2007

Low-level estimation at high-levels of abstraction in system-level design

Joseph Paul Schneider
Iowa State University

Low-level estimation at high-levels of abstraction in system-level design

by

Joseph Paul Schneider

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Diane Rover, Major Professor
Zhao Zhang
Andrew Miner

Iowa State University

Ames, Iowa

2007

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Embedded systems are becoming increasingly complex with shortening time-to-market demands. System-level modeling and design have been proposed to help embedded system development keep pace with this complexity. In a system-level design environment, a designer is able to delay critical design decisions until late in the design cycle, reducing the risk of making incorrect decisions which could require a costly redesign. New methods of estimating system-level performance must be devised to accommodate these needs.

In embedded systems composed of off-the-shelf parts, performance can be roughly estimated using part documentation. However, this process can provide poor estimates. Additionally, if the design includes a custom part, there may not be detailed documentation from which to gather performance estimates. The exhaustive gathering of estimates is error prone and tedious. In this thesis we present a novel estimation technique called minimal characterization for creating system-level estimation metrics. We show that estimates can be orders of magnitude more accurate, without any loss in fidelity, using a small number of source-level metrics. We show results from applying a source-level performance estimation technique generally used on software systems to a system-level design that is implemented in both software and hardware targets. Finally, we present a categorization of secondary execution factors which can greatly affect the accuracy of system-level estimates but have only been peripherally addressed in other approaches.

CHAPTER 1. OVERVIEW

Traditionally embedded systems are designed in a waterfall approach, where the hardware platform is first chosen with little concrete evidence that the hardware will achieve design goals. Software implementation often cannot begin until the hardware platform has been determined, and in some cases the software implementation must wait for the physical hardware platform to be finalized. It is only after the software is then implemented that the system designer has a first chance to measure the performance of the system. If the chosen hardware platform cannot supply the needed processing resources for the embedded application to meet its requirements, the entire design cycle must be restarted. Embedded system designers are turning to tools to keep productivity high while facing ever-increasing complexity and shortening time-to-market deadlines.

1.1 Introduction

The embedded design industry needs a completely new level of abstraction that hides details from designers and allows design decisions to be delayed as late as possible in the design cycle. To address this new level of abstraction, system-level design (SLD) languages and tools have been implemented, and a new methodology for designing embedded systems has been created. Delaying decisions, such as selection of processing elements or interconnect, not only decreases the risk of a redesign, but also allows the platform to be optimized for the desired behavior and costs. Optimization at the system-level requires new processing-element agnostic profiling techniques in order to provide as much information as possible, as early in the design as possible, without forcing the designer to commit to a

design decision. With efficient profiling techniques, a system-level design tool can even make design decisions automatically, choosing which functionality requires a dedicated hardware resource compared to functionality which can still meet design requirements if run in a traditional processor. A new type of profiler is needed which will provide just enough information to make the correct decision with the information available, without providing too much information that has cost the designer either time or implementation.

1.2 Motivation

The system-level profiler contemporary embedded designs require has several design constraints. It must work at all levels of abstraction, allowing the designer to get useful information at high levels of abstraction, in addition to determining the final performance of the system once design decisions are made. It must also be fast enough to allow a partitioning algorithm to profile large numbers of candidate designs quickly, a task called design-space exploration. Finally, the profiler must have enough fidelity so that correct design decisions are made based on feedback from the profiler result.

1.3 Thesis Statement

In order to provide fast and accurate system-level estimates, low-level information can be obtained from implementation-specific benchmarking. This allows more accurate estimates at high levels of abstraction. Additionally, the time required to acquire high-level estimates can be shortened by removing unimportant information. Designers also need to be aware of all performance effects included in their system-level estimates using a common set

of execution performance categories so they are able to utilize the system-level estimates wisely.

1.4 Approach

This thesis discusses several areas in which to improve system-level estimation. We begin by establishing a common language for estimation abstraction, using categories to group performance parameters in a system. Next, we use an established estimation technique which analyzes the system-level design at the source-code level to determine how much information we can take away from the model without losing estimation accuracy or fidelity. Finally, we can apply proposed estimation techniques to real performance data to determine if the resulting estimates provide more value than previously accepted estimates.

1.5 Contributions

The contributions of this paper are:

- A categorical analysis of performance estimation at all levels of abstraction
- A minimal characterization estimation technique reducing the number of necessary metrics to obtain a performance estimate of a system
- Application of a system-level source code performance analysis to real performance metrics

1.6 Thesis Organization

We begin by presenting a background of system-level design and system-level estimation in Chapter 2. Next, Chapter 3 presents the details of execution performance analysis and provides a discussion on the categories of secondary execution effects in

system-level estimates. Chapter 4 then develops a method of removing unimportant data from system-level estimation tables. Chapter 5 applies the developed method to real performance measurements, and Chapter 6 then presents related work. Finally, Chapter 7 summarizes the work and discusses future work.

CHAPTER 2. BACKGROUND

As embedded system design become more complicated, it becomes increasingly difficult to create designs that optimize both performance and cost. In a traditional workflow, the embedded system's hardware platform is designed before software work begins. Because the software is not available, hardware designers must make educated guesses regarding the computational complexity of their application. In order to guarantee that the hardware platform will provide the necessary computational resources to meet requirements when the software is completed, designers must often over-design a system by including more computational power than the application requires. System-level estimation uses an executable simulation of the application to gain insight into the computational complexity before making hardware or software decisions, allowing designers to make optimal decisions early in the design process.

2.1 System-level Design Overview

In order to discuss system-level estimation, a basic understanding of system-level design (SLD) is necessary. In section 2.1.1 some common SLD definitions are presented. In section 2.1.2 the basic system-level design refinement process is shown.

2.1.1 Definitions

A common terminology is useful when discussing SLD, since many of the concepts are abstract. The following are terms used throughout the document.

Design Space – The design space represents the entire set of possible hardware and software choices available for a design.

Model of Computation (MOC) - A level of abstraction and associated rules for modeling a system's behavior. The system model generally starts with an abstract MOC that is iteratively refined during design into less abstract models until a final implementation model is achieved.

Behavior- A set of computational instructions having been separated from any external communication. System-level models generally have many behaviors representing the computational work performed by the system.

Processing Element (PE) – A processing element is a computing resource, either software or hardware, that can be used to execute behaviors.

Weight table – A large set of metrics used to estimate the performance of a system-level design.

2.1.2 System-level Design Refinement Process

Abstraction through MOCs is the key tool used in SLD to limit the complexity of the system. By using estimation at each of the abstraction levels, the designer can make informed design choices based on the application's constraints, rather than guessing or over-designing. The abstraction level of the design is directly related to the degree of accuracy of the estimates that can be extracted. A standard set of MOCs are present in recent literature and this paper will adopt the same abstraction terminology.

In this system-level design flow, a designer will create an executable system-level design at the specification level. Once the functionality of the model has been verified, the designer performs a series of refinements, which helps define the hardware platform. At the

end of the design process, the designer has used profiling information at each level of abstraction to make profile-guided design decisions, resulting in an implementable design.

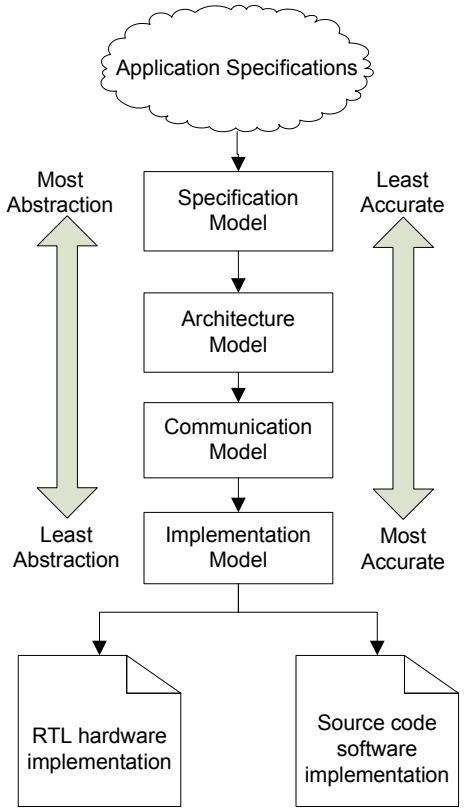

Figure 1: Trade-offs between models of computation

At the specification level, no knowledge of the implementation platform is known, but the functional behavior of the system is correct and can be simulated and verified. However, at this level none of the operations have any cost associated with them. This level of abstraction is untimed, meaning all operations occur in zero time. Although the specification model does not contain timed estimates, it is not entirely useless for system-level estimation. Using a data-flow graph constructed from this model, a designer can

estimate the number of operations, the distribution of the operations in both size and type, and the ability of the given application to utilize parallel computation.

The next level of abstraction, the architecture level, specifies the processing elements available to the system. Estimates at this level are timed and can provide a number representing the computational cost of executing behaviors on particular processing elements.

At the communication level, timed communication buses are added to the design to account for communication time between processing elements. The communication protocol, bus width, and arbitration methods all are specified and appropriate behaviors are added to the model. This refinement adds communication costs to the timing of the model.

Finally, the implementation level produces the partitioned design in the form of files that each processing element's toolset natively understands. This is generally C source code for software targets and VHDL for hardware targets. At this point, cycle-accurate simulations can be performed to obtain good estimates of system performance.

2.2 Weight-table Based Estimation Approaches

In this thesis we utilize a common estimation framework, which uses a static/dynamic hybrid estimation approach. In this type of estimation, the static portion of the estimation captures computational complexity for a single behavior, whereas the dynamic portion captures the control flow of the function. Together these two metrics give the total computational complexity of the function. In order to translate computational complexity into a performance metric such as time, a mapping is created for each processing element. In the following sections we describe each of these steps in detail, building the case for a

categorical analysis of deficiencies in system-level estimation and demonstrating the need for directed profiling.

2.2.1 Static Analysis

The static analysis begins with a source-level specification. The specification can represent the system in any of the abstraction levels. Each basic block is statically analyzed to generate an operation profile, which represents the computational complexity of the function without consideration of the control paths. The operator and the type of operands are considered together, and the frequency of each pair is counted and stored.

In the case of the estimator described in [24], the system-level design is specified in the SpecC language. To begin profiling, the design is scanned for C operators and keywords. An example of operators in this language are the common mathematical operators such as +, -, *, /, and %. Since the cost of implementing many of the operators depends on the type of operands used in the calculation, the profiler separates the statistics on each operator by type of operand. For example, it is useful to know if a multiplication is being performed with integer or floating-point operands, since the data type can greatly affect estimation results.

Several researchers have noted that performing static analysis in a high-level language such as C is difficult because the compiler is able to optimize away portions of the code. If the compiler is able to reduce the computational complexity of the code by replacing or eliminating operators in the original code, then the static profile may not accurately represent the post-compile computational complexity. To address this problem, the source code can be transformed to lower-level C code [18] or to a virtual instruction set [11]. In

both cases, this compilation allows a more accurate static analysis by accounting for compiler optimizations.

2.2.2 Dynamic Analysis

To perform the dynamic analysis, the system-level behavior description is instrumented at the basic-block level. The dynamic instrumentation is performed at the basic-block level, and is similar to the way traditional profilers such as gprof [25] capture the control flow of a system. The system is then compiled into an executable specification. The executable specification is then run, and the profiling results are communicated back to the estimator through a file or other communication method. By combining the static and dynamic information for a particular basic block, the total number of times each operator-operand pair is executed can be calculated.

In most cases, the execution of the specification will require a set of test data for input to the model. Note that the selection of the test data is expected to represent typical test data in order to optimize the system for the average case. However, choosing a test set which exposes the average computational complexity may not be appropriate for a system with a hard deadline, where the system has a specific performance requirement that must be met for safety reasons. In a hard real-time system, worst-case execution time (WCET) estimates are more appropriate [28]. The estimates dependence on choosing an appropriate set of test data represents a point of weakness in this estimation approach, since a designer may not be able to reasonably know whether the test data is appropriate for a system.

The execution profile generated by this process is useful immediately, as it contains information about the computational requirements of a design. For instance, graphing the

number of floating-point operations versus the number of integer operations can give the designer an idea of the sensitivity of the design to adding an FPU to the implementation platform.

2.3 Processing Element Characterization

As [7] describes, the execution profile vector can be multiplied by a vector mapping operator-operand pairs onto execution time, in either cycles or absolute time. This multiplication is mathematically simple and can be performed in a short amount of time. The inner product of the vectors gives the total execution time spent in a particular basic-block. Summing all the basic blocks together will give the execution time for the entire system. This abstraction is quite powerful, and several researchers have noted that a vector can be created to map operator-operand pairs onto power usage or area to obtain estimates of those metrics from a system-level design. This thesis also recognizes that this generalization is possible, but estimation of any factor other than execution time is outside the scope of this thesis.

The preferred implementation of this framework, shown in Figure 2, is to create a database of processing element characteristic vectors before estimation takes place, which can be retrieved and used in a system-level estimate on demand. During design space exploration, the cost of candidate architectures is estimated by combining the previously generated behavior characteristics and the processing element (PE) characteristics.

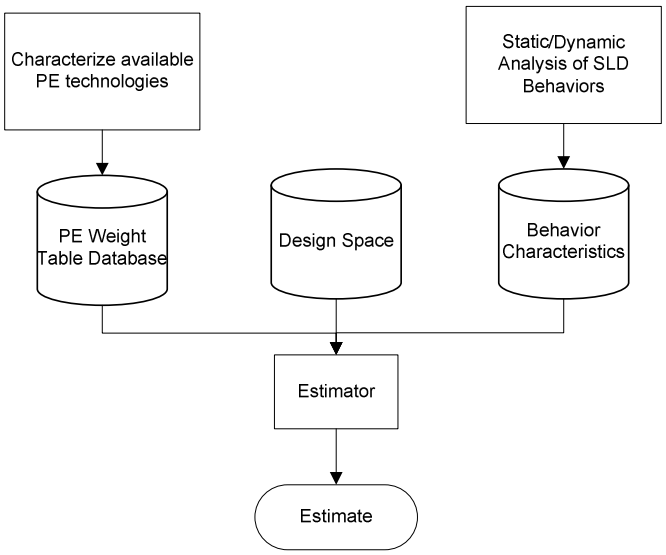

Figure 2: Preferred estimation framework

Generating the vector which maps operator-operand pairs onto execution time is a difficult problem which is not extensively discussed in current literature. As a starting point, [17] suggests that CPU processing elements should use the CPU's instruction set architecture (ISA) manual to determine cycles-per-instruction. However, this approach results in estimates which ignore caching affects, instruction-level parallelism, and other secondary effects which have a large impact on execution performance. A designer can easily imagine a system in which instruction-level parallelism may be the deciding factor in a design choice between two processors. In this case, a characterization based on the ISA manual would result in an incorrect design decision.

Another option for getting CPU operand timing is to create a large set of test-bench data. If the test-bench is analyzed both statically and dynamically using the profiling methods previously described, the computational requirements of the test-bench can be derived. Additionally, if the test benches are then run on actual hardware or in a co-

simulation environment, the actual performance of the test benches can be measured. Using this data, you can then perform a linear regression to determine the best-fit values for each of the operator-operand weights for the target platform in question

2.4 Summary

In this chapter we discussed the basic process of system-level design, the definitions of models of computation, and the relationship between models of computation and estimation. We also discussed the method by which we can estimate system performance by using a static and dynamic analysis of the system-level source code.

CHAPTER 3. EXECUTION PERFORMANCE ANALYSIS

In this section we introduce the execution performance factor categories, presented in a novel way to group execution performance factors. Then, in the context of execution performance factors, we introduce a performance analysis tool.

3.1 Categorical Analysis of Execution Performance Factors

3.1.1 Motivation

Accurate estimation in both the software and hardware domain, especially in a high-level language such as C, is difficult because we must estimate the impact of a large number of secondary effects. In order to clearly discuss these effects, a set of categories of execution performance factors (CEPF) is needed. Without clearly defining all factors included in the estimates, the estimates cannot be compared to each other. For example, if execution is estimated using a weight table that is based on the instruction set manual of a processor, those estimates cannot be compared to values measured from an actual platform because the platform includes many more secondary timing effects that have a non-trivial influence on the execution performance.

3.1.2 Categories of Execution Performance

The categories of execution performance factors are presented in Figure 2. The purpose of this figure is two-fold. First, it provides a common terminology for discussing estimation problems. Secondly, it encourages collaboration between the software and hardware estimation communities by exposing commonalities in the estimation process.

The top of Figure 2 shows the usual design process in action. A system-level design is specified, refinements are made, and the specification is then updated to reflect the refinements. Below the refinements are the steps toward implementation in two columns, one each for hardware and software platforms. Each of the categories represents a set of secondary factors that is present in the implementation, but is not explicitly present during design space exploration.

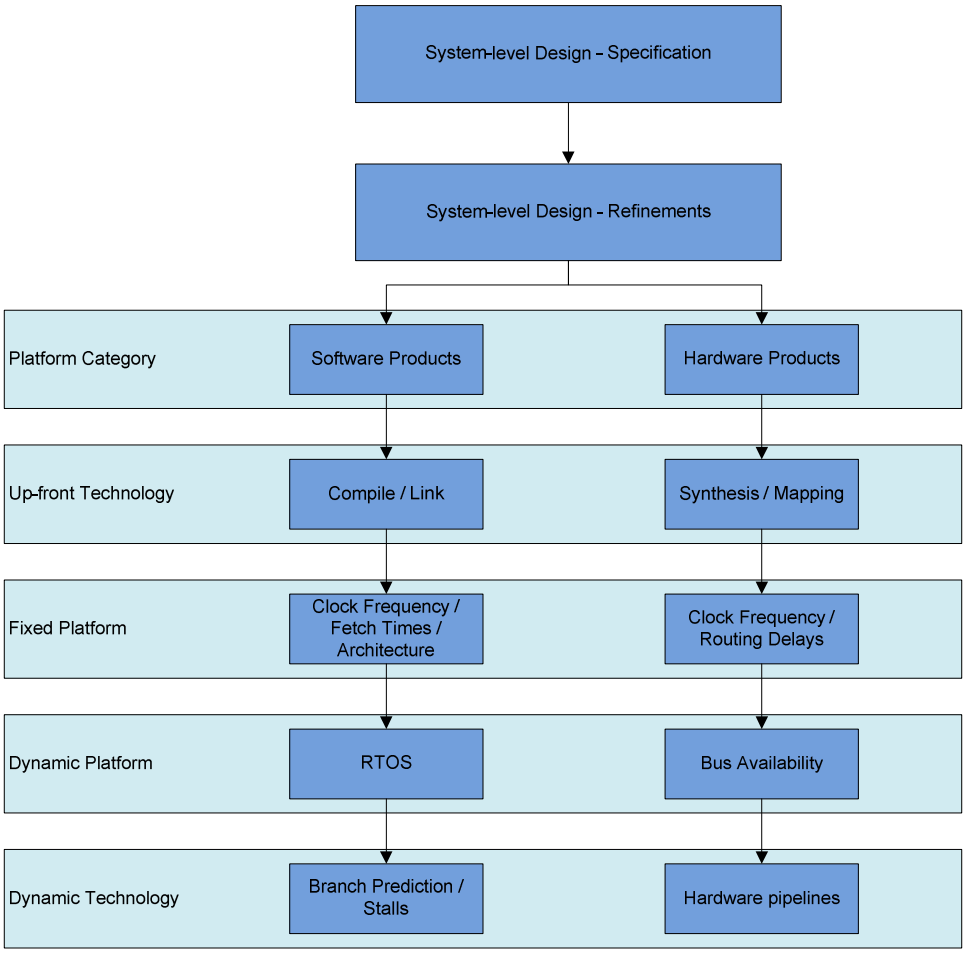

Figure 3: Categories of factors affecting execution performance not represented at the source-code level

3.1.2.1 Platform Category

The first set of factors is the platform category of the target PE. In order to implement the system-level design in a particular technology, it must be translated into a language suitable for standard software or hardware toolsets. That translation can affect the execution performance of the behavior.

Since most SLD languages are in a C-like format, the translation to a software target is trivial and the execution is not greatly affected. However, for a hardware target, consider the difficult problem of translating C code into VHDL. Several systems have been created to perform this translation and are available as commercial products today [29][30][31]. These translators, at a minimum, create a data-flow graph, calculate data dependencies, and schedule the operations in a given C function to maximize the resources available and minimize the schedule length in cycles. This minimizes the execution time of a specification by allowing operations to run in parallel, a translation which is not available on a simple software platform.

3.1.2.2 Up-Front Technology

The up-front technology category is the next area that affects execution time. After having been translated, the specification is ready to enter the tool chain for the given technology. In most cases, the tool a compiler for a software target, and a synthesis cycle for hardware targets. In both of these scenarios, the tool can perform a large number of instruction-level or register-transfer-level (RTL) optimizations as it generates its output. Examples of the types of tool optimizations that fit in this category would be loop unrolling

in a compiler, or unused bit elimination in a register that is too wide in a synthesis tool. In either case, the execution time can easily be affected.

After the hardware bitstream or software image is created, a fixed penalty is imposed by the platform and its surroundings. The clock frequency directly affects the speed with which both the hardware and software targets perform their work. In addition, software targets must fetch instructions over a bus of some kind, which may result in a stall of the CPU core while it waits for memory to be accessed. On the other hand, in hardware routing delays and delays imposed by the speed of the silicon used can affect the maximum clock speed. The speed of the clock affects the execution performance in both hardware and software implementation, but is not clearly modeled in some system-level languages.

3.1.2.3 Dynamic Platform

Next, dynamic platform factors which affect the execution time are considered. These factors generally represent services provided by the platform that have varying degrees of availability and can affect system performance. In the software realm, the real-time operating system (RTOS) is considered a dynamic platform factor, whereas hardware elements can be limited by bus availability. This category may require system-level knowledge in order to provide estimates of the platform as a whole.

3.1.2.4 Dynamic Technology

Finally, dynamic technology constraints model the per-cycle factors affecting execution performance. In a software target, this category includes the micro-architecture of the CPU, including any sort of predictions, cache hit rate, data stalls, or similar effects. In the hardware realm, an example of this type of factor is a pipelined hardware, in which the

throughput, and consequently performance, is greatly affected by the architecture of the hardware element. Note that pipelines can affect performance either positively or negatively based on how many data items are presented to the core at one time, and the number of times the pipeline must be flushed.

3.1.3 Summary

The categorical analysis of execution factors represents a contribution to the discussion of execution parameters. Previously it was not always clear whether execution estimates included all the factors necessary to compare estimates to real-world measurements. The categories provide a means to communicate exactly which factors are included in reported estimates.

3.2 SCE Performance Analysis Tool

The System-on-Chip Environment (SCE) tool is a system-level design environment developed by the Center for Embedded Computer Systems (CECS) at University of California, Irvine [19]. It provides a comprehensive set of system-level modeling and analysis tools, which we used for SLD performance analysis in this thesis. This section details how the performance analysis is performed at a high level, defines the CEPF categories that SCE supports, and defines the level at which we were able to report execution performance estimates.

3.2.1 SCE Overview

SCE, which stands for SoC Environment, is a SLD environment using the SpecC language for SLD specification. A screenshot of the tool is shown in Figure 4. SCE allows

the user to import functionally complete specification level designs which can then be refined through the MOCs into an implementation level design. Also within SCE is a source-code profiler implementation, the details of which are described in [18] and [24]. SCE uses the preferred estimation model presented in Figure 2 to measure system-level performance, meaning it has a processing element (PE) database which stores a weight table for each PE. As described in section 2.3, a hybrid static/dynamic source code analysis is used to determine the computational complexity of a behavior. Once the behavior is associated with a processing element, the behavior characterization is combined with the PE's weight table to determine estimated execution performance.

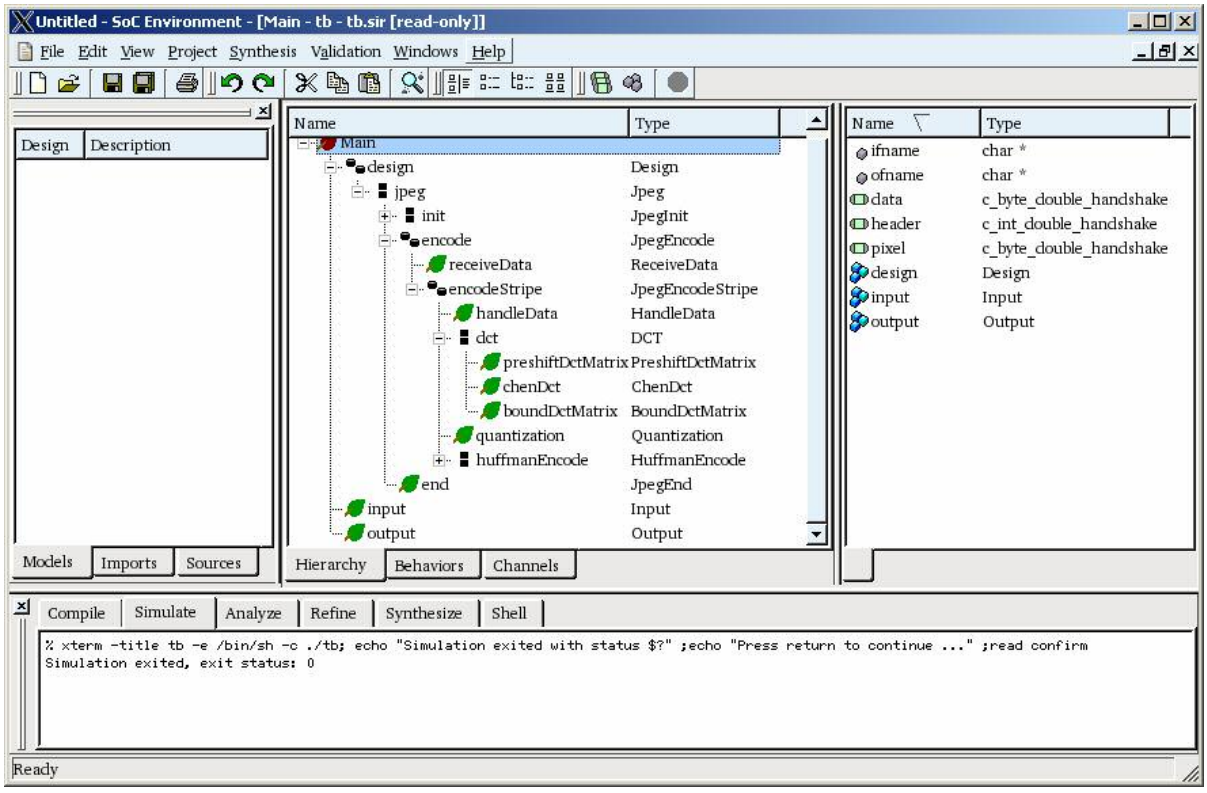

Figure 4: SCE tool for system-level design

3.2.2 PE Weight Tables in SCE

Of particular interest to this thesis is the PE weight table database. SCE provides a weight table of two dimensions, one for the computation "type" and the other for the computation "operation". A type represents the type of the operands being worked on, while the operation represents the operator performing the computation. For example, this approach allows a designer to provide different execution estimates for integer and floating-point multiplies.

When calculating a system-level estimate with this framework, the behaviors in the system are first characterized using static/dynamic methods discussed in section 3. The output of the tool is written to a tab-delimited file and includes the static, dynamic, and total counts for the behavior characterization.

An unfortunate drawback of this approach is that it is not clear how to fill in this weight table to provide useful estimates. A screenshot of the weight table entry screen is shown in Figure 5.

There are a total of 17 types and 79 operations for a total of 1343 different weights. How should a designer characterize a PE in this form to get good estimates?

One approach to fill in the weight table for a PE with meaningful values is to use the instruction set architecture documents associated with a processor. However, this method has several drawbacks:

- It only makes sense for software PEs. What would be the instruction set for a hardware target?
- It does not account for secondary execution effects

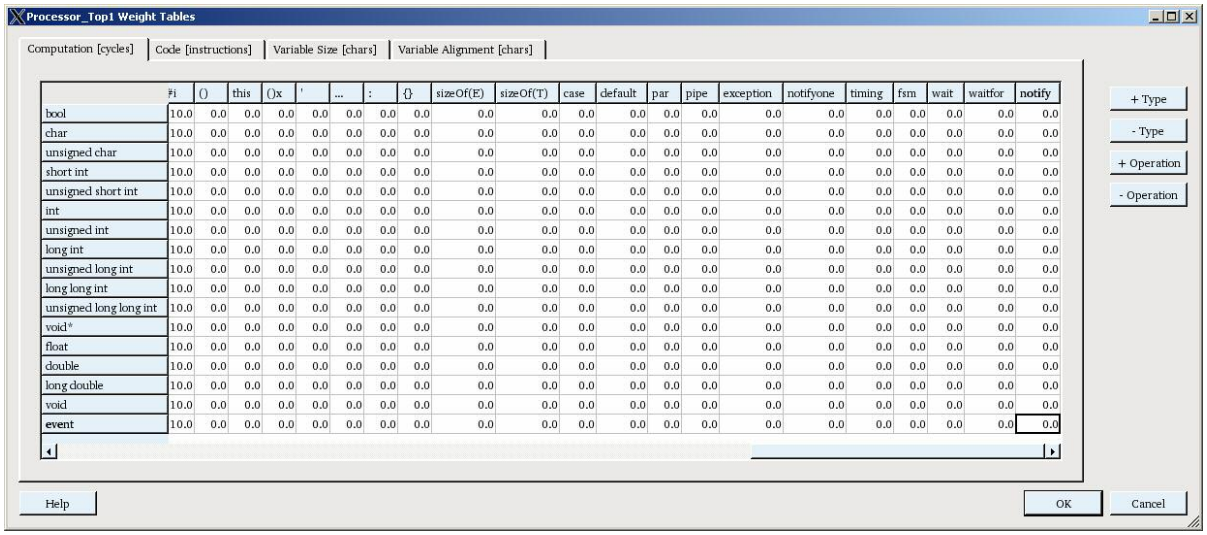

Figure 5: Weight table entry for a PE in SCE – 1343 parameters

- A human must perform an manual translation between the high-level SpecC operators and the low-level assembly instructions, a process that is open to developer interpretation
- A large weight table is required to perform estimates, even though it seems intuitive that much of the information will be redundant

Despite its drawbacks, using the instruction set architecture on software PEs as a first-pass estimator provides more value than no estimate. While [10] and [18] disagree on the fidelity of this approach, it seems intuitive that the instruction set should provide estimates that correlate with the execution time. However, [10] provides a least-squares estimation that provides much better accuracy which we adopt for testing.

3.2.3 Summary

In this chapter we introduced a novel categorization framework for factors affecting system-level performance. This can provide a common way to discuss execution

performance and provides insights into system-level performance factors affecting hardware and software. We also introduced the SCE tool for system-level design. We showed that it provides a useful environment for the study of system-level estimation techniques. A PE database with weight tables for each PE can be created using the SCE tool. These weight tables can then be used in system-level estimates to investigate design decision fidelity in multiple scenarios.

CHAPTER 4. MINIMAL CHARACTERIZATION

In this chapter we show that a large portion of the performance information is contained within a small set of source-level operators, allowing us to reduce the number of metrics needed to estimate performance. We evaluate the fidelity of the minimal characterization against a large benchmark, finding that it can indeed provide good fidelity.

4.1 Motivation

Our goal is to obtain reasonably accurate system-level performance estimates at a high level. Low-level performance information can be used in high-level designs to estimate high-level performance. However, it is desirable to limit the amount of low-level information needed. Limiting the amount of performance metrics allows the estimation to occur more quickly, a trait which is important when the design space is very large. Additionally, collecting performance information for PEs is expensive. Performance characteristics for PEs that do not physically exist may require time-intensive simulations, while PEs that do physically exist may require extensive test setups to acquire accurate timing. Finally, complete solutions for all performance characteristics require a large set of benchmark applications, a suite of which does not currently exist in a system-level format.

Figure 6 shows the spectrum of estimate accuracy for system-level design. Low-level-unaware estimates do not include any secondary estimation effects; this includes many currently reported models which use instruction set cycle timing. Minimally characterized models use measured performance data to create a model with enough accuracy to make a design decision; however these models omit a large number of characterization

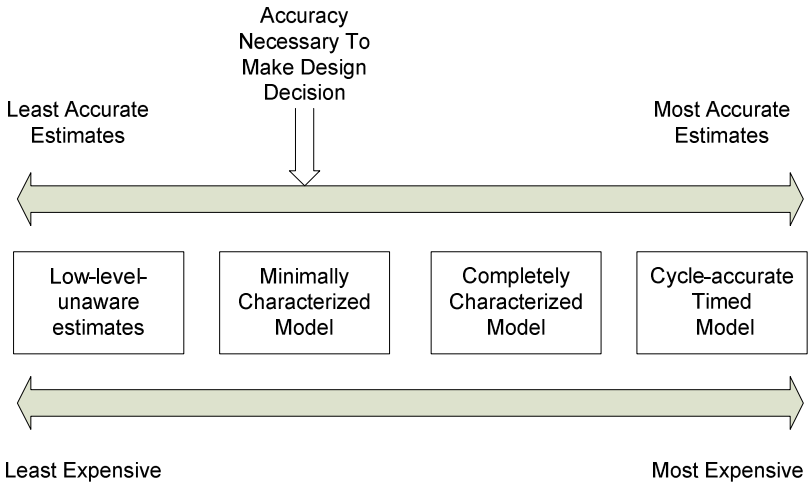

Figure 6: Estimation accuracy spectrum for system-level design

metrics which are unimportant to the design decisions. Completely characterized models are generated from measured performance, just as minimally characterized models are generated. However, extra effort is required to determine all performance metrics, even those unimportant to design decisions. Creating a cycle-accurate timed model requires a large amount of design effort, but it has even better accuracy than a weight-table based method because it accurately tracks the state of the system. In this section we focus on the creation of the minimally characterized model.

4.2 Experimental Setup

To test the feasibility of using a subset of the source-code characteristics to estimate execution performance, a JPEG (Joint Photographic Experts Group) encoding example provided by UCI's CECS group was used [26]. The JPEG encoder represents a real-world example and utilizes many of the SpecC language constructs used in the estimation process, such as parallel execution and pipelined execution, providing a good test bed for execution

estimation evaluations. We characterize the JPEG encoder using the SCE tool introduced in section 3.2.

During performance evaluation of the code, the SCE tool calculates the number of times each operator is executed by the design. Some of the operator labels can be cryptic, so the definitions for operators used in this section are shown below in Table 1.

Operator	Description
#i	Variable Access (read or write)
#1	Constant Access (read or write)
()	Function call
[]	Array access
{}	Basic-block

Table 1: Important operators used in source-level estimation and their descriptions

4.3 Applying Minimal Characterization

As a first step, the top-level behavior of the JPEG encoder was characterized in terms of operator frequency by the static-dynamic code analysis tool built into the SCE environment. This operation does not require any processing elements to be defined because we are simply gathering statistical data on the control and computational complexity of the JPEG encoder. The SCE environment can be configured to produce operator estimates in a tabular text-file form. A portion of the JPEG design's operator characteristics as reported by SCE is shown in Figure 7.

The listing in Figure 7 has two major sections: the first represents the total operation execution count for the entire run of the system, labeled "operation result"; the second is the "static operation result" section, which is a simple count of the operations in the function

```
                                        …

********************
      Name  Number      Op_s  Op_d  Tr_s  Tr_d  Mem_s Mem_d
      Main  1     0     7529246     0     0     3139  0

            Operation result:

                  Listed by operation types:
                  void  #1     #i      ()     []      f()    .          …
                  22680 877284 2787429 779189 488922  103310 61142

                  Listed by data types:
                  bool   char   unsigned char  short int    int        …
                  132671 146168 118440         18           7700231

            Static Operation result:

                  Listed by operation types:
                  void  #1    #i    ()    []    f()   .     ->    p++  …
                  1     337   954   236   100   184   138   3     42

                  Listed by data types:
                  bool  char  unsigned char  short int  int           …
                  81    40    24             12          1470


                                        …
```

Figure 7: Partial listing of JPEG encoder source-code characteristics

without regard to control structures. Each section also provides a breakdown of the data types associated with each operation. However, this thesis ignores any data-type-dependent effects on execution performance. We will be using the "operation types" sub-section of the "operation result" section, as this represents the total execution profile of running the benchmark.

At this point, we have to determine how to rank the operations in order of importance to our performance estimate. The assumption we will use in this paper is that operators that are executed the most are also the operators that are most important to our performance estimate. We acknowledge that this may not be a valid assumption in situations where some

operators are much more expensive than others, but an experiment to determine a better ranking system is outside the scope of this thesis.

After ordering the operators by frequency of execution, we applied the minimum characterization hypothesis by removing the operations that were executed the least number of times, effectively trading off estimation accuracy for a smaller set of operations. Here we chose to remove all but the top five most frequently executed operators, which are shown in Table 2.

Operation	Frequency
#i	1325344
#1	418343
()	382321
[]	244076
=	230355

Table 2: Highest frequency operations in the JPEG Encoder

We then wanted to view our system estimates using the minimal estimation and compare the minimally characterized estimates with the estimate we would obtain from using a completely characterized model. Since we have not yet shown how to derive operator weights from actual performance metrics, we arbitrarily pick weights for a fictional PE. In this PE, the actual cost of each operator is arbitrarily picked as 10.0 units. We then created five more PEs: the first PE only included weights for the top operation in Table 2, the next PE included weights for the top two operations, and so on up to a PE that included weights for all five operations. Because the weights of all other operations are set to 0.0 in these PEs, all other operators are effectively eliminated from the equations. The PE configuration is shown in Table 3.

PE Name	#i	#1	()	[]	=	All others
All Operations	10.0	10.0	10.0	10.0	10.0	10.0
Top5	10.0	10.0	10.0	10.0	10.0	0.0
Top4	10.0	10.0	10.0	10.0	0.0	0.0
Top3	10.0	10.0	10.0	0.0	0.0	0.0
Top2	10.0	10.0	0.0	0.0	0.0	0.0
Top1	10.0	0.0	0.0	0.0	0.0	0.0

Table 3: PE configuration for minimal characterization of JPEG encoder

Using these PEs, we re-estimated the system execution performance and arrived at the estimates shown in Figure 8. The graph shows that a large percentage of the system execution time is accounted for by including only the most frequent operators. However, we need to calculate the fidelity of the approach to know whether the technique is truly valid. An analysis of the fidelity of this approach is presented in the next section.

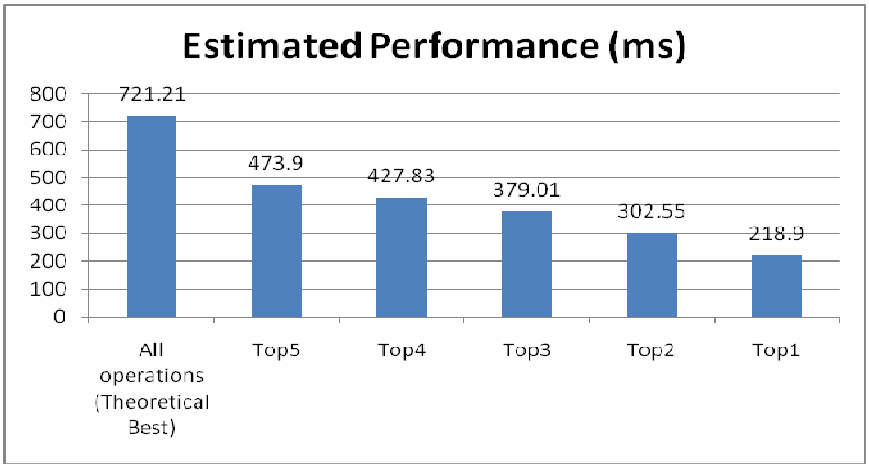

Figure 8: Estimated execution times for varying weight table configurations, grouped by test iteration

4.4 Fidelity of Minimal Characterization

In order to evaluate the fidelity of our process, design decisions made using estimates which were completely characterized must be compared to decisions made with the minimally characterized model. We considered a design choice between two processing elements, one a software PE and the other a hardware PE, which we label SW (software) and HW (hardware). This represents a common system-level design question: "Can the performance goal be met using a cheaper software solution, or is a more costly hardware solution necessary?"

To evaluate the fidelity of our approach, we duplicated an experiment which is reported in [7] by the CECS group. In this experiment, the JPEG encoder is evaluated for \

Configuration #	HandleData	Quantization	DCT	Huffman Encode
0	SW	SW	SW	SW
1	SW	SW	SW	HW
2	SW	SW	HW	SW
3	SW	SW	HW	HW
4	SW	HW	SW	SW
5	SW	HW	SW	HW
6	SW	HW	HW	SW
7	SW	HW	HW	HW
8	HW	SW	SW	SW
9	HW	SW	SW	HW
10	HW	SW	HW	SW
11	HW	SW	HW	HW
12	HW	HW	SW	SW
13	HW	HW	SW	HW
14	HW	HW	HW	SW
15	HW	HW	HW	HW

Table 4: Configurations of JPEG hardware

performance by assigning four behaviors to the SW and HW PEs in all 16 different configurations. The experimenters reported an "actual" execution performance, which was derived from cycle-accurate simulation, and an "estimated" execution performance which was calculated using a weight-table-based estimation method. The weight tables used in [7] were low-level-unaware, created using estimates of timing for each operator from product documentation, and thus we will label these estimates the "low-level-unaware model". We performed the same experiment, using a "low-level-aware model," which was trained using the results from the "actual" model. Table 4 shows each configuration, numbered to facilitate referencing.

4.4.1 Least-squares Model Generation

To generate the low-level aware model for estimation, we use a least-squares fitting technique for software estimation similar to the technique used in [10]. The mathematical model we needed to solve is shown in Eq. 1. The first matrix represents the operator characteristics for each of the 16 configurations given in Table 4. Each of the opSW and opHW values is a column-vector representing the execution profile for each configuration. The weightSW and weightHW represent the operator weight unknowns that we need to solve. The actTimes value is a row-vector with all of the measured performance times. Using the least-squares method, we then solved for the weights of all operators and inserted them into our estimation models as the low-level-aware model.

$$\begin{bmatrix} opSW_1 & opHW_1 \\ opSW_2 & opHW_2 \\ \dots & \dots \\ opSW_{15} & opHW_{15} \\ opSW_{16} & opHW_{16} \end{bmatrix} \begin{bmatrix} weightSW \\ weightHW \end{bmatrix} = [actTimes] \qquad \text{(Eq. 1)}$$

After generating the low-level-aware estimation model, we then used that model to re-estimate the execution performance of each of the 16 system configurations. Somewhat surprisingly, the resulting model characterizes the data well. The low-level-unaware and -aware models are compared against the actual execution performance in Table 5. The results demonstrate that our model predicts the execution time over two orders of magnitude better than the model that has no low-level knowledge. Additionally, our model uses only five characteristics of the source code to characterize the performance, whereas the other model had a sparsely filled estimation table that had over 500 operational costs defined. The estimates are shown on the graph in Figure 9.

	Max Error (ns)	Mean Error (ns)
CECS model	10910000	3717500
Low-level data with minimal characterization	446250	$1.6 * 10^{-8}$
Low-level data with subset of configurations used to train model	66000	4875

Table 5: Models using low-level information compared to traditional model

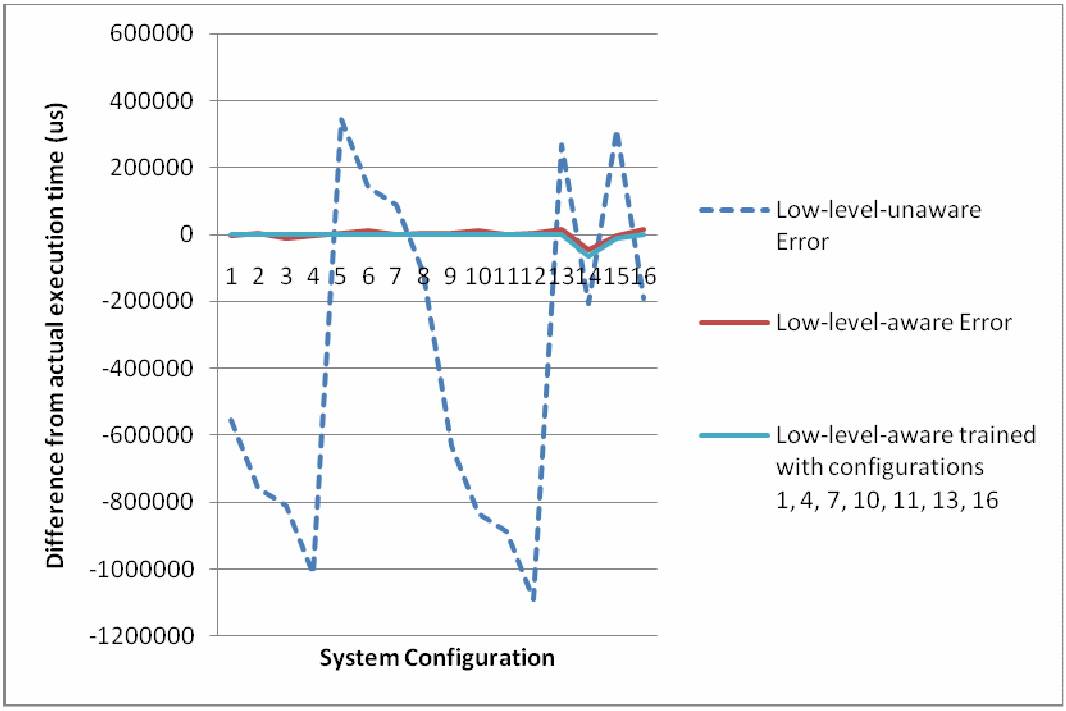

Figure 9: Error in estimates for low-level-unaware and –aware models

Having shown that a good fit is possible using only a small number of operators to characterize the system, we then wanted to see how the model would predict the performance if it was trained using only a subset of the total configurations. This is because it can be argued that estimating the same behaviors that the model was generated from can offer deceiving fits. After experimentation, we found if we choose the subset of configurations carefully so as to include each behavior on each PE at least once, we were indeed able to generate a model from a subset of the configurations and use it to accurately predict all configurations. We obtained a good fit using configurations 1, 4, 7, 10, 11, 13, and 16 for training the model. Using this model, the execution performance of the system was again estimated, and the results were nearly as good as the earlier results, proving that the model is capable of accurately predicting execution performance for configurations it has not seen

with very good accuracy. The results of this testing are shown in Figure 9 and Table 5 alongside the earlier results for comparison.

4.4.2 Fidelity Analysis

In order to define fidelity, first consider two design configurations. Using the estimated performance, the designer can choose the design configuration that performs the best. The designer can make the same choice by considering the actual performance of the system. If the design choice is the same using either the estimated performance or the actual performance, then the estimate has enough fidelity to accurately make design choices. By ranking the design configurations in order of performance, best to worst, any of the configuration rows can be selected. The configurations above that choice perform better, and the choices below perform worse. The same configurations should be above and below any chosen configuration for both the actual and estimated performance measures in order to achieve 100% fidelity. In some situations all the comparisons between configurations may not represent choices the designer needs to make, in which case the rankings may not be exactly the same, but the fidelity will still be 100%.

The fidelity of the low-level-aware model was compared to the fidelity of the estimates provided in [7] by simply ordering the configurations by estimated performance. The accuracy of the model can be sacrificed only if the reduced accuracy does not affect the relative performance of a design alternative when compared to other design alternatives. The results of this check are shown in Table 6. Using the table, we can see that the fidelity of the low-level-unaware model is correct in all cases except for configurations 15 and 8, whereas the low-level-aware model achieves 100% fidelity by matching all of the actual configuration

Actual	Low-level-unaware	Low-level-aware
16	16	16
14	14	14
15	8	15
8	15	8
13	13	13
6	6	6
7	7	7
5	5	5
12	12	12
10	10	10
11	11	11
4	4	4
9	9	9
2	2	2
3	3	3
1	1	1

Table 6: Design configurations ordered by performance from best to worst, shown to demonstrate fidelity

rows. This demonstrates how a low-level-aware model can achieve higher fidelity than a low-level-unaware model.

4.5 Summary

In this section we showed that minimal characterization is a promising tool for estimating system-level designs. In the given example, the accuracy of such a model was orders of magnitude better than a low-level-unaware model using many more operator weights. The fidelity of the approach was also shown to be better than a low-level-unaware model, showing that low-level-aware models can provide better design decisions in at least one case.

CHAPTER 5. PERFORMANCE DATA FOR LOW-LEVEL-AWARE ESTIMATION

In this chapter, we apply the minimal characterization to several system-level designs. We implement a set of benchmark programs in both hardware and software and gather performance metrics. We then use these performance metrics to generate a minimally-characterized weight table to predict system performance.

5.1 Creation of Benchmark to Gather Real-world Statistics

In order to evaluate estimation characteristics, a benchmark is needed as a baseline to characterize a given behavior in several target implementations. Several benchmark suites tailored to the embedded community are available such as the EEMBC [32], MiBench [33], and MediaBench [34], but none is provided as a system-level specification. We see the lack of a system-level benchmarking suite as a major deficiency in the field. The delay in producing a system-level benchmarking suite is likely due to the lack of agreement on a common implementation language, however the recent standardization of SystemC [35] may encourage the creation of such a testbench suite. We decided to adapt a small subset of the MiBench benchmarking suite based on constraints given in the following section.

5.1.1 Target Platform

In order to apply the estimation procedures to real-world targets, we used real hardware to collect some of our estimates. The platform we targeted was an FPGA (field-programmable gate array) development board from Digilent called the XUPV2P. This type of FPGA development board is ideal for system-level design experiments since it

incorporates a Xilinx Virtex II Pro FPGA. The Virtex II Pro includes two PowerPC CPU cores embedded in the FPGA fabric, allowing an experimenter to implement hardware and software systems in the same device. In the following experiments, we used the FPGA fabric as our hardware target and the PowerPC CPU core as our software target.

5.1.2 Constraints for System-level Benchmarking

The MiBench suite contains a large number of benchmarks for a variety of applications. Only a few of these benchmarks were straightforward to adapt into a system-level specifications based on the limitations of the SCE profiling tool and system-level language, discussed below. To choose the benchmarks, we created the following list of criteria:

- The software benchmark should be relatively easy to port to a hardware implementation, preferably through a CoreGen core implementing much of the logic for ease of translation.
- The benchmark should not rely on any standard C library math functions. This criterion was necessary because of a limitation of our profiler, namely that the SCE profiler does not profile behaviors that include any function calls. SCE requires that all functions to be profiled in a design be "clean." This means that functions must either be entirely function calls or entirely basic operations, a separation which keeps behaviors as either purely computational or aggregating, but not a combination of the two types [23].

- The number of benchmarks should be sufficient to solve a meaningful linear regression on a subset of the characteristics. For the requirements of this paper, meaningful is defined to be three benchmarks.

5.2 Benchmark Selection and Characterization

After applying these criteria, three benchmark functions were identified: square root, degree-radian conversion, and a DCT (discrete cosine transform) function. The square root and degree-radian conversion benchmarks are among the simplest benchmarks in the MiBench suite, and in fact the DCT implementation was not even a part of the MiBench suite but was brought in from the CECS JPEG encoder example to provide enough data the research in this section. Once selected, the benchmarks were manually translated into the system-level design language used by SCE. SCE was used to analyze the designs and calculated the dynamic operation execution statistics for each of the designs. The operator characterizations for each benchmark are shown in the following figures. In is notable that the characterizations are dominated by variable and constant accesses.

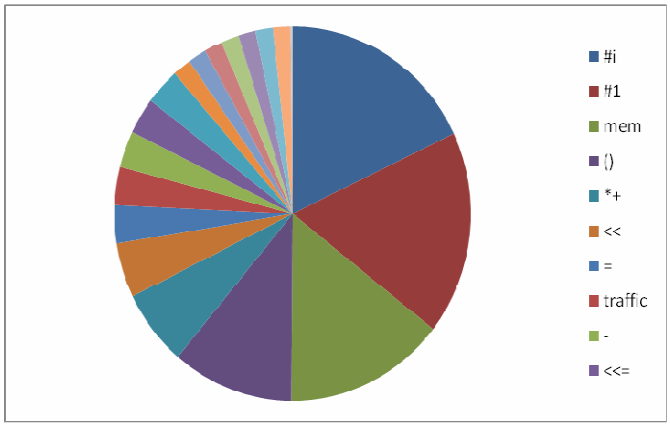

Figure 10: Square root benchmark operation frequency

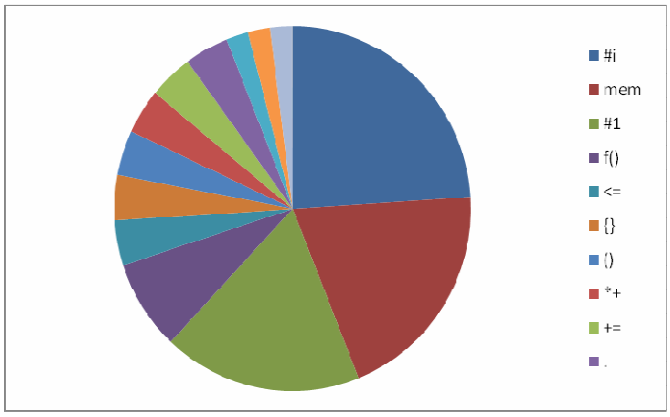

Figure 11: Radian-degree benchmark operation frequency

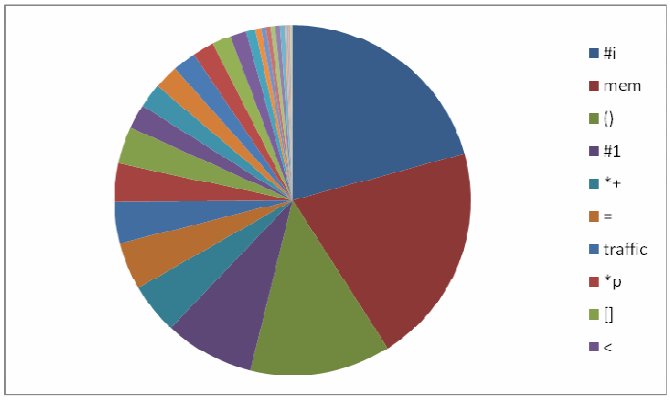

Figure 12: DCT benchmark operation frequency

5.3 Benchmark Implementations

In addition to the system-level model used for operator characterization, each benchmark was also refined to both a hardware and software implementations. For the hardware domain, optimized cores from a Xilinx-provided design library were used. In the software domain, the original C-source from MiBench or a manually translated C-source benchmark was used. The implementation performance of each of these designs was then

measured using techniques described in the next sections. Using the implementation performance along with the behavioral operational characteristics occurring most frequently in the behavior, we applied the minimal characterization methods shown in chapter 4 and were able to solve for a subset of the performance characteristics of each processing element.

5.3.1 Hardware Estimation

For the hardware target, performance was estimated by synthesizing the VHDL design. As part of the process of synthesis, the Xilinx synthesis tool estimates the maximum clock frequency at which a design will run. We use this maximum clock frequency estimate as our performance metric for hardware implementation. Then, to determine the number of clock cycles a hardware implementation of the benchmark would have to run, we counted by hand. The fact that we are not actually running the hardware implementation on the FPGA means that our hardware estimate only includes the up-front technology cost for the hardware implementation, as discussed in the CEPF sections of this thesis. The model used to gather performance characteristics in hardware is shown in Figure 13.

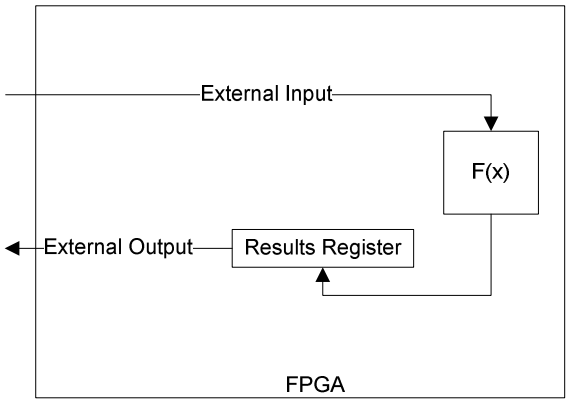

Figure 13: FPGA benchmarking setup

5.3.2 Software Estimation

For the software target, performance was measured using the cycle counter register built into the PowerPC 405. This register accurately reflects the number of cycles elapsed from the time that the processor began running. By taking the difference of the cycle count before and after the candidate function was executed, we can determine the real time taken to execute a function. Utilizing the CEPF categories presented in this thesis, this is a measurement of the up-front technology, fixed platform, and dynamic technology execution characteristics, meaning that the estimates include the effects of compiling, the clock frequency, instruction and data fetch times, and cache and other ILP (instruction-level parallelism) effects. The benchmarking setup used to test software performance is shown in Figure 14. Note the software case is more complicated than the hardware case, as the fetch patterns of the cache as well as the load of the bus directly have an impact on the speed at which the execution occurs. All software performance measurements were taken using the cycle counter register built into the PowerPC 405 architecture, ensuring that any measurement overhead was removed from the final results.

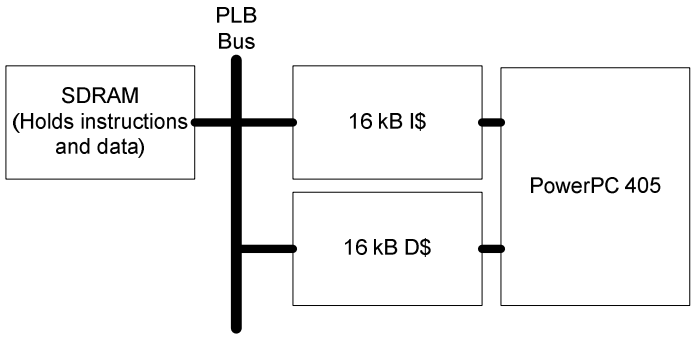

Figure 14: Software benchmarking setup

5.4 Calculation and Analysis of Operand Costs

To perform the experiment, both hardware and software implementations were measured for performance. The performance numbers were recorded both in cycles and time. Assuming that the most frequently occurring operations are most likely to affect execution performance, a subset of the operations was chosen to maximize the accuracy of the estimate. Since we had three benchmarks, all of which had different computational characteristics, all of the operators were not present in all of the benchmarks. In order to maximize the fit of our estimation model, we chose a subset of the operations which were used in all three benchmarks. The operations "#i" (variable access), "#1" (constant access), "{}" (basic block), "()" (function call), and "*+" (multiply-accumulate detection) were identified and used. The results of the minimal characterization are shown in Figure 15 and Figure 16. The residuals for the models are less than $1.0*10^{-9}$, indicating a good fit for the model.

One notable feature of these figures is that the cost of several of the operators is negative, which seems to indicate that a behavior will actually take less time to execute if it has more of a particular operator in it. However, it is important to remember that we are not solving for the operator costs here; these simply represent coefficients of an equation to fit a line to the data.

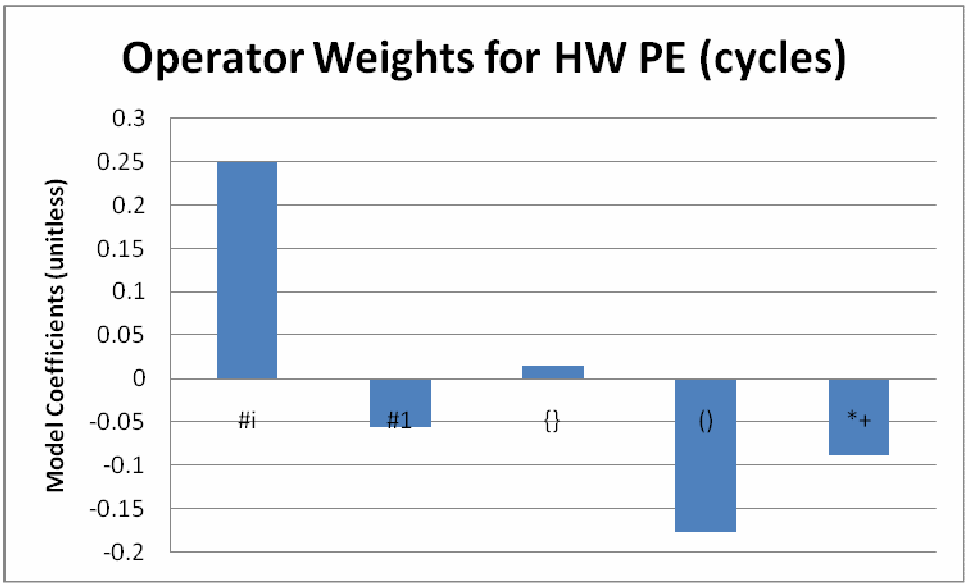

Figure 15: Estimated operator cost for hardware implementation

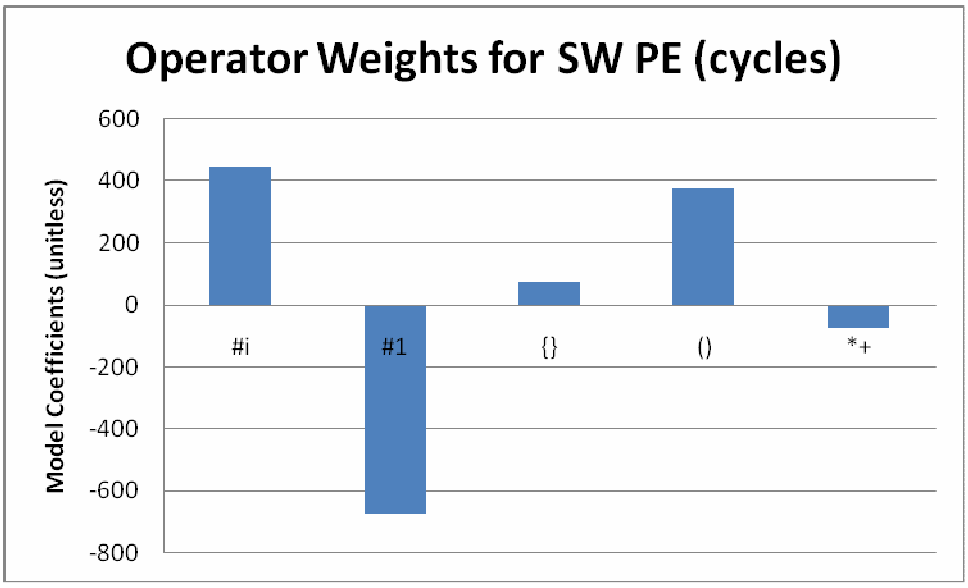

Figure 16: Estimated operator cost for software implementation

These models would then be used to estimate system performance in cycles, as given in Eq. 2 and Eq. 3 for hardware PE and software PE, respectively. Using the coefficients and

combining the computational complexity, you can arrive at estimates for the total system performance, S.

$0.249842*N_{\#i} + -0.05486*N_{\#1} + 0.015796*N_{\{\}} + -0.17576*N_{()} + -0.08744*N_{*+} = S$ Eq. (2)

$449.1195*N_{\#i} + -673.382*N_{\#1} + 75.40405*N_{\{\}} + 376.2772*N_{()} + -72.288*N_{*+} = S$ Eq. (3)

5.5 Summary

The models generated in this chapter are derived from real performance information from hardware and software targets. Thus, the models will correctly account for secondary execution performance factors as described in section 3, making the models more accurate. These models are also able to generate estimates more quickly, because they use a small number of operators.

Although we had wanted to be able to separate out the secondary execution factors in each weight table, the difficulty in creating a system-level benchmark suite prevented that work. As noted earlier, we the creation of a system-level benchmark suite as imperative to a system-level estimation tool. Generating this suite is a source of important future work, and once such a suite is created, an more detailed analysis of the secondary effects of execution performance can be conducted and reported.

CHAPTER 6. RELATED WORK

Source-level estimation is a mature topic that has been explored extensively for software targets, but less extensively for hardware targets. System-level estimation is a relatively new field, with new challenges in combining hardware and software estimation.

6.1 System-level Estimation

The team at Center for Embedded Computer Systems (CECS) at the University of California, Irvine has created a tool for system-level estimation, reported on in detail in a Ph.D. thesis [24] and more generally in work on retargetable estimation [7] and [8]. This work encompasses a broad scope of estimation, including the estimation of power, communication time between processing elements, synchronization between elements, and other performance metrics. This thesis uses the framework presented in their work as a basis for system-level estimation. We focus specifically on improving the execution performance estimation in this thesis, however the ideas should also be applicable to other performance metrics. The CECS estimation model does not include a discussion on high-level estimation of secondary execution effects, which is presented in this paper. At least one other paper [13] has mentioned this limitation, since it limits the ability to account for memory performance in a software PE and data fetches cannot be timed.

Several researchers have identified the compiler and translation from high-level languages to a low-level language as a source of noise when trying to create good estimation models. One approach to this problem is to first translate the high-level source code into an intermediate representation, with all compiler-level optimizations such as loop unrolling and elimination of unused variables already applied. The authors of [9] take this approach, using

the LANCE [18] compiler to turn the code into a form they term “three address code”. Each computation is separated into its own line so a one-to-one mapping between source code and assembly is possible. The researchers group operators into five categories: arithmetic, logical/bitwise, mult/div/mod, comparison, load/store. T. Kempf et al. [13] build on the work in [9] using an the instrumented version of the code that is compiled to a virtual processor, adding the ability to explicitly model memory timings between processing elements. The reported results using this method are quite accurate, but the simulation time is only a single order of magnitude less than cycle-accurate simulation, which makes the solution infeasible for early design space exploration.

Another similar approach to [13] and [9] is to create a virtual instruction set and compile the system-level design into the instruction set. In [11], estimation is realized by compiling C source code to the development machine's assembly code (to see high-level compiler optimizations), then translating the assembly back into low-level C with timing annotations. The authors also consider target compiler optimizations and target hardware optimizations. This thesis does not use this technique, but instead categorizes compiler and assembler optimizations as one of many factors affecting execution performance. Although this thesis was unable to explore individual execution factors, it assumes that these secondary factors can be accounted for in the weight-table based estimation approaches. A hybrid approach can also be imagined that would provide the benefits of both approaches. [22] also attempts to account for tool-specific optimizations but stops short of a general categorization.

The authors of [5] use MUSIC and GEODESIM to generate RT-level models of their test systems and to co-verify their results, respectively. Their example application, a motor controller, is specified in a system-level design language called SA-C. The design is

partitioned into two different processors and a hardware target with claimed accuracy around 2.5%. They describe the notion of profiling hardware targets at RT-level and back-annotation as the first two steps in a 4-step codesign sequence. The estimation uses both static and dynamic characteristics of the application behavior, and requires the creation of a database with profiling data from all possible design alternatives, which is similar to the approach take in this thesis.

As the field of system-level estimation grows, many tools are being created and abandoned. [6] provides a good overview of available toolsets, organized in two ways: by the portions of the Y-chart methodology they address, and by their level of abstraction. Many toolsets are available for system-level modeling.

Tools in [12] and [14] use estimation as a step in automated partitioning schemes. The partitioning decision in [12] is based on estimates of latency in both hardware targets using a directed acyclic graph of control flow, and software targets using a simple sum of the instruction timings. Presumably the software timings come from a processor manual. The algorithm uses a fixed platform for the target hardware, consisting of a FPGA and a CPU connected through a bus. In [14] the presented approach relies on choosing a target architecture and compiling or synthesizing behaviors for these architectures to derive performance estimates.

In this thesis we use the SpecC [36] system-level design language, however SystemC [35] is another system-level design language that has been gaining traction in recent years, especially with its recent standardization. The language is relatively new, however tool support is coming from some vendors and presumably system-level estimation will follow. [21] presents a SystemC-based system-level performance estimation method, using execution

dependencies to determine high-level synchronization points in the design. They define two types of execution synchronization, namely R/E (execution waiting for the reception of data) and E/S (sending of data waiting for execution). Using these concepts they are able to demonstrate a high-level performance estimate for data-dominated systems across multiple candidate architectures. We expect to see more System-C tools for estimation in the near future from vendors which should accelerate the rate at which system-level estimation research can proceed.

6.2 Hardware Estimation

Estimation work in the hardware domain is usually focused on power estimation and FPGA resource utilization estimation. [1] can predict performance for FPGA designs using floor plan, wire-delay, and clock path estimation, but requires synthesis to be performed and a RT-level description of the system available prior to estimation, which is costly during design space exploration. Presenting a partial solution, [3] predicts CLB usage for FPGA designs using a partially synthesized design, where the estimator attempts to predict the scheduling and binding of variables to speed estimation. However, this work is relatively limited to controller-type applications with binary-coded states.

Source-level performance estimation of hardware generally revolves around the generation of a data-flow or control-flow graph and estimating the number of cycles necessary to perform a given amount of work. [15] presents a high-level clock-period estimator for hardware targets by modeling the hardware ports as resources that must be scheduled. [16] demonstrates hardware estimation using generated CDFGs (control/data flow graphs) to expose parallelism, but a fairly restrictive fixed hardware interface. They use

integer linear programming (ILP), a technique borrowed from the popular software worst case execution time (WCET) estimator Cinderella [28], to estimate the hardware performance. This thesis does not include a measure of the control-flow of a hardware system, a point which is noted in future work and which we anticipate would lead to better hardware estimates.

Matlab [27] does not meet many definitions of system-level design language, since it cannot provide separation between computation and communication, among other requirements; however some researchers use it for high-level system specifications. [20] describes a method of gathering trace data from the execution of a high-level Matlab system model and using it to estimate the performance and size of the system on a hardware target. Each operation performed on a given bit-width of variables in the model is mapped to area, latency, and service rate. Their experiments use a data flow graph (DFG) to schedule the resulting system and determine an estimate for execution time for FPGA implementation.

6.3 Software Estimation

The best paper this author found on source-level software estimation was [17]. In their paper, the researchers provide a comprehensive mathematical model for estimating software execution. They break statements written in C into pieces called atoms, each of which is given a weight. The author also provides factors to account for compiler optimizations and other tool-dependencies on a per-atom basis. This work provides a good formal basis for software performance estimation. This thesis uses an estimation framework that is a subset of the framework presented in that paper.

[10] describes multiple software performance estimation techniques, comparing and contrasting several of the promising methods. The authors have a set of 35 programs they use to benchmark their analysis techniques, using a virtual instruction set for analysis. They first consider a simple least-weighted squares solution for mapping the behavior execution into cycles, however they find that its applicability is strictly limited to applications from the same domain, suggesting that the model predicts execution time only for programs that resemble the program used to generate the model. This issue was also noted and discussed in [2]. The authors then consider stepwise multiple linear regressions, where each regression is based on an application domain. The authors divide their tasks into one of two domains, either control- or data-dominated. Using a prediction metric of the ratio of "if" instructions to total instructions, they characterize the applications into control-dominated and data-dominated domains. The authors then conclude that they can achieve higher accuracy by applying different models to each of the two domains. Through statistical techniques, they are able to refine the predicting models from 25 dimensions down to 4 dimensions in the control-dominated case, and to 1 dimension in a specific data-dominated case. This thesis uses the least-squared approach presented in that paper for our estimates.

Another limitation of the estimation framework used in this thesis is the restriction that no library or function calls may be used inside of a profiled behavior. This is because, for many libraries, the source code is not available for profiling. This issue is discussed for software targets in [4], which attempts to improve software estimation by analyzing and characterizing the performance characteristics of library functions and system calls. Their work claims that a large majority of these functions can be stochastically modeled by conducting performance tests with different input data.

CHAPTER 7. CONCLUSION AND FUTURE WORK

In this thesis we have presented an evaluation of a weight-table based approach to estimation for system-level design. System-level design is becoming more complex in new embedded designs, and thus, we need to have good estimation frameworks to guide designers. While software and hardware estimation have been explored, system-level estimation has been slower to mature.

In this thesis, we argue that low-level information at high-level abstractions of the system design can provide better estimates than low-level-unaware estimates. We presented a minimal characterization method that makes accurate characterization feasible in a system-level design environment. The feasibility of this approach is two-fold: first, requiring a smaller number of metrics to be collected allows designers to spend less time measuring running simulations or measuring device performance; secondly, the time spent in the estimation algorithm is reduced by fewer metrics, making system-level design exploration faster. For the example presented in this thesis, we were able to clearly show that the accuracy and fidelity of low-level-aware estimates at high levels of abstraction were much more accurate with an increase in fidelity.

Finally, we presented a categorical discussion on secondary execution factors that have not been formalized. In previous work, these effects have been accounted for in extra coefficients and weights in estimation equations, but a top-down discussion of the effects was missing.

7.1 Future Work

As noted elsewhere, this thesis clearly showed that some areas of the field need improvement. This thesis identified the following areas of work:

- A system-level benchmark suite is necessary for future work comparing the system-level estimates to low-level performance values
- This thesis was neither able to explore the control or data flow graphs of hardware targets for estimation, nor was it able to group computation by the type of operator. Both of these factors could greatly increase the accuracy of system-level estimates.
- The measurements of execution performance made in this thesis certainly have some measurement noise. The effect of this measurement noise on system-level estimates is unknown and could be explored and quantified.
- With a larger set of benchmarks than was presented here, enough data should be present to separate the weight-tables into each of the categories of execution factors presented in this paper. With this data, we could then build a framework to characterize and estimate using all secondary execution factors.

BIBLIOGRAPHY

[1] M.Xu and F. J.Kurdahi, “ChipEst-FPGA: A tool for chip level area and timing estimation of lookup-table-based FPGA's for high level applications,” *Asia–Pacific Design Automation Conf.*, pp. 435-440, Jan. 1997.

[2] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," *Parallel Architectures and Compilation Techniques,* 2001, pp. 3-14.

[3] C. Menn, O. Bringmann, and W. Rosenstiel, "Controller estimation for FPGA target architectures during high-level synthesis," *15th International Symposium on System Synthesis*, 2002, pp. 56- 61.

[4] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "Library functions timing characterization for source-level analysis," *Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 1132- 1133.

[5] A. Baghdadi, N.-E. Zergainoh, W.O. Cesario, and A.A. Jerraya, "Combining a performance estimation methodology with a hardware/software codesign flow supporting multiprocessor systems," *IEEE Transactions on Software Engineering,* vol.28, no.9, Sep 2002, pp. 822- 831.

[6] D. Densmore and R. Passerone, "A Platform-Based Taxonomy for ESL Design," *IEEE Design & Test of Computers*, vol.23, no.5, May 2006, pp. 359- 374.

[7] L. Cai, A. Gerstlauer, and D. Gajski, “Retargetable Profiling for Rapid, Early System-Level Design Space Exploration,” *Proceedings 41st Design Automation Conference,* 2004, pp. 281-286.

[8] L. Cai, A. Gerstlauer, and D. Gajski, "Multi-metric and multi-entity characterization of applications for early system design exploration," *Proceedings of the ASP-DAC 2005, Design Automation Conference, Asia and South Pacific*, vol.2, 2005, pp. 944- 947.

[9] K. Karuri et al., “Fine-grained Application Source Code Profiling for ASIP Design,” *Proceedings 42nd Design Automation Conference*, 13-17 June 2005, pp. 329-334.

[10] P. Giusto, G. Martin, and E. Harcourt, “Reliable estimation of execution time of embedded software,” *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, 2001, pp. 580-588.

[11] J.R. Bammi et al., “Software performance estimation strategies in a system-level design tool,” *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, 2000, pp. 82- 86.

[12] G. Busonera et al., “Automatic Application Partitioning on FPGA/CPU Systems Based on Detailed Low-Level Information,” *9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools,* 2006, pp. 265-268.

[13] T. Kempf et al., “A SW performance estimation framework for early system-level-design using fine-grained instrumentation,” *Proceedings Design, Automation and Test in Europe,* Vol.1, 6-10 March 2006, pp. 6.

[14] F. Farrandi et al., “System-level metrics for hardware/software architectural mapping” *Second IEEE International Workshop on Electronic Design, Test and Applications,* 28-30 January 2004, pp. 231-236.

[15] B.K. Dwivedi et al., "Rapid Resource-Constrained Hardware Performance Estimation," *Seventeenth IEEE International Workshop on Rapid System Prototyping*, 14-16 June 2006, pp. 40- 46.

[16] Jun-hee Yoo et al., "Worst case execution time analysis for synthesized hardware," *Asia and South Pacific Conference on Design Automation*, 24-27 January 2006, pp. 6.

[17] C. Brandolese et al., "Source-level execution time estimation of C programs," *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, 2001, pp. 98-103.

[18] R. Leupers, "LANCE: A C Compiler Platform for Embedded Processors,", 2001, available from: http://servus.ert.rwth-aachen.de/lancecompiler/files/es2001.pdf, accessed on 25 March 2007.

[19] S. Abdi et al., "System-on-Chip Environment," tech. report CECS-03-41, Center for Embedded Computer Systems, Univ. of California, Irvine, 2003.

[20] P. Bjureus, M. Millberg, and A. Jantsch, "FPGA Resource and Timing Estimation from Matlab Execution Traces," *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, 2002, pp. 31-36.

[21] K. Ueda et al., "Architecture-level performance estimation for IP-based embedded systems," *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Vol. 2, 16-20 February 2004, pp. 1002-1007.

[22] A. Allara et al., "System-level performance estimation strategy for sw and hw," *Proceedings International Conference on Computer Design: VLSI in Computers and Processors*, 1998, 5-7 October 1998, pp. 48-53.

[23] A. Gerstlauer, “System-On-Chip Specification Style Guide”, tech. report CECS-03-21, Center for Embedded Computer Systems, Univ. of California, Irvine, 2003.

[24] L. Cai, “Estimation and Exploration Automation of System Level Design”, Ph.D. dissertation, Department of Information and Computer Science, University of California, Irvine, 2004.

[25] GNU gprof, 20 March 2007; http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html

[26] H. Yin, H. Du, T.-C. Lee, and D. Gajski, "Design of a JPEG Encoder using SpecC Methodology," tech. report ICS-TR-00-23, Center for Embedded Computer Systems, Univ. of California, Irvine, July 2000.

[27] Matlab, 20 March 2007; http://www.mathworks.com

[28] Cinderella, 20 March 2007; http://www.princeton.edu/~yauli/cinderella-3.0/

[29] S. Gupta, “SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations,” *Proceedings 16th International Conference on VLSI Design*, 4-8 January 2003, pp. 461-466.

[30] Celoxica Ltd., “Handel-C For Hardware Design,” White Paper, August 2002, available from: http://www.celoxica.com/techlib/files/CEL-W0307171L48-63.pdf, accessed on 20 March 2007.

[31] Catapult-C, 20 March 2007; http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/

[32] The Embedded Microprocess Benchmark Consortium (EEMBC), 20 March 2007; http://www.eembc.org

[33] M.R. Guthaus et al., "MiBench: A free, commercially representative embedded benchmark suite," *IEEE 4th Annual Workshop on Workload Characterization*, December 2001, pp. 3-14.

[34] L. Chunho, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," *Proceedings Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, 1997, pp. 330-335.

[35] IEEE Computer Society, "IEEE Std 1666-2005: IEEE Standard SystemC Language Reference Manual," *Design Automation Standards Committee*, 31 March 2006, available at http://standards.ieee.org/getieee/1666/download/1666-2005.pdf, accessed on 20 March 2007.

[36] R. Domer, A. Gerstlauer, and D. Gajski, "SpecC Language Reference Manual," Center for Embedded Computer Systems, Univ. of California, Irvine, 12 December 2002, available at http://www.cecs.uci.edu/~specc/reference/, accessed on 20 March 2007.