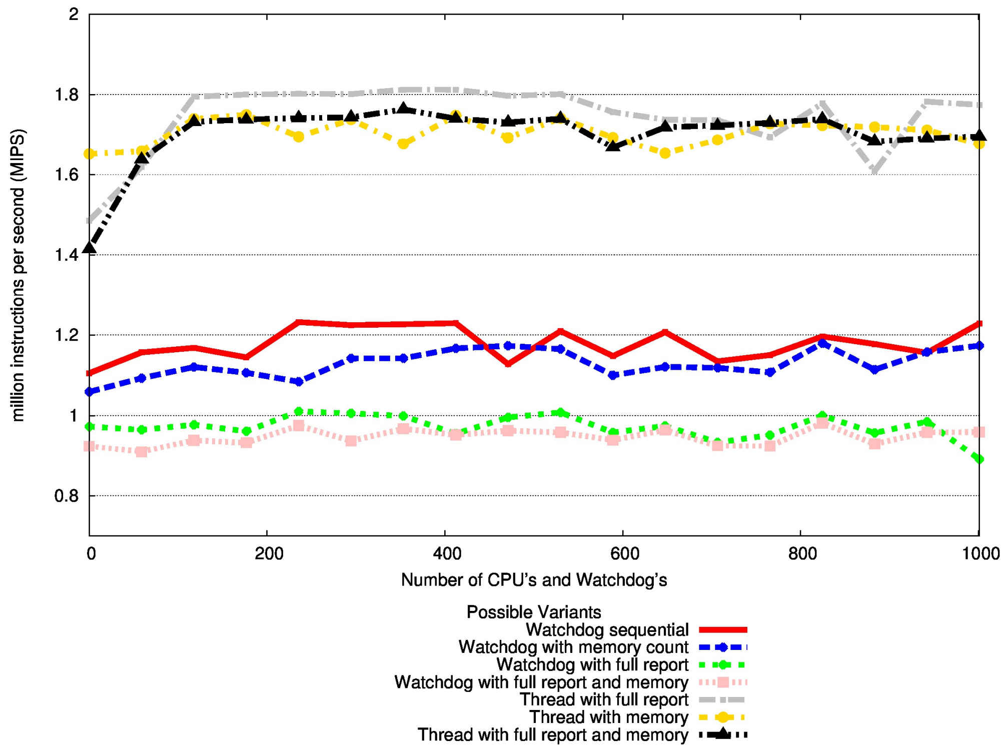


Figure 6.5 - Simulation performance as the simulated MPSoC scales from 1 to 1000 CPUs.

6.1.4 Thread extension accuracy

In order to verify the accuracy in this thread extension, presented in chapter 5, and if remains close to the sequential version described in chapter 3 all benchmarks were re-executed using this extension. Figure 6.6 shows the cycles from the referential board in red, in blue the estimation from the thread estimation, in yellow bars is visible the mismatch between the two.

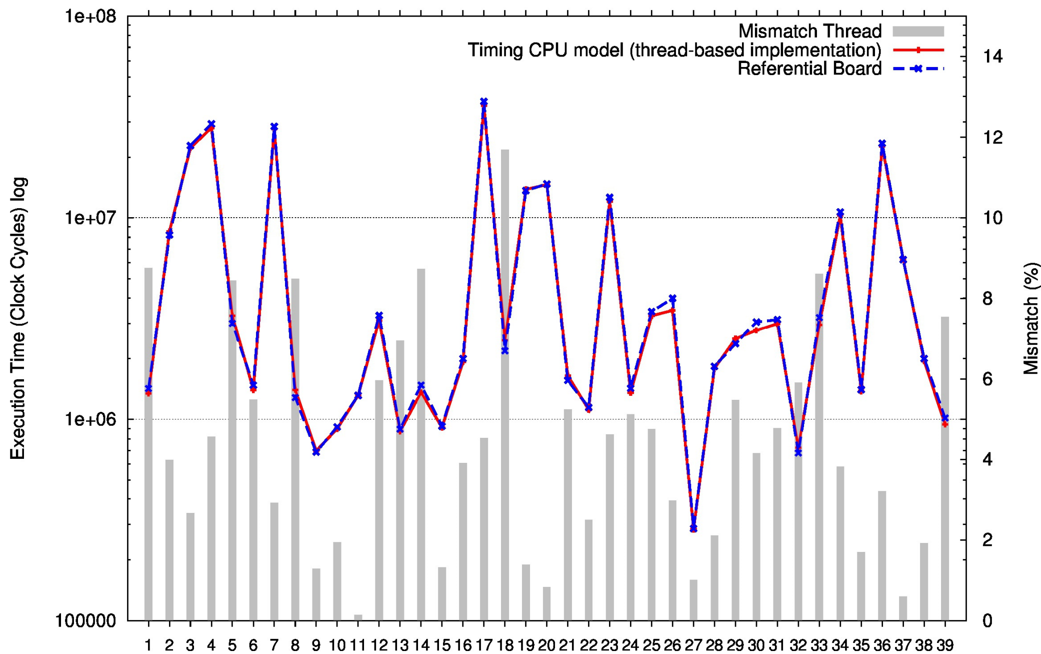


Figure 6.6 - Execution time comparison between real board and proposed thread extension.

Figure 6.7 shows the comparison between the timing model mismatch and the thread extension mismatch, remarking the majority of the 39 remains close to sequential estimation. The average error in this extension is 4.35 % while sequential version is approximately 4.31%.

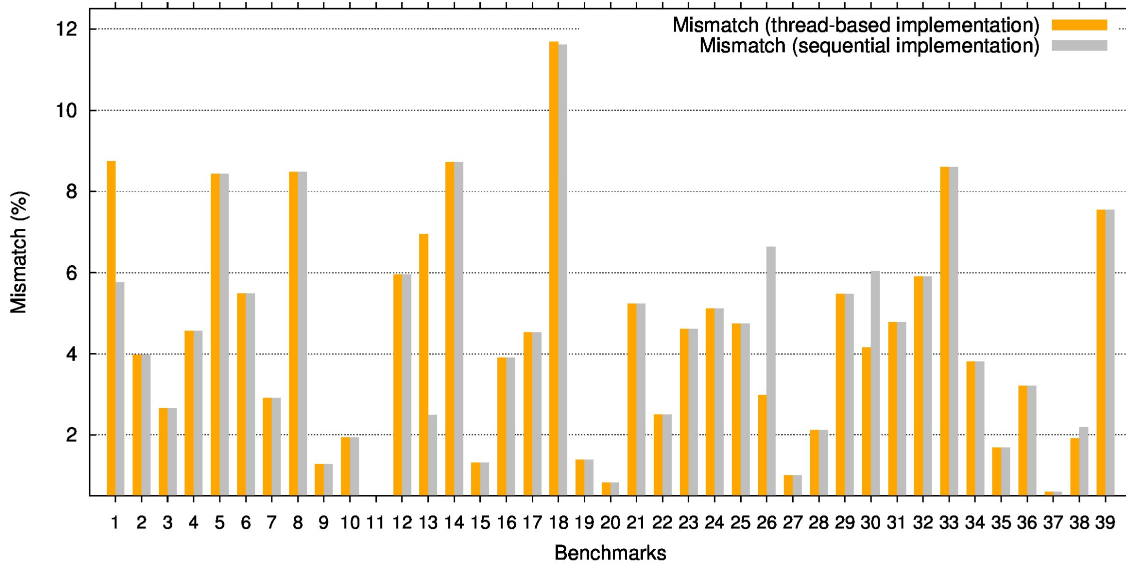


Figure 6.7 - Comparison between the timing model mismatch and the thread extension mismatch.

The mismatch between implementations occurs due the division cycle estimation, as already mentioned, section 3.3, the early termination algorithm deployed in ARM Cortex-M4F requires acquiring booth operands in order to estimate the cycle count. Nevertheless, when the thread-based version realizes the computation asynchronously, the operands may not be available in the register bank to comparison.

6.1.5 Thread extension speedup

Tread extension was intent to increase simulation speed by diving the work in several threads, as means to verify the gains obtained the individual speed to all benchmark was recorded. Figure 6.8 shows simulation speed in MIPS for each benchmark in booth main versions: Sequential in grey bars and thread-based extension in orange bars. Upon each benchmark is the speedup from one version to another. Noting an average speed of 1.5 MIPS presented by the thread extension and 1 MIPS in the sequential version.

The relative speed gain is due work migration from simulation flow to independent worker threads, minimizing the time spend in each fetch treatment. However, the speed gain was limited by the amount of work transferable to these threads, limitations imposed by the methods provide on the OVP to disassemble binary code, as seeing in section 3.4.1. These methods are not constructed to be thread safe, generating issues related to data races in multiple concurrent invocations.

Possible solutions were investigated across the model development, among than was suggested the creation of a custom disassemble function. However, this decision would imply in a large amount of work diverting from our main focus, additionally, the model is intent to be the most flexible as possible and easily ported to other architectures. Was considered the use of an extern disassemble program, for instance the provided by GCC compiler, despite the reasonably flexibility provide each disassemble operation would require an invocation of an extern program been extremely

slowly to our implementation purposes.

The most promisor solution encountered is embedded the proposed model directly in OVPSim engine. JIT-based simulators require a morphing phase to operate, in this phase the binary code is translated from the target machine to host machine binary, additionally during this phase the instructions are identified with enough information to implements our proposed OVP model. Also, the OVP morphing phase possesses a table containing the previous translations used to accelerate the simulation. However, in order to deploy this solution is necessary access to the privileged source code.

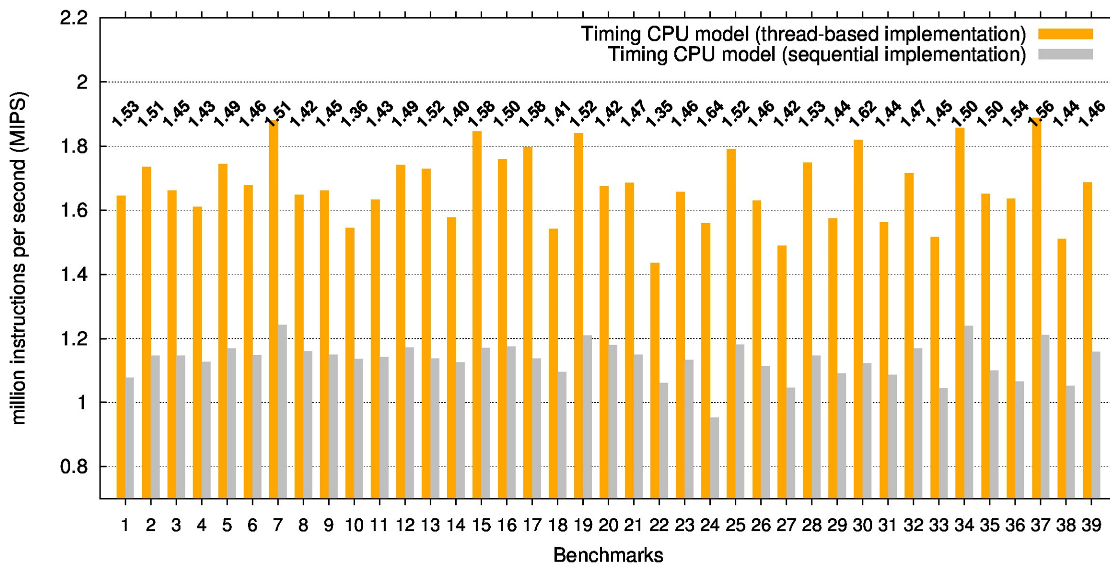


Figure 6.8 - Speedup comparison between timing model in sequential and thread extension versions.

6.2 ENERGY EXPERIMENTAL SETUP AND RESULTS

6.2.1 Test Planning

In order to demonstrate the effectiveness of the proposed approach, several benchmarks were selected and use to compare the accuracy of the model when compared with the Plasma RTL gate-level description. Application benchmarks that permit exploiting and assessing performance of embedded CPUs were selected from different research domains. For instance, the 11-selected applications of the Mälardalen WCET [Jan Gustafsson 2010] benchmarks vary in terms of execution time, number of loops, matrixes, and array size. The others are produced in-house using well-known algorithms. Since OVPSim uses the target CPU's binary code to perform the emulation on a host machine, all simulation scenarios were executed multiple times in order to capture meaningful results.

Executing each one in the OVPSim using the Watchdog energy model, and subsequently execute the benchmark in the Cadence Incisive and Cadence RTL compiler to acquire the energy spend, similar to made in characterization phase in section 4.1 and 4.2. Using the same metrics possible acquire the energy to each one of the 19 benchmarks displayed in Table 6.2.

Committed to ensure both platform runs most similar code as possible, although is not feasible execute exactly the same binary duo platform related initialization. However, this discrepancy is less than 500 instructions. Assuring a compilation using

same cross-compile, libraries, compilations flags is possible create almost identical binaries. Used compiler Mentor Graphics Sourcery Tools version 4.8.1 and flags -mips1 -g -Ttext 00000000.

Table 6.2 - List of used benchmarks in energy estimation.

#	Name	Suite	Description
A	Bfsh	House production	Blowfish is a symmetric-key block ciphe
B	BinarySearch	Mälardalen WCET	Binary search
C	BitManipulation	Mälardalen WCET	(ndes) Bit manipulation, shifts, array, and matrix calculations.
D	Bubble	Mälardalen WCET	Bubblesort program
E	Counts	Mälardalen WCET	Counts non-negative numbers in a matrix.
F	Crc	House production	Cyclic redundancy check computation
G	Edn	Mälardalen WCET	Integer Finite Impulse Response (FIR) filter calculations
H	Expint	Mälardalen WCET	Series expansion for computing an exponential integral function.
I	Factorial	House production	Factorial calculation
J	Fft	Mälardalen WCET	Fast Fourier Transform using the Cooly-Turkey algorithm.
K	Fib	House production	Fibonacci algorithm
L	Hanoi	House production	Tower of Hanoi solver
M	Harm	House production	Harmonics calculations
N	InsertSort	Mälardalen WCET	Insertion sort algorithm
O	MatrixInver	Mälardalen WCET	Inversion of floating point matrix
P	Mdc	House production	minimum common divisor
Q	Peakspeed	Impearas	Imperas development
R	Ud	Mälardalen WCET	Integer Calculation of matrixes
S	Usqrt	Mälardalen WCET	Integer Square root function

6.2.2 Accuracy results and comparisons

Figure 6.9 compares the energy consumption for each application benchmark, considering results obtained from gate-level simulation (i.e. Cadence RTL Compiler) and the proposed instruction-driven energy model in OVP. Gray bars correspond to the difference between each result, showing the high accuracy achieved with the proposed model (error below 8%).

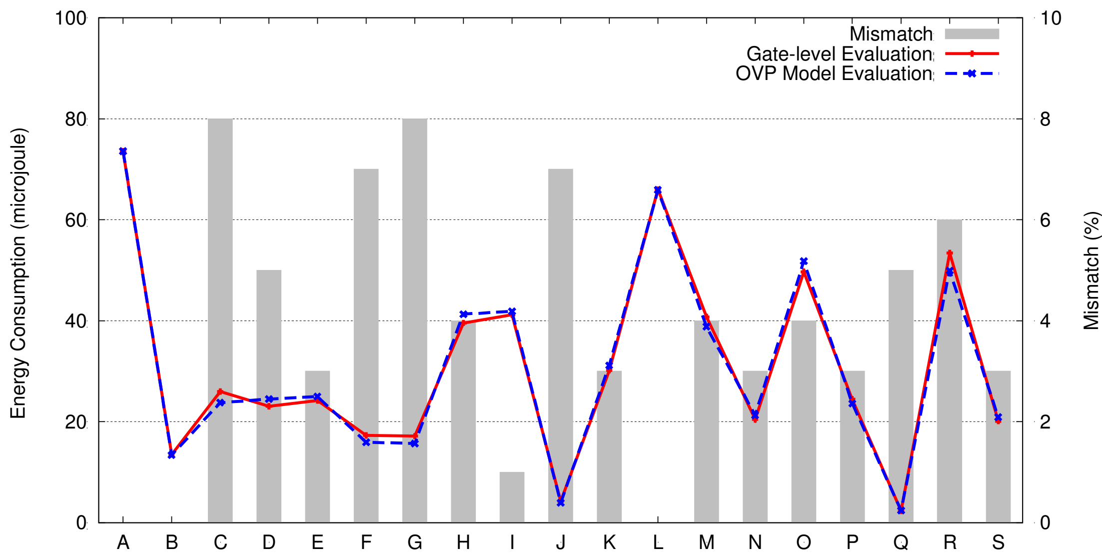


Figure 6.9 - Application benchmark energy consumption: gate-level simulation versus proposed instruction-driven energy model in OVP.

6.2.3 Relative speedup gain

Figure 6.10 presents the achieved simulation speeds in MIPS when comparing both the instruction-driven energy model in OVP and the gate-level simulation.

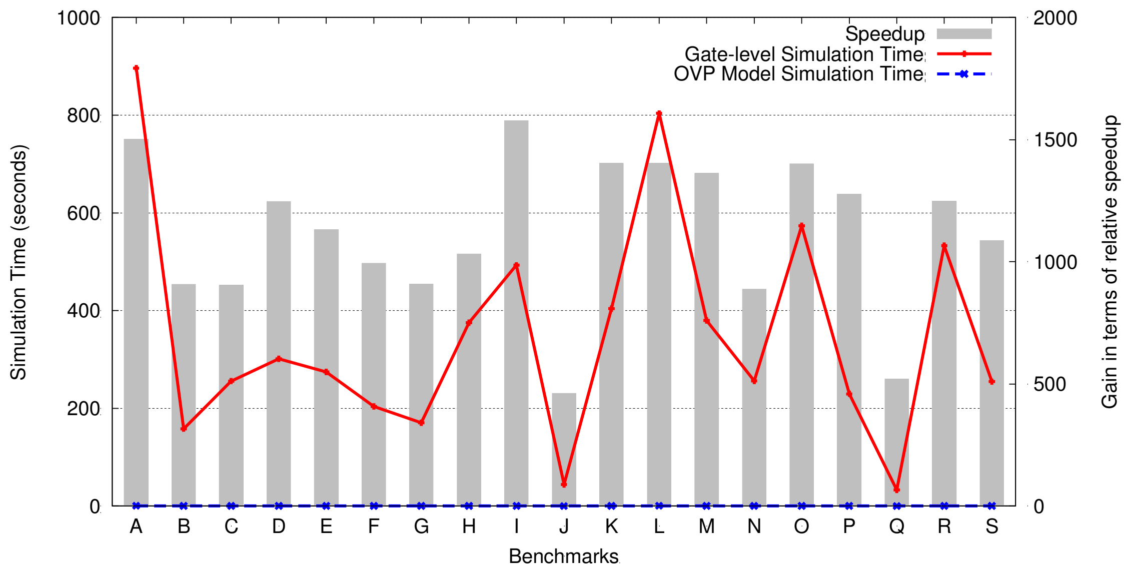


Figure 6.10 - Gain in terms of speedup: gate-level simulation versus proposed OVP energy model.

Results show that the gain in terms of speedup is in a wide range of 10x – 1500x (gray bars) depending on the application benchmark nature. Note that all analysis using our proposed energy model in OVP required less than a minute of simulation.

6.2.4 Application to Large Scale Systems

The proposed instruction-driven energy model was integrated into a NoC-based MPSoC model proposed in [Mandelli et al. 2013], in order to verify the its application to large scale systems. The underlying energy model was employed to evaluate the mapping process cost regarding different mapping heuristics by calculating the energy consumption during the execution of each of them. This experiment comprises exclusively the energy spend by the proposed mapping heuristics, providing to the

software engineer the ability to observe different algorithms to solve the same problem. Different scenarios were evaluated using the OVP model to compare three different heuristics: Nearest Neighbor (NN), first free (FF) and LECDN. The heuristics are distribute in scenarios using an 8x8 MPSoC size instance with 4x4 clusters. Each scenario executes 5 applications instances: 4 instances of a partial MPEG decoder (real application coded in C language), containing 5 tasks; and one instance of the Digital Time Warping (DTW, real application coded in C language), application, with 10 tasks.

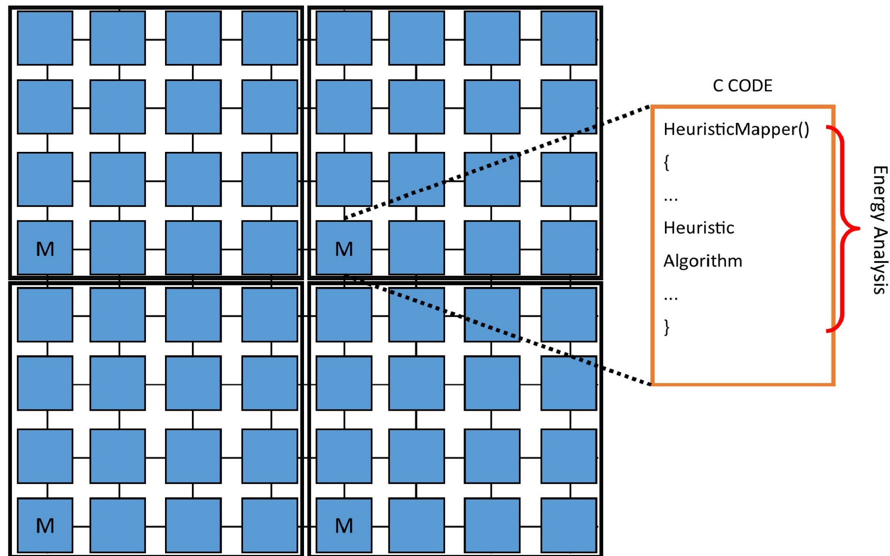


Figure 6.11 – Platform with four clusters used in the evaluation.

Nearest Neighbor (NN) heuristic was proposed by [De Souza Carvalho et al. 2010] considers only the proximity of an available resource to execute a given task. The LEC-DN heuristic [Mandelli et al. 2011] employs two cost functions: (*i*) proximity, in number of hops; (*ii*) communication volume among tasks. Differently from NN, which map the target task as close as possible to its source task, the LECDN considers the proximity of the target task to all tasks it communicates with already mapped. The last one, first free (FF) [Carvalho et al. 2009] selects the first free resource available.

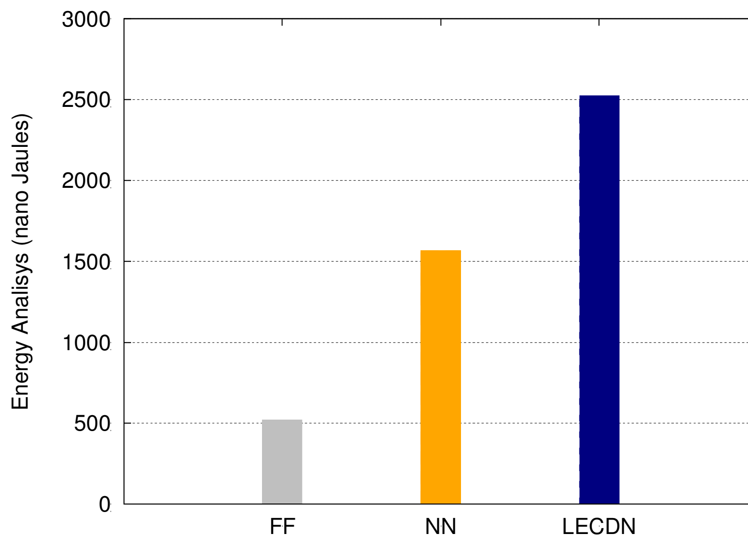


Figure 6.12 – Energy cost of the three mapping heuristics.

Results show that the LEC-DN is the heuristic with the highest energy consumption due to its complex algorithm. This increase (62.9% related to the NN) is considered by the Authors a small cost considering the benefits obtained by distributing the load in the system. The FF is the lowest energy consumption due his simplicity, however may generate the worst results when compared to the other heurist.

This experiment shows a potential application of the proposed energy model. We claim here, that this energy model can be used by software engineers to quickly modify and analyze, early at the design process, different software stack configurations, aiming to satisfy particular requirements of energy for specific functions (e.g. mapping heuristics), or even for a great variety of applications.

7 CONCLUSION

In this work, we have begun to address the challenge of making JIT-based simulators suitable for software performance estimation. In light of this, a watchdog model, incorporating timing and energy analysis were proposed and integrated into OVPSim simulator. The resulting approach offers the same design flexibility, setup and debugging features inherent to OVP, while enabling accurate timing and energy software evaluation. Thus, programmers can use the same simulator to have fast simulation and accurate software evaluation. As result, software engineers are able to estimate execution time and energy in early stages of development, improving the commitment with project constraints and reducing time to market. Additionally, the model is adequate to manage 1000-cores scenarios maintaining the scalability both to simulation speed and memory usage.

REFERENCES

Abril Garcia, A. B., Gobert, J., Dombek, T., Mehrez, H. and Petrot, F. (2002). Cycle-accurate energy estimation in system level descriptions of embedded systems. In *9th International Conference on Electronics, Circuits and Systems, 2002*.

ARM (2013). ARM® Cortex-M4 Processor Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0439d/index.html, [accessed on Apr 21].

Austin, T., Larson, E. and Ernst, D. (feb 2002). SimpleScalar: an infrastructure for computer system modeling. *Computer*, v. 35, n. 2, p. 59–67.

Bazzaz, M., Salehi, M. and Ejlali, A. (jul 2013). An Accurate Instruction-Level Energy Estimation Model and Tool for Embedded Systems. *IEEE Transactions on Instrumentation and Measurement*, v. 62, n. 7, p. 1927–1934.

Bellard, F. (2005). QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. , ATEC '05. USENIX Association. http://dl.acm.org/citation.cfm?id=1247360.1247401, [accessed on Jul 3].

Binkert, N., Beckmann, B., Black, G., et al. (aug 2011). The Gem5 Simulator. *SIGARCH Comput. Archit. News*, v. 39, n. 2, p. 1–7.

Bohm, I., Franke, B. and Topham, N. (jul 2010). Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *2010 International Conference on Embedded Computer Systems (SAMOS)*.

Borkar, S. (2007). Thousand Core Chips: A Technology Perspective. In *Proceedings of the 44th Annual Design Automation Conference*. , DAC '07. ACM. http://doi.acm.org/10.1145/1278480.1278667, [accessed on Jun 19].

Borkar, S. and Chien, A. A. (1 may 2011). The future of microprocessors. *Communications of the ACM*, v. 54, n. 5, p. 67.

Bose, P. (feb 2013). Is Dark Silicon Real?: Technical Perspective. *Commun. ACM*, v. 56, n. 2, p. 92–92.

Butko, A., Garibotti, R., Ost, L. and Sassatelli, G. (jul 2012). Accuracy evaluation of GEM5 simulator system. In *2012 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*.

Callou, G., Maciel, P., Tavares, E., et al. (jun 2011). Energy Consumption and Execution Time Estimation of Embedded System Applications. *Microprocess. Microsyst.*, v. 35, n. 4, p. 426–440.

Carvalho, E., Calazans, N. and Moraes, F. (oct 2009). Investigating runtime task mapping for NoC-based multiprocessor SoCs. In *2009 17th IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*.

Castillo, J., Posadas, H., Villar, E. and Martínez, M. (nov 2007). Energy Consumption Estimation Technique in Embedded Processors with Stable Power Consumption based on Source-Code Operator Energy Figures.

Ceng, J., Sheng, W., Castrillon, J., et al. (2009). A High-level Virtual Platform for Early MPSoC Software Development. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. , CODES+ISSS '09. ACM. http://doi.acm.org/10.1145/1629435.1629438, [accessed on Jun 20].

Chiang, M.-C., Yeh, T.-C. and Tseng, G.-F. (apr 2011). A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 30, n. 4, p. 593–606.

Davidmann, S. and Graham, D. (2014). Learning From Advanced Hardware Verification for Hardware Dependent Software.

De Souza Carvalho, E. L., Calazans, N. L. V. and Moraes, F. G. (sep 2010). Dynamic Task Mapping for MPSoCs. *IEEE Design Test of Computers*, v. 27, n. 5, p. 26–35.

Esmaeilzadeh, H., Blem, E., St.Amant, R., Sankaralingam, K. and Burger, D. (jun 2011). Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*.

Gray, I. and Audsley, N. C. (oct 2012). Challenges in software development for multicore System-on-Chip development. In *2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP)*.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., et al. (dec 2001). MiBench: A free, commercially representative embedded benchmark suite. In *2001 IEEE International Workshop on Workload Characterization, 2001. WWC-4*.

IBS (2013). International Business Strategies, Inc.2013.

Imperas (2014). Open Virtual Platforms (OVP). http://www.ovpworld.org/, [accessed on Apr 21].

Jan Gustafsson, A. B. (2010). The Mälardalen WCET Benchmarks: Past, Present And Future. p. 136–146.

Kalla, P., Henkel, J. and Hu, X. S. (oct 2003). SEA: fast power estimation for micro-architectures. In *5th International Conference on ASIC, 2003. Proceedings*.

Konstantakos, V., Chatzigeorgiou, A., Nikolaidis, S. and Laopoulos, T. (apr 2008). Energy Consumption Estimation in Embedded Systems. *IEEE Transactions on Instrumentation and Measurement*, v. 57, n. 4, p. 797–804.

Lee, D., Ishihara, T., Muroyama, M., Yasuura, H. and Fallah, F. (oct 2006). An Energy Characterization Framework for Software-Based Embedded Systems. In *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*.

Lee, S., Ermedahl, A., Min, S. L. and Chang, N. (2001). An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*. , LCTES '01. ACM. http://doi.acm.org/10.1145/384197.384201, [accessed on Mar 31].

Magnusson, P. S., Christensson, M., Eskilson, J., et al. (feb 2002). Simics: A Full System Simulation Platform. *Computer*, v. 35, n. 2, p. 50–58.

Mandelli, M. G., Da Rosa, F. R., Ost, L., Sassatelli, G. and Moraes, F. G. (dec 2013). Multi-level MPSoC modeling for reducing software development cycle. In *2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS).*

Mandelli, M., Ost, L., Carara, E., et al. (may 2011). Energy-aware dynamic task mapping for NoC-based MPSoCs. In *2011 IEEE International Symposium on Circuits and Systems (ISCAS).*

Marongiu, A. and Benini, L. (feb 2012). An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs. *IEEE Transactions on Computers*, v. 61, n. 2, p. 222–236.

Miura, N., Koizumi, Y., Take, Y., et al. (nov 2013). A Scalable 3D Heterogeneous Multicore with an Inductive ThruChip Interface. *IEEE Micro*, v. 33, n. 6, p. 6–15.

MPPA (2014). MPPA MANYCORE: a multicore processors family - Many-core processors - KALRAY. http://www.kalray.eu/products/mppa-manycore/, [accessed on Jun 25].

Nikolaidis, S., Kavvadias, N., Laopoulos, T., Bisdounis, L. and Blionas, S. (2003). Instruction Level Energy Modeling for Pipelined Processors. In: Chico, J. J.; Macii, E.[Eds.]. *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*. Lecture Notes in Computer Science. Springer Berlin Heidelberg. p. 279–288.

Papanikolaou, A., Wang, H., Miranda, M., Catthoor, F. and Dehaene, W. (2008). Reliability Issues in Deep Deep Submicron Technologies: Time-Dependent Variability and its Impact on Embedded System Design. In: Micheli, G. D.; Mir, S.; Reis, R.[Eds.]. *VLSI-SoC: Research Trends in VLSI and Systems on Chip*. IFIP International Federation for Information Processing. Springer US. p. 119–141.

Plasma (2014). Plasma CPU. http://plasmacpu.no-ip.org/, [accessed on Apr 21].

Sanchez, D. and Kozyrakis, C. (2013). ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. , ISCA '13. ACM. http://doi.acm.org/10.1145/2485922.2485963, [accessed on Apr 20].

Stattelmann, S., Ottlik, S., Viehl, A., Bringmann, O. and Rosenstiel, W. (jun 2012). Combining instruction set simulation and WCET analysis for embedded software performance estimation. In *2012 7th IEEE International Symposium on Industrial Embedded Systems (SIES).*

STM32F4 (2014). STM32F4DISCOVERY Discovery-STMicroelectronics. http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF252419, [accessed on May 18].

Sultan, S. and Masud, S. (jul 2009). Rapid software power estimation of embedded pipelined processor through instruction level power model. In *International Symposium on Performance Evaluation of Computer Telecommunication Systems, 2009. SPECTS 2009*.

Thach, D., Tamiya, Y., Kuwamura, S. and Ike, A. (mar 2012). Fast cycle estimation methodology for instruction-level emulator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P. and Gupta, A. (jun 1995). The SPLASH-2 programs: characterization and methodological considerations. In *, 22nd Annual International Symposium on Computer Architecture, 1995. Proceedings*.

Yourst, M. T. (apr 2007). PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *IEEE International Symposium on Performance Analysis of Systems Software, 2007. ISPASS 2007*.

Zhang, Y., Peng, L., Fu, X. and Hu, Y. (may 2013). Lighting the dark silicon by exploiting heterogeneity on future processors. In *2013 50th ACM / EDAC / IEEE Design Automation Conference (DAC)*.

APPENDIX A- TRABALHO DE CONCLUSÃO I

Fast and Accurate Evaluation of Embedded Applications for Many-core Systems

Felipe Rosa [1], Ricardo Reis[1], Luciano Ost[2]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

[2]Lirmm– University of Montpellier 2
Montpellier, Fr

{frdarosa,reis}@inf.ufrgs.br, ost@lirmm.fr

Abstract. *Many-core systems will be the embedded projects future, however this hardware-software architectures as the only viable solution if addressing designs constraints on cost, performance and power. With the purpose of provide these necessary tools, this work propose a rapid timing and power estimation model for many-core system. Results show that the accuracy of our timing model varies from 0,6% to 10,5% and for power 0,01% to 7,6% depending on the benchmark profile. Still reaching simulation average of 1.8 MIPS*

Resumo. *Sistema multiprocessados serão o futuro dos projetos de sistemas embarcados, no entretanto estas arquiteturas de hardware-software são viáveis soluções se atenderem as restrições de design em custo, desempenho e potência. Com o propósito de prover este ferramental necessário, este trabalho propõem um modelo de potência e temporal para sistemas multiprocessados. Os resultados mostram um precisão para o modelo temporal entre 0,6% e 10,5% e para o modelo de potência entre 0,01% e 7,6% dependendo do perfil do benchmark. Mantendo uma velocidade de simulação de 1.8 MIPS.*

1. Introduction

The many-core embedded systems will become the dominant hardware-software architecture. This is the most viable solution to addressing future designs constraints on cost, performance and power. The move to many-core paradigm as result of end of the single-core performance scaling principally pushed over the past couple of decades by increasing instruction-level parallelism (ILP) using several stratagems as: super pipeline, Out-of-Order architect, Speculative execution. However, simply leveling out ILP has little effect on most applications leading the performance versus silicon area cost unaffordable. Resulting in multicore systems shift.

This paradigm shift will increasing dramatically software design complexity is, resulting in new design challenges, such as improving the system's performance and programmability, these challenges impose more time and cost on the system's software development. In this context, software engineers are investigating alternatives to scale

up the system performance, while dealing with new challenges in many-core software development. Leading to the adopting of virtual platform frameworks aimed at functional verification like [“Open Virtual Platforms (OVP)” [S.d.]], capable of simulating embedded systems running real application code at the speed of hundreds of MIPS. Such simulators can achieve speeds approaching actual execution time, e.g. thousands MIPS at the cost of limited accuracy. They often focus on functional validation rather than architectural exploration. In light of this, the present work focuses on enhancing OVP capability by including power and timing models, making it suitable for software performance and power analysis.

The presented work is divided in 7 sections. In section 2 a state of the art in power and timing estimation. In 3, we introduce is discussed simulators performance, in more detail Open Virtual Platforms. Section 4 presents the timing estimation model and related results, as in 5 the power estimation model. Finishing with conclusion in the section 6 and referees in 7.

2. Related Work

In this section presents a state-of-art. Power and timing estimation is a broad field of exploration, in this work we will focus in models extensions of rapid frameworks (e.g. ISS and more higher levels of abstractions) in order to improve accuracy estimation. Almost no works, in this scope, cover both areas. As result, power and timing are separated in section 2.1 and 2.2 respectively.

2.1 Cycles.

Chiang et al. [Chiang et al. 2011] utilize the integration of QEMU and SystemC in order to allow faster clock-accurate evaluation when compared to RTL-based. Attaching a SystemC co-processor in the simulator framework, using the information coming from the DBT interface. A pipeline model was included into QEMU in [Thach et al. 2012], where authors proposed a two-phase approach (offline and online phases) to estimate the application performance. In the offline phase a cycle pre-estimation of the application execution time is performed. Using the computed information at dynamic adaption phase when CPU status and execution time of critical instructions are also taken in account, improving the approach accuracy (mismatch around 10%). A similar approach is presented in [Stattelmann et al. 2012], where worst-case execution time (WCET) analysis and QMEU are combined. In this work, the offline phase is composed by four steps, which produce a timing database that is used during the QEMU simulation. The drawback of such approaches is that they rely based on prior application profiling phases, which restricts its use when exploring large scenarios composed of diverse applications. Another disadvantage of them is that any software modification (e.g. changing the OS scheduling algorithm) implies re-running offline phases.

2.2 Power

Over the years, considerable number of different approaches have been proposed to create high-level power models aiming help software and systems engineers better explorer larger design spaces. They can be separated in two main groups taking into consideration their initial calibration low-level abstraction: Measurement-based methods and Simulation-based methods.

Measurement-based methods use data originated from field experiments usually using a precision resistor positioned between the power supply and the power input pin, thus instantaneous power is calculated. For example, [Bazzaz et al. 2013; Konstantakos et al. 2008; Sheayun Lee et al. 2001; Nikolaidis et al. 2003] applies this approach. The use of physical information aggregate precision, in order of 3%, however is needed additional hardware as high performance oscilloscopes and associated several different benchmarks to calibrate each instruction. Furthermore, another drawback of this technique is the difficulty to isolate separated modules inside the processor package duo the internal structure (e.g. Flash, Rom, SPI, AD, and DC).

In simulated-based techniques, the required information is extracted from low-level simulators (e.g. SPICE), using a hardware model to run the applications and calculate the energy consumption of each part of the system. In [Abril Garcia et al. 2002] an ISS as enriched with energy models based in the mean active in gate level and the energy by gate. In a similar work, [Sultan and Masud 2009] implements a model based in activity for a LEON3 processor.

[Kalla et al. 2003] create a tool called SEA in order to provide estimation of power and energy in a SPARC processor. This work uses a gate-level simulation to provide energy information to their model. The instructions were classified in memory, not-memory and specials, with a precision that is inside 5% for energy and inside 15% for power.

[Lee et al. 2001] combines linear programming, gate-level simulation and ISS to extract several parameters. However, instead of profiling separated instructions, the instructions were clustered in significant frames of fixed number of instructions.

Some approaches the level of abstraction are more behavioral like. For instance, by [Castillo et al. 2007] propose obtaining power estimations directly from the analysis of the source-code without requiring simulation or even compilation. A further higher-level approach is propose by [Callou et al. 2011] transforming the source code in Coloured Petri nets and combining with stochastic analysis to estimate an application power consumption. This couple of work presents a new level of abstraction that do not require a real hardware simulation, working only by the source code analyses.

3. Simulators

Test and debuggability becomes critical in embedded software development as designs complexity increase through heterogeneous multiprocessor System-on-Chip (MPSoC), resulting in the necessity of simulation before real implementation. Designers have the liberty to choose an abstraction level to simulate, managing performance and accuracy tradeoffs. Among then Circuit-level and Logic-level simulators are extremely time-consuming, demanding adoption of a fast platform simulator to suit time-to-market and cost constraints.

Event-driven and quasi-cycle accurate virtual platform frameworks like GEM5 target microarchitecture exploration since provides specific modeling details (e.g. instruction pipeline details, cache coherence protocols, etc.) [Binkert et al. 2011 p. 5]. Such simulators are not scalable to a large number of CPUs, specifically when it comes to usability, easy-of-modeling and simulation time (around 200 KIPS [Sanchez and Kozyrakis 2013]).

In contrast, simulators such as OVP that relies on just-in-time (JIT) dynamic binary translation (DBT) can achieve simulation speeds of up to 100 MIPS. DTB depend on emulation of one instruction set architecture (ISA) in another, through machine code online translation, enhancing performance in comparison to other simulation schemes. This performance gain comes at the expense of accuracy.

Due the limited simulation speed of event-driven cycle-accurate frameworks, simulators based on DBT become decisive to deal with today's application challenges, as well as to enable large scenarios evaluation. Simics ["Full System Simulation with Wind River Simics" [S.d.]], QEMU ["QEMU" [S.d.]] and the adopted OVPSim are examples of virtual platform frameworks that rely on DBT. Such simulators/emulators vary in modeling flexibility, simulation speed and accuracy.

3.1 Open Virtual platforms

OVP supports dozens of architectures (e.g. MIPS, ARM, x86, PowerPC) ramifying in several model variants (e.g. arm cortex-A9, cortex-M4F, etc.), as well peripherals (e.g. DMA, TIMER), and integration with System-C modules. Besides, of supplied models, the user is able to create customized models easily integrated in the platform.

OVP is composed of three main components: (*i*) APIs that enable modeling in C/C++ hardware components, (*ii*) library with a large number of CPU architectures and peripheral models, and (*iii*) the OVPsim simulator. OVPsim is a dynamic linked library marketed by Imperas, which supports the simulation of bus-based multiprocessor platforms only. OVPsim does provide instruction-level accuracy only resulting into inaccurate software performance.

In order to deploy this Multiprocessor capability, OVPsim implements a Round-Robin scheduling algorithm similar to a typical used in OS schedulers. The processor entity (PE) has time slice variable, typically 0.001 seconds, commonly equal shares are set to all PE. Converting this variable into a number of instructions that should be executed by that processor in a time slice, and then simulating for that number of instructions. The number is obtain multiplying the time slice by the processor nominal MIPS, 100 per default. OVP works in sequential way (i.e. simulating a unique processor at time, even if is hosted in a multi-core). Nevertheless, this algorithm inserts an issue related to the

synchronization between simultaneous events in different PE. For instance, if a PE sends a message to other in the middle of their time slice, the receiver PE only will be aware of the message at the begin of his time slice. In simulation tightly based in intercommunications between processor (e.g. HPC, NOC), the precision of results may be affected.

A possible addressable solution could be resizing the time slice, consequently the number of instruction execute each time by the PE. Decreasing the window size increases the precision, thus the level of fidelity in the simulation. Notwithstanding, the modification impacts in the performance obtain by the simulator due the cost in the context switch between PE. In order to observe this behavior a series of tests are prepared, using the same application in different scenarios. Ranging the time slice in 1 to 0.000001, therefore the instructions per window as between 1.000.000 to 1. Figure 1 shows the degradation in terms of simulation speed vs. time slice, notice a severe drop when the number of instructions per round approximate 5 thousand.

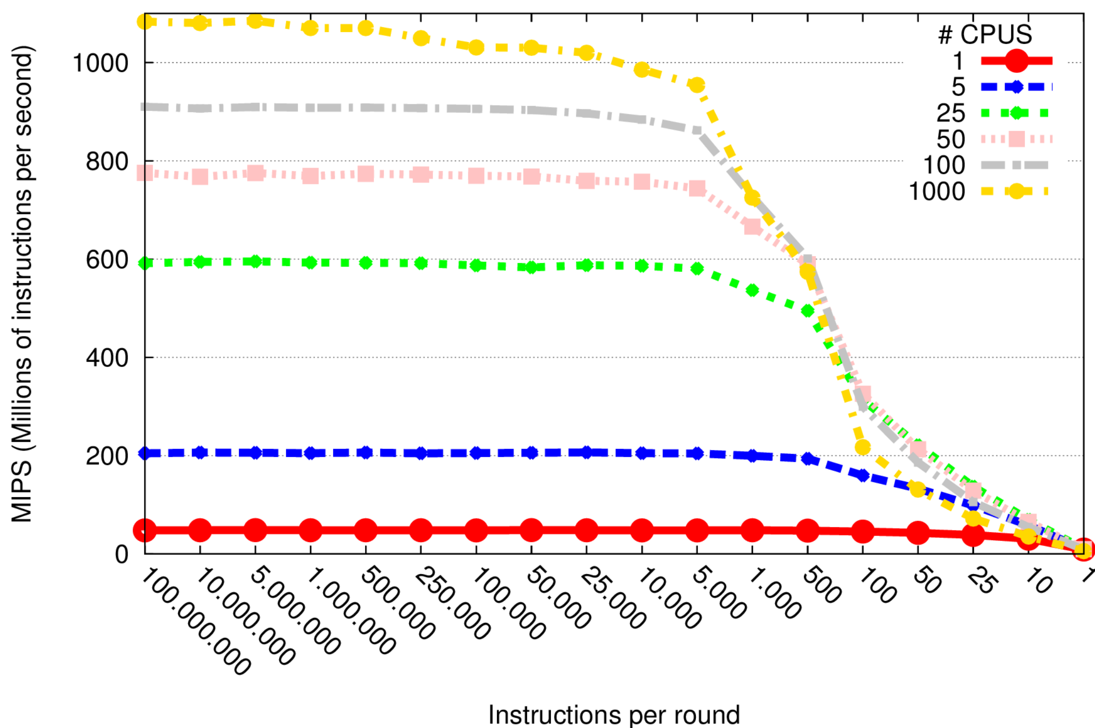


Figure 1 Scalability of Time Slice

4. Timing estimation

This section describes the proposed timing CPU model extension. As previously mentioned, OVPsim does not model cycle-accurate processors but rather instruction accurate processors, which provides inaccurate application execution time. In order to overcome this, the proposed model integrate a quasi-cycle accurate functionality in OVPsim. It relies on monitoring at run-time the instructions executed by a target CPU, employing a purely run-time based approach avoid huge amount of memory usage needed by trace-driven based approaches.

Developing the monitoring process based on OVP IPIs and integrated in a component called Watchdog, comprising three main modules: (*i*) assembly parser, (*ii*) a hash table with pre-characterized groups of instruction, and (*iii*) timing information. Calibrating both hash table and timing information according to an instruction set architecture (ISA).

4.1 Cycles Watchdog

The monitoring process bases in a special function supply by the OVP API called Callback. It is triggered when a predefined particular event occur and subsequently the simulator call a handler pointing to function provided by the programmer. The configuration of trigger events as memory access, bus access, etc. occurs at compilation-time, additionally allowing restrict this event a memory range instead entire address space.

After the simulation begin, whenever an instruction is fetched from the memory (1) is triggered a callback, thus activating the Watchdog. Inside the first module, the binary code of the instructions is acquired using the program counter (PC) register, thus the binary code is disassemble, divided in sub-strings, and identifies the instruction that must be executed (2). The identified instruction is employed as a hash table key to discovery which class (e.g. arithmetic, load, store) such instruction belongs (3). Thus, computing the necessary number of cycles to perform this instruction, considering the predefined timing information. Once, computed the cycle count, each instruction is executed in the CPU (4).

The number of cycles needed to execute each instruction can be affected by several conditions, such as content in the registers, last instructions executed, and address accessed, among others. Cycle *timing* for single *load and store* are examples of operations that affected by such environmental conditions. In such cases a normally 2 cycles load can be executed in a single cycle, since their address and data phases may be pipelined when the next instruction is an load or store, additionally this behavior can be chained thru multiple instructions (e.g. LDR R0,[R1,R5]; LDR R1,[R2]; LDR R2,[R3,#4] - normally four cycles total instead six.) [“ARM® Cortex-M4 Processor Technical Reference Manual” [S.d.]]. Treating these conditions with addition of internal logic and data structures, enabling to determine precise cycle counts even under such circumstances.

At the end of the simulation is possible retrieve the cycle estimation, besides about number of instructions grouped per class or individually, the entire information separated by PE. Practically realized entire online during the simulation, as result, the post-processing stage is almost negligible when compared with simulation.

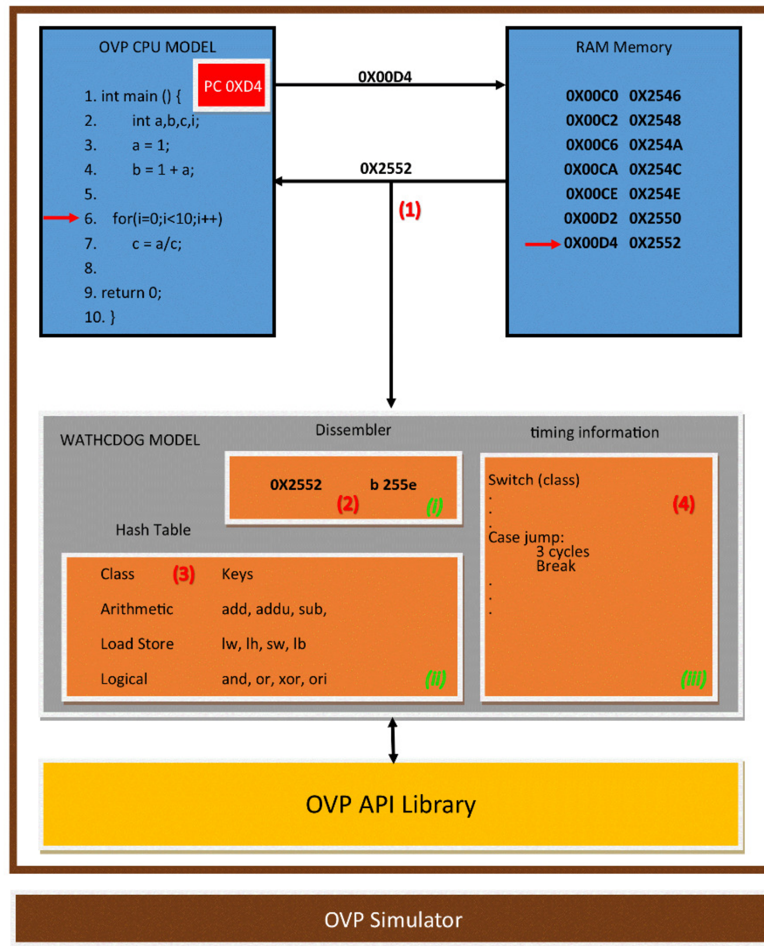


Figure 2 shows the block diagram and main flow of the run-time based approach.

In order to increase the simulation speed the timing model a parallelization level as inserted by porting the model to a thread based implementation (i.e. POSIX thread library). In order to achieve this goal, the cycle estimation was separated in two phases: A sequential and parallel using the OVPsim natural flow as referential. Fig. 2 shows the block if the parallelized version.

The first and sequential phase is execute concomitantly with the simulation, similarly whenever an instruction is fetched from the memory(1) a module is called, the binary code of the instruction is disassembled (2), and different of previous solution the result is stored in a buffer (3). This module is also responsible for manage the buffer, allocating memory chunks and when the buffer reach the predefined limit, create a new thread (4) passing the buffer an as argument.

Comprising the parallel phase, the new thread created is completely independent of the rest of the simulation, using local variables in most of the computation, however when it is necessary update the main data structure, which is shared, a *mutex variable* is use to implement a data mutual exclusion. The work realized is parser the instructions, analyzing each instruction of the buffer, extracting the mnemonic (5) and using as a key in the hash table (6). In this hash table is storage the information about what class

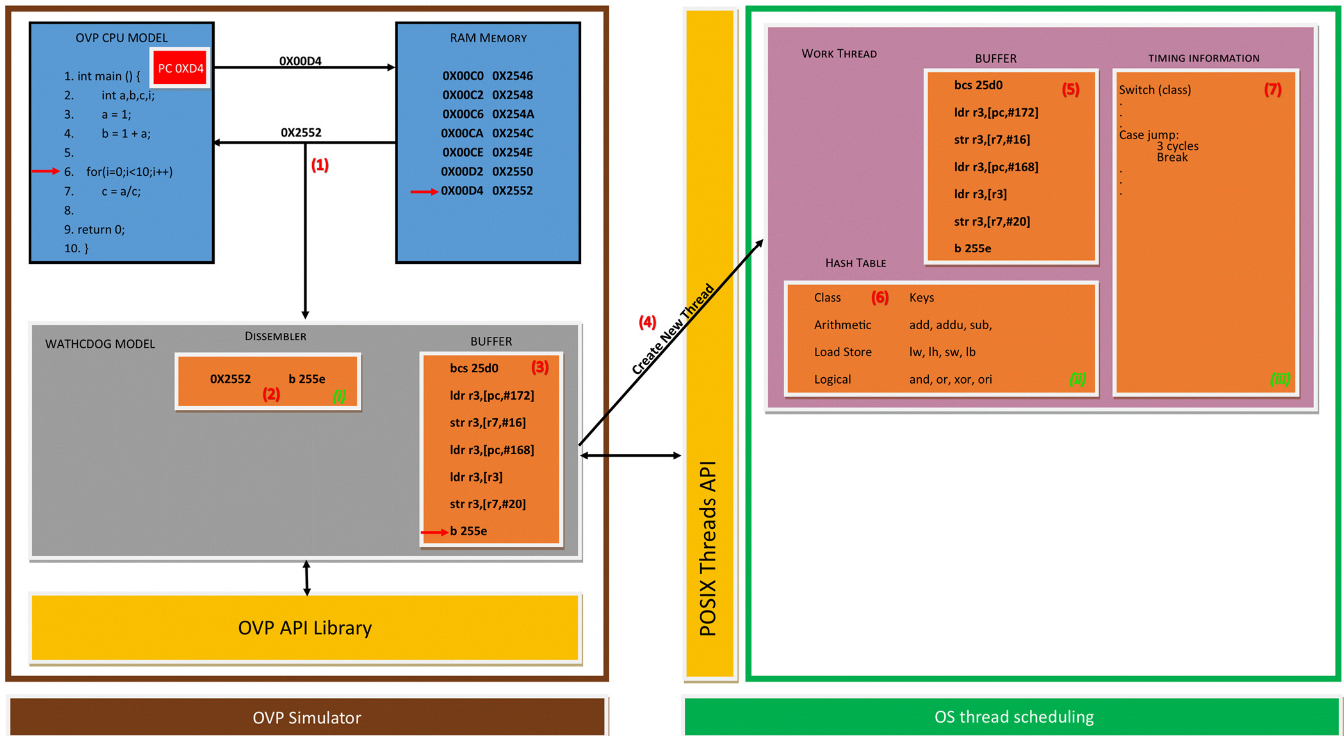


Figure 3 shows the block diagram with thread

belongs the mnemonic of the instruction, equally as previous mentioned (7). Thus, is computed the necessary number of cycles to perform this. Once, the cycle count is computed, the buffer index moves to the next instruction. When the data contained ends, the main data struct is update and consequently the thread is destroyed, deallocating the buffer.

Finished the simulation a thread barrier is use to synchronize all thread, thus similarly with the sequential way, the cycle estimation is available. From the viewpoint of the user booth approaches are transparent, therefore, the only perceptible change is the considerable speedup achieved.

4.2 Results

In order to demonstrate the effectiveness of proposed approach, a 32-bit ARM Cortex-M4 processor based on the ARMv7M architecture. In the case of study are used benchmarks from different domains, demonstrating the benefits towards the software evaluation facilities inherent to the proposed approach.

Measuring the accuracy comparing the estimation of the model with a STM32F4 Discovery board as illustrated in Figure 4; the reference platform board is built around a 32-bit ARM Cortex-M4F core running a FreeRTOS kernel version V.7.4.21 at 1 GHz. Among other features, ARM Cortex-M4F supports single precision floating-point unit (FPU) and power saving modes, which can be used for the development of energy-efficient embedded systems. Both Cortex-M4F and FreeRTOS are highly used in high-performance embedded system design, justifying the choice.

To provide relevant metrics, selecting application benchmarks that permit exploiting and assessing performance of embedded CPUs from different research domains. For

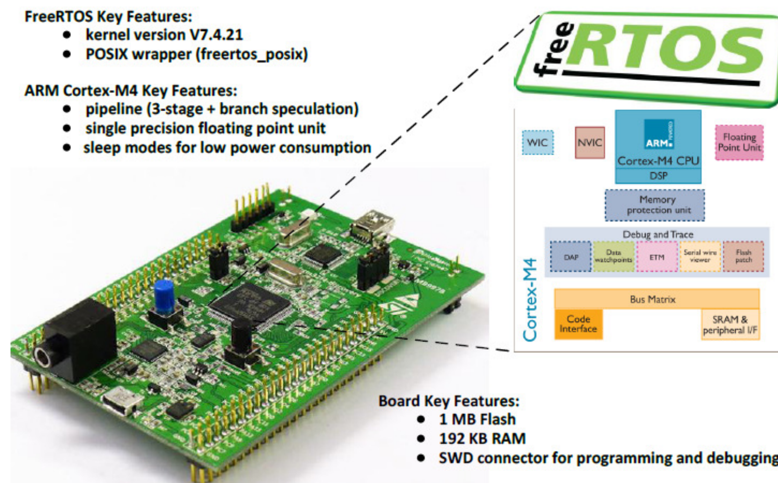
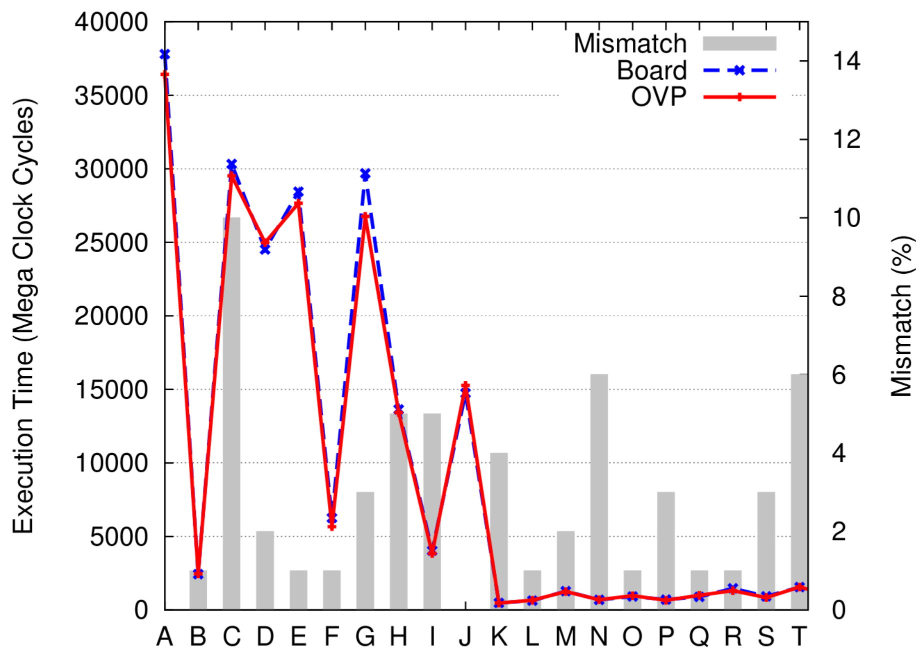


Figure 4 Adopted reference board platform. Proposed illustration integrates figures captured from their owners' websites

instance, the 13 selected application (i.e. from A to L, Fig. 6) of worst-case execution time (WCET) benchmarks vary in terms of execution time, number of loops, matrixes and array size [Jan Gustafsson 2010]. Such diversity allows observing the accuracy evaluation of the proposed approach under different conditions. Remaining applications also originates from different categories like biological (e.g. Smith Watermann) and telecommunication (e.g. CRC32).



A - Fibonacci | **F - Binary Sort** | **K - Factorial** | **P - SmithWaterman**
B - FIR | **G - Compression** | **L - FDCT** | **Q - Btree**
C - FFT | **H- CNT** | **M - InsertSort** | **R - BFSH**
D - BubbleSort | **I - EDN** | **N - MDC** | **S – Hanoi Tower**
E - ADPCM | **J - Expint** | **O - USQRT** | **T – Harmonic**
U – CRC32

Figure 5 Benchmark execution time comparison between real board and proposed timing CPU model (OVP)

To evaluate the accuracy of the proposed timing CPU model, in most cases each application benchmark was executed for at least 0.7 million instructions. Figure 5 shows that the benchmarks execution mismatch between the real board platform and the simulated timing CPU model in OVP is between 0.01% and 10.5%. Note that execution time mismatch is below 5% in 16 out of the 20 adopted benchmarks.

4.3 Scalability

The simulator has already demonstrate when expose to scenarios of hundreds of cores retaining the expected performance. As the proposed model alters the traditional execution flow, a performance decrease is expected. Previous mentioned OVP bases in Dynamic Binary Translation to active the speed demonstrated, extraction the velocity from this direct relationship between host and target ISA. Subsequently the instruction of the quasi-cycle accurate model is mandatory stop the execution at every executed instruction. Furthermore, executing several instructions in the host machine to implement the model.

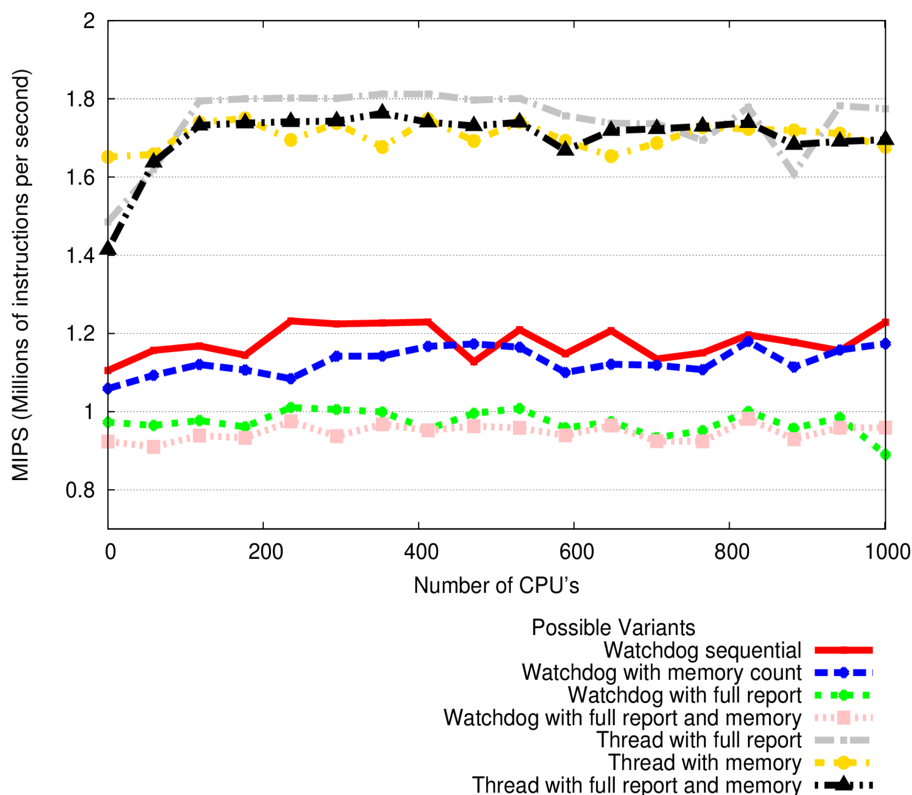


Figure 6 Simulation performance as the simulated MPSoC scales from 1 to 1000 CPUs

The model has two main variants, each one divided in four variants. First presented was the sequential, following the flow of OVPSim as describe in the section 5.1. Two others variants tested was sequential with memory count and sequential with full report. The memory count deploy callbacks in order to count the number of memory access in the memory, including fetched instructions, reads and writes. When full report is enable, every executed instructions is stored in data struct, demanding more processing time during the executing. The fourth variant is the union these two variants to create the sequential with full report and memory count. In order to characterize the scalability of

the parallel branch was create three subtypes, Thread with full report, Thread with memory count and the union of these two. The full report is performed is parallel, as result its impact is negligible when compares with a Thread version without full report, totaling seven variants.

In order to demonstrate the scalability of all proposed model variants and considering many-core systems scenarios, the number of CPUs as varied from one to a thousand. In each PE executes an instance of FFT. Results show in the Figure 6.

5 Power Analysis

In this section, is introduce a new instruction-level energy estimation model a for a Plasma processor [“Plasma CPU” [S.d.]].The Plasma processor is a 32-bit RISC processor based in the MIPS architecture with a 3-stage pipeline. It is important to highlight that the proposed profiling method is simple and transparent, applicable in any other processors cores with no considerable rework.

Contrary to most of the approaches, our methodology can be applied in optimize IP cores without previous knowledge of internal architecture our access to source code. Also in the propose energy model, the profiling phase is highly dynamic and easily migrate to others platforms in counterpoint to measurement-based method.

The approach applied is similar to the presented in the section 5, were the watchdog was presented. A run-time approach based in monitoring instructions executed by a target CPU. However, in the first case, almost information about cycle duration was available in the documentation, for this energy approach, the manufacturer does not offer the information about energy consumptions. As result, dividing the process flow in two: a profiling phase and the execution phase.

5.1 Characterization

First and most important phase is the characterization, the objective is measure the energy spent by each instruction belonging to the ISA. Three main steps compose the characterization flow: First step comprises developing benchmarks in order to profile the energy consumption for each instruction, noting the similarity between instructions it is possible divide the instructions in groups, taking into account their behavior in the data-path. For instance, there is a close relationship between instructions *add* and *addiu* and between *lw* and *sw*. Thus, dramatically decreasing the complexity and time spent in the instruction profiling stage.

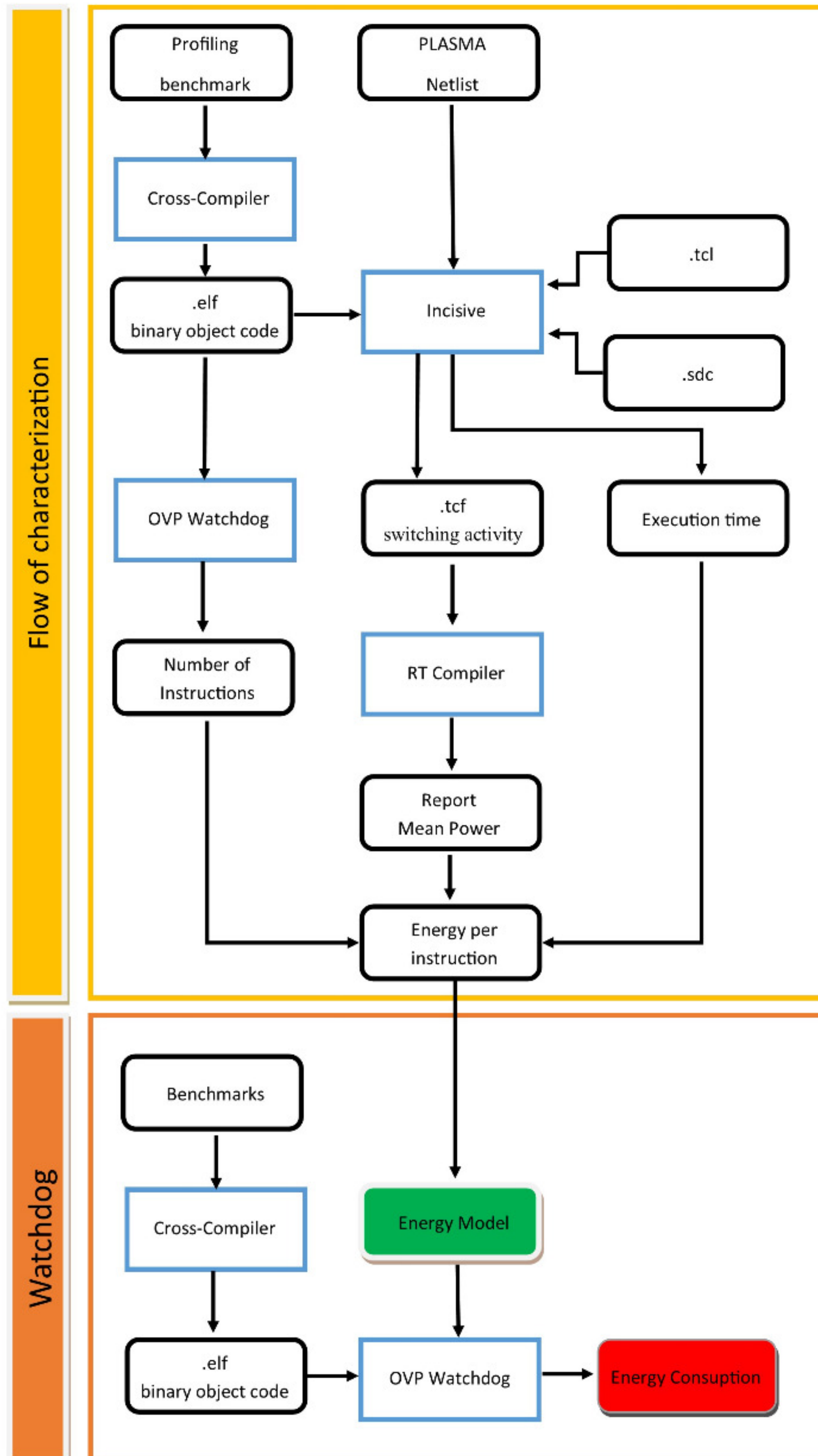


Figure 7 Characterization Flow

For the micro-architecture used in this work, we propose to split the instructions in seven groups: arithmetic, logical, move, branches, load/store and shifts. Furthermore,

placing the mnemonic 'move' among arithmetic instructions because it is a pseudo-instruction (performed by an *addi*). In addition, the multiplication and division instructions, which are arithmetic instructions, modeling as 12 arithmetic instructions each, since this Plasma version take 12 cycles to execute them. Nevertheless, a benchmark was created to characterize the pipeline stall as a *nop* instruction.

Next, one application was developed for each instruction group. These applications execute almost the instructions present in their respective group, including the several variations of the same instruction (e.g. *add, addi, addiu*, etc.). Although, to create an executable code is necessary blend instructions from different groups. Therefore, codding carefully these applications in a way that at least 90% of the executed instructions belong to the application target group. Previous experiments using higher percentages (e.g. 95%), showed a negligible difference. Each application executes, in average, 35 thousands instructions. Finally, the application is executed in the OVP simulator to verify its

correctness and to extract the exact number of executed instructions. After validating the application behavior in RTL simulation, passing to the second step. Still, before executing the second step, performing the Plasma logic synthesis with *Cadence RTL Compiler* tool targeting a 65nm low power library from ST Microelectronics. Subsequently , the Plasma netlist (i.e. gate-level description), the application object code, a *tcl* script and the *sdc* file containing the timing constraints are loaded into the *Cadence Incisive* simulation tool. Next, simulating until the application finishes its execution. As result, a *tcf* file is generated. This file contains statistic information about the switching activity of each cell and wire in the netlist. In addition, the exact execution time of each application is collected.

Finally, in the last step, the power evaluation takes as input the netlist, the *sdc* and the *tcf* files. Cadence RTL Compiler also performs this task; the tool reads the netlist and computes de power consumed by each cell based in their switching activity information in the *tcf* file. Subsequently, produces a report containing the average power consumption for the application. This information, associated with the execution time collected in the previous step, the total energy consumed is obtained. Then, with the

Table 1 Energy groups profiled

Groups	Power (mW)	Exec Time (us)	Energy (nJ)	# of inst	Energy per Inst (nJ)
Arithmetic	6,456	342,755	2212,826	34764,000	0,064
Jump	6,046	102,600	620,320	10224,000	0,061
Load-Store	4,094	1042,800	4269,223	48561,000	0,088
Logical	4,469	349,735	1562,966	35462,000	0,044
Move	3,129	480,725	1504,189	39363,000	0,038
NOP	257,155	2,141	550,569	26130,000	0,021
Shift	3,824	298,735	1142,363	30362,000	0,038

number of executed instructions and the total energy consumed, it is possible to calculate the energy consumed for each instruction. Reproducing this process for all instruction groups. The results of the characterization are shown in TABLE I.

After the characterization phase is finished, the next phase comprises building the energy model in the instruction-set simulator. The energy model relies on monitoring at run-time the instructions executed by the target CPU. Developing the monitoring process modifying the presented in the section 5, extending and enriching with energy models. The main modification inserted is the modification of the hash-table and the information contained.

***Table 2* Energy groups profiled**

	Name	Energie (mW)	Elapsed time (us)	Total consumption (nj)	Instructions	Estimated power (nj)	Error (%)
A	Bfsh	4,830	15235,505	73587,489	1035188,000	73581,398	0,0%
C	Binary search	5,092	2630,475	13394,379	196531,000	13503,721	0,8%
B	Bit Manipulation	5,388	4407,555	23747,906	380318,000	25974,705	8,6%
D	Bubble	5,296	4617,405	24453,777	336883,000	23079,633	6,0%
E	Counts	4,851	5147,435	24970,207	331364,000	24188,576	3,2%
F	Crc	5,327	2985,515	15903,838	254596,000	17284,408	8,0%
G	Edn	4,563	3437,145	15683,693	208368,000	17126,121	8,4%
H	Expint	3,730	11072,250	41299,493	448356,000	39551,980	4,4%
I	Factorial	4,724	8859,465	41852,113	493032,000	41173,594	1,6%
J	Fft	4,897	803,115	3932,854	54659,000	4256,579	7,6%
K	Fibonacci	5,141	6060,000	31154,460	418029,000	30180,719	3,2%
L	Hanoi	5,027	13108,105	65894,444	885305,000	66008,188	0,2%
M	Harm	4,075	9527,265	38823,605	435278,000	40763,266	4,8%
N	Insertsort	5,379	3953,825	21267,625	309159,000	20451,490	4,0%
O	MatrixMult	4,958	10446,895	51795,705	665043,000	49646,188	4,3%
P	Mdc	3,573	6597,485	23572,814	238784,000	24461,912	3,6%
Q	Peakspeed	5,164	471,545	2435,058	30108,000	2300,600	5,8%
R	UD	4,770	10464,475	49915,546	732831,000	53415,395	6,6%
S	Usqrt	5,107	4090,785	20891,639	298166,000	20105,066	3,9%

5.2 Experimental Setup

To demonstrate the accuracy of the instruction-level energy model, comparing the estimated energy consumption with a commercial tool for 19 benchmarks. The experimental tests were performed in the same way that was utilized in the characterization phase, in three steps, first run the target application in the OVP simulator and collect the prediction of our model, second run the targeted code in the Incisive Simulator and by last with help of RC compiler acquire the mean power of the application. With the previous knowledge of the number executed instructions, is possible measure the power consumption of the application.

OVP simulations, Incisive and RC compiler were executed in an Intel Xeon CPU W5580 8 cores x 3.20GHz – 32GB RAM. OVP release 20131018.0 and adopted scenarios were all compiled by gcc version 4.1.2 with full optimization enabled (i.e. -O3). Note that the code executed in the reference Incisive differs from the one executed in OVPSim in few instructions (less than a hundred), which are related to the boot

sequence in different platforms. An application set of 19 benchmarks that permit exploiting and assessing performance of embedded CPUs were selected from different research domains, shows in the Table 2. Observing the results, the mean error is approximately 4.33%. The comparison between gate-level and watchdog estimation is presented in figure 8, as is the mismatch.

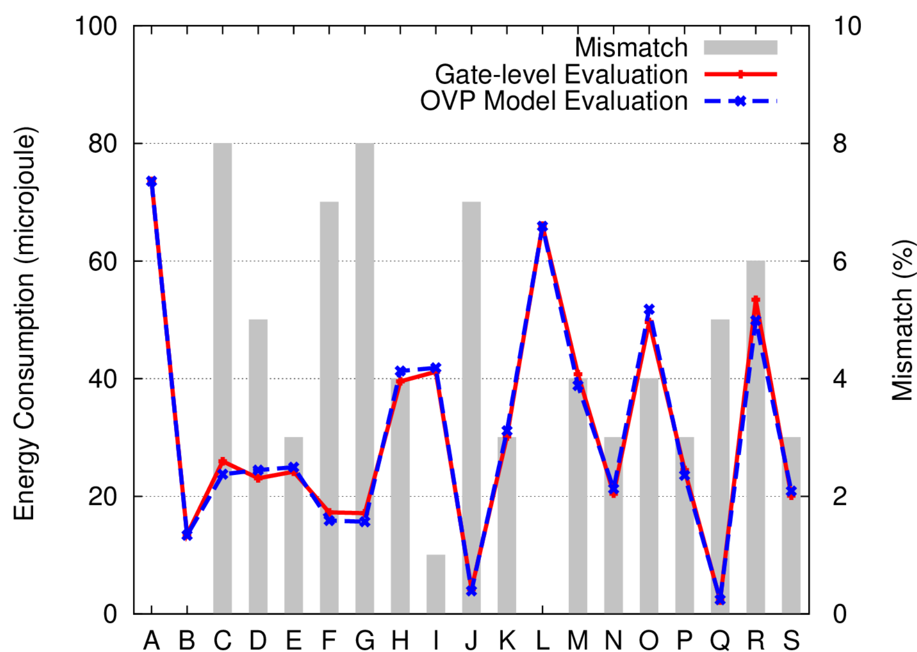


Figure 8 **Mismatch, gate-level and watchdog evaluation.**

6. Conclusion

This work presented extensions to the OVP framework by including an accurate timing and power estimation CPU model. The resulting approach provides the same design flexibility, setup and debugging features inherent to OVP, while enabling accurate software evaluation in design phase. Thus, programmers can use the same simulator to have fast simulation and accurate software evaluation.

The insights gained from these preliminary results call for improvements towards the following aspects: simulation time optimization, inclusion of instruction-based power models for CPUs, exploration of new techniques for online system management.

7. References

Abril Garcia, A. B., Gobert, J., Dombek, T., Mehrez, H. and Petrot, F. (2002). Cycle-accurate energy estimation in system level descriptions of embedded systems. In *9th International Conference on Electronics, Circuits and Systems, 2002*.

ARM® Cortex-M4 Processor Technical Reference Manual ([S.d.]). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0439d/index.html, [accessed on Apr 21].

Bazzaz, M., Salehi, M. and Ejlali, A. (jul 2013). An Accurate Instruction-Level Energy Estimation Model and Tool for Embedded Systems. *IEEE Transactions on Instrumentation and Measurement*, v. 62, n. 7, p. 1927–1934.

Binkert, N., Beckmann, B., Black, G., et al. (aug 2011). The Gem5 Simulator. *SIGARCH Comput. Archit. News*, v. 39, n. 2, p. 1–7.

Callou, G., Maciel, P., Tavares, E., et al. (jun 2011). Energy Consumption and Execution Time Estimation of Embedded System Applications. *Microprocess. Microsyst.*, v. 35, n. 4, p. 426–440.

Castillo, J., Posadas, H., Villar, E. and Martínez, M. (nov 2007). Energy Consumption Estimation Technique in Embedded Processors with Stable Power Consumption based on Source-Code Operator Energy Figures.

Chiang, M.-C., Yeh, T.-C. and Tseng, G.-F. (apr 2011). A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 30, n. 4, p. 593–606.

Full System Simulation with Wind River Simics ([S.d.]). http://www.windriver.com/products/simics/, [accessed on Apr 21].

Jabbar, M. H., Houzet, D. and Hammami, O. (sep 2013). Impact of 3D IC on NoC Topologies: A Wire Delay Consideration. In *2013 Euromicro Conference on Digital System Design (DSD)*.

Jan Gustafsson, A. B. (2010). The Mälardalen WCET Benchmarks: Past, Present And Future. p. 136–146.

Kalla, P., Henkel, J. and Hu, X. S. (oct 2003). SEA: fast power estimation for micro-architectures. In *5th International Conference on ASIC, 2003. Proceedings*.

Konstantakos, V., Chatzigeorgiou, A., Nikolaidis, S. and Laopoulos, T. (apr 2008). Energy Consumption Estimation in Embedded Systems. *IEEE Transactions on Instrumentation and Measurement*, v. 57, n. 4, p. 797–804.

Lee, S., Ermedahl, A., Min, S. L. and Chang, N. (2001). An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors. In *Proceedings of the*

ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems. , LCTES ’01. ACM. http://doi.acm.org/10.1145/384197.384201, [accessed on Mar 31].

Nikolaidis, S., Kavvadias, N., Laopoulos, T., Bisdounis, L. and Blionas, S. (2003). Instruction Level Energy Modeling for Pipelined Processors. In: Chico, J. J.; Macii, E.[Eds.]. *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*. Lecture Notes in Computer Science. Springer Berlin Heidelberg. p. 279–288.

Open Virtual Platforms (OVP) ([S.d.]). http://www.ovpworld.org/, [accessed on Apr 21].

Plasma CPU ([S.d.]). http://plasmacpu.no-ip.org/, [accessed on Apr 21].

QEMU ([S.d.]). http://wiki.qemu.org/Main_Page, [accessed on Apr 21].

Sanchez, D. and Kozyrakis, C. (2013). ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. , ISCA ’13. ACM. http://doi.acm.org/10.1145/2485922.2485963, [accessed on Apr 20].

Stattelmann, S., Ottlik, S., Viehl, A., Bringmann, O. and Rosenstiel, W. (jun 2012). Combining instruction set simulation and WCET analysis for embedded software performance estimation. In *2012 7th IEEE International Symposium on Industrial Embedded Systems (SIES)*.

Sultan, S. and Masud, S. (jul 2009). Rapid software power estimation of embedded pipelined processor through instruction level power model. In *International Symposium on Performance Evaluation of Computer Telecommunication Systems, 2009. SPECTS 2009*.

Thach, D., Tamiya, Y., Kuwamura, S. and Ike, A. (mar 2012). Fast cycle estimation methodology for instruction-level emulator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*.