2010

Embedded Processor Selection/Performance Estimation using FPGA-based Profiling

Fadi Obeidat
Virginia Commonwealth University

School of Engineering

Virginia Commonwealth University

This is to certify that the dissertation prepared by Fadi Obeidat entitled EMBEDDED PROCESSOR SELECTION/PERFORMANCE ESTIMATION USING FPGA-BASED PROFILING has been approved by his committee as satisfactory completion of the dissertation requirement for the degree of Doctor of Philosophy

---

Robert Klenke, Ph.D., Committee Chair, Department of Electrical and Computer Engineering, School of Engineering

---

Mike McCollum, Ph.D., Dept. Department of Electrical and Computer Engineering, School of Engineering

---

Afroditi V Filippas, Ph.D., Department of Electrical and Computer Engineering, School of Engineering

---

Kayvan Najarian, Ph.D., Department of Computer Science, School of Engineering

---

Vojislav Kecman, Ph.D., Department of Computer Science, School of Engineering

---

Date

EMBEDDED PROCESSOR SELECTION/PERFORMANCE ESTIMATION USING FPGA-BASED PROFILING

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by
Fadi Obeidat

Director: Robert Klenke
ASSOCIATE PROFESSOR, DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Virginia Commonwealth University
Richmond, Virginia
July, 2010

Abstract

In embedded systems, modeling the performance of the candidate processor architectures is very important to enable the designer to estimate the capability of each architecture against the target application. Considering the large number of available embedded processors, the need has increased for building an infrastructure by which it is possible to estimate the performance of a given application on a given processor with a minimum of time and resources. This dissertation presents a framework that employs the softcore MicroBlaze processor as a reference architecture where FPGA-based profiling is implemented to extract the functional statistics that characterize the target application. Linear regression analysis is implemented for mapping the functional statistics of the target application to the performance of the candidate processor architecture. Hence, this approach does not require running the target application on each candidate processor; instead, it is run only on the reference processor which allows testing many processor architectures in very short time.

To my wife, my son, and the family

Table of Contents

List of Figures

List of Tables

Acknowledgments

Since I started my Ph.D., I have had the chance to work on several projects which all added a great experience to me. I am blessed to have been guided and supervised by Dr. Robert Klenke who has offered me challenge and great support. I also want to thank Dr. Mike McCollum, Dr. Afroditi Filippas, Dr. Kayvan Najarian and Dr. Vojislav Kecman for their valuable feedback on my work. I am greatly thankful for Dr. Jerry Tucker and Dr. Rosalyn Hobson for their boundless support. My gratitude goes for my colleagues Jose Ortiz, Jeremy Cooper, Robert DeMott II, Abed Al-Raoof Bsoul, Jacob Berlier, and Brad Geltz who lent me their experiences and provided me with assistance repeatedly.

Chapter 1 Introduction

An embedded system is a special purpose computer working within a device to control/manage its functionality. For example, processors found in cell phones, digital cameras, medical equipments, cars and modern airplanes are all considered embedded systems. Because of the dramatic grown in embedded system applications, embedded processors have formed the largest class of computers. Figure 1.1 shows the market volume of the embedded processors compared to the other types of processors [1]. While they cover a wide range of applications, ranging from very simple applications such as monitoring the temperature for an air condition to very complex applications such as flight control systems, embedded processors vary in their characteristics, from 8-bit simple-pipeline architectures to 64-bit super-pipeline architectures. Consequently, and because of the very special functionalities embedded systems concerned with, it is very hard, or even impossible, to find an embedded processor that can be considered as the absolute best choice for all embedded applications. Hence, a careful trade-off should be considered when selecting an embedded processor.

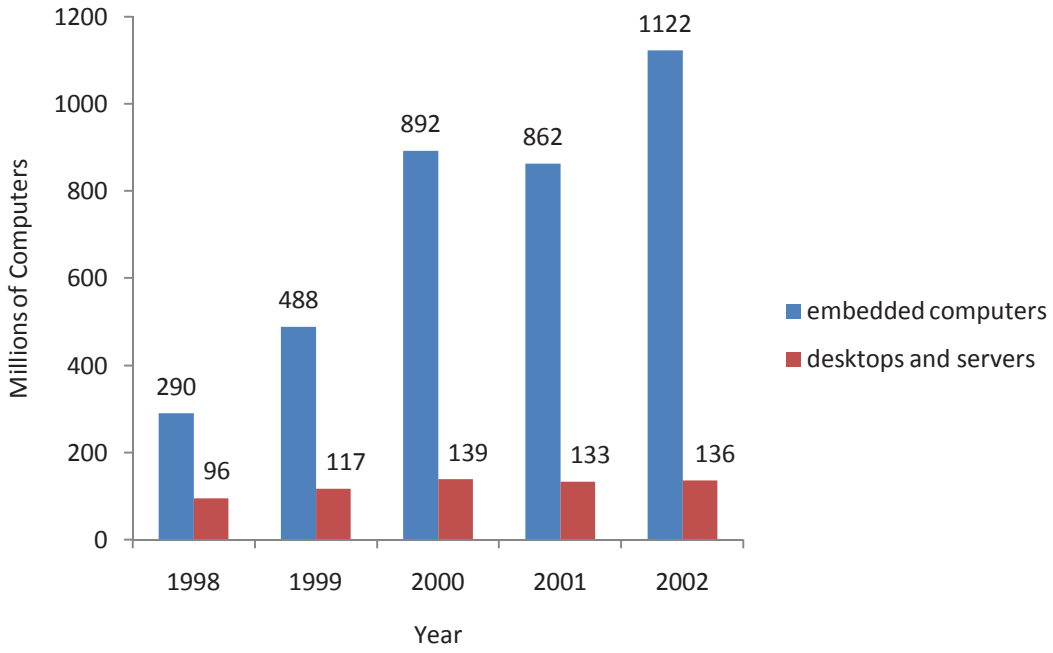


Figure 1.1. The number of embedded processors sold between 1998 and 2002 compared to desktop and server processors [1]

1.1. Overview

In embedded systems, selection of an appropriate processor to execute a given application is considered as a crucial decision. The processor must have enough processing power to meet the time constraints for executing the algorithms necessary to operate and control the system. However, the characteristics of the selected processor should not be significantly more powerful than required as this increases its cost, power consumption, heat production and other physical characteristics that must be kept to a minimum in most embedded systems [2]. Furthermore, adopting the most efficient and

effective algorithm for a given system and application is also at high level of importance. The algorithm should achieve the desired functionality, meeting the time and accuracy constraints of the embedded application, with a minimum of hardware resources. Accordingly, matching several algorithms demands with different processor architectures for modeling the performance of pairs of algorithms/processors is considered as very challenging task [3]. This is especially true at early stages of the design process where: "**there's no magic box into which you place the system requirements that spits out which processor to use**" *S. Rosenthal* [4].

1.2. Motivation

Embedded applications involve computations that are subject to real-time physical constraints. The response of an embedded system is determined by its reaction to the physical environment, specified by deadlines, throughputs …etc, and the execution on the physical platform, specified by processor speed, power …etc [5-6]. While the application's algorithm governs the reaction constraints of the system, it is the designer's responsibility to select the execution platform to achieve highest degrees of performance and efficiency. Hence, a key to embedded systems design is the ability to accurately predict the system's execution time for a given algorithm and set of inputs.

Selecting an appropriate processor for implementing the target algorithm should be compatible not only with the other hardware components such as sensors and actuators,

but also with the algorithm functional behavior and timing constraints. Degradation in any of these parts affects the performance of the whole system and may lead to undesired performance characteristics [7-8]. So, **it is very important to build an infrastructure by which it is possible to extract the dominant operations and required resources that characterize the system behavior**.

1.3. Challenges

Although embedded processors are generally considered less complex than desktop processors, most modern embedded processors implement common principles in computer architecture in order to gain the highest possible performance. Thus, most embedded processors include one or more of the following features: pipelining, branch prediction, cache …etc. These advanced architectural features introduce difficulties in evaluating the performance of not only desktop processors but also embedded processors. Moreover, factors like time-to-market, and the critical timing and functionality of embedded applications make evaluation of embedded processors more challenging.

Ideally, the time needed by a CPU to execute a program is given by the following equation:

$$\text{CPU time} = \sum_{i=1}^{n} \left(CPI_i \times C_i \right) \text{ x clock cycle} \qquad (1.1)$$

Where:

CPI_i = cycles per instruction-$type_i$

C_i = count of instructions of $type_i$ executed

n = number of instructions in the corresponding ISA

clock cycle = 1/(CPU clock rate)

However, this equation does not account for the clock cycle overhead coming from other aspects in the design, since it assumes that no pipeline stalls occur (no structural, data, or control hazards), and all instructions are available in the cache or first-level memory (all hits) [1].

Nowadays, most processor architectures adopt pipelining to enhance processor performance which in turn allows multiple instructions to run concurrently in the processor, one instruction per stage/functional unit. A simple pipeline is divided into three stages: instruction fetch, instruction decode, and instruction execution. This technique has been evolved continuously to achieve better performance. For example, the execution stage can be divided further into execution, memory access, and write back. Increasing the number of stages does not actually decrease the execution time for instructions; instead, it increases the number of overlapped instructions which in turn increases the processor throughput. However, increasing number of pipeline stages, or implementing parallel functional units that can work simultaneously in the same stage, means increasing the architecture complexity for handling the control between

stages/units. Theoretically, after filling the pipeline stages, the processor is supposed to finish one instruction every execution cycle. However, in practice, the implementation of the pipeline results in some constraints that limit achieving the maximum desired performance. Such constraints are called pipeline hazards where the flow of the instructions is halted for one or more execution cycles.

Pipeline hazards are divided into three types: structural hazards, data hazards, and control hazards. In structural hazards, instructions that are overlapping in different pipeline stages try to access the same architecture resource. For example, in a single memory processor architecture, both instructions and data are stored in the same memory. Thus, fetching an instruction and accessing the memory for loading or storing data cannot occur at the same time, which means that one instruction should wait until the other finishes accessing the corresponding stage. Data hazards occur when an instruction cannot be executed as scheduled because there is missing (dependent) data that is a result from a previous instruction. For example, if a load instruction is followed by an another instruction in which one of the source operands is the destination operand of the load instruction, then the following instruction should wait until the load instruction finishes loading the required data from the memory. In control hazards, the instruction that has just been fetched is not the one that needs to be executed due to some change in the instruction flow such as a branch misprediction.

Different software techniques (static: at compile time) and hardware techniques (dynamic: at run time) have been developed to reduce or eliminate pipeline hazards. For example, splitting the memory architecture into an instruction memory and a data memory solves the corresponding structural hazard. Forwarding results before finishing an instruction's execution helps to reduce the impact of data dependencies. Also, reordering the instructions at compile time can minimize data dependencies and reduce the penalty of branch mispredictions. Another technique to assist making correct branch decisions is to implement branch perdition units at hardware level, e.g. the simplest version is to assume branches always taken.

Implementing a cache as a first-level of the memory hierarchy includes many challenges regarding decisions such as cache size, block size, and level of the associativity. For example, increasing the cache size decreases the miss rate but may tend to increase the access time, i.e., increasing the block size decreases the miss rate (due to spatial locality[1]) but the miss rate tends to go up as the block size becomes too large relative to the cache size (this also tends to increase miss penalty). Also, increasing the level of cache associativity decreases the miss rate, however, the larger cache size, the less relative reduction on the miss rate (this also may tend to increase the access time).

[1] *Spatial locality*: the locality principle stating that "if a data location is referenced, data location with nearby address will tend to be referenced soon", while, *temporal locality* states that "if a data location is referenced then it tend to be referenced again soon" *D. Patterson* **[1]**.

All of the mentioned techniques, whether used to enhance the performance or to reduce the trade-offs in performance impacts, hold a lot of challenges when considering the performance evaluation of an architecture for several algorithms. For example, increasing the depth of the pipeline generally tends to achieve better performance, however, if the running algorithm behaves in such a way to produce large numbers of branch mispredictions, then, the penalty of the branch mispredictions on performance may dominate the performance of the system. On the cache side, different algorithms have different functional behaviors (instruction/data flow) that vary in the level of spatial and temporal locality, hence, the benefits of a given cache implementation are algorithm-dependent (as in pipeline architecture). In fact, other microarchitectural details (such as number of internal registers, etc) tend to have different performance impacts with different functional behaviors.

While the above discussion points to the difficulty of performance evaluation at the instruction-level, estimating the performance at source code-level (e.g., the C-level) holds more challenges. For example, the impacts of other tools involved in the system development (such as compilers) are hard to predict in the absence of the assembly code. In addition to the role of compilation in rearranging the code to eliminate data and control hazards, other issues such as predicting the final version of the code produced from different levels of optimization is hard to predict. Moreover, the hidden functional behavior of the library functions, which are sometimes only provided as binary files, also make detailed performance analysis a very challenging task [9]. Furthermore, some

operations at source code-level can be implemented using different functionalities. For example, multiplying or dividing an integer by a number to the power of two (2^n) is normally handled by shifting that number n-digits to the left or right, respectively.

Considering the obstacles in evaluating the performance of a single architecture for a wide spectrum of algorithms, things become more challenging when evaluating multiple different processor architectures with different instruction set architectures (ISAs), or even the same ISA with different microarchitectural details. ***Amdahl's* law** states that "**the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used**" [1]. This factor points to the difficulty in evaluating different processor architectures as each architecture tends to react differently to algorithm's demands, based on its ISA, available resources and processor organization. For example, if processor A includes a built-in floating point functional unit while processor B implements floating point operations using library functions (not supported by its ISA), it will not be a straightforward conclusion that processor A will be faster than processor B to execute a certain application, even if that application includes floating point computations in its code. This is because other factors such as memory architecture and the real time functional behavior of the target algorithm may dominate the performance. For example, the cache architecture of the processor B may, significantly, perform better behavior to exploit the spatial/temporal locality characteristics of the target algorithm, on the other hand, the algorithm run-time functional behavior may

access the portions of the code that include floating point operation infrequently (non-intensively).

1.4. Problem Statement

At the early design stages of an embedded system, exploring the performance characteristics of the target application by executing it on different hardware platforms is considered a very costly approach in terms of time and money. Such an approach requires acquiring the software development tools for each processor and the corresponding skills to use them, integrating the processor with other system components such as sensors/actuators and analog/digital convertors (or modeling their equivalent behavior), and possibly testing the system in the field or on a hardware in a loop simulation environment. Hence, **a main concern for embedded system designers is to reduce the set of algorithm/processor candidates by eliminating unacceptable alternatives, or better yet, to have a systematic approach by which they are able to perform a fair performance comparison based on the main parameters and characteristics of the given application and the candidate processor's architectures. Furthermore, this comparison needs to be done efficiently, with minimum of resources: time, tools and cost, especially since often the software development tools and microarchitectural specifications of the candidate architectures are not readily available to the designer.** *In this research, the efforts are constrained to model the performance of single-threaded algorithms which would constitute the main loop of the target*

application. At this point, the effects of multiple threads and interrupts on the performance of the main loop algorithm are not considered. Furthermore, the performance metric considered is limited to the execution time of the algorithm on the target processor – expressed as either processor clock cycles or actual processor execution time.

1.5. Contribution

A key question is whether, in order to make a correct design decision, the designer needs performance estimation with absolute accuracy[2], or a relative accuracy[3] amongst design alternatives. At the early stages of the design process, factors like the estimation speed, which can influence how many alternatives can be considered in the design process, are also very important. Hence, a tradeoff to minimize such important factors with an acceptable decrease in the level of accuracy is considered to be very helpful, as long as it is possible to accurately classify the relative performance of the candidate architectures. Later in the design process modeling at a more detailed level of abstraction can be done on a minimized list of processors to gain better accuracy in performance estimation.

[2] In this dissertation, the term absolute accuracy (or absolute performance estimation) is used to describe the performance of an application on certain processor architecture in terms of number of cycles (or units of time), regardless the performance of other processor architecture.

[3] In this dissertation, the term relative accuracy (or relative performance estimation) is used to describe the performance of an application on certain processor architectures in terms of better-worse (slower/faster) where the estimated time is used only for comparison purposes rather than an absolute estimation.

This research describes a method for analyzing the performance of a given application on different architectures based on running/profiling the application on a reference model (processor). Hence, the performance statistics resulted from this analysis can be used to assist in determining the functional behavior of the target application. This approach is unique in that there have been no efforts, tools or frameworks, up to this point in time, that use a reference processor in a similar way to predict the performance of other (foreign) architectures. The observed statistics produced by the reference model can be used to determine the application's dominant operations (such as integers, floating point, etc), which in turn allows exploring the hardware resources needed to be embodied in the system platform. Using analytical modeling techniques such as regression analysis, these statistics can be used (bound), along with a set of reference performance measurements of a domain of algorithms on the candidate architectures, for estimating the performance of these architectures against new algorithms/applications.

This research also introduces a novel FPGA-based instruction profiling technique. While FPGA-based profiling is a relatively new technique in the field and has been adopted recently in certain FPGA and software/hardware codesign approaches [10-14], the technique described herein relies on tracing the unique instruction flow of the functions-of-interest (considered as a function's footprint) rather than tracing the program counter (PC) value. This is a very important distinction which allows framework developed for

this dissertation to be applied against new algorithms with minimal efforts, and without the need for time-consuming reconfiguration of the FPGA.

A basic assumption in this research is that the reference performance measurements for a domain of algorithms on the candidate architectures have been previously extracted and published by the processor's vendor/manufacturer, so customers (embedded system designers) can access both the performance records, and the source code of these algorithms. For the purpose of producing results to evaluate the framework, these reference performance measurements were obtained by the author by running benchmark programs on the example processors that were evaluated. However, in the future, it is proposed that vendors/manufactures will extract and publish these results for the benchmark programs for use by designers in evaluating the processor for their specific application. Moreover, performance measurements reported by other users (on candidate processor architectures) for specific programs/benchmarks can be used for training purposes as long as the embedded system designer has an access to these programs/benchmarks so they can be run and profiled on the reference model.

It should be pointed out that the method described herein does not produce a 100% cycle-accurate estimation; however, this approach provides the embedded system designer with a high-level, very fast, performance modeling technique that can be used for initial processor selection and that requires minimal resources/knowledge of the application functional behavior and processor architecture.

1.6. Dissertation Organization

This dissertation is organized as follows: Chapter 2 discusses/surveys the performance modeling techniques in the areas of processor design, selection and embedded systems. Chapter 3 shows a problem analysis and discusses the theoretical background of the proposed framework. Then, reference model specifications and FPGA-based profiling are discussed in Chapter number 4. Chapter 5 illustrates experiments and results. Chapter 6 concludes, and Chapter 7 describes future work.

Chapter 2 Performance Modeling

Whether targeting desktop processors or embedded processors, performance modeling faces a set of common challenges (tradeoffs) and trends. Figure 2.1 summarizes the tradeoffs in the field of performance modeling. In general, as the level of the abstraction increases, more details are omitted/or more assumptions are made, hence, the accuracy of the model decreases. However, the lower the level of details employed to build a performance model, the more accuracy achieved, but, with a significant decrease in the estimation speed. This is true whether considering simulation-based modeling or analytical-based modeling.

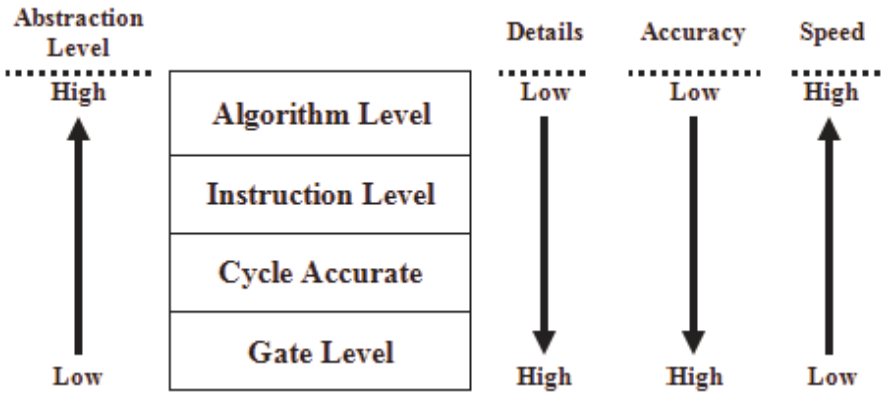


Figure 2.1. Performance Modeling Tradeoffs

Considering that, generally, each level of abstraction cannot achieve better accuracy than its adjacent lower level of abstraction, or faster estimation speed than its adjacent higher level of abstraction, efforts in performance modeling have concentrated on minimizing

the tradeoffs by either 1) modifying higher level of abstraction techniques to employ more details to achieve more accuracy than what they traditionally achieve to be as close as possible to the accuracy of the adjacent lower level of abstraction, (e.g. time annotation), or 2) simplifying the lower level of abstraction techniques to achieve faster estimation than what they traditionally achieve, to be as close as possible to the speed of the adjacent higher level of abstraction, (e.g. parallelism in cycle accurate simulation).

In addition to the accuracy and speed factors, it is desirable for performance modeling techniques to posses features such as structural simplicity, user friendliness, lower development cost, lower setup time, and flexibility to target new systems and applications. Such factors add still more tradeoffs in performance modeling techniques.

The first section in this chapter presents an overview of performance evaluation techniques used in processor design. The following section discusses general hardware and software performance evaluation techniques used in embedded system design. The third section discusses existing performance modeling techniques developed to assist in the processor selection decisions at early stages of an embedded system design – similar to the framework developed for this research. The final section presents a summary.

2.1. Performance Evaluation for Processor Design

In the development of new microprocessors, it is very important to validate the functionality and the performance of processors-under-development before proceeding in the design process or releasing the products into the market. Hence, performance evaluation is involved in several stages of the design, where going backward in the design process is very costly. A major challenge in processor performance evaluation is that "**one second of program execution on these processors involves several billion instructions and analyzing one second of execution may involve dealing with tens of billions pieces of information**" *L. John* [15].

Based on the classification shown in [15], performance evaluation can be classified into two main categories: performance measurements, and performance modeling. Performance measurement aims to emulate/verify the architectural functional and timing behaviors under a set of benchmark programs that characterize the target application domains at run-time. This technique is only possible if either actual system or its prototype, where the RTL design of the architecture-under-development is ported into FPGAs [14, 16], are available. By stressing the architecture with intensive workloads, different events/signals can be monitored to explore the bottlenecks in the design. In case of running workloads on a prototype, tools like Xilinx-ChipScope [17] can be used to trace the status of the processor [16], or alternately, the signals-of-interest can be routed externally for monitoring [14]. In case of using an actual architecture, performance

measurements can be implemented either at the hardware-level by accessing the on-chip counters (if equipped/available) or using logic analyzer to trace the machine status at the events-of-interest, captured via interrupts, or at the software-level using code instrumentation or software drivers. While using on-chip performance counters is considered very fast, it is limited to certain events that are actually accessible by such counters. Interrupts and code instrumentation can be implemented to measure a number of different events but they add a significant performance overhead – as they interfere intensively with program execution [11, 13, 15].

The second class of performance evaluation, that of performance modeling, is concerned with evaluating the performance of architectures-under-development. Performance modeling can be employed at the early stages of the design process where the actual architecture is not available and it is expensive to prototype all possible design choices, or if the signals-of-interest are hard measure on the actual hardware. Performance modeling can be further classified into analytical-based and simulation-based. Analytical modeling [18-19] has rarely been used due to the accuracy requirements and the level of implementation details needed for it to be employed [15, 20]. However, at the earliest stages of development, analytical modeling can be used as a decision support technique [20].

A framework dubbed the Processor Evaluation Cube (PEC) which helps in classification and comparison of a range of processor evaluation techniques is proposed in [21]. The

three axes of the PEC are: analysis, architecture and abstraction where each axis consists of two distinct points (see Figure 2.2). The analysis axis distinguishes methods employing static analysis or simulation; the architecture axis distinguishes methods evaluating single processor or multiprocessor target architectures; the abstraction axis distinguishes methods employing cycle-true evaluation or higher level execution time estimation techniques. The authors observed that **in most cases modeling the performance of single processors falls in the performance evaluation context and in few cases the selection is mentioned as an explicit goal of the performance estimation techniques** - in contrast with multiprocessor performance modeling which focuses in the selection process considering the performance of each single core has been implicitly modeled. In the research efforts surveyed in [21], it is shown that most (about two thirds) of the efforts targeting single processor performance estimation fall in the non-cycle-true static analysis category while, on the other hand, around half of the surveyed efforts targeting the performance estimation of multiprocessor architectures fall in the non-cycle true static analysis category. The remainder of the efforts fall on the (non or) cycle-true simulation categories. The survey argues that many different evaluation techniques may fall in the same category in PEC classification, though significant differences in performance analysis can be found to distinguish between them. The study also remarks that some techniques may fall in more than one category because they actually include the two options in an axis, nevertheless; based on the interpretation of these techniques, these could be classified into a single category. Finally, the authors point to the ability of adding more points to the existing axes (or adding more axes) for

tightening category boundaries, or expanding the classification paradigm to include new criteria.

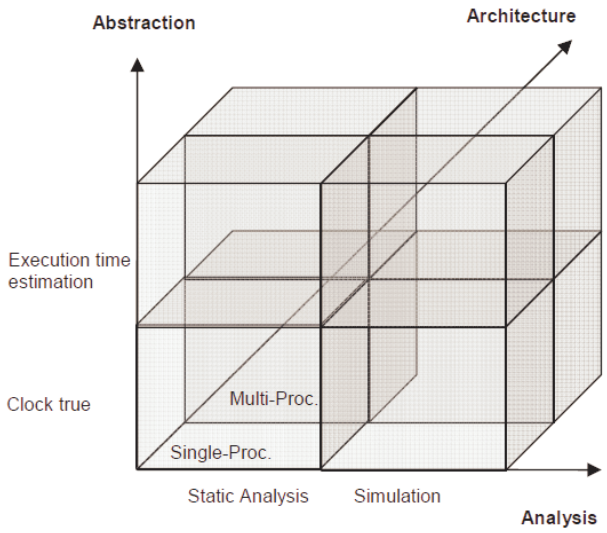


Figure 2.2. Processor Evaluation Cube [21]

In the area of simulation-based performance modeling of processors, cycle accurate simulation is considered the dominant technique used for hardware design space exploration [15, 20, 22-23]. Unlike functional simulators which are used to simulate only the functional behavior of an application on a certain processor, cycle accurate simulators [24] are concerned with handling both functional and timing behaviors of the target architecture, which in turn requires representing the impact of the micro-architectural details into the simulator structure, adding more complexity and slowing the simulation speed. Methodologies and techniques for cycle accurate simulation have been surveyed in [22]. The study argues that simulation speed and the level of accuracy are the most

important factors in the development of cycle accurate simulators where the more architectural details represented in the model and the larger benchmark set used for validation, the more accurate the results achieved. However the greater the level of detail, the slower simulation speed. Hence, such tradeoffs limit the efforts for developing cycle accurate simulators [15, 20, 22-23]. Techniques such as reducing the input set and using the sampling approach have been adopted to reduce the simulation workload [25-26], thus increasing the simulation speed, however, such techniques have direct impacts on the simulation accuracy. Recently, there have been interests in employing different parallelism (and partitioning) techniques for cycle accurate simulators to enhance the simulation speed by using multicore processors [27-31], or by using FPGAs to accelerate the time intensive portions of the cycle-accurate simulation [32-34]. However, such techniques face difficulties such as: 1) how to partition the jobs for maximum parallelism, 2) how to minimize the communications overhead, and 3) how to avoid deadlocks among shared resources [35-36].

The objective of the modeling techniques described above is to model/measure the performance of architectures-under-development at several stages of the development process for use by the processor developers themselves. On the other hand, this research aims to model the performance of an existing architecture that is already built, for the end user running an actual application. Hence, no direct comparison can be made between the framework developed in this research and the previously discussed modeling techniques. However, determining an application's

dominant operations using the techniques used this framework could be very valuable in determining the processing bottlenecks for the development of an application-specific instruction-set processor (ASIP).

2.2. Performance Evaluation for Embedded Systems Design

While desktop processors are generally very complex and hence need complex performance modeling techniques, factors like time-to-market, time critical/complex functionalities, and heterogeneous application domains have made estimating the execution time of software applications on embedded systems a crucial issue. To overcome the issues regarding traditional cycle accurate modeling techniques such as speed and complexity, new software performance analysis have been developed embedded systems design. Some of these methods are described in the following sections.

2.2.1. Open-Source Cycle Accurate Simulation for Embedded Processors

SimpleScalar [24] is an open source cycle accurate simulator which is widely used in academia. While it supports Alpha, PowerPC, and ARM instruction set architectures, it assumes a fixed pipeline structure and timing delays. Since it is implemented using very low level C-code with many macros, an extensive validation is needed whenever it is

modified with different parameters [37], which in turn makes it hard to retarget to a new processor architecture.

2.2.2. Time Annotation Techniques

In [38], timing delays for instructions are annotated from low level models (cycle-level models) back to the application source code. Hence, the original application can be simulated without the underlying architecture details with orders of magnitude faster run times and good level of accuracy. Another timing annotation technique called compilation-based simulation is shown in [39], where the assembler code is annotated with the timing delay (execution cost) of instructions that can be obtained from the datasheet or benchmarking. This allows obtaining timing information by running the application code on the host machine and simply adding up the number of cycles for each instruction instead of using cycle accurate simulation. Of course both of these techniques have reduced accuracy because the effects such as the interaction between instructions due to micro-architecture details are not accounted for.

2.2.3. Statistical Analysis Techniques

In [40], a non-linear Lazy statistical method is employed to predict the performance of embedded software running on the SPARC architecture. The performance model is defined as a set of functional models mapped into a set of possible architectures with

different memory latency and CPU speed parameters. A program's main parameters are extracted using an instruction-level profiler (SPARC-IPROF), and then a cycle-accurate simulator (TSS, built in C) is used to obtain the performance of each application/architecture pair. The model is then able to estimate the performance of an application on a certain trained processor configuration. Another statistical approach is shown in [41] where the source code is translated into simplified virtual instruction set, allowing program parameters can be extracted by running the application on a virtual instruction set simulator while performance estimates can be obtained using cycle accurate simulation. Linear regression is then used for building the prediction model.

Modeling the performance of an application on a specific processor whose cycle accurate and functional simulators have been tested using a set of benchmark programs is proposed in [42]. In this technique, linear regression is used to model the impacts of the application's parameters on the system performance. A total of 183 benchmark applications, most of them are DSP applications, have been used to train the model. The functional simulator has been enhanced to allow counting various events (e.g. counting different types of instructions and possibly also cache and memory accesses). The study shows a number of experiments on each one of which new parameters have been used to study how accurate the model is when employing such parameters. For different parameters sets, mean absolute errors ranging from 5.44% to 38.8%, (std. deviation = 7.12 to 57.7), with a maximum errors ranging from 26.31% to 518%, have been reported when testing DSP applications on ARM v5 implementation. This work shows that

increasing the number of parameters does not always lead to an increase in the performance modeling accuracy, hence, the efficiency of the selected parameters for modeling an application has to be considered for more accurate results and a more efficient regression model. Moreover, programs that had small size reported larger errors because their behavior can be far away from the general observed statistics that govern the performance prediction model.

2.2.4. Static Analysis of Embedded Software

In some real-time embedded systems applications, static timing analysis is mainly used to explore the worst case execution time (WCET) by detecting all possible scenarios for a program execution which in turn allows predicting the time cost for the worst case flow of the target program (or tasks). This is important for schedulability analysis, and to check the safety of the system to be certain that some circumstances cannot cause a system timing failure. Typically, WCET depends on analyzing the disassembled binary executable code to determine the structure of the program and to determine how each basic block of the program interacts with the hardware resources [8, 43-46]. A major issue in WCET analysis is that loops and conditional statements must be pre-determined, i.e. any dependency of the program's execution on real-time inputs/conditions should be pre-solved [8, 46]. Furthermore, as the design architecture and constraints become more complex, the number of the states increases quickly which in turn makes the analysis more difficult and time consuming.

Processor selection is not a goal of the performance modeling techniques discussed up to this point, instead, these techniques aim to replace the traditional cycle accurate simulation techniques used in the hardware design space exploration with new high-level, faster modeling approaches. However, techniques mentioned in [40-42] are among the techniques which have inspired this research's efforts.

2.3. Embedded Processor Selection based on Performance Estimation

A processor's performance, typically, can be characterized by a set of features like: clock rate, built-in functionalities (supported assembly instructions), level of pipelining, cache architecture, etc. Maximizing such features is considered to be one of the options to obtain more performance. However, unlike a desktop processor where "faster is better", in embedded systems, the design philosophy states that "**fast enough is good enough**" *T. Conte* [2], where, factors like cost, size, power consumption, heat production, limited memory resources and other physical characteristics constrain the designer's processor choices. In addition, in selecting a processor for a certain application, the non-functional requirements of the processor such as the portability and the user's familiarity with its development tools may dominate the processor selection decision [47-49]. An example of non-functional requirements affecting the processor selection decision can be found in [50].

2.3.1. Traditional Techniques

A study of the methods and guidelines for embedded processor selection is shown in [47]. The study found that the largest factor influencing an off-the-shelf processor selection decision is the performance, followed by other factors such as cost, and product time-to-market. The consideration of these factors, as the study shows, are typically based on a set of criteria such as benchmarking results, system requirements and resources, and the designer's familiarity with the development tools. In [2, 4, 47, 51], a set of tips based on the authors' experiences in embedded systems design are listed. These tips concentrate on key features of processor architectures that have a direct impact on the performance of the systems, such as the processor word size and the memory architecture. Such tips are very helpful when the embedded system designer has a deep knowledge of the application functional behavior/demands and processor architecture.

In [49, 52], a database of microprocessors is created to help in selecting an appropriate processor based on general specifications such as clock rate, memory resources, power consumption and price. Using this database, the designer can trade between the different features and narrow down the search space for the desired architecture.

While execution time cannot be indicated using only the clock rate of the system, it is important to know how many operations the CPU is capable of accomplishing per unit of time (or how many cycles each instruction needs to execute), measurements like MIPS

(millions of instructions per second) and MFLOPS (millions of floating point operations per second) have been introduced to help in evaluating the performance of processors. Such criteria indicate a processor's general performance. However, for different types of processors and different types of applications, MIPS/FLOPS may mislead the performance evaluation due to the different amount of work accomplished by different instruction set architectures (ISAs) for applications with characteristics [15, 53].

Reviewing the recorded performance measurements of different benchmark programs published by a processor's manufacturer can indicate the performance of the candidate processor in the same domain of applications. The accuracy of the performance estimation depends on the level of the representation (i.e. instruction distribution, branchs, cache behavior …etc) of the target application in the benchmark suite. However, designing a benchmark suite for embedded applications, is itself a very challenging task because benchmark programs should be non-redundant and comprehensive while at the same time, covering a wide domain of embedded applications which are very diverse [15, 20, 22-23, 54]. Currently, few benchmark programs are available for targeting embedded systems. The most popular embedded benchmark suites are the free open source MiBench [54], and the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) [55].

Using Rate Monotonic Analysis (RMA) to evaluate processors for use in real-time embedded applications is shown in [48]. This technique depends on high-level analysis of

the target architecture and the pseudo code of the target algorithm. Hence, a background in how compilers generate code is necessary to obtain a better evaluation. In this approach, the target algorithm is assumed to be divided into a set of tasks, where each task is assigned a run time value (duration) and frequency value (period). Based on analyzing the total utilization of the processor for all tasks, a decision can be made to accept or reject the candidate architecture. The pessimistic maximum allowed load is suggested to be 70% of the total processor load; however, to account for forgotten tasks and ones that are added at design time based on new requirements, the author suggests that a load of as low as 35% can be considered a fairly reasonable initial value. This technique needs the designer to have deep knowledge in both the functional behavior and time constraints of the target application and the microarchitectural details of the execution platform, which, at the early stages of the design process are hard to extract without having the software development tools of the candidate architectures (compilers, simulators, etc.).

2.3.2. Analytical Techniques

Techniques for performance estimation by evaluating the degree of matching between algorithm requirements and the processor architecture resources, is shown in [3, 56-57]. For example, in [3], a requirements matching technique was shown predicting the most suitable architectures, in terms of general performance (goodness), for three out of four MiBench algorithms. In this technique, a set of correlation functions are applied to the

algorithm/processor requirements/characteristics to estimate the utilization of the architecture resources, where, typical algorithm properties are extracted using platform-independent intermediate representation (SUIF), and the corresponding architecture is characterized using features such as the instruction set, pipelining, branch support, etc, where, due to the complexity, the cache factor is ignored.

In [58], the objective is to select the best architecture among a set of possible candidate processors (microcontrollers, DSPs and RISC microprocessors). The target application is modeled in an object oriented environment, where intermediate descriptions of the code are generated for three virtual machine types; microcontrollers, DSPs and RISC microprocessors. Next, by analyzing the application's characteristics such as the number of jumps, arithmetic and memory access operations, etc., the application behavior is classified into one of the following domains: 1- control intensive: many control instructions like a finite state machine (FSM), 2- data intensive: computations accomplished on internal registers like digital filters, or 3- memory intensive: such as list processing. Depending on this classification, a microcontroller, a DSP or a RISC architecture is selected for the implementation. The study does not show an estimation of the execution time but gives relative match of an application among a set of architectures.

In [9, 59], a performance analysis technique using an intermediate representation called Low Level Virtual Machine (LLVM) is proposed where the performance is indicated as an estimation to the number of executed instructions. In this technique, each LLVM

intermediate instruction and library function is correlated to the number of executed instructions on the target architecture that can be obtained by running executable code on an instruction set simulator for the target architecture. A mathematical model to estimate the performance of standard library functions is proposed by relating the number of the executed instructions for each library function to the size/number of parameters passed. By analyzing the functional behavior of the library functions, the study classifies them into two groups, input independent, where the performance is the same regardless to the input value, and input dependent, where the performance changes, linearly or non-linearly, based on the input nature. The proposed mathematical model has been verified by running the library functions a large number of times on a functional simulator and plotting the number of the executed instructions for each function against the input argument size/numbers. Although this approach is not shown to give very accurate results, it allows analyzing the performance of new applications at high-level without compilation and simulation tools.

In [60], a technique for embedded system performance estimation to assist in selecting a suitable processor for a given application is proposed based on application/processor architecture analysis. The evaluation is accomplished through two steps: 1- eliminating the unsuitable processors among a large set of processor candidates based on the application's and the processor's key features, 2- estimating the number of cycles needed to implement the target application on the rest of processor candidates. In this technique, the application (written in C) is profiled to determine the number of iterations for each

block of the source code. Then, the target application is translated into an intermediate format using SUIF (Stanford University Intermediate Format) to extract the main parameters characterizing the application (e.g. number of concurrent load/store operations). Using simple processor description format, each processor architecture is characterized through a set of specifications such as type/number of functional units and number of registers. Finally, analytically, the application code size and execution time are estimated based on the extracted profiling results, application parameters, and hardware characteristics. The authors reported estimates within 30% of the results obtained from lower level tools.

A neural network-based performance estimation technique is proposed in [61-63], similar to the concept in [42]. Figure 2.3 is a general outline of this approach, which eliminates the need to execute a specific target application on a cycle-accurate model of the processor in order to generate a performance estimate. However, a cycle-accurate model of the processor is still needed to determine the performance estimate for the set of benchmark applications used to train the regression model or the neural network. Because of the non-linear impacts of the hardware architectural details (such as pipeline, cache, and branch prediction units) on the system performance, the authors have proposed estimating such relationships using a neural network. The neural network is trained using the results of running a set of programs on a cycle accurate simulator (measuring the time) and a functional simulator with profiling capability (measuring the number of load/store, branch, integer and floating point operations). To estimate the performance of

a new application, the application needs to be run only on the functional simulator. Then, the extracted dynamic numbers of instructions for the target application are fed into the trained neural network for performance estimation. An application domain classification technique, by which applications are classified as control-flow or data-flow applications, has been used to enhance the estimation accuracy. Using 41 benchmark programs, this approach shows some level of estimation speed-up (resulting from the need to only execute the target application on the faster, functional processor model), with a moderate accuracy level [61]. For example, a speed-up up to 190 times in comparison with cycle accurate simulator has been achieved for the superscalar PowerPC-750 architecture where the estimation mean error was 6.41% with a maximum error ranges from -32.41% to 25.87% for the data flow domain (std. deviation = 9.54), and a mean error of 7.62 with a maximum error ranges from -49.37% to 24.96% for the control flow domain (std. deviation = 12.46). A higher level of accuracy has been achieved when testing the ADSP 218x processor, where for generic domain the results were 2.42% (avg), 4.89 (std. dev.), -18.1 to 18.88 (max error). The accuracy of the estimation results depends not only on the strength of the neural network structure but also on the size of the data set used to train the neural network and the selection of the parameters that govern the system and the processor architecture complexity. Moreover, the need for a cycle accurate simulator (and the software development tool for each platform) to train the neural network is considered a main obstacle of using this technique for algorithm/processor selection purposes.

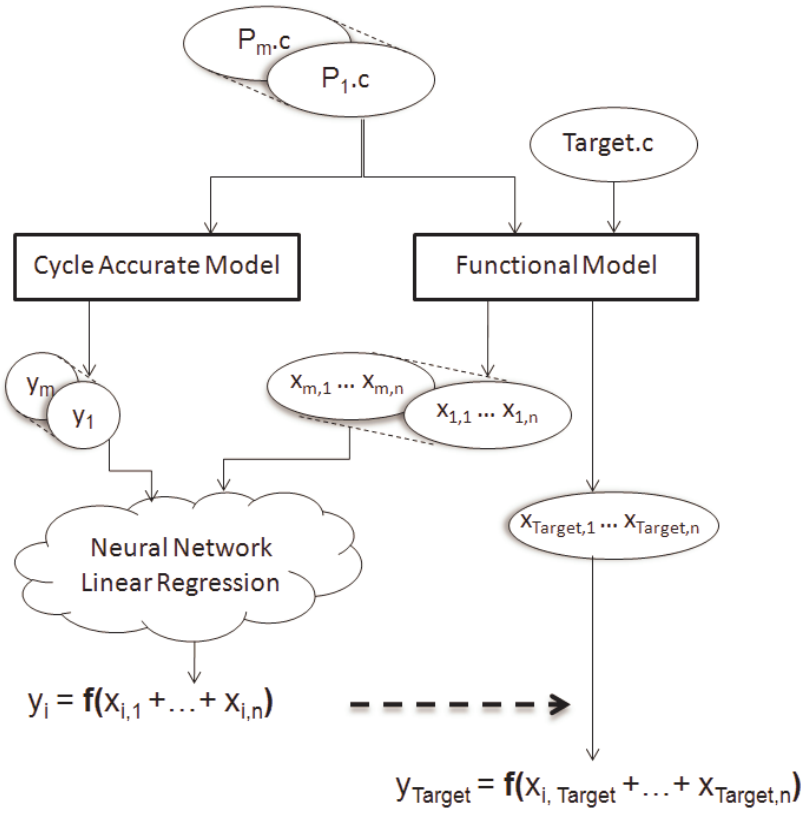


Figure 2.3 Statistical Analysis Techniques (neural network [61] or linear regression [42])

2.4. Summary

While the traditional techniques described in Section 2.3.1 above are generally simple, they do not give any statistical information that explains quantitatively how the presence or absence of certain features in the processor or requirements in the application affect the performance of the system. This lack of information, in turn, makes evaluating system alternatives a very hard task. On other words, such techniques can be used for decision support but cannot be used for detailed performance estimation purposes.

On the other hand, the analytical techniques described in Section 2.3.2 require running the target algorithms in a simulation environment, which inherently requires using the compilation tool of each candidate architecture. Furthermore, problems like integrating the simulation tool with other system components constraint such techniques to off-line analysis. It is clear that the neural network technique has been able to predict the performance of the target architecture with an acceptable level of accuracy using the functional parameters of the target algorithms. However, the need for the software development tools (compiler, functional simulator) for each target architecture significantly increases the cost and the setup time.

The approach developed in this research is similar to the neural network approach, however, linear regression for building the prediction model is used here in lieu of neural networks. However, the most important distinction is that in the technique developed in this research, no software development tools for target architectures are needed. Instead, the MicroBlaze platform is used as a reference processor, through which it is possible to predict the performance of foreign architectures. The use of the MicroBlaze as a reference processor also allows evaluating the performance of the application at the source code level with only the MicroBlaze software development tools being required.

Chapter 3 Problem Analysis

This chapter is organized as follows; section 3.1 provides a general background on the problem, section 3.2 discusses the theoretical analysis of the framework developed in this research, section 3.3 compares the developed framework directly to the most closely related existing methods, and section 3.4 gives a summery.

3.1. Background

In embedded systems, software has become a dominant design factor. As a result, modeling the performance of running software is one of the main design challenges, especially at a high-level of abstraction [64]. Moreover, considering the enormous number of ISAs and microarchitectures available for embedded systems makes this modeling a very challenging task. In [65-69], system-level design tools are provided which support a number of ISAs and microarchitectures. However, such tools are hard to extend or retarget [38]. Simulation tools such as [24] offer cycle accurate measurements, but they target specific ISAs and require avery low level hardware details, hence, they are hard to modify, even for the same ISA. Augmenting ISSs with timing information [38-39] requires compiling the target algorithm with the corresponding architecture's compiler, and analyzing the binary code and the hardware architecture, which in turn, requires disassembling the executable code and validating the ISS against cycle accurate models. Modeling new (or existing) ISAs or microarchitectures using architecture

description languages (ADLs) such as [70-74] is not within the scope of this research. Although such languages allow automatic generation of software development tools, such techniques are very time consuming since they require a complete specification of the target architecture.

As mentioned in Chapter 1, the fundamental question is whether absolute accuracy or relative accuracy is needed? While absolute accuracy has been always the ultimate goal for performance modeling, factors like time, cost, complexity, and the limited resources at the early stages of an embedded system design lead to the conclusion that an analytical framework that is able to achieve a relative accuracy to be a very useful resource. A challenging problem is how to collect quantitative statistics for the target algorithm to determine the dominant operations and the demanded resources that can assist in evaluating the performance characteristics of the system. The framework developed for this work and described herein, works at high-level of abstraction (C-level) where no ISA or microarchitectural details are required to model the performance of the candidate architectures, i.e., an algorithm written in C can run on a reference model, and quantitative statistics regarding system functional behavior can be collected using the developed profiling technique to assist in evaluating the performance of the target algorithm/architecture.

Implementing different ISAs and/or different microarchitectures in order to attempt to exploit their advantages for a certain class of applications has led to the development of an enormous number of embedded processors. Despite of their differences, however, similar design concepts can be found in most processor's implementations due to key design principles that considered as a rule of thumb in processor architectures. A survey of embedded RISC processors [75], which considered a subset of five types of processors, has shown that in most cases 1) the instruction size can is either 16 or 32 bits, 2) the address space is 32 bits, 3) the integer registers are 32 bits, 4) the I/Os are memory mapped, and 5) integer instruction sets are very similar. Moreover, when analyzing the operation of embedded applications, it can be stated that a small portion of code consumes a large portion of the execution time which is compatible with the *Pareto* principle. By their nature, embedded applications tend to spend 90% of the execution time in 10% of the code, dubbed "90-10 rule" [76-77]. While different ISAs/microarchitectures produce different performance characteristics, similar hardware design principles tend to have analogous impacts on performance.

Regardless the processor's ISA and microarchitectural details, when running software, processors tend to implement the same functionality, with different timing behaviors (and perhaps different precision). Considering the performance represented by the number of consumed units of time to execute a certain program, it is difficult, or even impossible, to map processor performance to another processor performance that has different ISA or/and different microarchitecture, based only on the performance measurements (as a

single number). For example, assume that processor **A** consumes $\mathbf{y_{A1}}$ units of time to execute program **1**, and processor **B** consumes $\mathbf{y_{B1}}$ units of time to execute the same program, then, it is possible to find another program (or a sequence of instructions) that consumes $\mathbf{y_{A2}}$ where $\mathbf{y_{A2}} = \mathbf{y_{A1}}$, while it is not mandatory that the new program will consume $\mathbf{y_{B1}}$ where $\mathbf{y_{B2}} = \mathbf{y_{B1}}$. Hence, the relationship between $\mathbf{y_{A1}}$ and $\mathbf{y_{B1}}$ that holds for the first program does not hold for the second program. However, if we analyze the logical execution of the running program at high-level (source code), both processors tend to follow the same logical execution flow to implement the same functionality for a given algorithm.

Since the performance of running software is a reflection of its functional behavior on a certain architecture, then, it is possible to evaluate the performance of an application by analyzing its functional behavior. The problem here is that although programs may follow the same functional behavior at a high-level, the real execution of programs is at a low-level in which different ISAs/microarchitectures have different specifications that lead to different execution paths, instruction flows, and timing constraints. The following section discusses the approach developed in this research to handle this issue.

3.2. Theoretical Analysis

The performance of an application on a certain processor is a result of the timing constraints of the algorithm's execution on the processor's microarchitecture, i.e., the timing behavior is characterized by the instruction flow through the hardware resources, which is subject to both logical and physical constraints. Hence:

$$\text{Performance} = f(\text{Functional Behavior, Microarchitecture}) \quad (3.1)$$

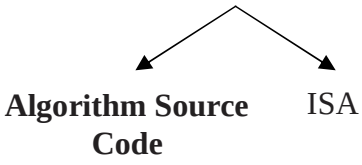
where performance is represented by a single number that represents a timing measurement (number of cycles or units of time), the functional behavior is the instruction flow of the target algorithm (at assembly-level), and the microarchitecture is the hardware (physical) implementation such as pipeline architecture, memory elements (including registers, cache and memory), system clock, …etc.

At the assembly level, a single program, before execution can be thought of as a set of **N** instructions, where the ordering between instructions represents the logical structure of the program. If the instruction I_i is the i-th instruction in the program structure, considering the address of instructions as a part of the instruction signature, then, for $k \neq j$, $I_k = I_j$ is not a valid expression, although they may have the same instruction type and the same operands, the address cannot be identical. On the other hand, a single program running on a processor can be thought of as an ordered trace of **M** instructions, where the

ordering between instructions represents their logical execution order (**M** could be less than, equal to, or greater than **N**). If the instruction I_i is the i-th instruction in the trace, considering the address of instructions as a part of the instruction signature, then, for $k \neq j$, $I_k = I_j$ is a possible option, i.e., an instruction can be called more than one time in the same program. However, for different ISAs, the quantities **N**, **M**, the type of instructions, and instruction flow are different since they are ISA-dependent.

The functional behavior (as expressed in 3.1) of executing an algorithm on an ISA can be represented by the following expression:

$$\text{Functional Behavior} = f(\text{Algorithm Assembly Code}, \textbf{Inputs}) \qquad (3.2)$$



The algorithm assembly code is the translated version of the source code based on the target ISA, which is platform-dependent, where the source code is the program written at high-level language (such as C) that decides the logical execution of the program (platform-independent). The inputs are the incoming data in through input ports. Inputs and Algorithm Source Code are bolded in the above expression because they can be considered common factors in our analysis.

At the level of source code, a single program before execution can be thought of as a set of **N** statements, where the ordering between statements represents the logical structure of the program. If statement S_i is the i-th statement in the trace, considering the logical order of the C statements, then, for $k \neq j$, $S_k = S_j$ is not valid for any j-k. On the other hand, a single program running on a processor can be thought of as an ordered trace of **M** statements, where the ordering between statements represents their logical execution order. If statement S_i is the i-th statement in the trace, considering the logical order of the C (i.e., the source code) statements, then, for $k \neq j$, $S_k=S_j$ is a possible option. Unlike assembly-level analysis, at the C-level, the quantites **N**, **M**, the statements, and the possible paths of program execution tend to be identical regardless the ISA of the target platform (ISA/platform-independent).

Although the real execution of a programs is at assembly level (binary executable code), analyzing a program at high level (or using an intermediate representation) to determine the logical functional behavior can assist in evaluating its performance [3, 9, 41, 56-57, 59-60]. As mentioned previously, in this research, the goal is not to build a cycle accurate model, since this requires, at least, having the target algorithm compiled into the target ISA, disassembling the binary code, and building a detailed microarchitecture model, instead, rather, analytical modeling is used to discover the relationship between the application's functional behavior and the processor's performance in executing it.

Figure 3.1 shows a general layout of analytical-based performance modeling. A performance relation can be defined as:

$$\mathbf{X} \times \mathbf{PM} \rightarrow \mathbf{y} \tag{3.3}$$

where **X** represents a set of input statistics ($x_1 \ldots x_n$), **PM** is the performance model, **y** is a single number that represents the estimated performance (number of cycles or units of time). As can be seen, inputs to a performance model can be functional statistics that characterize the behavior of an algorithm at assembly level, high level, or using an intermediate format representation. Such statistics can be obtained by profiling the target algorithm and, typically, report the frequency of events that have impacts on performance, such as number of times each instruction has been called, cache misses/hits …etc. On the other hand, the performance model represents a set of mathematical relationships the correlate **X** to **y**.

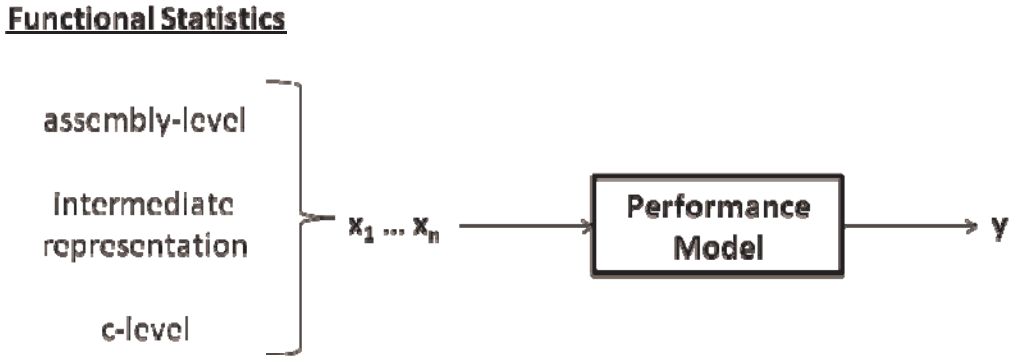


Figure 3.1. A Layout for Analytical Performance Modeling

Recalling expressions **(3.1)** and **(3.2)**, an application performance on a certain architecture can be expressed by:

$$\text{Performance} = f(\textbf{Algorithm Source Code}, \text{ISA}, \textbf{Inputs}, \text{Microarchitecture}) \quad (3.4)$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{\text{Functional Behavior}}$$

In the performance model layout shown in Figure 3.1, certain details that have impacts on performance are hidden. For example, considering the functional statistics that are available at assembly-level of the target architecture's ISA, the hidden information is related to the microarchitectural details (the fourth argument in expression 3.4). On the other hand, considering the functional statistics are gathered at a high-level or using an intermediate representation, more information is hidden in the model regarding the target ISA specifications. Consequently, the more details are hidden; either the more assumptions are made, hence less accuracy is achieved, or/and the more complex the model is.

Based on the above discussion, the following relations can describe the performance of a certain program-x (P_x) written in the C language on two different architectures, architecture$_1$ that implements ISA_1, and architecture$_2$ that implements ISA_2, where IR_x is an intermediate representation for the program, and $A_{x\_ISA(i)}$ is the assembly-level of the P_x using ISA_i:

For architecture$_1$:

1- At the assembly-level:

$P_x \rightarrow A_{x\_ISA(1)}$

$A_{x\_ISA(1)} \rightarrow$ Functional Behavior$_{1.0}$

Functional Behavior$_{1.0} \rightarrow y_{1.0}$

2- Using an intermediate representation

$P_x \rightarrow IR_x$

$IR_x \rightarrow$ Functional Behavior$_{1.1}$

Functional Behavior$_{1.1} \rightarrow y_{1.1}$

3- At the high-level:

$P_x \rightarrow$ Functional Behavior$_{1.2}$

Functional Behavior$_{1.2} \rightarrow y_{1.2}$

For architecture$_2$:

1- At the assembly -level:

$P_x \rightarrow A_{x\_ISA(2)}$

$A_{x\_ISA(2)} \rightarrow$ Functional Behavior$_{2.0}$

Functional Behavior$_{2.0} \rightarrow y_{2.0}$

2- Using an intermediate representation:

$P_x \rightarrow IR_x$

$IR_x \rightarrow$ Functional Behavior$_{2.1}$

Functional Behavior$_{2.1} \rightarrow y_{2.1}$

3- At the high-level:

$P_x \rightarrow$ Functional Behavior$_{2.2}$

Functional Behavior$_{2.2} \rightarrow y_{2.2}$

Remarks:

- While **y** represents a single number, this number cannot be used to map the performance of an architecture to other architectures, i.e. for different programs $y_{1.i}/y_{2.i}$ is not a constant.

- **y** is a dependent (single) variable, and **X** is a set of independent variables that characterize the Functional Behavior$_{j.i}$ which can be extracted by running/profiling P_x, IR_x, or $A_{x\_ISA(i)}$.

- **X** → **y** is a unidirectional relationship, i.e., **X** cannot be extracted from **y**.

Assumptions:

- While P_x and IR_x can be considered platform–independent, and $A_{x\_ISA(i)}$ is platform-dependent, then:

$$\text{Functional Behavior}_{1.0} \neq \text{Functional Behavior}_{2.0}$$

$$\text{Functional Behavior}_{1.1} = \text{Functional Behavior}_{2.1}$$

$$\text{Functional Behavior}_{1.2} = \text{Functional Behavior}_{2.2}$$

- Considering $P_x \rightarrow IR_x$ and $P_x \rightarrow A_{x\_ISA(i)}$ as bidirectional relationships, so that:

$IR_x \rightarrow P_x$ and $A_{x\_ISA(i)} \rightarrow P_x$ are valid relationships, then:

$$A_{x\_ISA(1)} \rightarrow P_{x'} \text{ , and } A_{x\_ISA(1)} \rightarrow IR_{x'}$$

$$P_{x'} \rightarrow \text{Functional Behavior}_{2.2'} \text{ , and } IR_{x'} \rightarrow \text{Functional Behavior}_{2.1'}$$

$$\text{Functional Behavior}_{2.2'} \rightarrow y_{2.2'} \text{ , and Functional Behavior}_{2.1'} \rightarrow y_{2.1'}$$

As a result,

$$A_{x\_ISA(1)} \rightarrow y_{2.2''}, \text{ and/or } A_{x\_ISA(1)} \rightarrow y_{2.1''}$$

$$\text{or } A_{x\_ISA(1)} \rightarrow y_{2.3}$$

In general,

$$A_{x\_ISA(1)} \rightarrow y_{i.2''}, \text{ and } A_{x\_ISA(1)} \rightarrow y_{i.1''}$$

$$\textbf{Or } \mathbf{A_{x\_ISA(1)} \rightarrow y_{i.3}}$$

where i denotes to $architecture_i$. The use of (‘) is to indicate the abstract in information/details resulted from the made assumptions, where $y_{i.1''}$ or $y_{i.3}$ represents the performance estimation of correlating the assembly-level functional statistics of an architecture to the performance of other architectures.

The above discussion leads to the consideration that functional statistics of a certain architecture to be thought of (with respect to other architectures) as 1) semi-high-level statistics, 2) semi-intermediate representation statistics or simply, 3) a reference format

that can be employed to estimate the performance of other archirectures. In other words, the functionality of a certain ISA can be mapped to other ISAs' functionalities which in turn can be used in the performance analysis of those other ISAs/microarchitectures.

3.3. Application Functional Statistics

In this research, functional statistics are those statistics that indicate the functional behavior of the running program which depends on the logical instruction flow rather than processor microarchitecture. Hence, statistics related to CPU core architecture (e.g. pipeline stalls), cache/memory architecture (eg. number of misses/hits) should not be among the functional statistics. This is important to eliminate the special impacts of the reference model on the performance model.

The functional statistics adopted in this research basically represent number of operations such as load/store, unconditional branch, conditional branch, return, simple ALU (eg. add, and), integer multiply, integer divide, and single/double-precision floating point operations (classified based on the cost of each operation). Such classification of operations eases the role of the regression modeling for handling the different time costs for each group of instructions. For example, in general, integer divide instructions cost more than integer add instructions, floating point operations cost more than integer operations, especially when there is no hardware support for floating point operations.

In addition to a possible deference in cost, the importance of classifying branch instructions into unconditional, conditional and return is that an unconditional branch (and return) are normally used to jump to (from) routine functions which can indicate jumping to a far distance. On other hand, conditional branches are used in the conditional statements (such as if, for loop, …etc) which normally indicate code execution close to the address of the branch instruction. Hence, such classification assists in accounting for the general behavior of cache/memory architectures which affect the performance significantly.

3.4. Linear Regression

In this research, $x_{i,1}…x_{i,n}$ are collected functional statistics from the reference model (MicroBlaze) for a program i, while y_i represents the real performance measurement on the actual target hardware/model (see Figure 3.2). It should be noted that the proposed framework does not consider $x_{i,1}…x_{i,n}$ to be identical with the functional statistics of the target architecture, however, it is assumed that such statistics can be correlated to each other.

For m number of observations (programs) and n number of features (parameters):

$$(y_1,x_{1,1},\ldots, x_{1,n}) \ldots (y_m,x_{m,1}, \ldots, x_{m,n})$$

a linear regression model can be represented by:

$$\boldsymbol{y} = \mathbf{XC} + \boldsymbol{\varepsilon} \tag{3.5}$$

where $\boldsymbol{y}$ is a vector of m dependent response variables (scalars), $\mathbf{X}$ is m-by-n matrix that consists of m number of rows in which each row has n number of independent prediction variables (features), $\mathbf{C}$ is a vector of n coefficients that govern the relation between $\boldsymbol{y}$ and $\mathbf{X}$, and $\boldsymbol{\varepsilon}$ is a vector of m random errors that model the uncontrolled features or experimental errors.

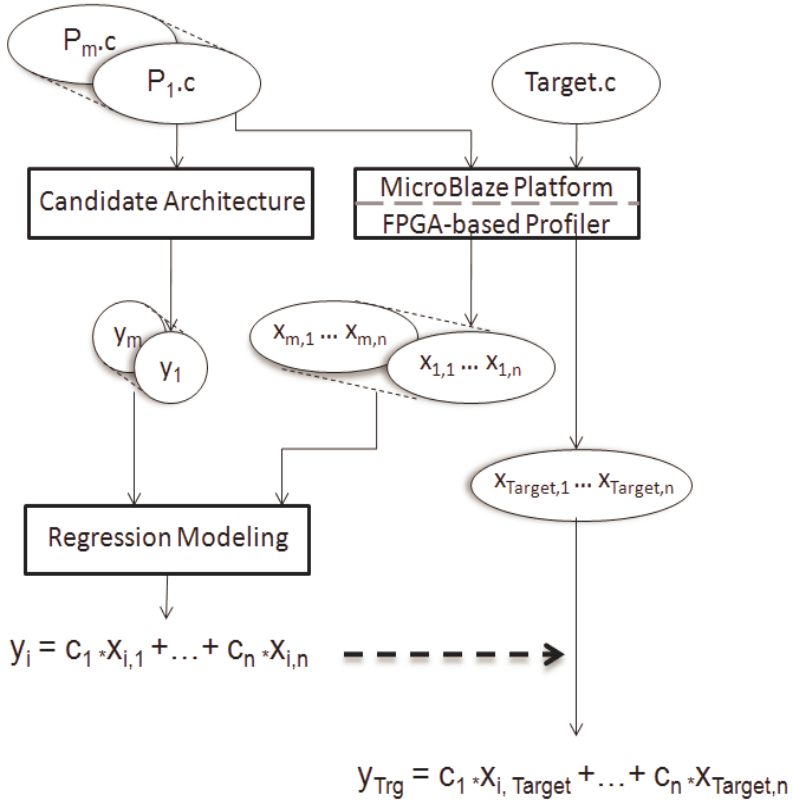


Figure 3.2. Framework Outline

Different methods can be used for fitting the model. For example, the least squares method works by minimizing the sum of the squares of the difference between the estimated and the actual values (errors or $\boldsymbol{e}$), i.e.,

$$||e||^2 = \sum_{i=1}^{m} e_i^2 \tag{3.6}$$

Linear regression using robust fit is less susceptible to outliers than normal least squares [78]. Robust fit implements a method called iteratively reweighted least squares where at the first iteration, each point is assigned equal weight and model coefficients are estimated using ordinary least squares. At subsequent iterations, weights are recalculated (using weighted least squares) so that lower weights are given to the points farther from model predictions in the previous iteration, i.e.,

$$||e||_w^2 = \sum_{i=1}^{m} w_i e_i^2 \tag{3.7}$$

Where *w* is the weight function. The process continues until the values of the coefficient estimates converge within a specified tolerance.

In this research, robust fit is implemented using the weight function *logistic* in which[4]:

$$w = \tanh(r) \,./\, r \tag{3.8}$$

the value r in the weight functions is:

$$r = \frac{e}{t\, s \sqrt{(1-h)}} \tag{3.9}$$

where *e* is the vector of errors from the prior iteration, t is a tuning parameter with a default value 1.205, s is an estimate of the standard deviation of the error term, and h is the vector of leverage values from a least-squares fit.

3.5. Prediction Models, Related Work

In [61-63], functional statistics regarding number of load/store, integer, floating point, and branch operations are collected using the ISS of the target architecture. The microarchitectural details of the target architecture are not modeled directly in this approach, instead, the performance model preserve the impacts of the microarchitectural details by training the model with a set of pairs of functional statistics and performance

[4] './' is array right division operation, i.e., A./B is the matrix with elements A(i,j)/B(i,j). A and B must have the same size, unless one of them is a scalar.

measurements for a domain of algorithms using neural network. A similar approach is shown by [42, 79] where more functional statistics regarding number of times each instruction has been executed and cache events are obtained using the ISS of the target architecture to build the performance model using linear regression. In [40], non-linear Lazy method is used to build a performance model where instruction-level statistics and different memory latency and CPU speed parameters form the input statistics for the model. In [41], the target algorithm is translated into virtual instruction set which consist of simplified RISC operations. Linear regression is used to model the performance based on the obtained virtual instruction statistics by running the algorithm on the virtual instruction set simulator. Other analytical methods using intermediate format representation for performance analysis are proposed in [3, 9, 56-58, 60].

The most related to the framework proposed in this research are [42, 61], see Figure 2.2 and Figure 3.2. However, unlike these approaches, different set of parameters are used here to ensure a better representation of the instruction set of the target architecture where instructions are classified based on the execution time rather than detailed (or general) instruction functionalities. Also, the proposed framework uses a real architecture to obtain the (real time) functional statistics to be employed, in opposite [42, 61], to model the performance of foreign ISAs and architectures. While it is hard to model a complete system by integrating an ISS with the other components of the system, integrating a real architecture that is capable of collecting real time statistics can handle this issue, although the functional statistics are not identical.

In [80-83], evaluating the performance of different hardware configurations is proposed where the inputs of predictive model are the architectural parameters. Neural networks technique is used in [80] while linear regression is the technique used in [81-82]. In [83], estimating the performance different versions of an ISA that consist of the same set of instructions a common (base) ISA has in addition to new set of customized instructions is proposed by running the target algorithm only on a cycle accurate simulator of the base architecture then analyzing the functional behaviors of the proposed ISA versus the base ISA functional behavior. However, it is not the scope of these techniques evaluating the performance of different algorithms/programs.

3.6. Summary

Unlike simulation-based performance analysis, "**analytic models can help to understand a system in ways that simulation does not**" *K. Skadron* [20]. While simulators tend to model the performance cycle-by-cycle or instruction-by-instruction, and require implementing a low-level hardware details; statistical analysis methods, on the other hand, handle such problem from a quantitative perspective through a set of functional/performance statistics which in turn abstract the level of details implemented in building the performance model.

Chapter 4 Reference Model and FPGA-based Profiling

This chapter is organized as follows; section 4.1 discusses the specifications of the reference model, section 4.2 shows the main profiling techniques, section 4.3 discusses our FPGA-based profiling methodology, and section 4.4 summarizes.

4.1. Reference Model Specifications

Using an ISS as a reference model suffers from a major problem that it is hard to model a complete system by integrating an ISS with the other components of the system. On the other hand, using a real architecture as a reference model faces the problem of the limited signals/events that can be monitored at real time. Fortunately, many soft-core processors have been released in this decade which are implemented using reconfigurable logic in an FPGA [84-86]. Most of these processors are not open-source, so a user cannot modify their implementation. However, the ability to read some of the internal signals has made the mission of extracting functional statistics from a real architecture much easier. Modifying or updating the state of the processor is not required in the framework, collecting functional statistics by tracing the instruction execution flow requires only reading processor's state rather than modifying it.

In this work, the softcore Xilinix-MicroBlaze [84] has been adopted as a reference architecture. In the MicroBlaze, the user has access to the system instruction/data busses, which in turn allows building an FPGA–based instruction-level profiler that can work in real-time without interfering with execution of the running program (see Figure 4.1). This is very useful for profiling embedded applications in the field, where the profiled statistics can be stored in a log file, and analyzed later on to determine the algorithm behavior (based on the functional statistics) over a range of the system states, which, in turn allows observing the dominant operations and demanded resources that characterize the system behavior. Moreover, the MicroBlaze has a flexible ISA consisting of the basic load, store, branch, arithmetic and logical operations with the ability of including shift, multiplication, division and floating point operations if the corresponding functional units are incorporated into the implementation. Hence, MicroBlaze can be configured to be as close as possible to the candidate ISA allowing a C-level command to be profiled by detecting its corresponding assembly instruction or library function. Double-precision floating point operations are not supported by MicroBlaze ISA, i.e., such operations are implemented using library functions. Hence, a function-level profiling mechanism had to be developed for detecting such operations (more details in section 4.3).

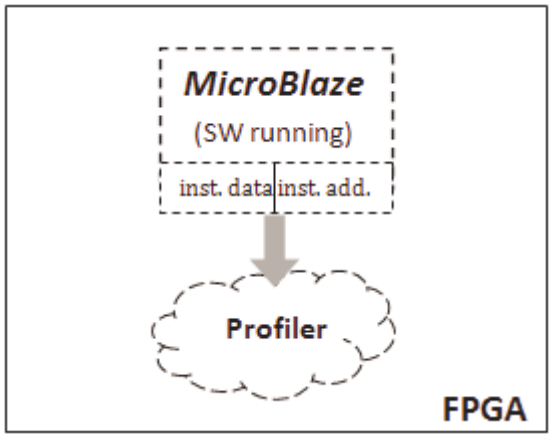


Figure 4.1. MicroBlaze/FPGA Reference-Model/Profiler Outline

4.2. Profiling

Profiling is a technique for exploring the time/computation-intensive portions in an application. It is widely used in hardware/software co-design and analytical modeling as a basic source for performance statistics, based on which designers make important decisions for partitioning the design implementation into hardware or software to achieve the desired performance and workload balance. Profiling techniques can be classified into three main categories [10-11]: software-based, hardware-based, and FPGA-based.

Software-based profiling: This technique can be implemented using a simulation environment or using the actual hardware. In the former type, an application can run without any modification on a simulation tool which can be programmed to record certain events-of-interest without interfering with the execution of the code, avoiding

adding performance/accuracy overhead. However, the overall performance/accuracy of this technique is constrained with the speed/accuracy of the simulation tool. Setting up the simulation to handle the system environment (input/output devices such as sensors/actuators) needs considerable efforts and becomes more challenging when the target application is a real-time application. On the other hand, adding instrumentation code to the target application, either at source code level or at the binary (executable) level enables profiling the application while it is running on the actual hardware. In this technique, the instrumentation code interrupts the program at certain events, functions or at a regular frequency and increments the associated counters to measure the consumed time and number of calls for events or functions-of-interest, e.g., *gprof* [87]. In addition to the accuracy and performance overhead of the inserted instrumentation code, such technique works at the function/block-level and it is generally not suitable for instruction-level profiling.

Hardware-based profiling: This technique relies on on-chip performance counters that are already built into some processors [88-89]. Hardware counters can be set to monitor certain internal events such as pipeline stalls and memory behaviors (e.g., cache misses/hits). Sampling the counters occurs periodically or on the events that increment them. However, this technique is constrained by the limited number of counters equipped with the processor, and with the type of events that are accessible by the counters [11]. Hence, this technique is generally impractical for instruction-level profiling.

FPGA-based profiling: Recently, there have been some efforts to exploit the ability of accessing the system buses of the soft-core processors for developing profiling tools to monitor programs running on these processors [10-14]. These techniques have been proposed in the context of hardware/software co-design and have achieved significantly faster and more accurate profiling results than the software-based profilers. In general, such techniques need no (or very limited) instrumentation code which in turn eliminates the performance overhead. For example, SnoopP [12], AddressTracer [13] and Airwolf [11] concentrate on function-level profiling by accessing the address bus (or program counter) to determine the time/computation-intensive portions of the application, so the user can determine the obstacles in his code and possibly move these portions into hardware to achieve better performance. The main difference between these techniques is the mechanism for setting the profiling counters/comparators. For example, SnoopP and AddressTracer obtain the addresses of the code segments-of-interest from the assembled source code (or the symbol table) and assign them to profiling comparators, while in the Airwolf, software drivers are inserted at the beginning and the end of each segment-of-interest in the source code to enable and disable the profiling counters, which in turn, minimally affects the profiling speed and accuracy. Statistics Module [14] is another FPGA-based profiling technique developed to monitor the cache and memory behaviors/events (such as cache misses/hits, memory read/write) for a specified code segment running on the FPGA implementation of the SPARC-V8 processor. The start and end addresses of the segment-of-interest are set via a software interface.

Unlike [10-14] which offered a very fast and accurate profiling technique for hardware/software co-design and system debugging, in this research, the proposed framework requires not only block-level functional statistics, but also instruction-level statistics. The following section explains the structure of the FPGA-based profiler developed in this research.

4.3. FPGA-based instruction-level profiling

In the MicroBlaze, **`Trace_Instruction`** is one of the trace signals grouped in the TRACE bus provided as an IP core by Xilinx. **`Trace_Instruction`** is a 32-bit output port which holds the binary code of the instruction being executed. A 1-bit signal called **`Trace_Valid_Instr`** is used to indicate whether this code is valid or not. Based on the instruction description in the MicroBlaze Processor Reference Guide [90], it is possible to use FPGA logic to decode the **`Trace_Instruction`** value in order to know more details about each instruction in the execution flow, such as instruction type, operands, etc. The address of each instruction can be determined by monitoring the 32-bit **`Trace_PC`** signal.

In this research, the MicroBlaze instruction set has been classified based on the functionality. This allows a profiling counter to be set to monitor the frequency of each instruction. However, for double-precision floating point operations which are not supported by the MicroBlaze ISA, a function-level profiling technique has been

developed. It is very important that this function-level profiler is able to work as an application-independent profiler, so each time the algorithm code is modified or a new algorithm is targeted, it is not mandatory to reconstruct/configure the profiler to handle such operations. For example, techniques mentioned in [10-14] are all application-dependent, i.e., all of them need either 1) setting the profiling comparators with new values (lower and upper bound addresses) for targeting the same functions with different applications (or whenever the application code is updated) where, in such cases, the location (address) of the targeted function in the application code changes, or 2) editing the software code to activate/deactivate the profiling counters for targeting new functions, adding performance overhead and more complexity. While the proposed framework aims to reduce the time factor in the estimation process, it is desirable to build a profiler that is able to work at the instruction-level and the function-level without the need for going through the FPGA design cycle each time the application is modified or changed. Unlike [10-14] which depend on accessing the address bus to achieve segment-level profiling for the same architecture, the proposed framework is able to detect functions based on monitoring the instruction trace for the target functions.

In this research, the implementation (assembly) code of different library functions on the MicroBlaze platform have been studied by disassembling the .elf (executable) files for several applications. It has been noticed that each of these functions has a sequence of instructions (first block) that can be considered as a unique trace (footprint) that differentiates these functions from each other. Also, it has been noticed that these

functions consist essentially of the same sequence of instructions when compiled/linked with different applications with very few differences. Those differences are related to some of the instructions that use intermediate values as absolute addresses. However, most of the instructions that use this type of addressing have the same relative offset values because they point to locations that have constant distances from the calling points. Hence, profiling a certain library function can be done by detecting its footprint (which is application-independent) rather than tracing its address (which is application-dependent), see Figure 4.2.

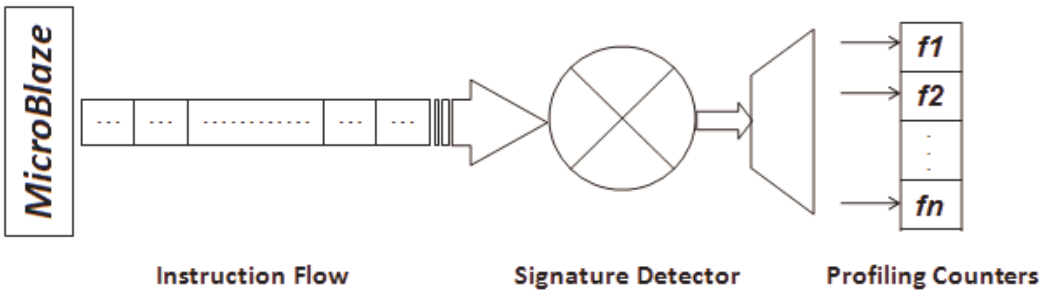


Figure 4.2. Application-Independent Profiling

For example, the first block of the double-precision floating point functions add, sub, mul, and *div* have 18, 18, 23, and 19 instructions, respectively. Table 4.1 shows the disassembled code of the double-precision floating point *div* function in one of the studied applications. Comparing this segment of code to different function implementations in different applications, the only difference is the first column of the table that contains the addresses of the instructions while the second and third columns

that contain the machine code and its equivalent assembly instruction, respectively, remain unchanged. In addition to that, comparing the machine code of this segment of code with the complete code of different applications, no match with the same sequence (except for the same function) has been detected. Therefore, this block of code can be considered a unique footprint for the double-precision floating point *div* function.

Table 4.1. Double-precision floating point div function -first 18 instructions

Ins Address	**Ins Code**	**Instruction**	
8700204C	**302**1FF98	addik	r1, r1, -104
87002050	**FA6**10054	swi	r19, r1, 84
87002054	**326**1002C	addik	r19, r1, 44
87002058	**114**70000	addk	r10, r7, r0
8700205C	**116**80000	addk	r11, r8, r0
87002060	**FAC**10058	swi	r22, r1, 88
87002064	**FAE**1005C	swi	r23, r1, 92
87002068	**12C**50000	addk	r22, r5, r0
8700206C	**12E**60000	addk	r23, r6, r0
87002070	**10D**30000	addk	r6, r19, r0
87002074	**FB0**10060	swi	r24, r1, 96
87002078	**30A**1001C	addik	r5, r1, 28
8700207C	**F9E**10000	swi	r15, r1, 0
87002080	**F94**10024	swi	r10, r1, 36
87002084	**F96**10028	swi	r11, r1, 40
87002088	**FB2**10064	swi	r25, r1, 100
8700208C	**FAC**1001C	swi	r22, r1, 28
87002090	**FAE**10020	swi	r23, r1, 32
87002094	**B9F**40694	brlid	r15, 1684

For simplicity and to avoid any unexpected change due to different intermediate values for absolute addressing, only the first three digits (bolded hex) of each machine code instruction, which contains the op-code and portions of first operands, has been considered as a representation for the instructions. Thus, by monitoring the MicroBlaze instruction bus, a double-precision floating point *div* operation can be detected by a sequence of first three digits of the machine code in the running application appearing as:

"302,FA6,326,114,116,FAC,FAE,12C,12E,10D,FB0,30A,F9E,F94,F96,FB2,FAC,FAE,B9F"

Based on the above results, modifying the application code or targeting the same function in a different application will have very small chance to produce the same sequence of code. It should be noticed here, that double-precision floating point add and sub functions share the same sequence of instructions for the first block, this should not be considered as an obstacle because both of these operations can be classified in the same category even though they can be distinguished from each other with more analysis.

While profiling a library function, other profiling counters should stop counting untill the function exits. The end of a library function can be detected by tracing its return operation. This can be implemented by registering the PC value once the corresponding library function is called, hence the returning address is the registered PC value plus 4. When detecting a beginning of a library function, the **`Trace_PC`** signal is registered

and all other profiling counters are turned off. Once the **`Trace_PC`** signal holds the registered **`Trace_PC`** plus 4 all counters turned on.

To evaluate the accuracy of this application-independent profiling mechanism, ten numerical algorithms [91] have been tested (see Table 4.2). Because the only way to validate the accuracy of this technique (in profiling double-precession floating point operations) is by comparing the profiling results with the manual analysis of the code, this technique was used to profile the single-precision floating point multiplication operation when it is implemented as a library function (the floating point functional unit is turned off) and validating the results with the number of floating point multiplication instructions that can be profiled when the floating point functional unit is turned on. **This experiment has shown 100% accuracy for the ten targeted algorithms.**

Table 4.2. Numerical C programs [91] used to test the application-independent FPGA-based profiling mechanism

Program Name	**Description**
bisection.c	Bisection method
rec_bisection.c	Recursive version of bisection method
newton.c	Sample Newton method
secant.c	Secant method
sums.c	Upper/lower sums experiment for an integral
trapezoid.c	Trapezoid rule experiment for an integral
rec_simpson.c	Adaptive scheme for Simpson's rule
euler.c	Euler's method for solving an ODE
taylor.c	Taylor series method (order 4) for solving an ODE
rk4.c	Runge-Kutta method (order 4) for solving an IVP

4.4. Summary

This chapter illustrated the features that make the MicroBlaze/FPGA platform the core of proposed performance modeling infrastructure where the MicroBlaze soft-core processor works as a reference functional model which is accessible via TRACE signals. The FPGA-based profiling mechanism developed in this research works as application-independent which in turn is very easy to retarget. This profiling technique can be added to the efforts of FPGA-based profiling for hardware/software codesign to build automatic function-level profiling using function's trace signature rather than address.

Chapter 5 Experiments and Results

In this chapter, experimental validation for the proposed framework is shown through targeting five different processor architectures and thirty three programs. Section 5.1 describes the experiment setup. Section 5.2 presents the results, and Section 5.3 summarizes.

5.1. Experiment Setup

5.1.1. Target Platforms

Including the Microblaze reference model, five different architectures (with different ISAs) have been used as target architectures for the proposed framework. The following sections describe the main characteristics of each architecture.

5.1.1.1. AVR32 Microcontroller

The AVR32 microcontroller [92] is a 32-bit load/store RISC architecture, with fifteen general-purpose 32-bit registers and Harvard memory architecture, with no cache implementation. The AVR's ISA consists of variable length instructions of 16 or 32 bits. The AVR32 core is a pipelined processor with three pipeline stages: fetch, decode and execution. All instructions are issued and completed in order. The pre-fetch unit

comprises the fetch pipe-stage, and is responsible for feeding instructions to the decode unit. The pre-fetch unit fetches 32 bits at a time from the instruction memory interface and places them in a FIFO pre-fetch buffer. At the same time, one instruction, either RISC extended or compact, is fed to the decode stage. The decode stage accepts one instruction each clock cycle from the pre-fetch unit. This instruction is then decoded, and control signals and register file addresses are generated. The execute pipeline stage performs register file reads, operations on registers and memory, and register file writes. It contains an ALU section, multiply section, and load-store section. The multiply section implements a 32 by 32 multiplier array, and 16x16, 32x16 and 32x32 multiplications and multiply-accumulates. For this experiment, the processor was clocked at 64.512 MHz, therefore incurring a 1 wait state penalty when accessing instructions or data from the flash memory non-sequentially.

5.1.1.2. PowerPC405 Microprocessor

The PowerPC405 (PPC405) used in this research is an embedded PPC405F6 processor core used in a Xilinx Virtex-4 FX12 FPGA [93]. The PPC405 is a reduced instruction set computer (RISC) with thirty-two 32-bit general purpose registers and Harvard memory architecture. The PPC405 has a five stage single issue execution pipeline, including fetch, decode, execute, write-back, and load write-back stages. Most instructions execute in a single cycle, including loads and stores assuming no-wait-state memory accesses. Multiply instructions take between 1 and 4 cycles, for 32-bit and 64-bit results

respectively, and divide instructions typically take 35 cycles. For this experiment, the Virtex-4 FX12 FPGA with the PPC405 core was contained on a Xilinx® Virtex®-4 FX12 Mini-Module. Separate data and instruction caches, each 16KB with 32-bytes per cache line were used in this system. Cache lines were connected to a memory controller which connected to a 100 MHz, 16-bit data bus SDRAM. The processor was configured to run at 300MHz and did not include a hardware floating point unit.

5.1.1.3. Gumstix Intel XScale® PXA255

The Gumstix Verdex XL6P [94] utilizes an Intel PXA270 microprocessor. This processor supports Intel Wireless MMX integer instructions, four 64-Kbyte memory banks, and an integrated LCD panel controller. It is also designed to be highly backward compatible with the PXA25x series. The particular version of the PXA270 used by the Gumstix utilizes a 600MHz clock frequency. This processor is an Intel XScale microarchitecture fully compatible with ARM architecture V5TE. The architecture contains write, fill, pend, and branch-target buffers. As well, it contains a multiply-accumulate coprocessor, 32-bit coprocessor interface, and core memory bus. The architecture is RISC with a seven to eight stage superpipeline. The MAC is capable of performing two simultaneous 16-bit SIMD multiplies. There is also a 128-entry branch target buffer to keep the pipeline filled with statistically correct options. A 2-Kbyte mini-data cache prevents thrashing of the 32-Kbyte cache. An eight entry write buffer means the core can execute while data is being written to memory.

5.1.1.4. PIC32 microcontroller

The PIC32 processor [95] implements the MIPS ISA which runs at runs at 80 MHz on a RISC architecture and has a Harvard memory architecture. It contains two different pipelines. The CPU core consists of five pipeline stages: fetch, execution, memory fetch, memory align, and memory writeback. Assuming there are no stalls in the pipeline, most instructions will be executed on every clock cycle. Once a multiply/divide instruction is issued, the CPU may either fetch the next instruction (while the multiply/divide pipline is still calculating a result) or stall waiting for the result from the pipeline. Stalling occurs if the CPU attempts to retrieve the result before the pipeline is finished. A dedicated hardware is used for the multiply and divide instructions For multiplications, 1 clock cycle can perform 16x16 or 32x16 operations and requires 2 clock cycles otherwise. Division requires between 11 to 32 clock cycles depending on the parameters used. This pipeline is iterative and multi-stage. The PIC32 also employs a 128-bit Flash memory designed to increase throughput. This module, combined with the 128-bit prefetch cache means that the CPU can prefetch that many bits of the next instructions and store them in the cache.

5.1.1.5. MicroBlaze Soft-Core Processor

As a target architecture, MicroBlaze been configured to consist of five pipeline stages (single-issue) running at 50MHz with a cache implementation of separate data and

instruction caches, each is 8KB. MicroBlaze [84] is a RISC architecture with thirty-two 32-bit general purpose registers and Harvard memory architecture.

5.1.2. Algorithms and Benchmarks

A total of thirty three programs/benchmarks from different application domains (data structure, control, FFT, sorting, and numerical algorithms) have been used in this research. Table 5.1 shows the type of each program, whether it includes floating point operations, and the source of programs. Around half of these programs include floating point operations in addition to other library function calls. Some modifications have been made on some of these programs such as changing C++ statements into C, and commenting the *printf* commands.

As in [42, 61], the *leave-one-out* cross validation technique has been used to generate the result data. In this method, each program is used once as a target application while all of the other programs are used as the training set for the linear regression method. For example, the data for application 1 is gathered by using applications 2 through 33 as the training set for the linear regression. The developed model is then used to predict the run time (number of cycles) for application 1 based on the profile of operations for that application. A similar procedure is then performed for each of the other applications.

Table 5.1. Programs and Benchmarks used for Experimental Validation

No.	Program Name	Brief Description	FP	Ref.
1	test_math	Small Math Benchmark	Yes	[96]
2	basicmath	Automotive/Industrial Control Benchmark	Yes	[54]
3	pbmsrch_small	Pratt-Boyer-Moore String Search (small)	No	[54]
4	pbmsrch_large	Pratt-Boyer-Moore String Search (large)	No	[54]
5	hanoi	Recursive Hanoi	No	[97]
6	bitcount	Automotive/Industrial Control Benchmark	No	[54]
7	dhrystone	Integer Benchmark	Yes	[98]
8	fft_test	FFT Application	Yes	[99]
9	sort_bubble[5]	Sorting Algorithm	No	[100]
10	sort_combo	Sorting Algorithm	No	[100]
11	sort_heap	Sorting Algorithm	No	[100]
12	sort_insert	Sorting Algorithm	No	[100]
13	sort_merge	Sorting Algorithm	No	[100]
14	sort_quick	Sorting Algorithm	No	[100]
15	sort_selection	Sorting Algorithm	No	[100]
16	sort_shell	Sorting Algorithm	No	[100]
17	test_tree[2]	Tree Algorithm	No	[101]
18	tree_traversal	Binary Tree Algorithm	No	[102]
19	short_path	Dijkstra's Algorithm	No	[103]
20	mst	Minimum Spanning Tree	No	[102]
21	bisection	Locating Roots of Equations (bisection)	Yes	[91]
22	euler	Euler's Method for Solving an ODE	Yes	[91]
23	fuzzy	Fuzzy Controller	Yes	[104]
24	newton	Locating Roots of Equations (newton)	Yes	[91]
25	rec_simpson	Adaptive Scheme for Simpson's Rule	Yes	[91]
26	rk4	Runge-Kutta method for solving an IVP	Yes	[91]
27	rk4sys	Runge-Kutta method for systems of ODEs	Yes	[91]
28	rk45	Runge-Kutta-Fehlberg for solving an IVP	Yes	[91]
29	secant	Locating Roots of Equations (secant)	Yes	[91]
30	Sums	Upper/Lower Sums Experiment (Integral)	Yes	[91]
31	Taylor	Taylor series method for solving an ODE	Yes	[91]
32	Taylorsys	Taylor series method for systems of ODEs	Yes	[91]
33	Trapezoid	Trapezoid Rule for Numerical Integration	Yes	[91]

[5] *bubble_sort* and *test_tree* performance measurements are not available for the PIC controller.

5.1.3. Floating-Point Implementation

Two main experiments have been implemented. In the first one, the set of programs have been run on the MicroBlaze platform without configuring the hardware floating point functional unit, i.e., all floating point operations have been implemented through library functions, and in the second one, the floating point functional unit has been enabled which is dedicated to implement single-precision floating point operations. The FPGA-based profiler has been set to collect the functional statistics at instruction-level where instructions that have the same cost have been grouped in the same category, for example arithmetic add/sub and logical and/or operations have the same functional category. In the second experiment, since double-precision floating point operations are implemented using library functions, the application-independent profiling mechanism developed in this research has been used to count these operations independently at function-level (single-precision floating point operations are profiled at the instruction-level).

5.2. Results

5.2.1. Ordinary Least Squares vs. Robust Fit

In this subsection, linear regression analysis using ordinary least squares method is compared with linear regression using robust fit method. Table 5.2 shows the results of modeling the performance with a software implementation of floating point operations

and Table 5.3 shows the results using a hardware implementation of floating point operations (double-precision floating point operations still implemented through library functions, however, they are profiled at the function-level).

In the error analysis, the relative error for an observation is the estimated value minus the observed (true) value all divided by observed value. In these tables the minimum (Min), median (Med), maximum (Max), mean absolute (MA), and standard deviation (Std) of the relative errors are reported for each architecture. In both experiments, linear regression using robust fit has shown better accuracy levels for all architectures. Since the robust fit method has shown a higher level of accuracy for all architectures (including the PPC), a detailed error analysis using robust fit method is used to generate the results in the following subsections for all programs and architectures.

Table 5.2. Error analysis summary using software implementation of floating point operations (SW FP)

	Error	AVR	PPC	GS	PIC	*MB*
least squares	Min	-173.82%	-44.47%	-138.24%	-1833.34%	*-1401.61%*
	Med	31.52%	6.68%	49.31%	-22.44%	*-14.52%*
	Max	5034.65%	59.86%	766.82%	883.02%	*362.33%*
	MA	393.89%	15.94%	133.19%	235.49%	*159.68%*
	Std	11.2585	0.1954	1.8653	4.9612	*3.1081*
robust fit	Min	-126.82%	-43.57%	-65.08%	-100.36%	*-153.14%*
	Med	-3.98%	1.05%	0.97%	-1.60%	*0.25%*
	Max	222.21%	29.94%	61.37%	1022.16%	*83.36%*
	MA	42.55%	10.97%	18.37%	110.04%	*20.66%*
	Std	0.6222	0.1524	0.2658	2.4331	*0.3605*

Table 5.3. Error analysis summary using hardware implementation of floating point operations (HW FP)

	Error	AVR	PPC	GS	PIC	*MB*
least squares	Min	-793.00%	-144.46%	-1346.67%	-214.28%	*-306.79%*
	Med	-1.78%	8.03%	0.51%	-19.43%	*10.44%*
	Max	204.80%	404.71%	587.12%	184.77%	*1453.81%*
	MA	78.65%	51.19%	167.33%	66.68%	*173.80%*
	Std	1.9082	0.9083	3.2399	0.8475	*3.6806*
robust fit	Min	-146.57%	-88.60%	-59.30%	-76.53%	*-191.75%*
	Med	0.23%	-0.19%	0.81%	-0.07%	*8.71%*
	Max	120.77%	28.74%	183.29%	206.24%	*139.44%*
	MA	29.46%	12.65%	26.02%	44.39%	*38.49%*
	Std	0.4795	0.2106	0.4623	0.6974	*0.5647*

5.2.2. Absolute Performance Estimation

To generate the data in Tables 5.4, 5.5, and 5.6, the reference model (MicroBlaze) has been configured without a hardware unit to support to the floating point operations, hence, the floating point operations in the target programs have been run through library functions and all applications were then profiled as integer programs. Table 5.4 shows the performance modeling results of the MicroBlaze as a target architecture – that is, the MicroBlaze reference model was used to predict the performance of the Microblaze itself on each of the individual applications (using leave-one-out validation). A total of 26 programs (out of 33) have been modeled with an absolute error of less than 30%. The overall mean absolute error was 20.66% (see Table 5.2).

Table 5.5 shows the performance estimations extracted from the PPC and GS performance models where a total of 31 programs (out of 33) and 26 (out of 33) have been modeled with an absolute error less than 30%, respectively, with an overall mean absolute error 10.97% and 18.37%, respectively (see Table 5.2). It is clear that the PPC model has achieved an accuracy level that is better than the reference architecture's performance model itself. This can indicate that PPC has a more structured internal architecture organization (CPU core, memory architecture, etc) than MicroBlaze, so it is precisely handled by linear regression. On the other hand, the GS model shows performance estimations that are close to the MicroBlaze model with smaller mean absolute error.

The AVR and PIC microcontrollers performance estimations are shown in Table 5.6. The AVR performance model has reported a total of 17 programs (out of 33) with an absolute error less than 30%, while for the PIC, a total of 14 programs (out of 31) have been modeled with an absolute error less than 30%. The overall mean absolute error for these architectures was 42.55% and 110.04%, respectively (see Table 5.2). It can be noticed that these models have achieved less accuracy levels than the performance models of the targeted microprocessors (PPC, GS, and MicroBlaze). Most of the large errors fall in the floating point programs rather than integer programs, which leads one to believe that mapping the functional behavior of the floating point library functions on MicroBlaze to their performance on these architectures (AVR and PIC) has major impacts on the accuracy of the performance models.

Table 5.4. Performance Measurements/Estimations (in cycles) of the Target Programs on the MicroBlsze Architecture (SW FP)

Test Program	True	Estimated	Error
test_math	464,133	470,883.64	1.45%
basicmath	j,522,597,219	2,552,488,554.20	1.18%
pbmsrch_small	524,681	598,233.48	14.02%
pbmsrch_large	11,891,537	9,086,534.15	-23.59%
hanoi	1,476,415,213	2,097,718,686.27	42.08%
bitcount	196,215,015	359,784,307.75	83.36%
dhrystone	50,559,615	63,848,686.99	26.28%
fft_test	4,125,776	3,655,644.20	-11.39%
sort_bubble	525,117,153	437,869,137.68	-16.61%
sort_combo	28,888,433	28,961,735.55	0.25%
sort_heap	35,306,101	47,295,117.62	33.96%
sort_insert	193,066,897	176,662,006.39	-8.50%
sort_merge	29,974,820	28,573,968.28	-4.67%
sort_quick	28,805,880	28,791,720.93	-0.05%
sort_selection	327,732,463	371,861,313.15	13.46%
sort_shell	38,343,582	42,793,604.34	11.61%
testtree	167,314,216	92,195,007.44	-44.90%
TreeTraversal	23,162,571	20,523,635.93	-11.39%
shortp	458,720	-243,766.97	-153.14%
MST	1,757,049	1,132,322.66	-35.56%
m_bisection	24,227,231	21,902,476.76	-9.60%
euler	2,549,097	2,289,789.06	-10.17%
fc_fuzzy	201,024	264,150.08	31.40%
run_newton	4,039,881	4,197,555.28	3.90%
rec_simpson	11,760,609	11,766,566.05	0.05%
rk4	1,565,932	1,673,947.19	6.90%
m_rk4sys	4,467,044	4,645,155.82	3.99%
m_rk45	2,392,586	2,209,983.63	-7.63%
m_secant	1,328,581	1,336,735.26	0.61%
m_sums	34,096,294	24,229,306.81	-28.94%
m_taylor	1,796,796	1,890,990.14	5.24%
m_taylorsys	3,984,357	4,648,694.65	16.67%
m_trapezoid	2,050,244	1,655,411.91	-19.26%
Total No. of Absolutes Errors < 30%			**26/33**

Table 5.5. Performance Measurements/Estimations (in cycles) of the Target Programs on the PPC and GS Architectures (SW FP)

	True		Estimated		Error	
Test Program	**PPC**	**GS**	**PPC**	**GS**	**PPC**	**GS**
test_math	219,244	79,860	184,347.56	61,291.80	-15.92%	-23.25%
basicmath	1,084,554,413	303,945,960	1,000,373,653.60	247,361,591.40	-7.76%	-18.62%
pbmsrch_small	320,594	315,900	323,761.27	231,283.80	0.99%	-26.79%
pbmsrch_large	10,553,679	7,144,620	6,166,667.91	3,781,689.60	-41.57%	-47.07%
hanoi	713,032,712	759,520,860	813,642,564.67	816,175,280.40	14.11%	7.46%
bitcount	123,068,146	128,166,660	138,992,144.80	111,700,026.00	12.94%	-12.85%
dhrystone	25,469,159	52,219,080	26,669,867.20	36,768,393.00	4.71%	-29.59%
fft_test	1,790,136	469,920	1,513,845.68	484,287.60	-15.43%	3.06%
sort_bubble	299,556,791	333,290,340	290,081,929.74	354,574,664.40	-3.16%	6.39%
sort_combo	18,019,577	21,027,000	18,275,491.85	20,860,513.80	1.42%	-0.79%
sort_heap	28,543,733	32,237,580	29,791,562.84	27,457,248.00	4.37%	-14.83%
sort_insert	119,801,539	142,036,080	112,464,376.21	131,127,838.20	-6.12%	-7.68%
sort_merge	21,823,751	24,226,620	20,421,056.88	26,411,697.60	-6.43%	9.02%
sort_quick	17,866,030	20,891,220	18,139,537.15	20,742,791.40	1.53%	-0.71%
sort_selection	225,379,510	247,117,140	238,056,617.17	267,826,087.20	5.62%	8.38%
sort_shell	23,906,588	27,179,040	26,130,808.46	28,022,577.60	9.30%	3.10%
testtree	65,063,518	65,517,240	58,584,116.90	80,768,655.00	-9.96%	23.28%
TreeTraversal	14,384,887	29,490,600	10,465,648.78	10,298,769.60	-27.25%	-65.08%
shortp	332,779	924,240	187,787.91	987,904.80	-43.57%	6.89%
MST	1,534,768	1,433,220	1,994,277.93	1,476,058.20	29.94%	2.99%
m_bisection	9,219,514	2,387,760	10,452,478.32	3,063,649.80	13.37%	28.31%
euler	1,553,806	407,700	1,570,160.41	657,907.20	1.05%	61.37%
fc_fuzzy	113,344	51,180	98,466.32	48,141.60	-13.13%	-5.94%
run_newton	2,540,266	703,080	2,255,936.12	271,764.00	-11.19%	-61.35%
rec_simpson	4,761,100	1,216,440	4,431,935.19	1,708,447.20	-6.91%	40.45%
rk4	891,223	214,320	954,856.01	201,850.20	7.14%	-5.82%
m_rk4sys	2,515,969	683,280	2,696,658.57	689,881.20	7.18%	0.97%
m_rk45	1,461,238	391,380	1,455,343.70	366,046.20	-0.40%	-6.47%
m_secant	590,605	149,760	554,452.56	154,360.20	-6.12%	3.07%
m_sums	10,391,569	3,132,420	11,323,997.58	4,270,004.40	8.97%	36.32%
m_taylor	1,012,423	235,500	1,081,746.67	246,615.00	6.85%	4.72%
m_taylorsys	2,138,506	503,580	2,471,179.42	494,688.00	15.56%	-1.77%
m_trapezoid	648,607	194,640	661,009.72	256,712.40	1.91%	31.89%
Total No. of Absolutes Errors < 30%					**31/33**	**26/33**

Table 5.6. Performance Measurements/Estimations (in cycles) of the Target Programs on the AVR and PIC Architectures (SW FP)

	True		Estimated		Error	
Test Program	**AVR**	**PIC**	**AVR**	**PIC**	**AVR**	**PIC**
test_math	16,380	47,474	52,778.70	189,854.15	222.21%	299.91%
basicmath	245,384,463	363,203,164	272,557,752.29	862,521,340.55	11.07%	137.48%
pbmsrch_small	495,430	1,049,464	350,806.08	656,532.28	-29.19%	-37.44%
pbmsrch_large	11,407,955	24,119,692	6,153,603.83	13,068,657.07	-46.06%	-45.82%
hanoi	977,267,544	2,365,573,644	454,866,735.39	1,474,411,961.26	-53.46%	-37.67%
bitcount	126,556,933	331,459,756	103,658,513.09	281,266,938.32	-18.09%	-15.14%
dhrystone	28,640,011	71,046,664	25,583,102.89	79,787,241.18	-10.67%	12.30%
fft_test	382,344	1,006,025	603,660.21	1,996,724.59	57.88%	98.48%
sort_bubble	315,061,856	*	343,351,642.81	*	8.98%	*
sort_combo	20,390,971	48,907,067	20,403,815.57	46,963,103.72	0.06%	-3.97%
sort_heap	30,209,333	48,771,974	24,612,609.02	67,259,805.06	-18.53%	37.91%
sort_insert	130,798,035	324,462,844	123,769,134.10	377,720,302.29	-5.37%	16.41%
sort_merge	21,429,322	56,012,230	26,355,642.56	54,171,824.25	22.99%	-3.29%
sort_quick	20,311,153	48,812,765	20,283,752.84	46,809,099.42	-0.13%	-4.10%
sort_selection	270,863,523	635,888,854	241,229,550.65	520,476,723.80	-10.94%	-18.15%
sort_shell	25,731,503	65,401,496	27,360,415.86	70,418,051.04	6.33%	7.67%
testtree	63,191,338	*	80,574,727.41	*	27.51%	*
TreeTraversal	11,894,146	10,322,272	7,725,222.10	29,896,431.01	-35.05%	189.63%
shortp	255,275	562,831	509,446.52	320,694.37	99.57%	-43.02%
MST	1,424,579	4,212,812	-382,080.65	1,357,401.70	-126.82%	-67.78%
m_bisection	2,056,700	921,344	3,400,991.17	10,338,956.21	65.36%	1022.16%
euler	503,206	1,867,011	334,941.75	1,923,898.66	-33.44%	3.05%
fc_fuzzy	19,652	59,714	42,361.79	111,925.11	115.56%	87.44%
run_newton	731,497	2,752,529	1,434,990.15	601,955.55	96.17%	-78.13%
rec_simpson	941,432	1,090,673	903,952.61	2,229,477.60	-3.98%	104.41%
rk4	255,955	826,618	396,939.57	813,387.54	55.08%	-1.60%
m_rk4sys	769,278	2,839,195	575,603.41	2,494,666.00	-25.18%	-12.13%
m_rk45	440,258	1,501,858	604,306.02	1,354,346.72	37.26%	-9.82%
m_secant	123,223	56,228	187,827.27	523,946.96	52.43%	831.83%
m_sums	2,466,152	3,723,755	1,581,318.40	-13,244.64	-35.88%	-100.36%
m_taylor	288,541	1,068,907	175,312.23	1,110,062.53	-39.24%	3.85%
m_taylorsys	618,626	2,237,903	450,601.64	2,278,483.99	-27.16%	1.81%
m_trapezoid	148,426	225,361	138,694.68	48,611.13	-6.56%	-78.43%
Total No. of Absolutes Errors < 30%					**17/33**	**14/31**

The data in Tables 5.7, 5.8 and 5.9, was produced by configuring the reference model (MicroBlaze) with a hardware unit to support to the floating point operations; hence, the single-precision floating point operations in the target programs have been implemented directly by the MicroBlaze ISA on hardware and profiled at instruction-level, while double-precision floating point operations have been profiled at function-level. Table 5.7 shows the performance modeling results of the MicroBlaze as a target architecture. A total of 19 programs (out of 33) have been modeled with an absolute error less than 30%. The overall mean absolute error was 38.49% (see Table 5.3). This degradation in the accuracy level is caused by the fact that that the MicroBlaze configuration with hardware floating point support as a target architecture is different than the MicroBlaze configuration as a reference model without hardware floating point support, where in the former experiment, both configuration were the same (no hardware floating point support).

Table 5.8 shows the performance estimations extracted from the PPC and GS performance models where a total of 30 programs and 24 (out of 33) have been modeled with an absolute error less than 30%, respectively, with an overall mean absolute error 12.65% and 26.02%, respectively (see Table 5.3). This shows a slightly decreasing in accuracy than previous experiment (see Table 5.5). However, it is still clear that both the PPC and the GS models have accuracy levels better than the MicroBlaze performance model itself, which can be attributed to the internal organization of these architectures.

The AVR and PIC microcontroller performance estimations are shown in Table 5.9. The AVR performance model has reported a total of 22 programs (out of 33) with an absolute error less than 30%, while for the PIC, a total of 18 programs (out of 31) have been modeled with an absolute error less than 30%. The overall mean absolute error for these architectures was 29.46% and 44.39%, respectively (Table 5.3) which in turn shows a significant enhancement in the accuracy level over the previous experiment (Table 5.6), although still less than PPC, and GS microprocessor models. This shows that increasing the level of abstraction of floating point operations has enhanced the AVR and PIC models by hiding the detailed software implementation of these operations which in turn achieved a better correlation between the functional behavior on the MicroBlaze and the performance on AVR and PIC. In fact, considering only the integer programs (around half of the training set), the mean absolute error for AVR is 9.90% and for PIC is 22.27%.

As the differences among the target ISAs explains a the different accuracy levels achieved by each model, the different internal architecture of microcontrollers (AVR and PIC) from the other processor architectures (especially memory architecture) shows a significant impacts on the performance estimation results. Such models can be further improved by enlarging the set of training programs which in turn can be able to stress such model intensively [42].

Table 5.7. Performance Measurements/Estimations (in cycles) of the Target Programs on the MicroBlaze Architecture (HW FP)

Test Program	True	Estimated	Error
test_math	464,133	485,928.74	4.70%
basicmath	2,522,597,219	3,319,900,549.05	31.61%
pbmsrch_small	524,681	597,745.52	13.93%
pbmsrch_large	11,891,537	16,495,432.28	38.72%
hanoi	1,476,415,213	2,232,638,016.64	51.22%
bitcount	196,215,015	8,165,840.21	-95.84%
dhrystone	50,559,615	73,389,089.40	45.15%
fft_test	4,125,776	2,981,875.74	-27.73%
sort_bubble	525,117,153	483,325,351.21	-7.96%
sort_combo	28,888,433	27,599,829.54	-4.46%
sort_heap	35,306,101	22,230,511.02	-37.03%
sort_insert	193,066,897	209,887,329.60	8.71%
sort_merge	29,974,820	32,094,397.03	7.07%
sort_quick	28,805,880	27,486,837.61	-4.58%
sort_selection	327,732,463	448,413,503.13	36.82%
sort_shell	38,343,582	47,297,278.71	23.35%
testtree	167,314,216	121,374,192.37	-27.46%
TreeTraversal	23,162,571	26,902,480.28	16.15%
shortp	458,720	920,952.41	100.77%
MST	1,757,049	-1,612,131.77	-191.75%
m_bisection	24,227,231	18,485,981.91	-23.70%
euler	2,549,097	3,402,861.48	33.49%
fc_fuzzy	201,024	481,327.00	139.44%
run_newton	4,039,881	7,481,176.37	85.18%
rec_simpson	11,760,609	7,239,907.74	-38.44%
rk4	1,565,932	1,312,926.29	-16.16%
m_rk4sys	4,467,044	3,725,536.43	-16.60%
m_rk45	2,392,586	3,089,381.35	29.12%
m_secant	1,328,581	1,555,957.20	17.11%
m_sums	34,096,294	22,762,709.41	-33.24%
m_taylor	1,796,796	2,021,707.96	12.52%
m_taylorsys	3,984,357	4,825,190.92	21.10%
m_trapezoid	2,050,244	1,455,221.24	-29.02%
Total No. of Absolutes Errors < 30%			**19/33**

Table 5.8. Performance Measurements/Estimations (in cycles) of the Target Programs on the PPC and GS Architectures (HW FP)

	True		Estimated		Error	
Test Program	**PPC**	**GS**	**PPC**	**GS**	**PPC**	**GS**
test_math	219,244	79,860	242,879.01	66,183.60	10.78%	-17.13%
basicmath	1,084,554,413	303,945,960	1,085,855,714.33	306,511,992.60	0.12%	0.84%
pbmsrch_small	320,594	315,900	298,279.90	249,297.00	-6.96%	-21.08%
pbmsrch_large	10,553,679	7,144,620	6,042,293.56	4,056,127.20	-42.75%	-43.23%
Hanoi	713,032,712	759,520,860	917,988,396.15	1,022,147,844.60	28.74%	34.58%
bitcount	123,068,146	128,166,660	122,141,026.46	286,545,038.40	-0.75%	123.57%
dhrystone	25,469,159	52,219,080	25,602,635.90	38,689,438.80	0.52%	-25.91%
fft_test	1,790,136	469,920	1,627,144.13	452,868.60	-9.10%	-3.63%
sort_bubble	299,556,791	333,290,340	317,907,078.62	326,038,733.40	6.13%	-2.18%
sort_combo	18,019,577	21,027,000	18,019,716.85	21,024,967.80	0.00%	-0.01%
sort_heap	28,543,733	32,237,580	29,511,610.51	29,307,313.80	3.39%	-9.09%
sort_insert	119,801,539	142,036,080	109,997,079.29	143,193,020.40	-8.18%	0.81%
sort_merge	21,823,751	24,226,620	20,643,273.20	24,653,478.60	-5.41%	1.76%
sort_quick	17,866,030	20,891,220	17,870,961.05	20,884,214.40	0.03%	-0.03%
sort_selection	225,379,510	247,117,140	259,702,804.46	276,841,786.20	15.23%	12.03%
sort_shell	23,906,588	27,179,040	25,645,438.87	28,410,225.00	7.27%	4.53%
Testtree	65,063,518	65,517,240	55,152,408.73	72,177,763.80	-15.23%	10.17%
TreeTraversal	14,384,887	29,490,600	9,032,744.44	12,001,549.80	-37.21%	-59.30%
Shortp	332,779	924,240	321,601.87	937,919.40	-3.36%	1.48%
MST	1,534,768	1,433,220	1,511,024.39	4,060,152.00	-1.55%	183.29%
m_bisection	9,219,514	2,387,760	7,039,421.07	1,327,096.20	-23.65%	-44.42%
Euler	1,553,806	407,700	1,626,229.99	460,797.60	4.66%	13.02%
fc_fuzzy	113,344	51,180	92,050.25	52,630.20	-18.79%	2.83%
run_newton	2,540,266	703,080	289,536.02	290,289.60	-88.60%	-58.71%
rec_simpson	4,761,100	1,216,440	4,747,007.52	1,191,232.80	-0.30%	-2.07%
rk4	891,223	214,320	1,018,776.97	344,653.20	14.31%	60.81%
m_rk4sys	2,515,969	683,280	2,157,194.24	521,401.20	-14.26%	-23.69%
m_rk45	1,461,238	391,380	1,396,286.01	347,796.00	-4.44%	-11.14%
m_secant	590,605	149,760	676,945.31	191,766.60	14.62%	28.05%
m_sums	10,391,569	3,132,420	10,371,461.15	3,095,692.80	-0.19%	-1.17%
m_taylor	1,012,423	235,500	1,118,598.95	264,724.20	10.49%	12.41%
m_taylorsys	2,138,506	503,580	2,517,965.08	702,514.80	17.74%	39.50%
m_trapezoid	648,607	194,640	665,853.46	206,386.80	2.66%	6.04%
Total No. of Absolutes Errors < 30%					**30/33**	**24/33**

Table 5.9. Performance Measurements/Estimations (in cycles) of the Target Programs on the AVR and PIC Architectures (HW FP)

	True		Estimated		Error	
Test Program	**AVR**	**PIC**	**AVR**	**PIC**	**AVR**	**PIC**
test_math	16,380	47,474	-7,628.71	95,855.15	-146.57%	101.91%
basicmath	245,384,463	363,203,164	188,611,775.91	327,889,557.37	-23.14%	-9.72%
pbmsrch_small	495,430	1,049,464	455,590.46	1,057,277.16	-8.04%	0.74%
pbmsrch_large	11,407,955	24,119,692	12,154,006.82	24,081,119.59	6.54%	-0.16%
Hanoi	977,267,544	2,365,573,644	894,002,561.71	2,202,465,424.49	-8.52%	-6.90%
Bitcount	126,556,933	331,459,756	117,438,227.33	331,635,237.53	-7.21%	0.05%
dhrystone	28,640,011	71,046,664	30,230,882.74	77,510,381.34	5.55%	9.10%
fft_test	382,344	1,006,025	607,447.52	1,508,579.40	58.87%	49.95%
sort_bubble	315,061,856	*	309,115,131.30	*	-1.89%	*
sort_combo	20,390,971	48,907,067	20,436,992.36	49,123,013.85	0.23%	0.44%
sort_heap	30,209,333	48,771,974	31,655,964.57	76,187,072.94	4.79%	56.21%
sort_insert	130,798,035	324,462,844	132,127,949.11	352,938,883.96	1.02%	8.78%
sort_merge	21,429,322	56,012,230	22,742,926.72	55,931,166.45	6.13%	-0.14%
sort_quick	20,311,153	48,812,765	20,249,577.30	48,559,999.67	-0.30%	-0.52%
sort_selection	270,863,523	635,888,854	266,549,984.35	465,920,205.47	-1.59%	-26.73%
sort_shell	25,731,503	65,401,496	24,787,390.82	65,357,640.13	-3.67%	-0.07%
Testtree	63,191,338	*	65,934,767.25	*	4.34%	*
TreeTraversal	11,894,146	10,322,272	12,752,761.44	31,610,813.54	7.22%	206.24%
Shortp	255,275	562,831	38,353.34	547,170.87	-84.98%	-2.78%
MST	1,424,579	4,212,812	1,594,812.23	3,568,174.69	11.95%	-15.30%
m_bisection	2,056,700	921,344	4,067,035.26	2,624,656.70	97.75%	184.87%
Euler	503,206	1,867,011	345,429.84	459,293.40	-31.35%	-75.40%
fc_fuzzy	19,652	59,714	43,386.40	103,598.86	120.77%	73.49%
run_newton	731,497	2,752,529	193,161.56	645,982.79	-73.59%	-76.53%
rec_simpson	941,432	1,090,673	1,272,627.68	2,434,473.43	35.18%	123.21%
rk4	255,955	826,618	361,915.90	645,295.37	41.40%	-21.94%
m_rk4sys	769,278	2,839,195	845,020.32	1,958,265.00	9.85%	-31.03%
m_rk45	440,258	1,501,858	308,672.47	395,908.88	-29.89%	-73.64%
m_secant	123,223	56,228	43,605.76	142,766.50	-64.61%	153.91%
m_sums	2,466,152	3,723,755	3,316,060.18	4,124,231.76	34.46%	10.75%
m_taylor	288,541	1,068,907	246,627.57	739,461.33	-14.53%	-30.82%
m_taylorsys	618,626	2,237,903	533,215.08	2,036,852.76	-13.81%	-8.98%
m_trapezoid	148,426	225,361	166,913.66	260,886.62	12.46%	15.76%
Total No. of Absolutes Errors < 30%					**22/33**	**18/31**

5.2.3. Relative Performance Estimation

In previous subsection the errors have been analyzed in terms of absolute performance estimation, i.e., the exact number of processor cycles needed for program execution. In this subsection, a relative performance analysis is shown, i.e., the modeling technique is used to compare processor architectures with each other in terms of which achieves better or worse performance for each application. Table 5.10 lists the observed (true) time (in μs) for each tested program for all architectures and compare these measurements with the best performance estimations shown in the previous subsection (MicroBlaze SW FP implementation to model PPC and GS and MicroBlaze; and MicroBlaze HW FP implementation to model AVR and PIC), where the true (and estimated) time in μs is obtained by multiplying the number of true (and estimated) cycles by the clock rate for each architecture. Each processor of the candidate architectures is compared individually with the rest of candidate processors. The last major column in the table shows whether the estimated relative performance of each pair of architectures matches their actual relative performance or not. A letter 'y' is used to indicate that "yes, the relative performance of this processor was classified correctly" and the shaded cells are used to indicate a wrong relative performance classification. In each row, each wrong classification case has unique number that associates it with the pair of measurements (subscripted by the same number) from which the relative performance is extracted. For example, in the first row (for program number 1), the actual performance measurements show that AVR consumes more time than GS ($254_1 > 133_1$, or GS better than AVR), and

PPC consumes more time than PIC (731_2>593_2, or PIC better than PPC). However, the estimated performance wrongly indicates that AVR consumes less time than GS (AVR better than GS), and PPC consumes less time than PIC (PPC better than PIC). The rest of the relative performance estimations in this row are correct (y), i.e. the relative performance estimations matches the relative performance measurements. The AVR vs. GS case is numbered by '1' and the PPC vs. PIC case is numbered by '2'. The associated performance measurements with these cases are subscripted by the same numbers so easily can be referred to for more analysis. In case of a performance measurement involved in more than one wrong case, this measurement is subscripted by the numbers of the wrong cases separated by a comma ','. For example, in row number 25, the AVR performance measurement is involved in two wrong relative performance estimations (1: AVR vs. PPC, 2: AVR vs. PIC) so this measurement appears as "$14{,}593_{1,2}$".

The table shows that the proposed performance modeling framework has been able to classify the performance of the candidate processors correctly for a major portion of the tested programs. There are a total of 322 relative performance comparisons and out of those, a total of 293 cases were predicted correctly. For example, the total correct estimations for comparing AVR vs. PIC is 26 out of 33, and for comparing PIC vs. MB is 25 out of 31. All other relative performance evaluations include only three or less misclassifications. Moreover, analyzing the misclassified cases, it easily can be noticed that the performance measurements in most of these cases are close to each other for each pair of measurements. For example, in rows number six and seven, the two incorrect

cases are for comparing PIC vs. MB ($4{,}143{,}247_1$ vs. $3{,}924{,}300_1$) and PPC vs. GS ($84{,}897_1$ vs. $87{,}032_1$). In the first case, the actual performance difference is only 5.3% and in the second case it is only 2.5%.

Table 5.11 is built based on the assumption that all of the target architectures run at the same cock rate, so the relative performance can be obtained directly by comparing the performance measurements (estimations) in number of cycles instead of converting them into units of time (μs). While this is not a realistic assumption even if the system clocks in these architectures are set (or configured) to run at the same clock rate, i.e., still the overall performance is subject to be governed by the other microarchitectural details such as memory speed; this theoretical assumption is introduced to see how accurate the framework is in case of targeting different architecture that run at the same clock rate. In this table, a total of 276 cases were predicted correctly out of 322 relative performance comparisons where for the most of the misclassified cases, it easily can be noticed that the performance measurements are close to each other for each pair of measurements. For example, in the worst case estimations when comparing AVR vs. GS (a total of 11 misclassifications) and comparing PPC vs. PIC (a total of 8 misclassifications), only three cases out of these misclassifications (for each pair) are for a difference of larger than 20% in their performance (rows no. 11, 18, 22 for AVR vs. GS, and rows no. 18, 22, 23 for PPC vs. PIC).

Table 5.10. Relative Performance Analysis based on Time Comparison

Program No.	True Time (us)					Estimated Time (us)					Correct Comparison									
	AVR	PPC	GS	PIC	MB	AVR	PPC	GS	PIC	MB	AVR ^ PPC	AVR ^ GS	AVR ^ PIC	AVR ^ MB	PPC ^ GS	PPC ^ PIC	PPC ^ MB	GS ^ PIC	GS ^ MB	PIC ^ MB
1	254_1	731_2	133_1	593_2	9,283	-118	614	102	1,198	9,418	y	**1**	y	y	y	**2**	y	y	y	y
2	$3,803,703_1$	$3,615,181_1$	506,577	4,540,040	50,451,944	2,923,670	3,334,579	412,269	4,098,619	51,049,771	**1**	y	y	y	y	y	y	y	y	y
3	7,680	1,069	527	13,118	10,494	7,062	1,079	385	13,216	11,965	y	y	y	y	y	y	y	y	y	y
4	$176,835_1$	35,179	11,908	301,496	$237,831_1$	188,399	20,556	6,303	301,014	181,731	y	y	y	**1**	y	y	y	y	y	y
5	15,148,616	2,376,776	1,265,868	$29,569,671_1$	$29,528,304_1$	13,857,927	2,712,142	1,360,292	27,530,818	41,954,374	y	y	y	y	y	y	y	y	y	**1**
6	1,961,758	410,227	213,611	$4,143,247_1$	$3,924,300_1$	1,820,409	463,307	186,167	4,145,440	7,195,686	y	y	y	y	y	y	y	y	y	**1**
7	443,949	$84,897_1$	$87,032_1$	888,083	1,011,192	468,609	88,900	61,281	968,880	1,276,974	y	y	y	y	**1**	y	y	y	y	y
8	$5,927_1$	$5,967_1$	783	12,575	82,516	9,416	5,046	807	18,857	73,113	**1**	y	y	y	y	y	y	y	y	y
9	4,883,771	998,523	$555,484_1$	*	$10,502,343_1$	4,791,591	966,940	590,958	*	8,757,383	y	y	*	y	y	*	y	*	**1**	*
10	316,080	60,065	35,045	611,338	577,769	316,794	60,918	34,768	614,038	579,235	y	y	y	y	y	y	y	y	y	y
11	468,275	95,146	53,729	$609,650_1$	$706,122_1$	490,699	99,305	45,762	952,338	945,902	y	y	y	y	y	y	y	y	y	**1**
12	2,027,499	399,338	236,727	4,055,786	3,861,338	2,048,114	374,881	218,546	4,411,736	3,533,240	y	y	y	y	y	y	y	y	y	y
13	332,176	72,746	40,378	700,153	599,496	352,538	68,070	44,019	699,140	571,479	y	y	y	y	y	y	y	y	y	y
14	314,843	59,553	34,819	610,160	576,118	313,889	60,465	34,571	607,000	575,834	y	y	y	y	y	y	y	y	y	y
15	4,198,653	751,265	411,862	$7,948,611_1$	$6,554,649_1$	4,131,789	793,522	446,377	5,824,003	7,437,226	y	y	y	y	y	y	y	y	y	**1**
16	398,864	79,689	45,298	$817,519_1$	$766,872_1$	384,229	87,103	46,704	816,971	855,872	y	y	y	y	y	y	y	y	y	**1**
17	979,528	216,878	$109,195_1$	*	$3,346,284_1$	1,022,054	195,280	134,614	*	1,843,900	y	y	*	y	y	*	y	*	**1**	*
18	$184,371_1$	$47,950_2$	$49,151_2$	$129,028_1$	463,251	197,680	34,885	17,165	395,135	410,473	y	y	**1**	y	**2**	y	y	y	y	y
19	$3,957_{1,2,3}$	$1,109_{1,4}$	$1,540_{2,5}$	$7,035_6$	$9,174_{3,4,5,6}$	595	626	1,647	6,840	-4,875	**1**	**2**	y	**3**	y	y	**4**	y	**5**	**6**
20	$22,082_1$	5,116	2,389	52,660	$35,141_1$	24,721	6,648	2,460	44,602	22,646	y	y	y	**1**	y	y	y	y	y	y
21	31,881	30,732	3,980	11,517	484,545	63,043	34,842	5,106	32,808	438,050	y	y	y	y	y	y	y	y	y	y
22	7,800	5,179	680	23,338	50,982	5,355	5,234	1,097	5,741	45,796	y	y	y	y	y	y	y	y	y	y
23	305_1	378_1	85	746	4,020	673	328	80	1,295	5,283	**1**	y	y	y	y	y	y	y	y	y
24	$11,339_1$	$8,468_1$	1,172	34,407	80,798	2,994	7,520	453	8,075	83,951	**1**	y	y	y	y	y	y	y	y	y
25	$14,593_{1,2}$	$15,870_{1,3}$	2,027	$13,633_{2,3}$	235,212	19,727	14,773	2,847	30,431	235,331	**1**	y	**2**	y	y	**3**	y	y	y	y
26	3,968	2,971	357	10,333	31,319	5,610	3,183	336	8,066	33,479	y	y	y	y	y	y	y	y	y	y
27	11,925	8,387	1,139	35,490	89,341	13,099	8,989	1,150	24,478	92,903	y	y	y	y	y	y	y	y	y	y
28	$6,824_1$	$4,871_1$	652	18,773	47,852	4,785	4,851	610	4,949	44,200	**1**	y	y	y	y	y	y	y	y	y
29	$1,910_1$	1,969	250	703_1	26,572	676	1,848	257	1,785	26,735	y	y	**1**	y	y	y	y	y	y	y
30	38,228	34,639	5,221	46,547	681,926	51,402	37,747	7,117	51,553	484,586	y	y	y	y	y	y	y	y	y	y
31	4,473	3,375	393	13,361	35,936	3,823	3,606	411	9,243	37,820	y	y	y	y	y	y	y	y	y	y
32	9,589	7,128	839	27,974	79,687	8,265	8,237	824	25,461	92,974	y	y	y	y	y	y	y	y	y	y
33	2,301	2,162	324	2,817	41,005	2,587	2,203	428	3,261	33,108	y	y	y	y	Y	y	y	y	y	y
										Total Correct:	**26**	**31**	**28**	**30**	**31**	**29**	**32**	**31**	**30**	**25**
										Total Comparisons:	**33**	**33**	**31**	**33**	**33**	**31**	**33**	**31**	**33**	**31**

Table 5.11. Relative Performance Analysis based on Number of Cycles Comparison

Program No.	True No. of Cycles (K Cycles)					Estimated No. of Cycles (K Cycles)					Correct Comparison									
	AVR	PPC	GS	PIC	MB	AVR	PPC	GS	PIC	MB	AVR ^ PPC	AVR ^ GS	AVR ^ PIC	AVR ^ MB	PPC ^ GS	PPC ^ PIC	PPC ^ MB	GS ^ PIC	GS ^ MB	PIC ^ MB
1	16	219	80_{1}	47_{1}	464	-8	184	61	96	471	y	y	y	y	y	y	y	**1**	y	y
2	245,384	1,084,554	303,946	363,203	2,522,597	188,612	1,000,374	247,362	327,890	2,552,489	y	y	y	y	y	y	y	y	y	y
3	495	321	316	1,049	525	456	324	231	1,057	598	y	y	y	y	y	y	y	y	y	y
4	$11,408_{1}$	10,554	7,145	24,120	$11,892_{1}$	12,154	6,167	3,782	24,081	9,087	y	y	y	**1**	y	y	y	y	y	y
5	977,268	713,033	759,521	2,365,574	1,476,415	894,003	813,643	816,175	2,202,465	2,097,719	y	y	y	y	y	y	y	y	y	y
6	$126,557_{1,2}$	$123,068_{1,3}$	$128,167_{2,3}$	$331,460_{4}$	$196,215_{4}$	117,438	138,992	111,700	331,635	359,784	**1**	**2**	y	y	**3**	y	y	y	y	**4**
7	28,640	25,469	$52,219_{1}$	71,047	$50,560_{1}$	30,231	26,670	36,768	77,510	63,849	y	y	y	y	y	y	y	y	**1**	y
8	382_{1}	1,790	470_{1}	1,006	4,126	607	1,514	484	1,509	3,656	y	**1**	y	y	y	y	y	y	y	y
9	315,062	299,557	$333,290_{1}$	*	$525,117_{1}$	309,115	290,082	354,575	*	437,869	y	y	*	y	y	*	y	*	**1**	*
10	20,391	18,020	21,027	48,907	28,888	20,437	18,275	20,861	49,123	28,962	y	y	y	y	y	y	y	y	y	y
11	$30,209_{1}$	$28,544_{2}$	$32,238_{1,2}$	48,772	35,306	31,656	29,792	27,457	76,187	47,295	y	**1**	y	y	**2**	y	y	y	y	y
12	$130,798_{1}$	119,802	$142,036_{1}$	324,463	193,067	132,128	112,464	131,128	352,939	176,662	y	**1**	y	y	y	y	y	y	y	y
13	$21,429_{1}$	$21,824_{1}$	24,227	56,012	29,975	22,743	20,421	26,412	55,931	28,574	**1**	y	y	y	y	y	y	y	y	y
14	20,311	17,866	20,891	48,813	28,806	20,250	18,140	20,743	48,560	28,792	y	y	y	y	y	y	y	y	y	y
15	$270,864_{1}$	225,380	$247,117_{1}$	635,889	327,732	266,550	238,057	267,826	465,920	371,861	y	**1**	y	y	y	y	y	y	y	y
16	$25,732_{1}$	$23,907_{1}$	27,179	65,401	38,344	24,787	26,131	28,023	65,358	42,794	**1**	y	y	y	y	y	y	y	y	y
17	$63,191_{1}$	$65,064_{1}$	$65,517_{2}$	*	$167,314_{2}$	65,935	58,584	80,769	*	92,195	**1**	y	*	y	y	*	y	*	**2**	*
18	$11,894_{1,2,3}$	$14,385_{1,4,5}$	$29,491_{2,4,6,7}$	$10,322_{3,5,6,8}$	$23,163_{7,8}$	12,753	10,466	10,299	31,611	20,524	**1**	**2**	**3**	y	**4**	**5**	y	**6**	**7**	**8**
19	255_{1}	333_{2}	924	563	$459_{1,2}$	38	188	988	547	-244	y	y	y	**1**	y	y	**2**	y	y	y
20	$1,425_{1,2}$	$1,535_{3}$	$1,433_{1,4}$	4,213	$1,757_{2,3,4}$	1,595	1,994	1,476	3,568	1,132	y	**1**	y	**2**	y	y	**3**	y	**4**	y
21	$2,057_{1}$	9,220	$2,388_{1}$	921	24,227	4,067	10,452	3,064	2,625	21,902	y	**1**	y	y	y	y	y	y	y	y
22	503_{1}	$1,554_{2}$	$408_{1,3}$	$1,867_{2,3}$	2,549	345	1,570	658	459	2,290	y	**1**	y	y	y	**2**	y	**3**	y	y
23	20	113_{1}	51	60_{1}	201	43	98	48	104	264	y	y	y	y	y	**1**	y	y	y	y
24	731_{1}	$2,540_{2}$	703_{1}	$2,753_{2}$	4,040	193	2,256	272	646	4,198	y	**1**	y	y	y	**2**	y	y	y	y
25	941	4,761	$1,216_{1}$	$1,091_{1}$	11,761	1,273	4,432	1,708	2,434	11,767	y	y	y	y	y	y	y	**1**	y	y
26	256	891	214	827	1,566	362	955	202	645	1,674	y	y	y	y	y	y	y	y	y	y
27	769	$2,516_{1}$	683	$2,839_{1}$	4,467	845	2,697	690	1,958	4,645	y	y	y	y	y	**1**	y	y	y	y
28	440_{1}	$1,461_{2}$	391_{1}	$1,502_{2}$	2,393	309	1,455	366	396	2,210	y	**1**	y	y	y	**2**	y	y	y	y
29	123_{1}	591	150	56_{1}	1,329	44	554	154	143	1,337	y	y	**1**	y	y	y	y	y	y	y
30	2,466	10,392	$3,132_{1}$	$3,724_{1}$	34,096	3,316	11,324	4,270	4,124	24,229	y	y	y	y	y	y	y	**1**	y	y
31	289	$1,012_{1}$	236	$1,069_{1}$	1,797	247	1,082	247	739	1,891	y	y	y	y	y	**1**	y	y	y	y
32	619	$2,139_{1}$	504	$2,238_{1}$	3,984	533	2,471	495	2,037	4,649	y	y	y	y	y	**1**	y	y	y	y
33	148	649	195	225	2,050	167	661	257	261	1,655	y	y	y	y	y	y	y	y	y	y
										Total Correct:	**28**	**22**	**29**	**30**	**30**	**23**	**31**	**26**	**28**	**29**
										Total Comparisons:	**33**	**33**	**31**	**33**	**33**	**31**	**33**	**31**	**33**	**31**

5.2.4. Discussion

The related research work presented in [42, 61] targets different processor architectures and different application domains. Hence, it is hard to have a direct comparison between such frameworks and the framework presented in this dissertation, especially in terms of the accuracy levels achieved by each framework. Moreover, in this research the functional statistics used to model the candidate processor architectures are obtained from a foreign architecture (the reference model) and a diverse set of application domains, while in those other research efforts, such statistics are obtained from native models (ISSs of the target architectures) for a consistent set domain of applications which in turn (native models and same applications domain) tend to have a better correlation between the functional statistics and the performance measurements, which produces inherently better accuracy. However, the proposed framework herein is considered to be significantly more practical than those frameworks for meeting the objective of this research considering the limited resources at early stages of design process and the need for a fast, flexible (easy to retarget), and high level performance modeling analysis technique. That is, the method used in this dissertation can be used to predict the performance of an application on any processor for which the appropriate benchmark data has been obtained using the reference model (Microblaze) without requiring this new application to be actually compiled and executed on the target processor.

5.3. Summary

Different processor architectures tend to have different timing behaviors. Considering a certain architecture as a reference model, it is possible to extract relationships to correlate the functional statistics of the reference architecture to the performance of the target architectures. While each architecture can have its own relationship with the reference model, it is expected that different program parameters and regression methods are needed to model such relationships. In this chapter, it has been shown that different processor architectures with different ISAs have been able to be modeled using a reference architecture with reasonable and useful accuracy levels.

Chapter 6 Conclusions

In this dissertation, the MicroBlaze/FPGA platform has been introduced as a reference model/real-time profiler to produce quantitative statistics for a given application. While it can be used to determine the dominant operations/functions that govern the target application, the proposed framework along with reference performance statistics on the target architectures have been used for mapping the performance of the reference model (MicroBlaze) to the candidate architectures. Developing the FPGA-based profiler to provide application-independent functional profiling was a major contribution to this framework in terms of it being retargetable to different processors and applications.

Three of the five targeted architectures have reported mean absolutes errors ranging from around 10% to 20% on a set of thirty three programs. Even with the lower level of accuracy reported by the other architectures, the proposed framework has been able to predict the performance of these architectures relative to the others correctly for a major portion of the training set. While it is not a cycle accurate modeling technique, such an approach is considered very useful at the early design stages of embedded systems to assist in the selection of an appropriate algorithm/processor with a minimal knowledge of the application structure and processor architecture.

Chapter 7 Future Work

In this dissertation, a new and novel performance modeling technique has been proposed. This framework can be subject to more efforts to explore its capabilities/limitations. Following are a set of points which can be considered for more research efforts:

7.1. Different Statistical Analysis Approaches

It would be quite interesting to implement different statistical analysis techniques (e.g. nonlinear regression, neural networks, etc) to explore how these modeling techniques perform for different architecture models. As of now, two linear regression methods have been used to validate the theoretical concept of the proposed framework on five different architectures. While different architectures have different relationships with the reference model, it is expected that different analytical models would give variant accuracy levels for each architecture. Moreover, more/different application parameters/events can be monitored to test their effectiveness on different performance models. Domain classification and automatic feature selection are among of the techniques that can be augmented with the framework proposed in this dissertation. In addition, the more programs/benchmarks used, the more better the validation is.

7.2. Multi-Core Processors

Techniques for single processor evaluation can be tailored for the evaluation of multiprocessor architectures. While most of the work done to model the performance of single processors is considered in the context of performance evaluation rather than the processor selection (as an explicit goal), most multiprocessor evaluation proposals are focusing, in contrast, on the selection process as it is assumed that the performance of each single core has been already modeled. Hence, it would be worthy to use the framework developed in this research for decision support at a high-level of abstraction for multi-core processors.

7.3. Power Estimation

While power factor is a major concern in the design of embedded systems, a useful addition would be to extend the proposed framework to model the power consumption of different processor architectures. While this is quite a challenging task, it would be worthy to have a single framework to handle both performance and power modeling at with a minimum of resources.

7.4. Hardware/Software Codesign

While the scope of this dissertation is to construct a framework to assist in selecting an appropriate processor for a certain application, the FPGA-based profiling technique developed for this purpose provides a new, flexible application profiling approach in which modifying the application code needs no (or minimal) changes to the FPGA configuration. That is, the technique developed herein that traces the function's footprint is more flexible than previous techniques of tracing the function's address. This new profiling mechanism can be used in hardware/software codesign to explore the computational intensive portions of the code and their impact on the execution time of the overall application. However, more analysis is needed to measure the FPGA limitations regarding the number functions/events that can be targeted in a single configuration.

References

[1] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3 ed.: Morgan Kaufmann, 2007.

[2] T. Conte, "Choosing the Brain (s) of an Embedded System," *IEEE Computer*, vol. 35, pp. 106-107, 2002.

[3] J. Kreku, *et al.*, "Mappability Estimate: A Measure of the Goodness of a Processor-Algorithm Pair," in *International Symposium on System-on-Chip.* , 2003, pp. 119-122.

[4] S. Rosenthal. Selecting an Embedded Processor Involves both Simple and Nontechnical Criteria. Electronic Article, Available: http://www.sltf.com/articles/pein/pein9706.htm

[5] T. Henzinger and J. Sifakis, "The Discipline of Embedded Systems Design," *IEEE Computer*, vol. 3, p. 4, 2007.

[6] T. Henzinger, "Two Challenges in Embedded Systems Design: Predictability and Robustness," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 366, p. 3727, 2008.

[7] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*: Addison-Wesley Educational Publishers Inc, USA, 2009.

[8] A. Hergenhan and W. Rosenstiel, "Static Timing Analysis of Embedded Software on Advanced Processor Architectures," in *Design, Automation, and Test in Europe (DATE '00)*, 2000, pp. 552-559.

[9] A. Ray, *et al.*, "Estimating Processor Performance of Library Function," in *Second International Conference on Embedded Software and Systems*, 2005, p. 6.

[10] J. Tong and M. Khalid, "A Comparison of Profiling Tools for FPGA-based Embedded Systems," in *Canadian Conference on Electrical and Computer Engineering, CCECE 2007* , 2007, pp. 1687-1690.

[11] J. Tong and M. Khalid, "Profiling Tools for FPGA-Based Embedded Systems: Survey and Quantitative Comparison," *Journal of Computers*, vol. 3, p. 1, 2008.

[12] L. Shannon and P. Chow, "Using Reconfigurability to Achieve Real-Time Profiling for Hardware/Software Codesign," in *ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, 2004, pp. 190-199.

[13] E. Saada, *et al.*, "FPGA-Based Real-Time Software Profiler for Embedded Systems," *International Journal of Computers, Systems and Signals*, vol. 9, p. 28, 2008.

[14] R. Hough, "The Statistics Module: A Portable, Cycle-Accurate Profiling Engine for FPGA Designs, Master's Thesis, May 2007," *Statistics*, vol. 2007, p. 22, 2007.

[15] L. John, "Performance Evaluation: Techniques, Tools and Benchmarks," in *The Computer Engineering Handbook*, ed, 2001.

[16] P. Wang, *et al.*, "Intel® Atom™ Processor Core Made FPGA-Synthesizable," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays* 2009, pp. 209-218.

[17] Xilinx. *ChipScope*. Available: http://www.xilinx.com/tools/cspro.htm

[18] D. Noonburg and J. Shen, "A Framework for Statistical Modeling of Superscalar Processor Performance," in *Third International Symposium on High-Performance Computer Architecture* , 1997, p. 298.

[19] D. Sorin, *et al.*, "Analytic Evaluation of Shared-Memory Systems with ILP Processors," *ACM SIGARCH Computer Architecture News*, vol. 26, pp. 380-391, 1998.

[20] K. Skadron, *et al.*, "Challenges in Computer Architecture Evaluation," *IEEE Computer*, vol. 36, pp. 30-36, 2003.

[21] R. Holsmark and S. Kumar, "Processor Evaluation Cube: A Cassification and Survey of Processor Evaluation Techniques", *A Technical Report, Embedded Systems GroupDepartment of Electronics and Computer Engineering,School of Engineering, Jönköping University*, 2004.

[22] D. Lilja, "Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations," *IEEE Transactions on Computers*, vol. 55, pp. 268-280, 2006.

[23] J. Yi, *et al.*, "The Future of Simulation: A Field of Dreams," *IEEE Computer,* vol. 39, pp. 22-29, 2006.

[24] T. Austin, *et al.*, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer,* vol. 35, pp. 59-67, 2002.

[25] A. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters,* 2002.

[26] K. Skadron, *et al.*, "Branch Prediction, Instruction-Window Size, and Cache Size: Performance Trade-Offs and Simulation Techniques," *IEEE Transactions on Computers,* vol. 48, pp. 1260-1281, 1999.

[27] S. Wallace and K. Hazelwood, "Superpin: Parallelizing Dynamic Instrumentation for Real-Time Performance," in *International Symposium on Code Generation and Optimization, CGO '07*, 2007, pp. 209-220.

[28] W. Qin, *et al.*, "A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-Compiled Instruction-Set Simulation," in *4th International Conference on Hardware/Software Codesign and System Synthesis,* 2006, p. 198.

[29] J. Donald and M. Martonosi, "An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation," *Computer Architecture Letters,* vol. 5, pp. 14-14, 2006.

[30] D. Penry, *et al.*, "Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-Processors," in *The 12th International Symposium on High-Performance Computer Architecture 2006*, 2006, pp. 29-40.

[31] J. Lee, *et al.*, "FaCSim: A Fast and Cycle-Accurate Architecture Simulator for Embedded Systems," *ACM SIGPLAN Notices,* vol. 43, pp. 89-100, 2008.

[32] D. Chiou, "FAST: FPGA-based Acceleration of Simulator Timing Models," in *Workshop on Architecture Research Using FPGA Platforms,* 2005.

[33] E. Chung, *et al.*, "A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations using FPGAs," in *16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays* 2008, pp. 77-86.

[34] Y. He, *et al.*, "FPGA-based Equivalent Simulation Technology (FEST) for Clustered Stream Architecture," in *13th Asia-Pacific Computer Systems Architecture Conference, ACSAC.* , 2008, pp. 1-8.

[35] K. Wang, *et al.*, "Parallelization of IBM Mambo System Simulator in Functional Modes," *ACM SIGOPS Operating Systems Review,* vol. 42, pp. 71-76, 2008.

[36] M. Pellauer, *et al.*, "Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on FPGAs," in *IEEE International Symposium on Performance Analysis of Systems and software, ISPASS 2008* , 2008, pp. 1-10.

[37] R. Desikan, *et al.*, "Measuring Experimental Error in Microprocessor Simulation," in *28th Annual International Symposium on Computer Architecture* 2001, p. 277.

[38] T. Meyerowitz, "Single and Multi-CPU Performance Modeling for Embedded Systems," Dissertation, University of California, Berkeley, 2008.

[39] M. Lajolo, *et al.*, "A Compilation-Based Software Estimation Scheme for Hardware/Software Co-Simulation," in *Seventh International Workshop on Hardware/Software Codesign, (CODES '99),*, 1999, pp. 85-89.

[40] G. Bontempi and W. Kruijtzer, "A Data Analysis Method for Software Performance Prediction," in *Design, Automation, and Test in Europe (DATE'02)*, 2002, p. 0971.

[41] P. Giusto and G. Martin, "Reliable Estimation of Execution Time of Embedded Software," in *Design, Automation, and Test in Europe (DATE '01)*, 2001, p. 0580.

[42] B. Franke, "Fast Cycle-Approximate Instruction Set Simulation," in *11th International Workshop on Software & Compilers for Embedded Systems*, 2008, pp. 69-78.

[43] K. Chen, *et al.*, "Retargetable Static Timing Analysis for Embedded Software," in *14th International Symposium on Systems Synthesis* 2001, p. 44.

[44] Y. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, p. 1477, 1997.

[45] J. Engblom, *et al.*, "Worst-Case Execution-Time Analysis for Embedded Real-Time Systems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4, pp. 437-455, 2003.

[46] R. Wilhelm, *et al.*, "The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, pp. 1-53, 2008.

[47] A. Silva and R. Oliveira, "Methods and Guidelines for Embedded System Processor Selection," in *International Conference on Industrial Applications*, 2006.

[48] D. Thomae, "Processor Selection Using Rate Monotonic Analysis," *Embedded Systems Programming*, vol. 9, pp. 34-51, 1996.

[49] B. Martin-del-Brio, *et al.*, "Self-Organizing Maps for Embedded Processor Selection," *Microprocessors and Microsystems*, vol. 29, pp. 307-316, 2005.

[50] R. H. Klenke, *et al.*, "A High-Throughput Processor for Flight Control Research Using Small UAVs," in *25th AIAA Aerodynamic Measurement Technology and Ground Testing*, San Francisco, California, 2006.

[51] J. Vaglica, *et al.*, "How to Select a Microcontroller," *IEEE Spectrum*, vol. 27, pp. 106-109, 1990.

[52] N. Parimala and S. Reddy, "Processor Selection for Embedded System Design," *International Journal of Computers & Applications*, vol. 30, pp. 348-353, 2008.

[53] P. Lapsley, *et al.*, "Evaluating DSP Processor Performance, *White Paper from Berkeley Design Technology, Inc*," 1996.

[54] M. Guthaus, *et al.*, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *IEEE 4th Workshop on Workload Characterization*, 2001, pp. 3-14.

[55] *EDN Embedded Microprocessor Benchmark Consortium (EEMBC)* Available: http://www.eembc.org/home.php

[56] J. Soininen, *et al.*, "Mappability Estimation Approach for Processor Architecture Evaluation," in *20th IEEE NORCHIP Conference*, 2002, p. 171.

[57] J. Soininen, *et al.*, "Mappability estimation of architecture and algorithm," in *Design, Automation and Test in Europe* 2002, p. 1132.

[58] L. Carro, *et al.*, "A Design Methodology For Embedded Systems Based On Multiple Processors," in *Distributed And Parallel Embedded Systems*, 2001, p. 33.

[59] A. Ray, *et al.*, "Practical Techniques for Performance Estimation of Processors," in *Fifth International Workshop on System-on-Chip for Real-Time Applications*, 2005, pp. 308-311.

[60] T. Gupta, *et al.*, "Processor Evaluation in an Embedded Systems Design Environment," in *VLSI Design*, 2000, pp. 3-7.

[61] M. Oyamada, *et al.*, "Applying Neural Networks to Performance Estimation of Embedded Software," *Journal of Systems Architecture*, vol. 54, pp. 224-240, 2008.

[62] M. Oyamada, *et al.*, "Software Performance Estimation in MPSoC Design," in *Asia and South Pacific Design Automation Conference, ASP-DAC '07*, 2007, pp. 38-43.

[63] M. Oyamada, *et al.*, "Accurate Software Performance Estimation using Dmain Cassification and Nural Ntworks," in *17th Symposium on Integrated Circuits and System Design*, 2004, pp. 175-180.

[64] (2004, *The International Technology Roadmap for Semiconductors (ITRS)*. Available: http://public.itrs.net/

[65] J. Henkel, *et al.*, "COSYMA: A Software-Oriented Approach to Hardware/Software Codesign," *Journal of Computer and Software Engineering*, vol. 2, p. 314, 1994.

[66] F. Balarin, *et al.*, *Hardware-Software Co-Design of Embedded Systems: the POLIS Approach*: Springer Netherlands, 1997.

[67] *Cadence, VCC2.1 Production Documentation*. Available: http://www.cadence.com/us/pages/default.aspx

[68] *Seamless: Ideal Platform for HW/SW Integration*. Available: http://www.mentor.com/products/fv/seamless/

[69] *CoWare: Model Library*. Available: http://www.synopsys.com/Tools/SLD/Pages/default.aspx

[70] A. Halambi, *et al.*, "EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability," in *Design, Automation & Test in Europe (DATE '99)*, 1999, pp. 31-45.

[71] S. Pees, *et al.*, "LISA—Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures," in *36th Annual ACM/IEEE Design Automation Conference (DAC'99)*, 1999, pp. 933-938.

[72] V. Rajesh and R. Moona, "Processor Modeling for Hardware Software Codesign," in *12th International Conference on VLSI Design - 'VLSI for the Information Appliance'* 1999, p. 132.

[73] G. Hadjiyiannis, *et al.*, "ISDL: An Instruction Set Description Language for Retargetability," in *34th Annual ACM/IEEE Design Automation Conference, (DAC'97)*, 1997, pp. 299-302.

[74] S. Chandra and R. Moona, "Retargetable Functional Simulator using High Level Processor Models," in *13th International Conference on VLSI Design*, 2000, pp. 424-429.

[75] S. Przybylskic, "A Survey of RISC Architectures for Desktop, Server, and Embedded Computers," in *Computer Organization and Design: The Hardware/Software Interface*, ed.

[76] H. Chia Hsiang, *et al.*, "Energy Saving Based on CPU Voltage Scaling and Hardware Software Partitioning," in *13th Pacific Rim International Symposium on Dependable Computing, PRDC* , 2007, pp. 217-223.

[77] G. Stitt, *et al.*, "Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems," *ACM Transactions on Embedded Computer Systems*, vol. 3, pp. 218-232, 2004.

[78] *MatLab Statistics Toolbox*. Available: http://www.mathworks.com/access/helpdesk/help/toolbox/stats/bq_676m-2.html#bq_676m-19

[79] D. Powell and B. Franke, "Using Continuous Statistical Machine Learning to Enable High-Speed Performance Prediction in Hybrid Instruction-/Cycle-Accurate Instruction Set Simulators," in *7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, 2009, pp. 315-324.

[80] E. İpek, *et al.*, "Efficiently Exploring Architectural Design Spaces via Predictive Modeling," *ACM SIGOPS Operating Systems Review*, vol. 40, p. 206, 2006.

[81] P. Thazhuthaveetil, "Construction and Use of Linear Regression Models for Processor Performance Analysis," in *The 12th International Symposium on High-Performance Computer Architecture*, 2006.

[82] B. Lee and D. Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction," in *12th International Conference on Architectural Support for Programming Languages and Operating Systems* 2006, p. 194.

[83] S. Dey, *et al.*, "An Approach to Software Performance Evaluation on Customized Embedded Processors," in *21st International Conference on VLSI Design, VLSID.* , 2008, pp. 111-117.

[84] *Xilinx MicroBlaze*. Available: http://www.xilinx.com/tools/microblaze.htm

[85] *Altera NIOS* Available: http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[86] *LatticeMico32*. Available: http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/index.cfm

[87] *gprof*. Available: http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

[88] *AMD Athlon Processor, x86 Code Optimization Guide*. Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf

[89] D. Gove. Using UltraSPARC-IIICu Performance Counters to Improve Application Performance. Available: http://developers.sun.com/solaris/articles/pcounters.html

[90] Xilinx, "MicroBlaze Processor Reference Guide."

[91] D. K. Ward Cheney, *Numerical Mathematics and Computing*, Fifth ed.

[92] *AVR32 uC*. Available: http://www.atmel.com/dyn/resources/prod_documents/doc32002.pdf

[93] *PPC405* Available: http://www.xilinx.com/support/documentation/user_guides/ug011.pdf

[94] *GumStix*. Available: http://pubs.gumstix.com/documents/PXA%20Documentation/PXA270/PXA270%20Processor%20Developer%27s%20Manual%20%5B280000-002%5D.pdf

[95] *PIC32*. Available: http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2601

[96] *SimpleScalar*. Available: http://www.simplescalar.com/

[97] A. Aburto. *hanoi*. Available: ftp.nosc.mil/pub/aburto

[98] R. P. Weicker, *Dhrystone*, 1996.

[99] C.-T. F. a. (C). Available: http://en.literateprograms.org/Cooley-Tukey_FFT_algorithm_(C)?oldid=15458

[100] *Sorting Algorithms*. Available: http://www.concentric.net/~ttwang/sort/sort.htm

[101] M. A. Weiss, *Data Structures and Algorithm Analysis in C*, 1997.

[102] P. J. Deitel, *C How to Program* vol. 2010.

[103] *Dijkstra's Algorithm*. Available: http://www.dreamincode.net/code/snippet2032.htm

[104] *Fuzzy Controller Program*. Available: http://www.ece.osu.edu/~passino/FCcode.txt