ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Methods for Rapid Selection of Processors for Constraint-Aware Embedded Systems

Abhijit Ray

School of Computer Engineering

A thesis submitted to the Nanyang Technological University in fulfillment of the requirement for the degree of Doctor of Philosophy

2008

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Abstract

Embedded systems design is moving from ASIC based implementations to using commercial off the shelf processors. ASICs tend to have long design cycles and they do not offer much flexibility for later design changes. With the rapid advances in embedded systems and microprocessor technology, the traditional methods of selecting a processor for an embedded system may not be effective. The techniques have to keep up with the advanced novel architectures that are constantly being developed. Processor selection is an important part of embedded system design. The current methods of performance evaluation which are mostly based on instruction set simulators are quite accurate but they suffer from the drawback of consuming large amounts of time. Moreover, as processor technology improves, the construction of such instruction set simulators becomes more complex. Hence, with advances in technology, the evaluation process becomes slower. Therefore to speedup the process, a higher level view of the underlying architecture is desirable. Estimation, instead of a detailed performance evaluation, can help in narrowing down the choice to a smaller set of processors. Fast processor performance estimation can easily lead to large saving in design time. Estimation at a higher level does incur the cost of decrease in accuracy. But this disadvantage is easily compensated for by the increase in estimation speed, thereby allowing us to take into consideration more processors.

Current processor performance estimation methodologies generally lag behind

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

the current state of the art in hardware technology. This research was mainly motivated by the need to come up with a technique which can estimate performance at a higher level and hence can easily be modified to cater to the new innovations. Treating the processor architecture at a higher level allows us to ignore details that does not affect the performance, while capturing the important ones. A proposal for using performance estimation instead of a detailed performance evaluation to help in the selection of processors for an application is suggested in this work.

This dissertation proposes an estimation framework which is based on processor performance estimation on a higher level. It does not require the application to be compiled for every processor under consideration. The estimation methodology is based on the intermediate representation of the application. The intermediate level instructions are used to approximate the compiled code. In this process intermediate level instruction which do not have much influence on the performance are filtered out. Estimation of library function was done by characterizing the performance of the functions based on the input. The IPC (instructions per cycle) was estimated by bounding the IPC values depending on the average basic block length.

Performance estimation results on the MiBench benchmark suite show that estimation at higher level can give very fast output though with a loss in the accuracy, which depending on the requirements may be acceptable. Performance for the MiBench benchmark suite was estimated for the ARM, PISA and the PowerPC architectures.

Processor selection strategy based on the performance estimation techniques is proposed. It is proposed that the processor with the least shortfall in performance as compared with the required performance, be selected. This ensures that an over-performing (and hence expensive) processor is not selected and that the size

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

of the hardware area required to bring the performance to the required level is the least.

A fast hardware/software partitioning algorithm is also presented. Efficient hardware/software partitioning is crucial towards realizing optimal solutions for constraint driven embedded systems. The size of the total solution space is typically quite large for this problem. A Knapsack model based partitioning algorithm is provided. In particular, the technique consists of a method to split the problem into standard 0-1 knapsack problems in order to leverage on the classical approaches. The proposed method relies on the tight lower and upper bounds for each of these knapsack problems for the rapid elimination of the sub-problems, which are guaranteed not to give optimal results. Experimental results show that, for a wide range of problem sizes, the solution can be found found by solving a very small number of sub-problems.

A design framework for a constraints-aware embedded system was also proposed. This involves using the processor selection framework (which in turn uses the performance estimation techniques developed) to identify a suitable processor. Processor selection is followed by hardware-software partitioning in an iterative manner, increasing or decreasing the area depending on whether the performance constraints are satisfied or not.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Acknowledgments

First of all, I would like to thank my thesis supervisor Dr. Thambipillai Srikanthan for his guidance, support and encouragement. I am very grateful to him for his patience during this research. It was indeed a privilege to have the opportunity of working under him.

I would also like to thank Dr. Wu Jigang for all the suggestions and help in writing the papers. I am also grateful to him for proof reading this thesis. Thanks are also due to Mr. Lam Siew Kei for his insightful comments during the many discussions we had. It was a rewarding experience being in the company of both Dr. Wu and Siew Kei.

Thanks also to Ms. Merilyn Yap, Ms. Nah, Mr. Rande Heng and Mr. Chua Ngee Tat for all the administrative help and the computing support provided.

I am also grateful to Mr. Ashish Panda and Mr. Santanu Dash for agreeing to proof read this thesis and also for all the meaningful and the not so meaningful discussions we had. I also thank Dr. Kugan Vivekanandarajah for all the help and discussion which were very invaluable for this research work. Thanks also to Mr. Rohit Nagarajan and Ms. Chandni Patel for being a great source of help, support and much fun.

I would also like to thank Mr. Shamim Akhtar for initially advising me to pursue graduate studies here at Nanyang Technological University. I would also like to thank Mr. Kolli Naveen for just being a friend and also for all the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

selfless help and support. I would be ever grateful to Mr. Mohan Davasahayam Sundaram and his wife for providing me with a place to stay during this period and their wonderful daughter Ms. Zita who managed to graduate from crawling to walking faster than I could complete this thesis.

Finally, I would like to thank my family. My parents for their continuous support of my decisions throughout my life. My sister and brother for being the wonderful persons that they are. This thesis is dedicated to all of them.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Contents

Abstract . . . i

Acknowledgements . . . iv

1 Introduction **8**

1.1 Embedded Systems Constraints . . . 9

1.2 Motivation . . . 10

1.3 Research Objectives . . . 13

1.4 Major Contributions . . . 15

1.5 Organisation of the Dissertation . . . 15

2 Related Work **19**

2.1 Introduction . . . 19

2.2 Classification of Estimation Techniques . . . 20

2.3 Static Analysis . . . 23

2.4 Architecture Description Languages . . . 27

2.5 Instruction Set Simulators . . . 32

2.6 Tracing . . . 37

2.7 Profiling . . . 39

2.8 Worst-Case Execution Time . . . 41

2.9 Limitations of Existing Work . . . 47

2.10 Summary . . . 49

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3 Intermediate Representation For Performance Estimation 50

3.1 Introduction 50

3.2 Background 51

3.2.1 Performance estimation and performance evaluation 51

3.2.2 Abstraction level for estimation 52

3.3 Estimation Technique 54

3.3.1 Instruction Classification and Estimation 57

3.3.2 Estimation of Library Functions 62

3.4 Comparison With Current Techniques 64

3.5 Experimental Results 66

3.6 Summary 69

4 Performance Estimation of Library Functions 70

4.1 Introduction 70

4.2 Library Functions 71

4.3 Methodology 72

4.3.1 Classification of library functions 73

4.4 Experimental Work 79

4.5 MiBench Estimation Results 84

4.5.1 Tools and Environment Used 84

4.5.2 LLVM 85

4.5.3 Experimental Procedure 86

4.5.4 Estimation Results 86

4.6 Summary 90

5 Estimation of IPC (Instructions Per Cycle) 93

5.1 Introduction 93

5.2 Estimation Methodology 94

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

5.3 Estimation Results 97
5.4 Summary 106

6 Framework for Processor Selection and HW-SW Partitioning 107
6.1 Introduction 107
6.2 Processor Selection 108
6.3 HW-SW Partitioning Framework 112
6.3.1 Hardware Software Partitioning 114
6.3.2 Model of the physical problem 115
6.3.3 Problem Splitting 116
6.3.4 Algorithm Description 122
6.3.5 Experimental Works 124
6.4 Design Framework 127
6.5 Summary 129

7 Conclusion 130
7.1 Dissertation Summary 130
7.2 Further Enhancements 133

A Estimation Results for Library Functions 136

B MiBench Benchmark Suite 146

C The LLVM Compiler Infrastructure 149

D SimpleScalar Tool-set 151

E Publications 153

References 154

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

List of Figures

1.1 Moore's Law 11
1.2 Time to Market 12
1.3 Global Embedded Systems Market, 2003 – 2009 [9] 13

2.1 ADL-driven design automation of embedded processors [24] 28
2.2 Taxonomy of ADLs [24] 30
2.3 Taxonomy of Simulation Tools [42] 33
2.4 Interpreted Simulation 34
2.5 Compiled Simulation 35
2.6 Relation between possible executions and flow information [58] . . 43

3.1 Relative runtime of simulators at different levels of detail 53
3.2 Coding effort of simulators at different levels of detail 54
3.3 Basic Flow 55
3.4 Test program for estimating switch instruction 59
3.5 LLVM intermediate code for the switch statement 59
3.6 ARM assembly code for the switch statement 60
3.7 Estimation for LLVM instructions 63

4.1 Library function estimation overview 78
4.2 Estimates for *sqrt*() 81

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.3 Number of instructions vs. number of characters printed for *printf*() 83
4.4 Number of instructions vs. number of items sorted 84
4.5 Distribution of Errors for ARM 90
4.6 Distribution of Errors for PISA 91
4.7 Distribution of Errors for PPC750 91

5.1 IPC vs cache size for ARM 97
5.2 IPC vs cache size for PISA 98
5.3 Cache miss rate vs cache size for ARM 99
5.4 Cache miss rate vs cache size for PISA 100
5.5 Upper and lower limits of IPC for ARM 101
5.6 Upper and lower limits of IPC for PISA 101
5.7 IPC Estimates for ARM 102
5.8 IPC Estimates for PISA 102

6.1 Processor Selection 109
6.2 Processor selection flow 113
6.3 Running time vs. Problem size 126
6.4 Design Flow 128

A.1 Simulation results for *fscanf*() 142
A.2 Simulation results for *fprintf*() 144
A.3 Simulation results for *printf*() 145

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

List of Tables

2.1 Application Parameters [20] 26

3.1 LLVM instructions and their ARM equivalent 58
3.2 Estimates for ARM (without stdlib) 66
3.3 Estimates for PISA(without stdlib) 68
3.4 Estimates for PPC750(without stdlib) 69

4.1 Function Classification 77
4.2 putchar results 79
4.3 putchar estimates 80
4.4 *printf*() estimates 82
4.5 *qsort*() estimates 83
4.6 Estimates for ARM 86
4.7 Estimates for PISA 88
4.8 Estimates for PPC750 89

5.1 IPC Estimates for ARM 103
5.2 IPC Estimates for PISA 104

6.1 Running time (ms) of the algorithm for different problem sizes . . 125
6.2 Number of subproblems solved for different problem size and area constraints. 125

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

A.1 Simulation results for *cos*() 137
A.2 Simulation results for *sin*() 138
A.3 Simulation results for *exp*() 139
A.4 Simulation results for *pow*() 140
A.5 Simulation results for *sqrt*() 140
A.6 Simulation results for *rand*() 141
A.7 Simulation results for *fscanf*() 142
A.8 Simulation results for *fprintf*() 143
A.9 Simulation results for *printf*() 144

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 1

Introduction

Embedded system designers have to deliver under very strict constraints like low cost, short time to market, low power etc. On the other hand, the number of options available to the designers has increased manifold. Additionally, there is a demand for more functionality on embedded devices. It is common for cell-phones nowadays to have features like a camera, radio, game console in addition to its primary function as a phone. Due to the availability of a large number of implementation, the job of the embedded systems designer has become tougher. The designer now has to evaluate performance of the many choices that are available. Increasingly a greater part of the embedded applications is being implemented in software (i.e. programs running on microprocessors) for reasons mentioned in section 1.2. Selection of processors from those available in the market is the additional problem faced by the designer.

This chapter briefly discusses embedded system and their special requirements which differentiate them from general purpose computing systems. The motivation for this research work is discussed next, followed by the research objectives and the contributions.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

1.1 Embedded Systems Constraints

Embedded systems are computing platforms embedded inside other devices and are not perceived as general purpose computers. These days they are to be found everywhere, in mobile phones, consumer electronics, household appliances and automobiles. The embedded applications generally run partly on software and partly on hardware. The software part is usually a microprocessor running a program while the hardware part can be ASICs or FPGAs.

Due to its dissemination among various types of application domain, embedded systems are more affected by market constraints than general-purpose units [1]. These special requirements of embedded systems have been specified by Koopman in [2]. These include

- Real time/reactive operation: In critical embedded devices the system needs to respond within a short time. Failure to do so means the system has failed.
- Small size, low weight : Many embedded systems are contained within other larger systems hence there are additional constraints on its size. Also embedded devices which a person carries with him/her have size restrictions on them.
- Safe and reliable : In mission critical applications like flight control systems, the reliability of the system is essential as its failure can lead to severe damage.
- Low cost : Embedded systems have to be really cost effective. Decisions increasing costs by even a few cents attract management attention [3].
- Low power : Many embedded systems do not take power from the mains, instead it takes power from alternative sources like a battery. Hence a low power design is very important.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

- Short time to market : Profits in the embedded system market is very much dependent on the time it takes for product launch. This can be seen from the diminished profit for delayed entry in the graph 1.2.

1.2 Motivation

The software content of embedded systems grows exponentially, mostly from the migration of application-specific logic to application-specific code, to cut down product cost and time to market [4]. The demand for more functionality along with the need for shorter development cycle has also led to larger portions of the application to be moved from hardware to software. All these have resulted in the increasing importance of microprocessors in embedded system design. While earlier the desired high performance could only be achieved by using ASICs, now the same performance can be obtained using microprocessors. The long design cycle and high cost of deep submicron designs make the ASIC driven approach problematic [5]. And as microprocessors are more flexible than ASICs, this has also resulted in the moving of significant part of the application to software, i.e., programs running on microprocessors.

With the rapid advances in microprocessor technology (following the famous *Moore's Law* [6], see Fig 1.1) the number of processors available in the market has multiplied many times and this number can only grow in future. Today, the number of microprocessors shipped for embedded applications far exceeds the number shipped for personal computers [7]. In 2003 there were more than 100 32-bit embedded processors on sale. The numbers for 16-bit and 8-bit processors are even higher and is likely to grow [8]. Thus the embedded system designers are faced with the increasingly difficult problem of having to decide on a suitable processor for an application.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

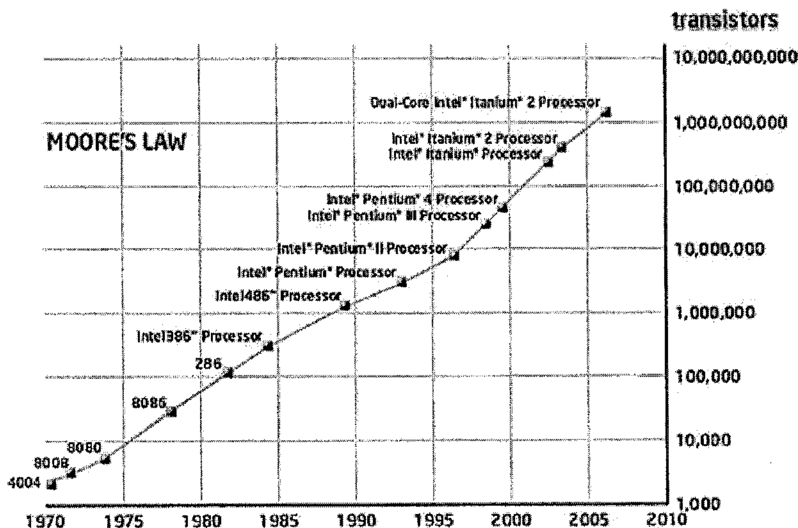

Figure 1.1: Moore's Law

The current methodology of using simulators for performance estimation is time consuming and this will get worse as more and more microprocessors become available in future. The use of simulators also means that the necessary compilers tools have to be developed. Hence the cost of compiler tools for each architecture is added to the development cost. The simulators also have to be modified to incorporate the advances in microprocessor technology. Any novel architecture feature has to be incorporated in simulators. Hence there is an additional cost of simulator development.

Time to market considerations are also very important for embedded systems. This can be seen from the revenue vs time to market graph (see Fig 1.2). There is a revenue loss due to delayed market entry. Hence it may not be viable to use simulator based approaches on the wide range of processors available in the market.

While the required performance can always be achieved using a high perfor-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

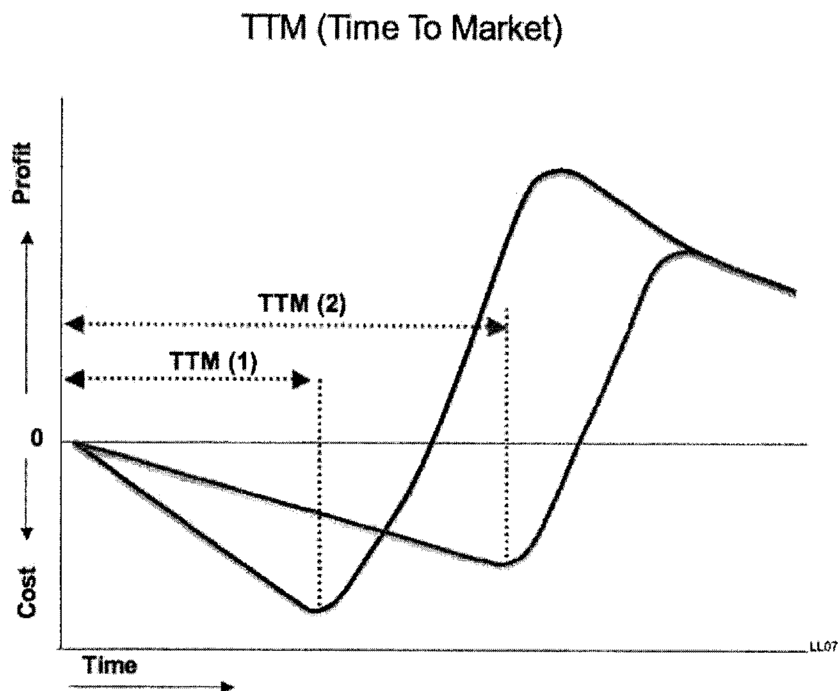

Figure 1.2: Time to Market

mance processor, this is not always desirable as high performance processors are more expensive. Thus leading to a higher unit cost of the product. And as mentioned in section 1.1, the cost effectiveness is particularly important for embedded systems. Hence an embedded system designer would want to achieve the desired performance using the cheapest possible processor.

The embedded system market is growing exponentially. Fig 1.3 shows that the global embedded systems market. The embedded system market will only grow faster as computing spreads to all parts of our lives.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

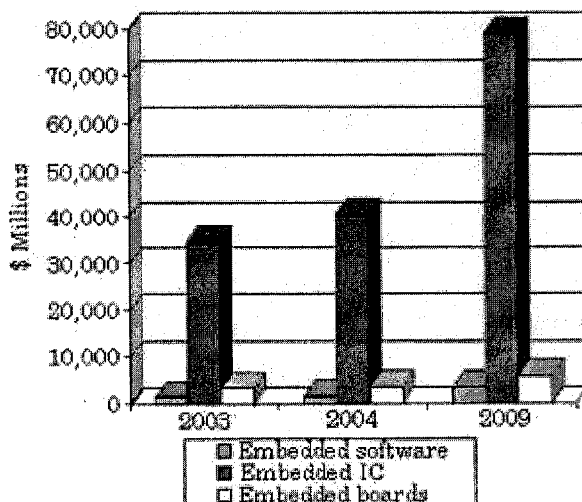

Figure 1.3: Global Embedded Systems Market, 2003 – 2009 [9]

1.3 Research Objectives

Estimation at a level of abstraction higher than what is used in the traditional methods, can potentially save design time by short-listing a smaller set of processors from the large number of processors available in the market. Detailed performance evaluation using a combination of compilation and simulation on an instruction set simulator, may not be a viable option given the wide range of processors available in the market. The challenge is to provide a framework to help the embedded system designer in selecting or short listing a few processors, on which the designer can concentrate further, rather than spend his/her time in the detailed evaluation using compiler and simulators. Given an application in a high level language like $C/C++$ and a representative input data, the framework attempts to provide a quick way to short list a smaller range of processors for further detailed analysis.

The objective of this research is to propose efficient methods to provide perfor-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

mance estimates of processors for a given application without actually compiling the application for that processor and executing it. Taking advantage of the optimizations performed on the intermediate format allows us to estimate the actual compiled code quite well. The aim is to provide a fast method to estimate the performance of a large number of processors without resorting to cycle accurate performance figures. Fast estimation techniques can help in short-listing appropriate processor which can then be subjected to detailed evaluations.

The intermediate level code of an application was used to estimate the number of target instructions in the final compiled code. For this the intermediate level instructions were classified into three categories, based on whether the instruction has a direct equivalent in the target instruction set and also a category of instructions which are not much relevant for performance estimation as they are usually executed only a few times at most. The instructions which do not have direct equivalent in the target instruction set are estimated by breaking down into simpler instructions from the target instruction set. Thus an estimate of the compiled code is obtained. This combined with the output from a target independent profiler was used to arrive at the performance estimates for the application. A technique to estimate the performance of standard library functions is also proposed. The technique is based on obtaining the characteristics of performance depending on the input.

A technique was also developed to estimate the IPC (instructions per cycle) of an application running on a processor. The upper and lower limits of the IPC was estimated and these limits on the IPC were used to find a bound on the IPC for any average (dynamic average) basic block length.

A simple partitioning method was also devised at part of this work. This partitioning technique is based on the splitting the hardware software partitioning problem into many smaller knapsack problems. Tight upper and lower bounds

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

on the solution of these sub-problems led to elimination of most of them. Experimental results show that in most cases the optimal solution can be arrived at by solving only a few of these sub-problems.

1.4 Major Contributions

The contributions in this dissertation can be summarized as:

1. A performance estimation methodology based on machine independent intermediate level code and profiling. This helps in avoiding expensive simulation based performance evaluation.

2. A methodology to estimate the processor performance executing standard library functions. The method does not require access to the source code.

3. A IPC (instructions executed per cycle) estimation framework, which is based on calculating the upper and lower limits of the IPC.

4. A novel hardware-software partitioning framework based on the 0-1 Knapsack problem. The partitioning framework also uses the proposed performance estimation technique.

5. A processor selection framework based on performance estimation.

6. An embedded system design framework which integrates the processor selection framework and the hardware-software partitioning framework.

1.5 Organisation of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 starts with the different applications of performance estimation. Classification of different esti-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

mation techniques is provided. A survey of the current state of the art in the field is presented next. A brief survey of different simulation techniques is also presented. This is followed by an in-depth study of other estimation techniques including tracing and profiling. Shortcomings of current performance estimation techniques when large numbers of processor architectures are available, are explained.

The proposed performance estimation methodology is described in chapter 3. A comparison of performance estimation and performance evaluation is provided. The consequences of using varying abstraction levels during estimation is discussed. The technique for arriving at the performance estimates of each intermediate level instruction is presented. A classification of instructions is described, which is used for performance estimation. A high level performance estimation scheme is provided. The estimation process ignores processor details which does not affect the performance much. For example processor details like *endianness* are ignored. The proposed scheme is compared with current performance estimation techniques. Experimental results for performance estimation without the consideration of standard library functions was provided. The results show that without incorporating estimation for standard library functions the errors in estimates can be as high as 99%. Performance estimates of standard library functions is the focus in chapter 4. The main objective is to develop a fast estimation technique which can help us short list a smaller number of processors from a wide range of processors available in the market.

In chapter 4, the performance estimation of standard library functions is discussed. For performance estimates of standard library functions, the number of instructions executed is used as the metric. Estimates of standard library functions cannot be easily obtained using techniques described in chapter 3. The problems associated with performance estimation of standard library functions

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

are explained. An alternative estimation strategy, which does not require the availability of the source code for the library functions, is described. Classification of standard library functions, depending on whether they depend on the input or not is provided. The estimation techniques for these different classes of functions are explained in detail. Experimental work done to arrive at the estimates for a few standard library functions is presented along with the estimation results for these functions. Experimental results for the benchmark applications are also provided.

Chapter 5, describes the proposed methodology for estimation of IPC (instructions executed per cycle) of an application running on a particular processor. Reasons for pipeline stalls, which lead to IPC being less than 1 is explained. Techniques for the calculation of upper and lower limits on the IPC is explained. The reason why performance estimate in terms of the number of instructions executed may not be sufficient for judging a processor's performance is explained with an example. Experimental work to minimise the cache effects is described with results. A technique to estimate the IPC of an application running on a processor is presented. The average basic block length is used to arrive at rough upper and lower limits for the IPCs.

In chapter 6 a processor selection framework based on selection of a processor which just fails to satisfy the the performance constraints, is presented along with the reasoning for it. A hardware software partitioning strategy based on the knapsack model is also described in this chapter. An efficient hardware software partitioning technique based on the Knapsack model is proposed. It was shown that by examining the upper and lower bounds of the sub problems, it is possible to rapidly eliminate the large number of sub-problems that do not contribute to optimal solutions. Experimental investigations demonstrate that a substantial reduction in the number of sub problems that require processing is possible,

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

thereby providing for an efficient means to partitioning of hardware and software. An embedded system design framework is presented, which is based on processor selection and then further tightening or relaxing the constraints depending on whether the constraints are met or not. And finally in chapter 7 the conclusions and the future work are presented.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 2

Related Work

2.1 Introduction

This chapter discusses previous work in the field of performance estimation of processors. Approaches based on ADLs (architecture description languages), instruction set simulators and worse case execution time analysis are primarily discussed. In ADL based techniques, different architecture description languages have been developed which are able to describe the various behavioural and structural aspects of the processor. These descriptions are later used to build compiler tools and simulators, which are then used to obtain the performance estimates.

The second area of related work is based on using instruction set simulators for performance estimation. Instruction set simulators are software models for the actual processor on which an application can be executed to obtain the performance estimates.

The third area of related work concerns worst case performance of a particular processor. Guaranteed worst case performance is especially important for hard real time systems where inability to perform a task within the deadline is equivalent to a system failure.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Software estimation is crucial for hardware software co-design. [10] lists the different applications of performance analysis in general (i.e., both hardware and software).

1. **Design Validation** : Performance analysis helps in ensuring that the design meets the performance requirements. This is necessary so that time and effort are not spent in designing a system which will not meet the requirements.

2. **Design Decision and System Optimization** : Performance estimation of the different components helps in mapping which of the tasks are allotted to which parts like in hardware-software partitioning. It can also assist in selection of processors. Performance estimates maybe used to optimize other system parameters such as cache and buffer sizes.

3. **Real Time Schedulers** : Performance bounds are required for scheduler algorithms to provide the best possible schedule.

4. **Compiler Optimization** : Performance bounds can help guide the compiler to optimize the software performance.

2.2 Classification of Estimation Techniques

The software estimation techniques can be classified into two major categories [11] : Static Estimation and Dynamic Profiling. There is another classification based on the level of abstraction at which the estimation is done [12]. Based on this there can be the following two main categories: Source level and Object level. A brief description of each of the above categories is provided below.

1. **Static Estimation**

   Static estimation does not require any execution of the application. This

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

obviously makes it impossible to get accurate estimates for most of the applications which contain loops that execute a variable number of times, or applications containing recursive functions. As explained in [13, 14] the problems are due to :

(a) Loops : The estimation of execution times is difficult for loops due to the fact that the number of times a loop gets executed cannot be inferred just from the loop condition most of the time. Moreover the demand of stack space cannot be determined before runtime.

(b) Recursive functions : In applications having recursive function calls the depth of recursion cannot be ascertained beforehand. A related problem of using recursion in real-time applications is that the demand of stack space cannot be determined before run-time

(c) Parameters and pointers to functions can reference functions of distinct timing properties. Because of the dependence of the maximum execution time on the different functions it does not make sense to assign the maximum time it takes to evaluate a function to the maximum time for a function referenced. Moreover pointers to functions provide a means for the implementation of recursion.

(d) Micro-architecture features: It is difficult to model complex micro-architectures, like cache, branch predictors, superscalar architectures [14].

So these restrictions are placed on static estimation techniques. To get around the limitations of not having unbounded loops, user annotation are necessary to set upper bounds for loops and dynamic structures [11]. There are other difficulties in static analysis. Even when loops etc. are bounded, program path analysis is difficult as the number of possible paths can be

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

quite large, especially in the presence of nested loops and branches in loops [14].

2. **Dynamic Estimation**

   In dynamic estimation the application is executed to get a better picture of the application characteristics. Dynamic profiling keeps track of the sequences of all executed instructions and data references. As a result, dynamic estimation is more accurate than static estimation. Moreover, the restrictions on static estimation like no recursive function calls etc., do not hold in dynamic estimation. Dynamic profiling requires an instruction set simulator of the target processor to obtain the application runtime parameters. It also requires the building of all the compiler tool chains etc. to compile the application so that it can be run on the instruction set simulator.

3. **Source Level Estimation**

   In source level estimation, the performance estimates are obtained from the source code of the application. This is quite a high level of abstraction to estimate the performance. Some source level estimation is done for the worst case execution time analysis. A model of the timing behaviour of the processor is built and together with an analysis of the structure of the code, an upper bound on the WCET (Worse Case Execution Time) of the code is obtained [15].

4. **Object Level Estimation**

   In object level estimation, the application is compiled to the object code of the target machine and the performance estimates are obtained from the resultant executable. This requires the availability of the compiler tools, like compiler, assemblers and linkers for the target architectures. Instruction set

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

simulator and trace based estimation comes under this category.

2.3 Static Analysis

Static timing analysis is essential in real-time systems development, where the schedulability analysis of programs with hard real-time constraints depends on the estimated extreme case performance [16]. Static analysis is inherently difficult. It was shown in [17] that the number of possible paths can easily become unmanageably large. For example consider the following code :

```
for (i=0; i < 100; i++) {
    if ( rand() > 0.5 )
       j++;
    else
       k++;
}
```

In the preceding code, the loop has 2^{100} possible paths depending on which branch is taken in each iterations of the loop. If the timing cost to increment j is equal to the timing cost to increment k, then all of these paths are worst case paths [17].

As explained in section 2.2, static estimation is not so accurate and has limitations. Many approaches have tackled the problem at a coarse grain focusing mainly on the WCET [18]. But due to the issues with static estimation mentioned above, it is quite hard to efficiently obtain the WCET. To get around these drawbacks, the author in [13] proposes some restrictions

- Programs must not contain any (direct or indirect) recursions. Recursive algorithms have to be either replaced by iterative ones or transformed into non recursive schemes by applying program transformation rules

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

- The absence of function variables and parameters is enforced in programs for which the execution times have to be calculated. Calls of subroutines via variables or parameters have to be substituted by explicit subroutine calls.

- In third generation programming languages every semantic that can be programmed with *gotos* can also be achieved with the standard language constructs - sequences, loops, and iterations. As a consequence the elimination of *gotos* does not result in any restrictions on programming.

- Since loops are fundamental for the implementation of almost every algorithm, one cannot eliminate loop constructs from programming languages. Nevertheless it has to be guaranteed that every loop terminates within a specified amount of time. As a consequence loop constructs which force the programmer to give some information about the time spent in a loop have to be introduced. Loops of this kind are called bounded loops. The authors introduced two types of bounded loops:

  1. Loops with a specified limit for the number of iterations.
  2. Loops which are bounded by the time limit that must not be overrun at run-time.

A source level estimation of C programs is proposed in [18], where the estimation is made from the source code and the number of times each high level constructs are executed. The execution counts are obtained from source level profiling of the application. This has the disadvantage of not considering the compiler optimizations. In [19], the authors partially compile the application and annotate the partially compiled code with timing delays to arrive at the estimation. The source based estimation compiles the code to a virtual instruction

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

set, then this instruction set is characterized for a target processor using a set of benchmarks to generate solutions for a system of linear equations based on summation of instruction execution time.

As has been pointed out, the source level based estimation techniques have problems considering compiler optimization in the estimation process. To overcome this in [4], the authors propose a framework, where the application is compiled to the assembly code level. This assembly output is used to generate a C simulation model annotated with timing information. As this takes into account all the architectural effects (instruction scheduling, register allocation, address modes, memory accesses, ...), the accuracy is higher than that of [19]. However this technique also suffers from the requirement of compiler tools for all the processors.

In [20], the authors extract application parameters from the intermediate format. The parameters extracted are basic block size, number of multiple-accumulate operations, ratio of I/O instructions to total instructions. They then try to match these parameters to processors. To obtain the dynamic characteristics of the application, the program is executed under the *gprof* profiler. The application parameters and the relevant processor architecture features, that the authors tried to match is provided in Table 2.1.

In [21], the best and worst case times for each basic block is determined. This is followed by a longest or shortest path analysis on the program control flow graph to obtain the upper and lower bounds. As this did not result in sufficient accuracy, feasible and false paths were identified. The feasible and false paths are defined as:

Definition A *feasible program path* is a path in the flow graph corresponding to a possible sequence of basic blocks when the program is executed, i.e., leading from the first to the last basic block of a program.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 2.1: Application Parameters [20]

Parameter	Relevant Processor Architecture Features
Average block size	Branch Penalty
Number of multiply-accumulate operations	Multiply-accumulate as a custom operation
Ratio of address computation instructions to data computation instructions	Separate address generation ALU
Ratio of I/O instructions to total instructions	Memory bandwidth requirement
Average arc length in the data flow graph	Total number of registers
Unconstrained ASAP scheduler results	Operation concurrency and lower bound on performance

Definition A *false program path* is a path in the program flow graph which cannot be executed under any input condition.

In addition to loop bounding as was done in [13, 22, 23], the user is also required to annotate the false paths. This was useful as the author's observed that many embedded programs or at least parts of such programs have a single feasible program path. This they called the single feasible path (SFP) property. The basic block or path segment is simulated using a cycle accurate simulator (this makes the technique not perfectly static). This result is used to estimate the performance using the sum-of-basic-blocks model.

Definition Let a program consist of N basic blocks, with c_i as the execution count of basic block bb_i and t_i execution time of basic block bb_i, $i = 1, 2, \ldots, N$. Then the sum-of-basic-blocks model assumes for the total program execution time T:

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$T = \sum_{i=1}^{N} c_i \times t_i \tag{2.1}$$

2.4 Architecture Description Languages

A considerable amount of research has been devoted to machine description languages. ADLs enable design automation of embedded processors as shown in 2.1 [24]. Research in this is mainly driven by the need to develop retargetable compiler tool set. These architecture description languages describe the processor. This description is sufficient for automatic generation of compilers, assemblers, linkers, simulators and debuggers etc. This helps in separating the codes of such tools into architecture specific and architecture independent parts. The architecture independent parts can be reused across a wide range of architecture, thereby saving time and effort. The designers of these architecture description languages, have to trade off between complexity of the tools and the level of the abstraction. More generality is obtained by lowering the abstraction level [25].

Traditionally, architecture description languages have been classified into three categories: structural, behavioural and mixed. This classification is based on the nature of the information provided by the language [25]. This classification is based on the *content* of the description written in a particular ADL. ADLs can also be classified based on their *objective.* Objective-oriented classification is based on the purpose of an ADL. Based on the objective ADLS can be classified into: simulation-oriented, synthesis oriented, compilation-oriented and validation oriented [24].

Following are the different types of ADLs based on the content:

1. **Structural Languages** : Structural languages define the structure of the processor. The different hardware resources of the processors are described.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

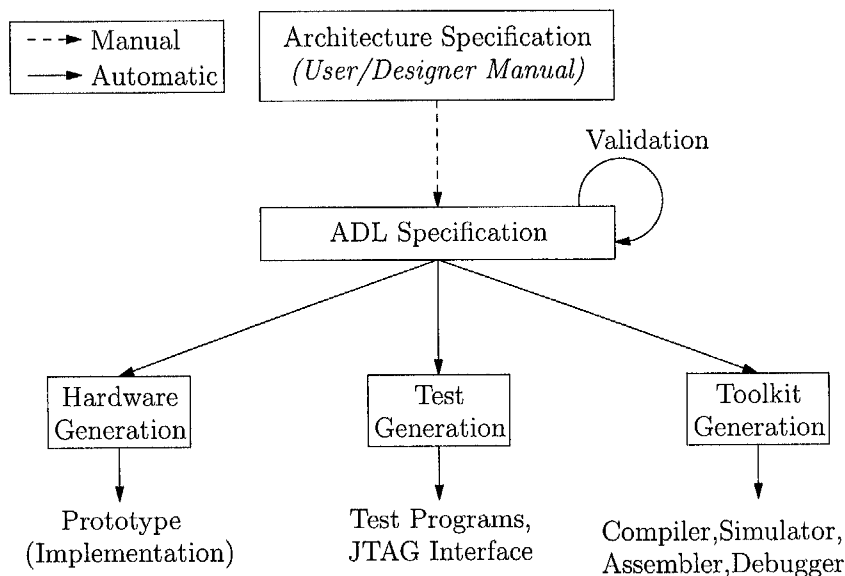

Figure 2.1: ADL-driven design automation of embedded processors [24]

Structure refers to the nature of the communication paths within the system; the structure of the hardware system is described in terms of its "primitive" functional and storage components and the manner in which they are interconnected [26].

2. **Behavioural Languages** : Behavioural language describe the dynamic aspects of the processor. In contrast to structural languages, behavioural languages ignores the detailed hardware structure [25].

3. **Mixed Languages** : Mixed languages extend behavioural languages by including abstracted hardware resources in the description [25].

Similarly following is the objective based classification of ADLs:

1. **Compilation Oriented** : Compilation oriented ADLs are primarily focused on the automatic generation of retargetable compilers. Retargetable

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

compiler generators generally read a description of the different processors which is written in a particular ADL and then generate a compiler for the particular processor. Other related tools like assemblers, loaders etc. are also similarly automatically generated. The key idea is the automatic generation of the tools.

2. **Simulation Oriented** : Simulation oriented ADLs are used to auto-generate simulators. Different ADLs are better suited for generation of simulators at different levels of abstraction. Behavioural ADLs can enable generation of functional simulators while structural ADLs are suited for cycle-accurate simulator generation [24].

3. **Synthesis Oriented** : Synthesis oriented ADLs are used for hardware generation. Structural ADLs are better suited for hardware generation as they describe the various interconnects. Mixed ADLs are also used for hardware generation as they also include the structural information.

4. **Validation Oriented** : These are used for test generation for functional validation of embedded processors.

Fig 2.2 shows the classification of ADLs

MIMOLA (Machine Independent MicrOprogramming LAnguage) is the common input language for a variety of CAD tools for the design of VLSI systems, developed at the University of Dortmund. It is similar to common hardware description languages (HDL) like VHDL or Verilog, but on a higher level of abstraction [27]. Hardware structures are modeled as netlist of hardware components, while the behavioural description is done using a PASCAL-like program [28, 29]. Hierarchical design styles are supported, with the assumption that each hardware structure consists of hardware elements, which can be described by their structure or by their behaviour [29]. One advantage of MIMOLA is that the same

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

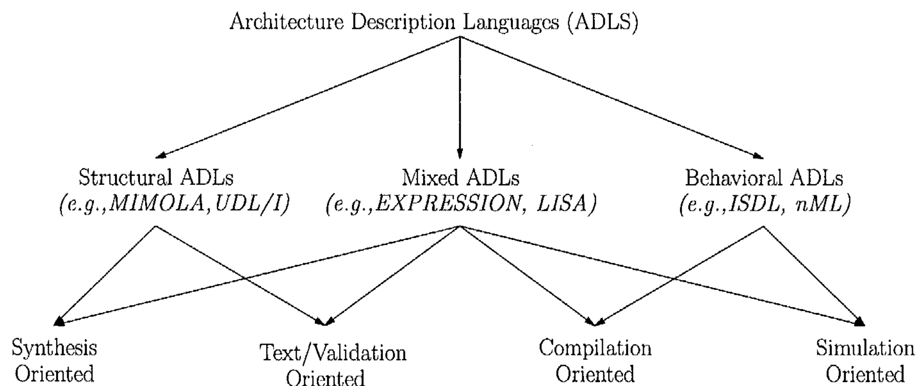

Figure 2.2: Taxonomy of ADLs [24]

description is used for both processor synthesis and code generation. MIMOLA descriptions are generally very low level and laborious to write [30].

In nML, a skeleton of the target structure is constructed by declaring all storage entities. The register transfers between these storage entities describe the exact execution behaviour of the machine [31]. nML has been used to generate retargetable instruction set simulator. It has also been used to build a retargetable code generator, CHESS [32]. nML however does not directly support multi-cycle or multi-word instructions.

Sim-nML is an extension of nML which includes the timing of various operations and a resource usage model. The main idea behind the resource usage model is that, as an instruction executes, it holds a set of resources like functional units etc. Capturing this model helps in the study of the performance of the processor [33, 34, 35, 36].

ISDL (Instruction Set Description Language) provides support for explicit constraints, which simplifies description of processors with instruction level parallelism. Without explicit constraints, every legal combination of the micro instructions has to be listed, which makes the writing of processors with instruction

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

level parallelism, very laborious [37]. HMDES is a hardware description language used in the Trimaran system. This language describes a processor architecture from the compiler's point of view and specifies the instruction format, resource usages and reservation tables, latency information etc [38].

Another architecture description language EXPRESSION [30], supports building of cycle accurate simulators. In EXPRESSION the system is described both in terms of its instruction set and its structure. It also supports specification of detailed constraint information in the form of reservation tables needed for optimizing compilers which is not available in languages like LISA [39]. Reservation tables are needed by the compilers scheduler to test for resource conflicts between operations whose execution cycle overlap. Unlike HMES, in which the user has to laboriously enter the reservation on a per operation basis, the reservation table is automatically generated from the structural information in EXPRESSION.

PD-XML consists of a collection of three main entities. The first captures information about components that store information, such as register files, stacks, external memory, or block RAMs on FPGAs; hence is called the store entity. The second entity describes the instruction set and is called the inst entity; it may include pseudo instructions which are decomposed by a compiler into several operations that are executed by the processor. The third is the resource entity and it contains information about physical resources available in a micro-architecture such as ALUs, cache control, fetch and decode units. An instruction set architecture (ISA) description of a processor mainly involves the store and inst entities; the micro-architecture description requires all three entities. An ISA description should: (1) expose the capabilities of an instruction set to the programmer and compiler writer, as well as (2) provide a functional specification of the instruction set for implementation by micro-architectures. As such an ISA description is made up of store and inst entities. The bit-width of registers and instruction

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

formats are included at this stage to allow for the generation of binary code. PD-XML allows for extensible descriptions for both instruction set architectures and their micro-architecture implementations [40]. The description is based on XML, which makes it easily extensible.

2.5 Instruction Set Simulators

An instruction set simulator is a tool that runs on a host machine to mimic the behaviour of running an application program on a target machine[41]. While they are mostly used to validate architecture design and to evaluate architectural design decisions during design space exploration, they can also be used to run an application to obtain the performance estimates for a particular processor. These simulators are useful as compiled code can be directly executed on them and gives as output the execution statistics, like time taken, number of cycles, IPC (instructions per cycle) etc.

Simulators are classified into *functional* simulators and *performance* simulators. Functional simulators implement the instruction set architecture, which can be used to verify that the application being simulated gives the correct results. Additionally they may check for correct alignment and access permission for each memory reference. Performance simulator on the other hand, implements the micro-architecture and hence is able to provide detailed timing results [42]. Functional simulators are further classified into *trace-driven* and *exec-driven*. In trace based simulation a trace of program execution is obtained, which is later analyzed to obtain the different simulation statistics. In execution based simulation, the application is actually executed and the simulation data are obtained on the fly. These are more difficult to implement compared to trace-driven simulators. Performance simulators, are of two types: *Instruction schedulers* and *Cycle*

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

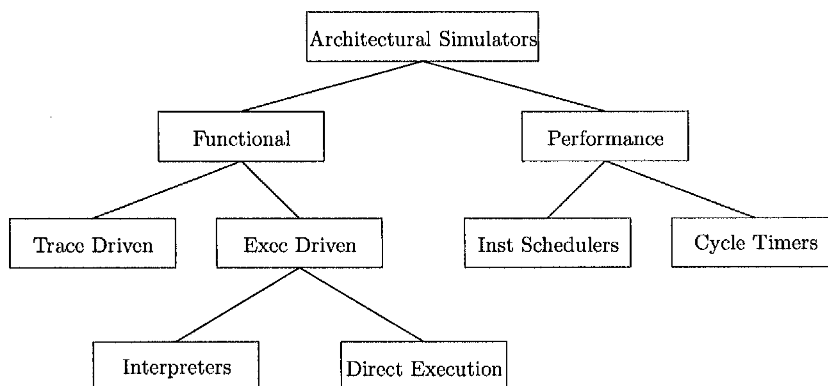

Figure 2.3: Taxonomy of Simulation Tools [42]

timers. Instruction schedulers are constraint based schedulers, which schedule the different instructions, based on the availability of micro-architecture resources. The simulator implemented by [36], falls under this category, which uses a resource usage model to schedule the different instructions. Cycle timers track the micro-architecture state for each machine cycle. These are suitable for detailed micro-architecture simulation, but are slower than the less detailed ones. *Sim-outorder* included with the SimpleScalar Architectural Research Tool Set, is one such cycle accurate simulator. Fig 2.3 shows the taxonomy of the different simulators.

SimpleScalar [43] is a popular instruction set simulator which can simulate arm, powerpc and alpha architectures. Shade [44] is an instruction set simulator and a custom trace generator. Current Shade implementations run on SPARC systems and simulate the SPARC (Versions 8 and 9) and MIPS I instruction sets [45]. The SimIt-ARM [46] package contains an instruction-set simulator and a cycle-accurate simulator for the StrongARM architecture. *ppc750sim* is a POWERPC 750 (G3) simulator written in SystemC. It is intended to simulate precisely the hardware of the POWERPC 750 microprocessor, and achieves a 15% inac-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

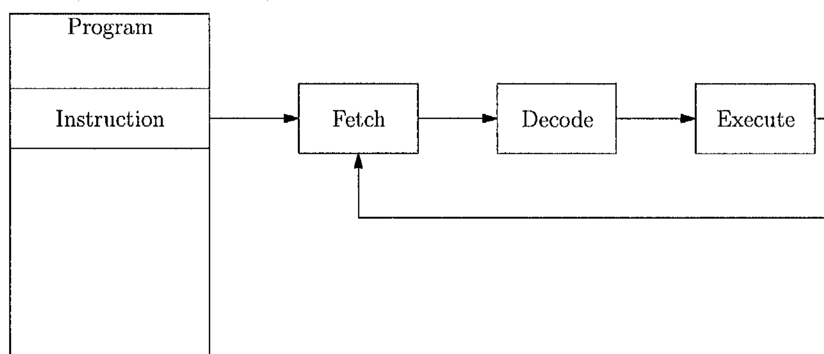

Figure 2.4: Interpreted Simulation

curacy on SPEC CINT 2000 compared to a real POWERPC 750 microprocessor [47].

In addition to these, there are commercial instruction set simulators, which are provided by the respective processor manufacturers. The RealView Developer Suite Instruction Set Simulator (RVISS), provides accurate simulation of ARM and Thumb core-based processors [48].

In [49, 50], the authors classify instruction set simulators into two categories: interpretive simulation and compiled simulation. Fig. 2.4 shows the three main stages in a traditional interpretive simulation. In interpreted simulation, the program is stored in the simulators memory in the same way as it would be in the simulated computer. The simulator program repeatedly fetches instructions from memory, using the opcode to select a routine to execute that will simulate the effects of that opcode, as well as maintain statistics, e.g. instruction frequency, execution time, etc [50].

In compiled simulation on the other hand, the time consuming decoding step is moved from the runtime to compile time [49]. The basic flow of a compiled simulation is shown in Fig. 2.5. The compiled simulation method is faster than

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

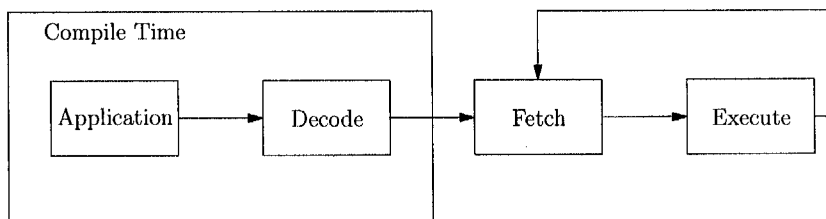

Figure 2.5: Compiled Simulation

the interpretive simulation as in the former the majority of time is spent in fetching and decoding the operations. Typically simulators based on interpretive techniques execute from ten to a hundred instructions for each interpreted instruction [50]. Some speedup is also achieved by doing the instruction scheduling during compile time [51].

In [11], the authors used an extended version of the DLX software simulator *dlxsim*. The simulator tracks stalls and instruction mix per function. For estimating cache performance, memory trace of the application and the cache parameters are fed to a cache simulator *dinero*, which allows easy configuration of cache parameters like block size, associativity. Execution time of the CPU is decomposed into three separate components.

$$CPU_{time} = CPI_{total}\left[\frac{cycles}{instr}\right] \times IC\left[\frac{instr}{program}\right] \times t_{clock}\left[\frac{sec}{cycle}\right] \tag{2.2}$$

- Clock cycle time (t_{clock}) is the inverse of the CPU clock rate. It depends on the processor and the technology it uses.

- Instruction count (IC) is the total number of instructions executed. This was measured with the simulator *dlxsim+*.

- Cycles per instruction (CPI) is the number of cycles taken to execute a single instruction. The CPI_{base} can be calculated by summing the product

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

of each individual component (CPU_i) by the fraction of occurrences (Mix_i) of that instruction in the program

$$CPU_{base} = \sum_{i=1}^{n} CPI_i \times \frac{IC_i}{IC} = \sum_{i=1}^{n} CPI_i \times Mix_i \tag{2.3}$$

Due to pipeline, cache and interface stalls the overall CPI increases.

$$CPI_{total} = CPI_{base} + CPI_{pipeline} + CPI_{memory} + CPI_{interface} \tag{2.4}$$

For measurement of the overall CPI the authors first measure the increase in CPI attributed to pipeline stalls caused due to data and control dependency

$$CPI_{pipeline} = CPI_{data_stall} + CPI_{control_stall} \tag{2.5}$$

Next the penalty caused by memory hierarchy is measured by feeding the memory trace to the cache simulator. The output of the simulator is used in the formula

$$CPI_{memory} = \frac{accesses}{IC} \times missrate \times penalty_{cache} \tag{2.6}$$

Accesses per instruction and miss-rates are reported by the cache simulator. Cache penalty is dependent on the processor architecture. Next the overhead caused by the hardware-software interface is measured by

$$CPI_{interface} = \frac{calls}{IC} \times penalty_{interface} \tag{2.7}$$

Instruction set simulators suffer from the problem that they are very slow compared to the normal execution. [41] attempts to make simulation faster by moving time consuming decoding process from run time to compile time. This technique works only when the program is not modified during runtime. For example, ARM processor uses two instruction sets (normal and reduced bit-width) and dynamically switches to different instruction-set at run time [49]. [52] tries to reduce the simulation time by grouping source instructions and translating them as a unit.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

2.6 Tracing

In tracing an application is executed and a log of the instructions executed and or memory accesses is maintained. This log or trace as it is called is used to obtain the performance estimates. Traces are of two types. One is the instruction trace which is a sequence of addresses of instructions executed, and the other is the memory trace which is a sequence of addresses of memory locations the program refers to while executing. Memory traces are used to simulate caches and memory systems [36]. Typically tracing is implemented by embedding instrumentation code into the program [53].

Traces are difficult to obtain since if the system is modified to measure it, the values measured may be made invalid. For example, introduction of instrumentation code may affect the behaviour of the execution and the data obtained may be vastly different from what would have been observed if no instrumentation codes were inserted. Following are the traditional trace gathering techniques [54]:

1. **Single-stepping and breakpoints** : Instruction sequences can be obtained by using the debugging features built into the processor. By enabling the processor's single-stepping mode or by setting instruction breakpoints at each basic block, debugging interrupts can be used to record the execution trace. Every time an interrupt occurs, the relevant data can be logged to a file, and then the control can be returned to the point where the execution of the application was interrupted.

   This method suffers from the overhead of processing every interrupt and consequently the performance is severely degraded. Moreover the repeated execution of the debugging interrupt handler code will impact many of the processor's performance-enhancing caches.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

2. **Instruction inlining** : In instruction inlining instrumentation code is inserted into each basic block. Although this technique successfully avoids the overhead of frequent interrupt processing, there is still the problem of getting a trace of a modified application and not the original executable.

3. **Hardware monitoring** : In this the processor is monitored using logic analyzers and other hardware monitors. While this may be able to generate near perfect traces, most processors unfortunately do not provide information on the pins that directly identify the instructions being executed. Monitoring the system bus for instruction fetch activity won't work either, since most of the fetches will be satisfied in the first or second level cache. Of course, all instruction fetches can be forced to appear on the bus by disabling the caches, but this will inevitably alter the original system's behaviour. Furthermore, it might be difficult to distinguish normal instruction fetches from speculative execution, prefetching, and data reads.

4. **Processor simulation** : The traces are obtained using a software simulator for the particular processor under consideration. The difficulty in this method lies in building a fast and efficient simulator which models the processor accurately. Moreover, if the application being simulated needs real-time continuous input, this method may not be suitable as a software simulator will run considerably slower than a real processor.

There are different ways to configure the interface between trace generation and simulation.

1. **Files** : One way is to store the trace data in a file and then use it later to simulate. It has the disadvantage of taking up huge disk spaces for storing the trace data. The problem of huge trace files is avoided in the other configurations listed below. While these configurations overcomes the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

disadvantage of using huge file spaces, the trace is not available for later use and has to be regenerated every time it is required.

2. **Unix Pipe**: The generated trace can be fed to the simulator using the Unix file pipe mechanism [53].

3. **Shared Memory**: The generated trace is written to a shared memory which is used by the simulation process. Since writing to shared memory is faster than writing to the file system, generating the trace concurrently with simulation adds less than 1% overhead to the total simulation time [55].

4. **Unix Sockets** : In the socket mechanism on the other hand, the trace is generated on a machine different from the machine where the simulation is run and the trace data are transferred over Unix sockets. In this case as the trace generation and simulation is performed on different machines, trace generation does not add any additional time to the simulation [55].

In addition to the disadvantages of tracing mentioned earlier, it also has the disadvantage that only non-speculative or committed instructions are recorded in the trace. Hence, the effects from speculatively executed instructions from incorrect branch paths are lost [56]. [55] claims that tracing is largely inadequate for simulation in the modern architectures and can lead to inaccurate results. Tracing does have an advantage though. Once execution trace has been generated, multiple pipeline architectures can be evaluated using the single trace [51].

2.7 Profiling

Profiling is the collection of a program's performance characteristics by measuring the events happening while the program is being executed. The data are gathered

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

using the following two techniques

- **Statistical profile** : A sampling profiler probes the target program's program counter at regular intervals using operating system interrupts. Sampling profiles are typically less accurate and specific, but allow the target program to run at near-full speed.

- **Instrumentation** : Some profilers instrument the target program with additional instructions to collect the required information. Instrumenting the program can cause changes in the performance of the program, resulting in inaccurate results.

Statistical sampling is less disruptive to program execution, but cannot provide completely accurate information. Code instrumentation, on the other hand, may be more disruptive, but allows the profiler to record all the events it is interested in. Specifically in CPU time profiling, statistical sampling may reveal, for example, the relative percentage of time spent in frequently-called methods, whereas code instrumentation can report the exact number of time each method is invoked.

In [57], instrumentation code is inserted at the *Three Address Code Intermediate Representation* level where all C operators and the majority of memory accesses are visible. Additionally, high level standard IR optimizations, such as constant propagation, constant folding and loop invariant code motion, can be performed on this IR. Such optimizations prevent chances of false prediction (such as counting operations that will be eventually eliminated by compiler optimizations) in estimating cycle counts and memory accesses. The instrumenter adds a line of code after each IR operation to increase cycle counter by the operation cost. This cost can be assigned by the user. The total execution cost is given by the following formula

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$Cycles = \sum_{i=1}^{n} E(O_i) \times C(O_i) \tag{2.8}$$

where $E(O_i)$ and $C(O_i)$ are the execution count and the cost of an operator O_i, respectively.

The instrumenter also inserts function calls to intercept and report all accesses to different source level data elements, such as arrays, structures and global variables, found in any application. Usually memory accesses, for an application written in a high level language like C, originate from four sources :

1. Accesses to global *scalar and composite variables* (i.e. structures) and arrays
2. Accesses to a function's local *composite* variables
3. Accesses to *dynamically allocated memory* on the heap
4. Accesses during building-up and cleaning-up of a function's stack frame

The local scalar variables are usually allocated in registers, and the number of memory accesses caused by them is often negligible. The 3-AC IR makes the first three kinds of memory accesses explicit by converting all global accesses and local composite accesses to pointer *dereference* operations.

2.8 Worst-Case Execution Time

Worst-case Execution Execution Time (WCET) analysis finds an upper bound for the time needed to execute a program. This is important for hard real time systems, where failure to perform within the timing constraints can be fatal. An operation performed after the deadline is, by definition, incorrect. WCET analysis therefore tries to provide a time value within which the application will finish execution under any circumstances. For WCET estimates to be valid, the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

estimates must be *safe*, i.e., guaranteed not to underestimate the execution time. To be useful, they must be *tight*, i.e., provide low over-estimations [58].

According to [58], there are three main phases in WCET estimation :

- ***program flow analysis*** : This phase determines the possible flow of the program. Information like the number of iterations of a loop, dependencies if any between if-statements etc. are extracted in this phase, without paying any attention to the time taken for the execution of any of these control units.
- ***low level analysis*** : In this phase the time taken for the execution of each atomic unit of flow is determined. The hardware features like instruction cache, branch prediction etc have been analyzed.
- ***calculation*** : In this phase the WCET estimates are calculated based on the information determined from the previous two phases.

There are three main categories of calculation methods proposed in literature [58]:

- ***path-based*** : The final WCET estimate is generated by calculating times for explicitly represented paths in a program, searching for the path with the longest execution time.
- ***tree-based*** : The final WCET is generated by a bottom-up traversal of a tree representing the program.
- ***implicit path enumeration technique (IPET) based*** : The control flow of a program and its atomic execution times are represented using algebraic and/ or logical constraints. The WCET estimate is calculated by maximizing an objective function while satisfying all constraints.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

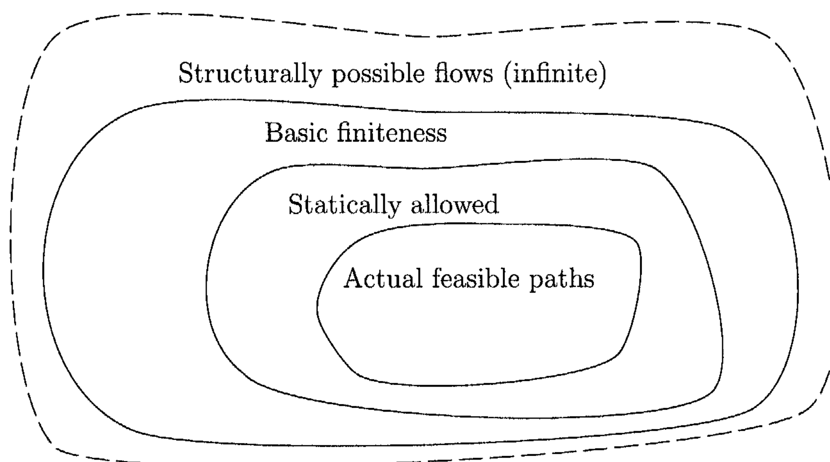

Figure 2.6: Relation between possible executions and flow information [58]

It would be impossible to analyze all the possible paths for an application. As this takes into consideration all possible paths and loops can theoretically execute arbitrary number of times and hence this set is usually infinite. This set is made finite by introducing *basic finiteness information*, where all loops are bounded with some upper limit on the number of executions. Further information can be provided to narrow down the set of possible execution paths to a small set of *statically feasible paths.* These set of paths are illustrated in Fig. 2.6 [58].

The flow information can be provided both at the source code level or the object code level. If provided at the source code level, the information must be mapped to the object code to be used in the WCET calculation [58]. It was shown in [59, 60] that in the presence of compiler optimizations this problem is non-trivial.

Cinderella, a WCET estimation tool described in [17], uses the executable code of the application to construct the CFG (control flow graph) and then uses the CFG along with the original source code to output the annotated source code.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

If x_i is the number of times basic block B_i is executed when the program takes the maximum time to complete and c_i be the running time of this basic block in the worst case, then the worst case timing is given by the maximum value of :

$$\sum_{i=1}^{N} c_i x_i \tag{2.9}$$

The value of the expression is ∞ since any x_i can take the value of ∞. For this reason restrictions must be imposed on the programs. One of the common restrictions usually imposed is that the bounds on the number of iterations of each loop must be specified. This however will provide a very pessimistic estimate as in general the upper bound on each x_i is rarely achieved simultaneously. There are two factors which may preclude the upper bounds from being achieved simultaneously. These are illustrated by the following example code fragment:

```
for(i = 0; i < k; i++) {
    if(OK)
       do_something();
    else {
       do_something_else();
       OK = true;
    }
}
```

The first is that the program structure itself will impose conditions that make this impossible. In the above example code the loop has an upper bound of k iterations. The loop body itself consists of of a single if-then-else statement.

If the functionality of the program is ignored, in the worst case both the *then* part and the *else* can have k iterations. However, the mutual exclusion of

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

the *if-then-else* prevents this from happening simultaneously. Thus the *program structure* imposes conditions on what can and what cannot happen in terms of execution counts of basic blocks. The second factor that comes into play is the *program functionality*. This deals with what the program is computing. In the above example, the execution of the *else* part sets a condition that precludes the *else* part from being executed again. Thus, the execution count of the *else* part is at most 1. This type of information is not easily obtained from the program in most cases, but is something that the writer of the program can provide with relatively less difficulty. These two constraints are used to formulate an ILP to find the maximum value of the expression given in Eq. 2.9. The solution of the ILP provides the worse/best case execution time depending on whether the expression is maximized or minimized. It requires the user to provide the loop bounds for all the loops present in the application. One interesting technique of estimating the performance of each basic block is related to the fact that each execution of a basic block is not constant. The authors does a micro-architectural analysis of the processor to find the execution time for each basic block. Eq. 2.9 assumes that all executions of a basic block take the same amount of time. This is not true as the first execution of a loop may result in cache misses for all the instructions in the loop but subsequent iterations may result in cache hits. Hence assuming all iterations result in cache misses is overly pessimistic. Hence the authors consider the first iteration of a loop as a separate basic block.

[61] contends that WCET analysis has had hardly any industrial impact. The author attributes this to the high complexity of implementing and using the proposed WCET approaches. In addition, WCET research is always lagging behind the advances in micro-processor technology- whenever WCET research manages to deal with the features of one hardware generation, the next generation of processor and hardware architectures equipped with novel speedup features are

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

already there.

Many factors make WCET analysis difficult [62], including

- **Pipelining**: Hardware architecture features like pipeline helps speed up application execution by overlapping the execution of different instructions. But pipelining also makes WCET analysis difficult. When program execution is pipelined, the execution of an instruction is not only dependent on that particular instruction but also on the execution history of the program.
- **Caches**: Caches help improve memory access time. But this also make estimation difficult as the execution time of an instruction depends on the contents of the cache, which again is dependent on the execution history of the application. It has been shown that in dynamically scheduled processors, a cache miss in some cases can result in shorter execution time than a cache hit [63].
- **Improvement of average case by manufacturers**: As processor manufacturers optimize for the average case of the major benchmarks, the WCET analysis becomes hard to predict, even more so when the different features interact [15].

For advanced processors, there is a difference between average case and observed worst case due to the speed up introduced by the advanced processors like caches, pipelining and out of order execution. When applications have data dependent memory accesses or paths these differences increase. There is inherent variability of the execution time of programs. These difficulties in estimating the WCET, lead to pessimistic estimation of the performance.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

2.9 Limitations of Existing Work

The approaches mentioned in the previous sections have one or more of the following drawbacks

- **Conversion** : In many of the approaches, the application needs to be written in a specific language or the application has to be converted to a specific format. For example in [64], the system has to be described in SDL, a system level specification language. Similarly in [13, 22, 23], the initial program has to be annotated with bounds on the loops. As a result, the initial application in its original form is not suitable for estimation purpose.

  In trace driven estimation, the executable program, usually, needs to be embedded with instrumentation codes to generate the instruction or the memory traces.

- **Slow**: Most instruction set simulators provide a detailed model of the processor internals, like pipelines, bus contention. As a result the simulators are quite slow. This means a lot of time has to be spent in executing the program on it to arrive at the performance estimates. This again will be an acute problem when many processors have to be considered. In [18], the execution counts of the different programming constructs are obtained by executing the code under the single step mode, which can be $100 - 300$ times slower than a normal run.

  In trace based estimation too, the cost of recording every instruction and data address as the application executes is quite high. A simple tracing system examines every instruction as a program executes. This approach is inefficient and makes the program run slowly [36]. Run time of trace generation has been found to be 50-1000 times slower relative to original executable [53].

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

- **Compiler Tools** : All the instruction simulator based approaches require the building of compiler tool chains, like compilers, assemblers, linkers, and the simulator for all the target processors to be considered. Although machine description languages help automate the process of generation of all these tools, building them for all the processors will be quite time consuming. Also the machine description files for all these processors need to be written in one of these languages. In [4], the application needs to be compiled before the estimation.

- **Suboptimal Code** : Some architectures like DSPs do not lend themselves to automatic compiler generation of optimal machine code from high level language description [65]. Hence, the approach based on building of compiler tool chains may lead to suboptimal code and result in pessimistic estimation.

- **Complexity** : WCET analysis suffers from complexity of the analysis. In general the number of paths to be analyzed for an exact WCET analysis of a piece of code grows exponentially with the number of consecutive branches in the control flow of the analyzed code [61].

- **Supports limited architectures**: Most of the instruction set simulators simulate a limited range of architectures. Simplescalar supports ARM, Alpha, Powerpc and the PISA architectures. The SimIt-ARM simulator only supports the StrongArm architecture.

- **Compiler optimization**: In source based estimation like [18], the approach does not take care of the compiler optimizations. This can result in considerable error in a poorly written code.

- **Limited high level language constructs** : In the static estimations

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

the application is restricted and not allowed to have unbounded loops and recursions. Functions pointers are not allowed in [13]. In [66], the methodology supports only a subset of C and C++ with *gotos* not allowed, also it places certain restrictions on the *switch* statement.

- **Trace Size**: In trace driven estimation, the size of the trace generated can be quite high. A 10-million-instruction-per-second (MIPS) processor produces up to nearly 70 megabytes of trace per second of execution. This makes the trace for long time execution of the program difficult to store [36].

- **User Intervention** : User intervention is required in some of the existing techniques, especially the WCET based techniques requires the user to provide bounds on the loops.

2.10 Summary

This chapter provided a survey of the various performance estimation techniques currently in use. Simulation based approaches can be very time consuming, which is even more time consuming when a large number of processors have to be considered. Tracing is faster than simulation but they may suffer from other disadvantages like huge trace sizes. Hence a fast estimation procedure which does not require building of compiler tools for each processor should help in reducing the total design time.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 3

Intermediate Representation For Performance Estimation

3.1 Introduction

With the increase in the complexity of embedded system, more and more parts of the system are now being implemented in software. Implementing in software also significantly reduces the time to market. As explained in chapter 2, the existing work in processor performance estimation concentrates in increasing the accuracy of the estimates. Hence, they are quite time consuming. As the number of commercial off the shelf processors increases in the market, the processor selection time will increase significantly.

A typical embedded program runs on a processor with parts of the program running in hardware so that performance requirements are met. It has been shown that partitioning of an application into hardware and software improves the performance [67]. It is important to select a processor whose performance is close to the performance requirements of the application. Selecting a processor which under-performs by a huge margin means a very large piece of hardware

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

have to be used to meet the requirements. On the other hand, an over-performing processor would be very expensive which would reflect in the higher unit cost of the product. Additionally, it is important to select a good processor as software development cost is around *to* to 75% of the embedded system project cost [68]. The current estimating methods rely heavily on compiling and then executing the application on a instruction set simulator. This is a time consuming process and even more so when you have to consider a large number of processors and have to evaluate their performance. There is also the added cost of the development tools itself. Toward this end a processor performance estimation methodology is proposed.

Our estimation methodology relies on converting the given application to the LLVM [69] intermediate format. LLVM (low level virtual machine) format is a low level intermediate format which while being low level is also machine independent. In the proposed methodology, the application is converted to the LLVM intermediate format which is then used to obtain the performance estimates. The intermediate format of the application is used to obtain a rough estimate of the compiled code. This provides us with the estimated number of instructions in each of the basic blocks. The LLVM intermediate format can also be executed under the environment. This gives us the execution frequencies of the different basic blocks. The execution count along with the estimated instruction is used to obtain the final performance estimates.

3.2 Background

3.2.1 Performance estimation and performance evaluation

As discussed in chapter 2, a detailed performance evaluation may take a large amount of time. Hence the same may not be a feasible way to select a processor

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

from the wide range available. Detailed performance evaluation may also be a more expensive process due to the cost of the development tools and the simulator. A comprehensive performance evaluation would involve measurement of all performance parameters of a processor while executing a particular application. It would typically consider the effects of cache misses, branch mis-prediction, input-output latencies, power consumption. While a detailed evaluation can give more and accurate data for analysis, it requires more effort to obtain these performance parameters.

In performance estimation a few parameters which have large effect on the performance are considered to obtain a rough estimate of the performance. Performance estimation without using a detailed simulation can easily shorten the time taken for arriving at the performance figures. Although the simulation results will be more accurate than estimation, more number of processors can be taken into consideration.

3.2.2 Abstraction level for estimation

The current methodologies for performance estimation uses varying level of abstraction for the processor. For example, in [15], the estimation is done from the source code, while in simulation based approaches like [36], the estimation is done at the object code level. The estimation at the source code level, is based on calculating the performance figures for each of the high level programming statements in a particular programming language. While at the object code level, the calculation of performance is done for all the instructions present in the compiled code. The latter is more accurate than the former but is also more complex. Similarly for instruction set simulators, the simulators may just implement a simplified model of the processor which executes the instructions or it can be a detailed implementation which models the cache, instruction pipeline, branch predictors

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

etc. The simplescalar tool set includes within simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction [43]. The functional simulator, *sim-fast*, is the least detailed simulator but is also the fastest. Then at a lower level of abstraction are, *sim-cache* and *sim-cheetah*, which are functional cache simulators. The most complicated and detailed simulator, *sim-outorder*, is also as expected the slowest. In other words, a more detailed and accurate simulator would take more time to execute than a less accurate one. Fig. 3.1, shows the execution times for the different simulators which model the processor at different abstraction levels. The execution times are for one of the benchmark from the MiBench Benchmark suite [70], simulated on a Sun machine. As can be seen the most detailed simulator, *sim-outorder*, is slower than the least detailed one, *sim-fast*, by an order of two [1].

Not only the execution time is larger for a detailed simulator but the coding effort is also higher. *Sim-fast* and *sim-safe* are implemented using less than 400 lines of code, while the code for *sim-outorder* is about 4000 lines of code [71]. Fig. 3.2, shows the coding effort involved in the different simulators of the simplescalar tool set.

3.3 Estimation Technique

In our estimation technique, the application is converted to the intermediate format and the intermediate code is used to estimate the number of instructions generated in target instructions for each of the different intermediate instructions. The performance characteristics is used to obtain the estimates for the standard

[1]The Simple Scalar user guide [71] claims that *sim-outorder* runs about an order of magnitude slower than *sim-fast*

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

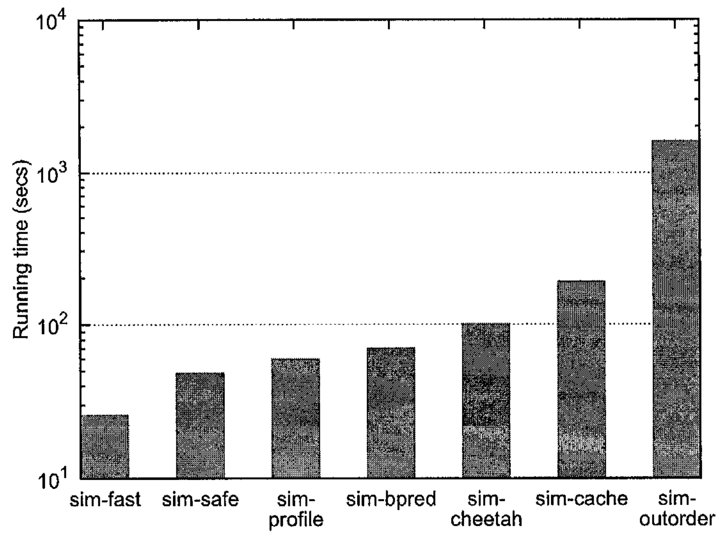

Figure 3.1: Relative runtime of simulators at different levels of detail

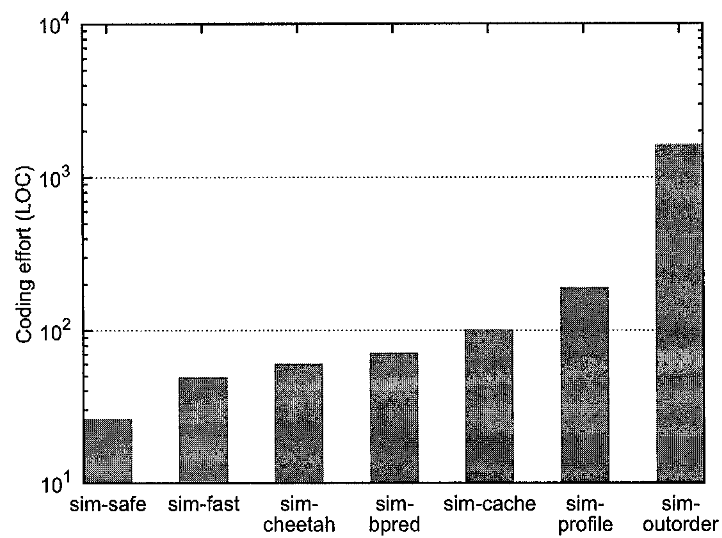

Figure 3.2: Coding effort of simulators at different levels of detail

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

library function. The IPC (instructions per cycle) of the application is estimated by analyzing the variation of IPC with average basic block length. The estimation is performed using an intermediate format which is low-level enough and still processor independent. The Low Level Virtual Machine (LLVM) format [69] is used. LLVM is a compilation framework for research in compiler optimizations across the entire lifetime of a program. The LLVM virtual instruction set is a low level program representation using simple RISC-like instruction. LLVM incorporates a virtual machine which can run the LLVM object codes from which profile data can be easily extracted. This was used to obtain the execution counts of the different basic blocks in the intermediate format. The estimation methodology is graphically presented in Fig. 3.3. As can be seen from the figure, first the application is converted into the LLVM intermediate format and the program details are extracted from the intermediate format, and then it is executed on the virtual machine to generate profile output. The output of the above two steps is used by the estimator to provide the performance estimates. The methodology is described in detail below:

The application is assumed to be provided as a C/C++ program. This does not limit the methodology in any way. There are front-ends available to convert C/C++ programs into LLVM format. Front-ends for Java and Scheme are under development. The application is converted into the LLVM format using the GCC front-end available from the LLVM web page[72]. The front-end converts the input programs in C/C++ into a bytecode file.

LLVM comes with lots of inbuilt passes. These passes can be run on the bytecode file and after it does the required transformations, produces a new bytecode file. After the bytecode file is obtained, it is run through a series of machine independent optimisations. The optimisations performed on the bytecode file are namely, Aggressive Dead Code Elimination, Constant Propagation, Dead In-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

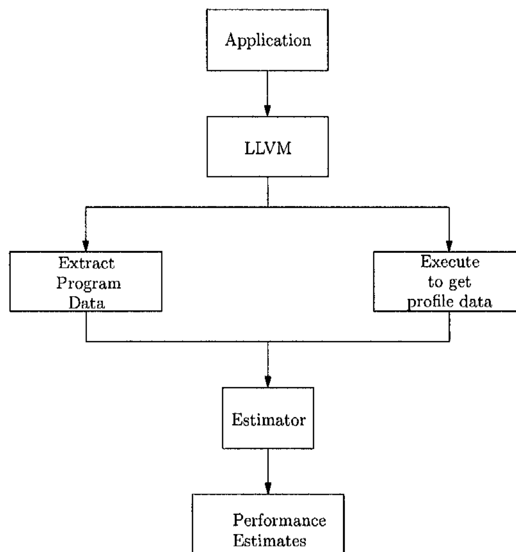

Figure 3.3: Basic Flow

struction Elimination, Global Common Subexpression Elimination etc.

After the bytecode file has passed through these machine independent optimizations, the intermediate format instructions for it is extracted. A small extraction pass was written for this purpose. The pass iterates through all the instructions and prints out the relevant information, like the number of arguments in a function call. A sample of the extracted information is given below

```
Basic Block name: __main.entry
Inst: call  called function: printf num_args:1
Inst: call  called function: SolveCubic num_args:6
Inst: load
Inst: setgt
...
```

In brief the above intermediate format conveys the information that it is the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

intermediate format for the basic block named "__main.entry". It consists of the instructions a) **call** instruction: which calls the function *printf*() with one argument, b) **call** instruction: which calls the function *SolveCubic*() with six arguments, c) **load** instruction, d) **setgt** instruction and so on.

After the initial intermediate code is passed through processor independent optimizations, the resultant intermediate format instructions is used to estimate the number of instructions generated in the final machine code that would be produced. Many programming constructs were compiled, both till intermediate format and right up to the final executable format. The results were analyzed and the intermediate as well as the final object codes produced were compared. This helped obtain a correlation between the intermediate code and the machine code. Since the LLVM intermediate code is low-level, most of the LLVM instructions have direct counterpart in the machine instructions. The process of estimation for different LLVM instruction is explained in the following sections.

3.3.1 Instruction Classification and Estimation

For our estimation purpose, the LLVM instructions are classified into the following three categories.

1. *Simple Instructions* : Instructions which have a direct implementation in the target architectures instruction set.
2. *Compound Instructions* : Instructions which do not have a equivalent instructions in the target architectures instruction set and hence is broken down into simpler instructions from the target architecture's instruction set.
3. *Functions Calls* : These are instructions which invoke a function.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

This classification of the LLVM instructions are for a particular target. The classification may be different for different targets. Simple instructions for the arm architecture is shown in table 3.1. The LLVM instructions on the left hand side of the table can be implemented by the arm equivalent instruction on the right. Hence these LLVM instructions result in one instruction in the final object code after compilation.

Table 3.1: LLVM instructions and their ARM equivalent

LLVM instruction	ARM equivalent
br	b, bl
add	add, addc, qadd, qdadd
sub	sub, sbc, rsb, rsc, qsub, qdsub
mul	mul, mla, umull, umlal, smull, smlal
setcc	cmp, cmn
and	and
or	orr
xor	eor
shl	lsl, asl
shr	lsr, asr
load	ldr, ldm
store	str, stm
call	bl

To estimate the number of instructions generated for each LLVM instructions in an application, the instruction is checked to see which categories it belongs to. For simple instructions there are direct equivalents in the target instruction set and hence they produce one instruction per LLVM instruction in the compiled

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

object code.

For the compound instructions, programs containing these instructions were compiled to the final object code and the resulting instructions from these LLVM instructions were studied. The procedure for the 'switch' LLVM instruction is demonstrated below. The C program shown in Fig. 3.4 was used as a test program (only relevant portion is shown):

```
switch(a){
      case 1: printf("1");break;
      case 2: printf("2"); break;
      default: printf("none");break;
      }

      return 0;
```

Figure 3.4: Test program for estimating switch instruction

The code shown in Fig. 3.4, when compiled by the LLVM compiler resulted in the LLVM intermediate format shown in Fig. 3.5 (only relevant portions are shown). The same code in ARM assembly is shown in Fig. 3.6 (only relevant portions are shown).

```
switch uint %tmp.5, label %label.2 [
         uint 2, label %label.1
         uint 1, label %label.0 ]
label.0:
    ... %printf( ... )
    ret int 0
label.1:
    ... %printf( ... )
    ret int 0
label.2:
    ... %printf( ... )
    ret int 0
```

Figure 3.5: LLVM intermediate code for the switch statement

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library


```
      cmp    r3, #1
      beq    .L4
      ldr    r3, [fp, #-28]
      cmp    r3, #2
      beq    .L5
      b      .L6
.L4:
      ldr    r0, .L8
      bl     printf
      b      .L3
.L5:
      ldr    r0, .L8+4
      bl     printf
      b      .L3
.L6:
      ldr    r0, .L8+8
      bl     printf
.L3:
      mov    r0, #0
      ldmea  fp, {fp, sp, pc}
```

Figure 3.6: ARM assembly code for the switch statement

As can be seen from the assembly listings, one 'switch' statement in the LLVM format results in, as many 'cmp' instructions as there are 'cases' in the switch statement, twice as many 'branch' statements as there are 'cases' and one more 'branch' if there is a 'default' case. Hence a 'switch' statement in the LLVM format can be estimated to be equivalent to n_a instructions in the compiled code for ARM, where n_a is given by:

$$n_a = 3 * n_c + d \tag{3.1}$$

where n_c is the number of cases in the 'switch' statement and d is 1 if there is a default case and 0 if there is no default case. Similar experiments were performed for the other compound instructions. Other programming constructs were analyzed, which could possibly give rise to different machine codes. The results were used to come up with a framework to estimate the number of instructions that a compiler will produce for each of the compound instruction.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

obtaining the estimates of standard library functions a simpler and a more direct technique was used, which is explained in chapter 4.

Fig. 3.7 gives the flow for estimation of LLVM instructions for the ARM. The instruction is checked first if it is a simple instruction, i.e., those which have direct equivalent in the target architecture, if so the estimate is one, for other instructions the model derived from our experiments are used as estimate.

LLVM has a profiler, which can execute the bytecode file and can give the execution count of each basic blocks in the bytecode. After the execution count of basic blocks is obtained, the method described in Fig. 3.7, is used to obtain the estimates for each basic blocks. The estimates for each basic block along with the profiler outputs is used to calculate the estimates for the execution of the whole application under a given set of inputs.

3.3.2 Estimation of Library Functions

This is fine for user defined functions as the source code is available, which can be compiled to the intermediate format and estimates obtained. But, that would be a very time consuming process as there are many standard library functions. Moreover, it would also require the source code of the whole standard library, which would need to be compiled first and then estimated. Hence for obtaining the estimates of standard library functions, a simpler and more direct technique is used, which is explained later in chapter 4.

Similar analysis was done with lots of different programming constructs, which could possibly give rise to different machine codes. The results were used to come up with a framework to estimate the number of instructions in the target code that a compiler will produce.

While this worked for all different intermediate language instructions, there were problems while estimating the code for standard library functions. For

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

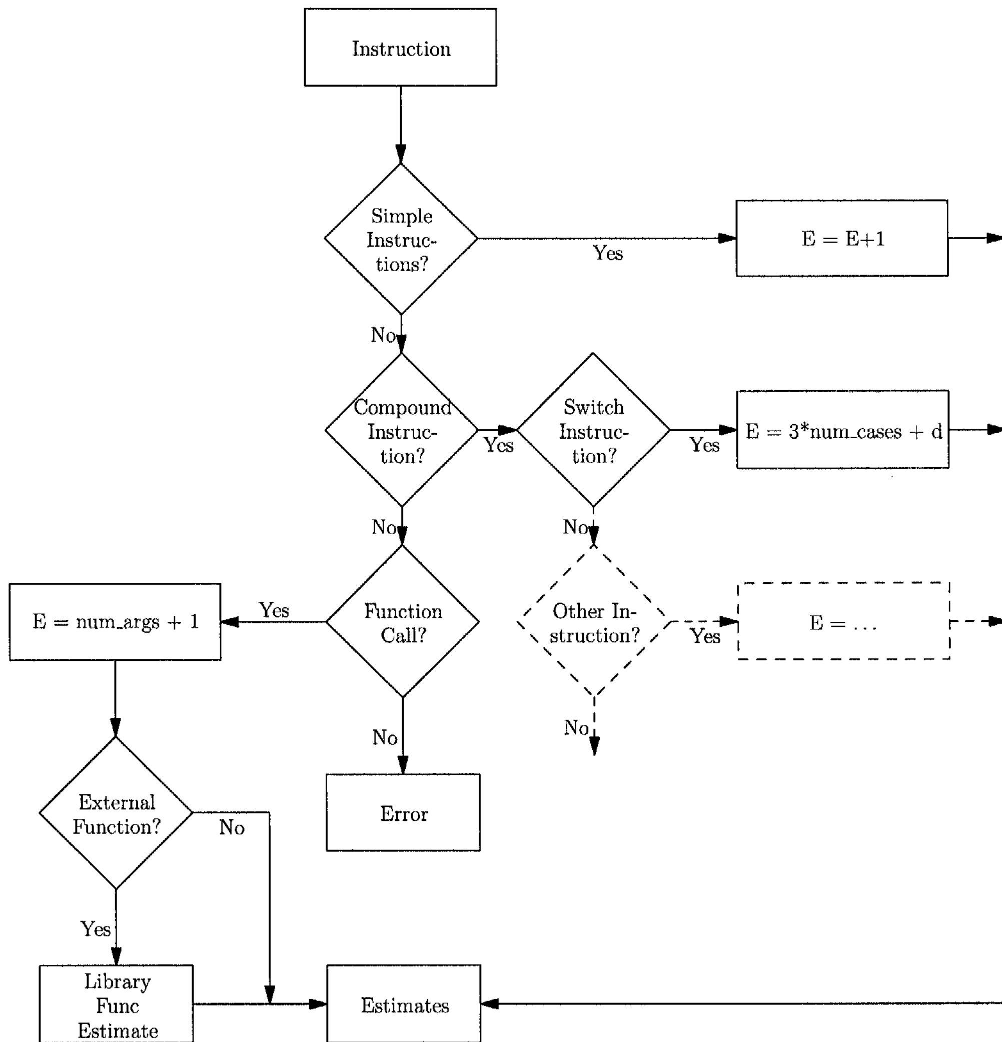

Figure 3.7: Estimation for LLVM instructions

example for a call to the math library function *sqrt*(), the intermediate format just puts in a call to the function. But the performance of a processor should also include the estimates for the standard library functions. For this one way would be to compile all the standard library functions with LLVM and then use

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

similar technique as described above to obtain estimates. The methodology used for estimating the performance for the standard library functions is described in chapter 4.

3.4 Comparison With Current Techniques

The proposed methodology for performance estimation tries to circumvent some of the problems present in the current performance estimation techniques. It is successful in reducing the effects of some of these. The framework loses out in terms of accuracy when compared to compilation-simulation based methods. But this loss of accuracy is not critical when the aim is to short-list a few processors from a large number of processors available in the market. The short-listed processors can be evaluated in more detail using a simulator if more accurate performance estimation statistics are required. Listed below is the comparison of the proposed estimation framework compares with other traditional methods of performance estimation.

- **Conversion:** The framework is able to take as input, applications in any of the high level languages supported by the LLVM compiler infrastructure. The application need not be converted into methodology specific formats like SDL [64]. The application also do not need to be annotated with bounds on the loops [13]. Similarly in tracing driven estimation the application is embedded with instrumentation codes, which is not required in the proposed framework. Moreover, in some of the estimation frameworks, all features of the high level language like unbounded loops and *gotos* are not allowed. Some place restrictions on the *switch* statement. The proposed framework does not place any such restrictions on the given applications.

- **Compiler Tools:** All the compilation-simulation based methods needs a

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

processor dependent compiler and a simulator. So for each of the processor under compilation it needs a compiler-simulator pair. In the proposed framework, the application needs to be compiled to an intermediate format, but only one compiler LLVM is required. LLVM is used to convert the application into a intermediate format and the application information is extracted from the intermediate bytecode file.

- **Compiler Optimizations:** In source based estimation the estimation is done using the source code of the application. Hence, these cannot take into account the compiler optimizations. In the proposed framework, LLVM performs some optimizations on the application before outputting the intermediate bytecode. Hence, the framework is able to benefit from the compiler optimizations performed. This can be substantial if the original program is poorly written. The optimizations performed by LLVM are machine independent. Hence the framework is not able to take into consideration the machine dependent optimizations. In the compilation-simulation based techniques, the application is compiled to the object code of the target architecture and hence machine dependent optimizations are taken into account. As a result the estimates using compilation-simulation based techniques are more accurate.

- **Speed:** The proposed framework is considerably faster than compilation-simulation based techniques. This is because the compilation-simulation based techniques model the micro-architecture like the pipeline, caches etc.

- **Accuracy :** The instruction set simulators model the detailed micro-architecture. The simulators execute object code compiled for the particular processor under evaluation. As a result, the machine dependent optimizations are taken care of. Hence, compilation-simulation based methods are

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

more accurate than the proposed performance estimation framework. In fact in this work the performance estimates are compared with the statistics obtained from an instruction set simulator.

3.5 Experimental Results

Estimation results are provided for the different programs from the MiBench Benchmark suite for the ARM, PISA and the powerpc750 architectures. The estimates for ARM is provided in Table 3.2. The estimates for PISA is provided in Table 3.3 and the estimates for powerpc750 is provided in Table 3.4.

Table 3.2: Estimates for ARM (without stdlib)

benchmark	estimated	simulated	% error
bf_d_large	239189830	1392687761	82.83
bf_d_small	23120252	133968261	82.74
bf_e_large	239189801	1392443854	82.82
bf_e_small	23120223	133944573	82.74
bitcount_large	605815862	962542514	37.06
bitcount_small	40450917	64310652	37.10
crc_large	292723235	-834329382	91.54
crc_small	15057539	178043283	91.54
dijkstra_large	121327214	407673995	70.24
dijkstra_small	24068958	94149199	74.44
fft1_large	30917452	522288020	94.08
fft1_small	2867871	40293453	92.88
fft2_large	31408988	523041666	93.99

Continued on next page

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 3.2 – continued from previous page

benchmark	estimated	simulated	% error
fft2_small	6206162	83587734	92.58
patricia_large	14188324	1063410226	98.67
patricia_small	2288366	172335801	98.67
qsort_large	7675068	412246475	98.14
qsort_small	3346496	10219366	67.25
rawcaudio_large	692115744	839617631	17.57
rawcaudio_small	35600749	43217281	17.62
rawdaudio_large	574360512	639980400	10.25
rawdaudio_small	29143589	32948055	11.55
search_large	1698823	6649941	74.45
search_small	73710	291809	74.74
sha_large	142851552	192679486	25.86
sha_small	13718243	18532610	25.98
susan_c_large	19470574	24210800	19.58
susan_c_small	879577	1694333	48.09
susan_s_large	363669857	479733371	24.19
susan_s_small	23760598	31834686	25.36

As can be seen from the tables 3.2, 3.3 and 3.4 the errors are quite high, going up to as much as 99%. The high error in this case is due to the absence for estimates for the standard library functions. Hence applications which have a higher number of standard library function calls have a high percentage error. For this reason techniques were developed for performance estimation of standard library functions, which is described in chapter 4.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 3.3: Estimates for PISA(without stdlib)

benchmark	estimated	simulated	% error
basicmath_large	37885936	7093875349	99.47
basicmath_small	1517060	175911718	99.14
bf_d_large	228878287	417478732	45.18
bf_d_small	22129645	40279949	45.06
bf_e_large	228878261	420807391	45.61
bf_e_small	22129619	40599487	45.49
dijkstra_large	121238161	282747929	57.12
dijkstra_small	24043356	60810922	60.46
qsort_large	7275064	638660954	98.86
qsort_small	3326492	46470006	92.84
rawcaudio_large	692089130	719154333	3.76
rawcaudio_small	35599377	36716477	3.04
rawdaudio_large	574333898	613888296	6.44
rawdaudio_small	29142217	31409305	7.22
search_large	1694087	5176444	67.27
search_small	73519	216462	66.04
sha_large	142749265	145585470	1.95
sha_small	13708414	13999338	2.08

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 3.4: Estimates for PPC750(without stdlib)

benchmark	simulated	estimated	% error
bitcount_small	53746909	44701325	16.83
bitcount_large	804447921	669379259	16.79
susan_small_s	29550466	25780945	12.76
susan_large_s	450080360	394222223	12.41
dijkstra_small	59571172	28938841	51.42
dijkstra_large	243272102	145879511	40.03
rawcaudio_small	82198837	45575021	44.55
rawdaudio_small	64637240	32224906	50.14
rawcaudio_large	1634938886	885976743	45.80
rawdaudio_large	1276938562	634220785	50.33

3.6 Summary

A high level performance estimation scheme is provided. The estimation process ignores processor details which does not affect the performance much. For example processor details like *endianness* were ignored. While the proposed technique for performance estimation is fast, it comes at the cost of accuracy. Experimental results for performance estimation without the consideration of standard library functions was provided. The results show that without incorporating estimation for standard library functions the errors in estimates can be as high as 99%. Performance estimates of standard library functions is discussed in chapter 4. The main aim is to come with a fast estimation technique which can help us identify a smaller number of suitable processors from a wide range of processors available in the market.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 4

Performance Estimation of Library Functions

4.1 Introduction

In chapter 3, the proposed methodology for performance estimation of processors was described. The experimental results show that, the results are inaccurate if the estimation for standard library functions are not taken care of. In this chapter the methodology for performance estimation of standard library functions is described. For the estimation of standard library functions, they are classified into three categories based on whether their performance is dependent on the function arguments or not. Experimental results are also provided for the estimation for some of the standard library functions. Section 4.2 explains why performance estimation of library functions is difficult and describe why the estimation techniques given in chapter 3 may not be suitable for the performance estimation of standard library functions. The classification of standard library functions is explained in section 4.3. The section also shows how each of them is modeled in the estimation framework. Experimental works for performance

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

estimation of standard library functions are given in section 4.4. Experimental results of performance estimates for a few standard library functions compared with results obtained with an instruction set simulator is presented next. The results are used to get the performance estimates for the different benchmarks in the MiBench benchmark suite.

4.2 Library Functions

Commonly used routines are provided as library functions which can be invoked from an application. Many such related functions are combined into one library. Library function saves a lot of developer's time as all these routines do not need to be hand-coded every time they are needed. Previously tested and proven implementations can be used whenever required. Moreover the developer need not write the same function again and again whenever a functionality is required. This effectively promotes code reuse. This results in better developer productivity as the same code for a particular function can be used again and again after being written once. The application just contains function call when the services of such routines are required. The application source is then compiled and linked with the library files so that those function calls are resolved correctly. Most programming languages come with their own set of libraries which perform commonly used operations, like input/output, common mathematical functions, string manipulations etc.

The methodology described in chapter 3 works well for programs whose source code is provided. The problem arises when the applications contain calls to library functions like $sqrt()$, $printf()$, $fscanf()$, etc. This is because applications do not include code for the library functions, they just have references to the different library functions. The compiler just inserts calls to these functions where

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

required. Hence, in the intermediate format obtained, only the calls to such functions are present. As the source code of the functions are not present, it makes the performance estimation of these functions hard. But the performance of a processor should also include the estimates for the standard library functions. One way to estimate these functions would be to compile all the standard library functions with LLVM and then use similar technique as described above to obtain estimates. The problem is further complicated by the fact that these standard libraries are often distributed as pre-compiled binaries, with no available source codes to help us analyze them. If the source code is available, the method presented in chapter 3 is sufficient to obtain the estimates for standard library functions too. Theoretically, the pre-compiled binaries can be disassembled to obtain their assembly listing, which is the equivalent source code at a lower level. But, this is a very cumbersome process. Moreover this would require a disassembler for every processor which needs to be evaluated as disassemblers are processor specific programs. Hence, the methodology described in chapter 3 is not a feasible way of estimating software performance of standard library functions, and a new framework has to be developed for estimation of library functions.

4.3 Methodology

As explained in the previous section, a new technique has to be developed for the estimation of library functions. In the absence of source codes of these library functions, the only thing that can be done besides disassembling them is to execute them and analyze their performance characteristics. Based on the execution data of these functions, the functions are classified into three categories. The classification is described in section 4.3.1. The estimates for the different

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

function is based on this classification. For some functions the performance is constant irrespective of the argument provided.

For other functions, the estimates are obtained as a formula depending on the size of argument. The exact formula is derived experimentally by executing the functions with different arguments and analyzing the performance. Some functions are executed only a few times when an application runs. Hence, they have only a small effect on the overall performance of the application and thus need not be analyzed.

4.3.1 Classification of library functions

For estimation of library functions, the standard library functions are classified into three categories:

1. *Independent functions*: These are the library functions for which the performance is independent of the argument provided. This also includes functions which do not have any argument.

2. *Dependent functions*: In these functions the performance of the function depends on the argument.

3. *Irrelevant functions*: These are the functions, whose performance does not usually have *much* effect on the performance of the whole application. This category includes' functions like *exit()*, which can execute only once, i.e., when the program terminates.

The first two categories are classified based on whether the argument to the functions affects its performance. The third category, i.e., the irrelevant functions, does not imply that these functions are not important to the application. It just means that their performance does not influence the overall performance of

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

the application much. This classification is subjective in the sense that while performance depends on the input, the variation in performance with the input is small for some of the functions while for others the variation is more.

For the purpose of obtaining software estimates of standard library functions, the GNU C Library [73] was studied, as it is one of the most popular C library and it has been ported to a wide range of architectures. Moreover the benchmarks were also linked with glibc, for execution on the instruction set simulator. While this work describes only the performance of the ANSI C library functions, the methodology is general enough for similar analysis to be done for other libraries.

It was found that functions like $sqrt()$ belongs to the first category, i.e., the runtime of the function $sqrt()$ does not vary too much with the argument. On the other hand functions like $printf()$, $scanf()$ depend very much on the argument. This is evident from the fact that the time taken for the $printf()$ function to run will depend on the number of inputs it is invoked with. In fact even when the number of inputs is constant, the running time also depends on what is to be printed. For example if the application calls the $printf()$ function with an integer argument, the time taken will be smaller if "1" has to be printed, compared to if "1000000" had to be printed.

Functions declared in $< ctype.h >$ like *isalnum(), isalpha()*, etc. also belong to the class of independent functions. This is because these functions carry out a fixed number of comparisons to obtain the result, thus having a constant performance. All functions which do not have any argument like *fflush()*, obviously belong to the first category as there are no arguments on which the performance may depend. As a result functions like *clock(), rand(), getchar()* falls in the first category. For the functions which belong to the first category the performance is independent of the argument, their performance is approximately constant for different arguments. Hence, these can be modeled using a fixed number of in-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

structions being executed.

For the dependent functions, a different technique is used. In the case of *printf*(), for example, the number of instructions executed would obviously be dependent on the number of characters printed, and it was hypothesized that the relationship would be linear. As shown in the experimental results later this is true. So the *printf*() function was modeled using the following formula.

$$N = F + n * c \tag{4.1}$$

In the above formula, F represents the fixed cost of invoking the function, c is the cost of printing a single character, n is the number of characters to be printed by the function call. Using the formula the total cost of the function invocation was arrived at. It will be shown in the experimental results in the next section, that this formula quite accurately models the *printf*() function. Similarly other functions which are expected to show linear relationship, like *scanf*(), *fprintf*(), etc were modeled.

Some functions in the standard library do not vary linearly with the argument size, for example *qsort*(). The number of instructions executed when *qsort*() is called depends both on the size of the argument (here it is the number of arguments passed to the function) and the relative ordering of the argument based on the sorting criteria. It has been shown that the running time of quick-sort is $O(n^2)$ and $\Omega(n \log n)$. Study of the *qsort*() implementation in *glibc*, [73], revealed that the *qsort*() function in *glibc* is not a pure quick-sort implementation. Instead it uses interesting modifications, like using insertion sort once the partition size gets smaller than a particular threshold value. It also uses the median-of-three to chose the pivot value, this reduces the probability of choosing a bad pivot value. This results in a runtime of $O(n)$ for almost all cases [74].

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

There are many standard library functions whose performance does not have much influence on the overall performance of the application which invokes them. Functions like $fopen()$, $fclose()$, are generally executed only once at program start and program end. Similarly functions like $exit(), abort()$ can get executed only once, i.e., when the function exits. So all these functions have negligible impact on the overall program performance, and hence can be safely ignored. It does not matter whether the performance of such functions depends on the input or not. Such programs were classified into the third category.

An instruction set simulator was used to obtain the number of instructions executed for comparison with the estimates. Test programs were written which calls the standard library functions and then the program was executed on the instruction set simulator. The simulator used was simplescalar [43].The compiled program was executed on simplescalar and the number of instructions executed was obtained.

Sim-safe/sim-fast etc. were used which runs very fast compared to the detailed simulator like sim-outorder, but are less accurate than sim-outorder. However, as the estimates obtained from the rest of the code is not as accurate as those produced by sim-outorder, not much accuracy is gained accuracy by using a detailed but a very slow simulator. Test programs were written in such a way that the targeted library function gets called a large number of times. This was done so that the instructions like program startup instructions etc, which cannot be attributed to the library function, gets amortized and a more accurate result is obtained. After the program is executed with the instruction set simulator, the result in terms of the number of instructions executed is obtained. Similar experiments were performed for other standard library functions.

From the analysis of the different library functions, they were classified the three categories defined in the beginning of this section. Table 4.1 shows the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 4.1: Function Classification

Category	Functions
Independent Functions	isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper(), fgetc(), feof(), fputc(), getc(), getchar(), clock(), rand(), srand(), time(), difftime(), putchar()
Dependent Functions	sscanf(), fscanf (), fprintf(), printf(), sprintf(), gets(), scanf(), fread(), fwrite(), malloc(), strchr(), strcmp(), strcpy(), strlen(), atof(), atoi(), atol(), qsort()
Irrelevant Functions	abort(), exit(), open(), close(), fopen(), fclose()

classification of library functions into the three categories.

The estimation for the first category of functions is easy. Test programs can be written to execute these functions a large number of times, so that the effects of other instructions get amortized over the number of instructions. The functions in the third category need not be analyzed as their performance has negligible effect on the performance of the application. For the functions in the third category the program needs to be run with different argument size to obtain the performance characteristics of the functions. This relates the performance of the function to the size of the argument and lets us formulate the performance of the library functions in terms of the size of the argument. After the constant performance of the *independent* functions are obtained and the performance of the *dependent* functions are formulated, the performance of the library functions can be estimated. Figure 4.1 is the flow for estimation.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

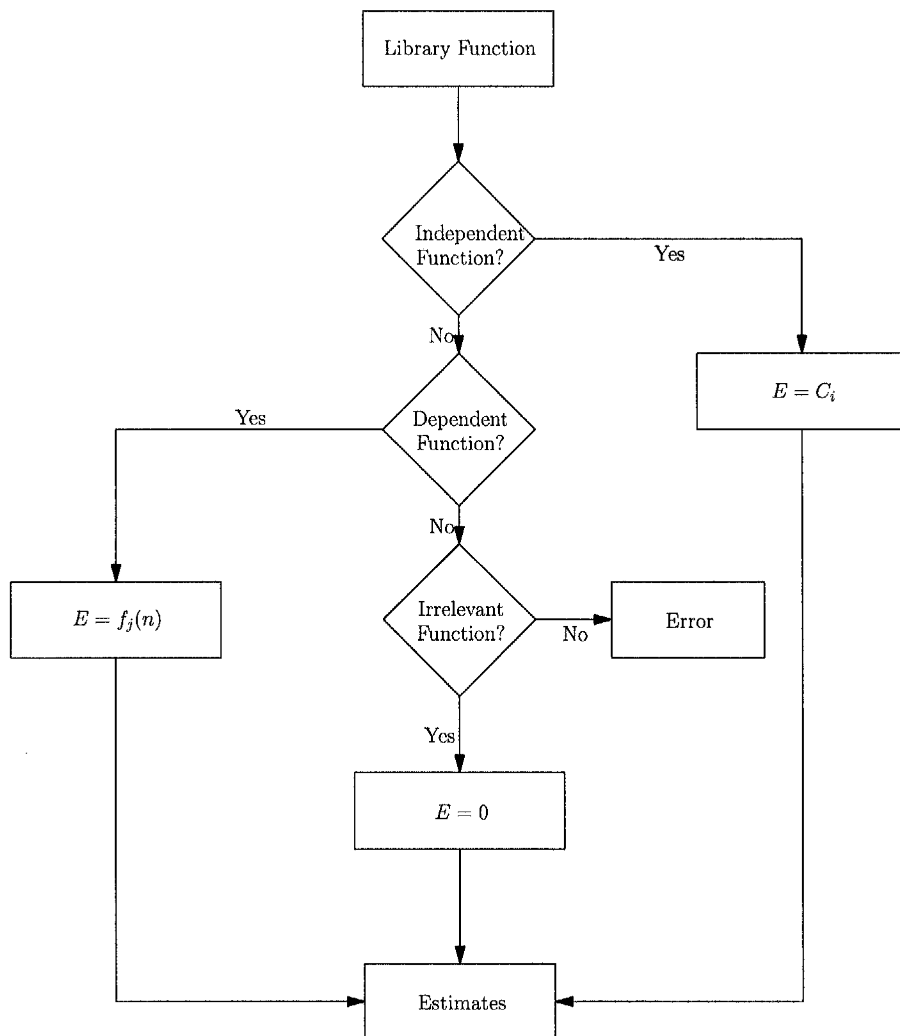

Figure 4.1: Library function estimation overview

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 4.2: putchar results

Input character	# of instructions executed
a	63
b	63
.	.
.	.
y	63
z	63

4.4 Experimental Work

This section describes the experimental work performed to obtain performance of the standard library functions. For estimation of the *independent* functions, it is enough to just run those functions a large number of times to obtain the performance. This is required so that the number of instructions executed during startup and exit of the test program gets averaged over a large number of iterations. When the total instructions executed is divided by the number of iterations, the contribution of program startup and exit is insignificant when the number of iterations is large. For instance a program which invoked *putchar* 1000000 times was written. The program takes the character to be printed as input. The program was executed with different characters as input. More specifically the inputs were the characters $a - z$. The number of instructions executed for each run of the program was divided by the number of times the function was invoked, i.e., 1000000 in this case, to give the number of instructions executed as part of the execution of that function call. The results are shown in Table 4.2

The results show that the number of instructions executed for each run of *putchar* is almost constant or in other words, the performance of the *putchar*

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 4.3: putchar estimates

Input character	# of instructions estimated	# of instructions executed	% error
0	63	63	0.0
1	63	63	0.0
2	63	63	0.0
3	63	63	0.0
4	63	63	0.0
5	63	63	0.0
6	63	63	0.0
7	63	63	0.0
8	63	63	0.0
9	63	63	0.0

function is independent of the input, i.e., the character printed. This validates the earlier classification of *putchar*() in the *independent* functions category.

To verify further, the number of instructions executed was estimated to be 63 (obtained from Table 4.2), and was compared with simulation results for different characters as input (not the ones in Table 4.2). These estimates and simulation results are shown in Table 4.3. The table shows that the estimation error for the *putchar* function is close to 0.

Similarly test programs were written to find the number of instructions executed for the function *sqrt*() for different iterations. The result is shown in Fig 4.2. The number of iterations is plotted on the x-axis with a logarithmic scale as the number of iterations was varied across a large range. As can be seen from the graph, as the number of iterations increase, the average number of instructions executed settle to a stable value. This is because the number of instructions ex-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

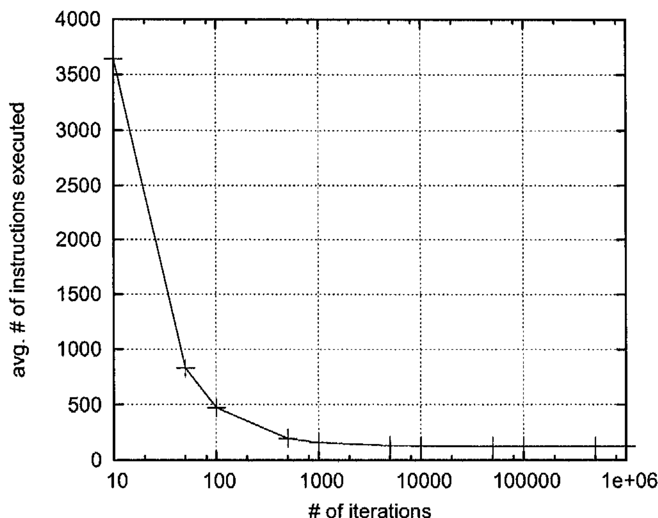

Figure 4.2: Estimates for *sqrt*()

ecuted during the program startup and exit gets averaged over a larger number of iterations. The number of instructions executed per invocation was found to have stabilized at 50000 iterations. The same experiment was repeated with 100 different random values as arguments and the mean was found to be 126 with a standard deviation of 0. This confirms the hypothesis that number of instructions executed in a *sqrt*() call is independent of the argument.

Next the results for the estimation of *printf*() is presented. Small test programs were written to print varied number of characters to the standard output. Assuming linear dependence of performance with the number of characters printed, two of the results were used to obtain the values of F and c in equation 4.1. After these values were obtained they were used to estimate the number of instructions executed. The results are shown in Table 4.4.The number of instructions executed was plotted against the number of characters printed in Fig 4.3. As can be seen the plot is almost a straight line and shows that the hypothesis of

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 4.4: *printf*() estimates

# of chars. printed	# of instructions executed	# of instruction estimated	% error
1	339	342	-0.98
3	358	362	-1.39
5	384	383	0.10
10	438	435	0.64
15	494	486	1.46
20	552	538	2.46
25	594	590	0.67
30	642	641	0.06

linear relationship between the number of instructions executed and the number of characters printed is correct. The Table 4.4 also shows that the percentage error of the estimates are within ± 2.5.

Similar test programs were written to call *qsort*() with varying array sizes to be sorted. Then the performance (as measured by the number of instructions executed) was modeled according to a linear dependence on the argument size i.e., the size of the array to be sorted. The test programs were run on the simplescalar to obtain the number of instructions executed. The resulting data were used to generate the estimates shown in Table 4.5. Also the number of instructions executed was plotted against the number of items sorted to show the linear relationship between them in Fig 4.4.

Experiments similar to those explained in this section were performed for many standard library functions. These results for the different functions are provided in appendix A.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

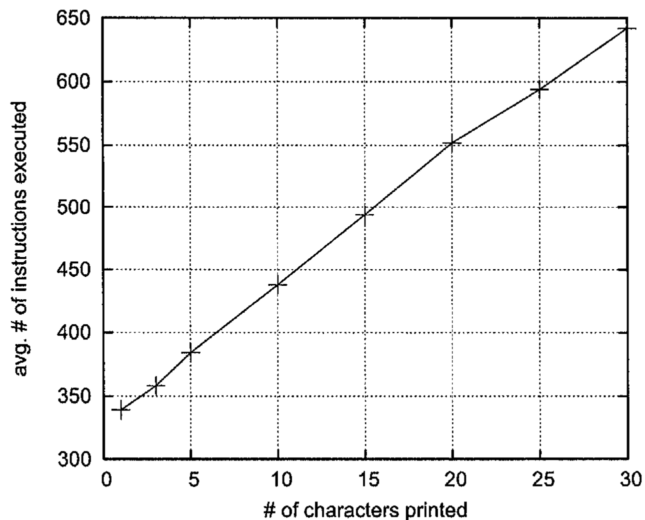

Figure 4.3: Number of instructions vs. number of characters printed for *printf*()

Table 4.5: *qsort*() estimates

# of items sorted	# of instructions executed	# of instruction estimated	% error
100	43080	42000	2.51
200	77121	78000	-1.14
300	112170	114000	-1.63
400	149525	150000	-0.32
500	182133	186000	-2.12
600	222957	222000	0.43
700	263202	258000	1.98
800	301283	294000	2.42
900	337319	330000	2.17
1000	370175	366000	1.13

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

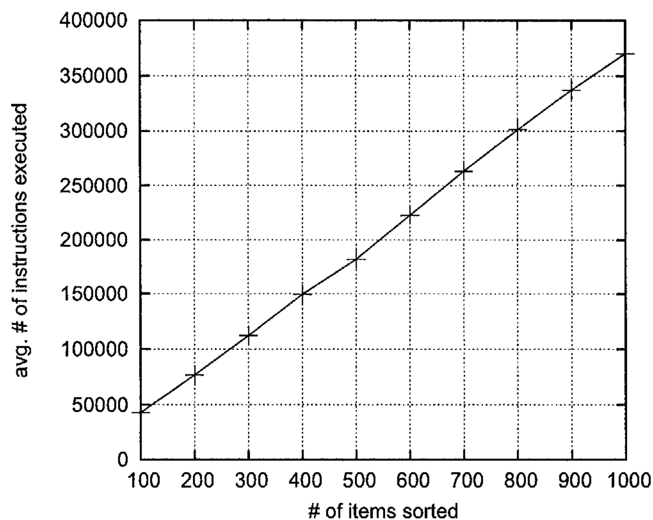

Figure 4.4: Number of instructions vs. number of items sorted

4.5 MiBench Estimation Results

For a methodology or a framework to be useful it has to be validated against a range of applications. The accuracy of the estimates has to be compared with results of other estimation schemes. This chapter presents such experimental works. Evaluation of the proposed methodology with respect to accuracy and also the time taken to arrive at the estimates is presented. A brief description of the tools and the environment used in the experiments is also provided.

4.5.1 Tools and Environment Used

The different tools and environment used in the experimental work is described here. A slightly more detailed description is provided in the appendices B, C and D.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.5.2 LLVM

LLVM is a compiler infrastructure developed at the department of computer science of University of Illinois at Urbana-Champaign. LLVM was used to convert the application to the low level intermediate format, from which application data were extracted. It was also used to run the bytecode files produced to obtain the execution frequency of the different basic blocks.

SimpleScalar

The experimental results should ideally have been compared with actual results obtained from execution of the benchmarks on real processors under study. But a real processor is not ideal for experimentation. It is not easy to vary system variables like cache size [75]. Because of this difficulty the SimpleScalar instruction set simulator was used for the experiments.

The SimpleScalar tool set is a system software infrastructure used to build modeling applications for program performance analysis, detailed micro-architectural modeling, and hardware-software co-verification [76]. SimpleScalar was used to measure the accuracy of the estimation framework. The benchmark used for the experiments were executed on SimpleScalar to obtain the performance figures. This were used as the actual performance figures when comparing with the estimates.

MiBench

MiBench is a free, commercially representative embedded benchmark suite [70]. The benchmark suite includes a set of typical embedded application. The framework was tested on the MiBench benchmark suite.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.5.3 Experimental Procedure

The estimation framework developed was used to obtain the performance estimates of the MiBench benchmark suites for three instruction set architectures, i) ARM , ii)PISA and iii) PPC750. First the benchmark applications were converted into the LLVM format using the LLVM compiler. The intermediate language format instructions for each basic blocks was extracted from the format using a LLVM compiler pass developed for the purpose. The extracted data were stored in xml format for ease of use later during the estimation process. The intermediate bytecode was then executed on the LLVM virtual machine and profiling statistics was obtained and stored again in xml files. The intermediate instructions and the profiling statistics was later used to obtain the performance estimates.

4.5.4 Estimation Results

Tables 4.6,4.7 and 4.8 show the experimental results. The estimates are compared with the SimpleScalar and PPC750 simulator simulation results.

Table 4.6: Estimates for ARM (no. of instructions)

benchmark	simulated	estimated	% error
basicmath_small	34337252	33807827	1.54
basicmath_large	153968114	147769221	4.02
bitcount_small	49671054	47736543	3.89
bitcount_large	743681421	714951488	3.86
qsort_small	43608758	39828189	8.67

Continued on next page

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 4.6 – continued from previous page

benchmark	simulated	estimated	% error
qsort_large	737923418	648426723	12.13
crc_small	73981239	69812195	5.63
crc_large	1437936500	1357171331	5.61
susan_small	68075222	59736962	12.25
susan_large	1130126888	993681995	12.07
dijkstra_small	64930886	68714178	5.82
dijkstra_large	272660486	264889316	2.85
blowfish_small_e	52415857	53195574	1.49
blowfish_small_d	52408177	53195707	1.50
blowfish_large_e	544060765	552636294	1.58
blowfish_large_d	543979692	552636427	1.59
sha_small	13544323	13195094	2.58
sha_large	140892866	137382880	2.49
stringsearch_small	161498	140741	12.85
stringsearch_large	3682705	3428762	6.90
rawcaudio_small	37695304	42476088	12.68
rawdaudio_small	30162442	28819377	4.45
rawcaudio_large	732520508	825729166	12.72
rawdaudio_large	586079072	568012320	3.08
patricia_small	103926712	92613136	10.89
patricia_large	640423162	539617650	15.74

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 4.7: Estimates for PISA (no. of instructions)

benchmark	simulated	estimated	% error
bitcounts_small	43607079	44709924	2.53
bitcounts_large	659000010	669387858	1.58
susan_small	28655709	29371773	2.50
susan_large	466581542	494538457	5.99
stringsearch_small	186057	151779	18.42
stringsearch_large	4449396	3691922	17.02
blowfish_small_e	37373591	43725435	17.00
blowfish_small_d	37054038	43725531	18.00
blowfish_large_e	387245044	453773239	17.18
blowfish_large_d	383916370	453773335	18.20
sha_small	13200925	14661675	11.07
sha_large	137290035	152652647	11.19
qsort_small	41868586	31899122	23.81
qsort_large	568948084	637629717	12.07
dijkstra_small	54879500	48900534	12.23
dijkstra_large	255787178	204229593	20.16
patricia_small	134125678	109499955	18.36
patricia_large	826922998	646257876	21.85

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 4.8: Estimates for PPC750 (no. of instructions)

benchmark	simulated	estimated	% error
bitcount_small	53746909	47701485	11.25
bitcount_large	804447921	714379419	11.20
susan_small_s	29550466	36243553	22.65
susan_large_s	450080360	554379063	23.17
dijkstra_small	59571172	48926857	17.87
dijkstra_large	243272102	246511724	1.33
rawcaudio_small	82198837	75704420	7.90
rawdaudio_small	64637240	59906347	7.31
rawcaudio_large	1634938886	1471702575	9.98
rawdaudio_large	1276938562	1173420487	8.10

Errors in performance estimation can be due to a number of reasons. Sophisticated compiler optimisations, which the methods described in chapter 3 can not account for, can lead to the estimates being higher than the actual number of instructions executed. Secondly, the estimation was done using the LLVM intermediate code. Some processors may have some instructions which may not have a corresponding LLVM equivalent and moreover it can be possible that one target instruction is implemented by a combination of LLVM instructions. For example, ARM has multiply accumulate instruction which does not have a single LLVM instruction and must be implemented by a combination of LLVM instructions. This special instructions can not be modeled using our estimation technique and produces errors.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

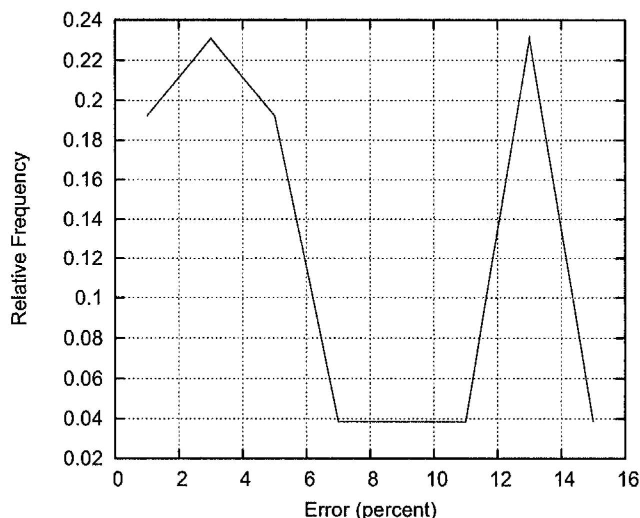

Figure 4.5: Distribution of Errors for ARM

As there are multiple causes of error and they may be additively contributing to the error, it may be interesting to look at the error distribution. The error distribution is shown in figures 4.5, 4.6 and 4.6. For ARM, it is observed that the error distribution has two modes at 5% and 13%. The error distribution of PISA and PPC750 estimates have modes at 17.5% and 7.5%.

4.6 Summary

Estimates of standard library functions cannot be easily obtained using techniques described in chapter 3. As libraries are sometimes provided as pre-compiled binaries, the source code is not available. This creates additional problems. Therefore an alternative strategy was developed which does not require the availability of the source code for the library functions. The different standard library functions were classified depending on whether they depend on the input or not. The func-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

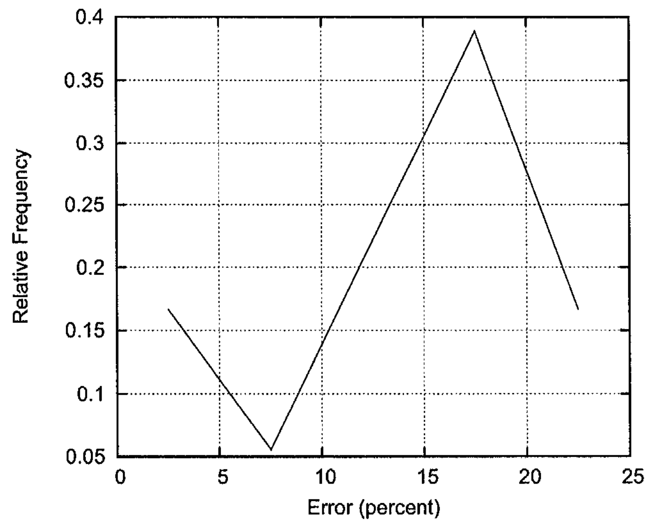

Figure 4.6: Distribution of Errors for PISA

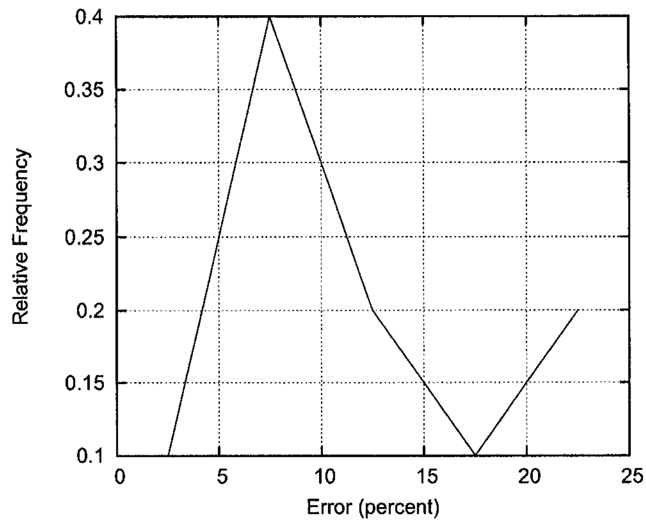

Figure 4.7: Distribution of Errors for PPC750

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

tions which do not depend on input are estimated to have a fixed performance for any input. The other function's estimates are obtained by analyzing the variation of performance with the input.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 5

Estimation of IPC (Instructions Per Cycle)

5.1 Introduction

The performance estimates in the previous chapters were in terms of number of instructions executed. In this chapter it is shown that the number of instructions executed is not a sufficient parameter to judge the performance of a processor. It is also important to estimate the IPC (instructions per cycle), to evaluate the performance of the different processors. This chapter describes the methodology for the estimation of IPC. The methodology is based on the calculation of the upper and lower limits of IPC for different basic block lengths. These bounds are used to arrive at the IPC estimates after obtaining the average basic block lengths. IPC estimation results are shown for ARM and PISA.

Section 5.2 describes the estimation methodology and explains the reason why only the number of instructions executed is not enough for performance estimation. The section also explains why IPC can be less than 1 for RISC processors. In section 5.3, the experimental work done for the estimation of IPC

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

is presented.

5.2 Estimation Methodology

IPC is defined as the number of instructions that the CPU executes per clock cycle.

$$IPC = \frac{1}{CPI} \tag{5.1}$$

where CPI is the number of cycles per instruction. IPC is a term to describe processing speed. The higher the rating, the better the performance.

In RISC (Reduced Instruction Set Computer) processors, IPC can at most be 1. It can be higher in the advanced CPUs like the *superscalar* and *VLIW* (Very Long Instruction Word) processors. As mentioned earlier, the numbers of instructions executed were estimated. While it can give rough indication of how much time a processor will take in executing a given application, the indications may not be always accurate. This can be seen from the following example.

Assume two processors, P_A and P_B, take same number of instructions N to run an application A. Looking at just the number of instructions, it would seem that both the processors would have similar performance. But if processor P_A has an IPC of I_A and processor P_B has an IPC of I_B, the number of cycles C_A and C_B taken for both the processors to execute the application would be

$$C_A = \frac{N}{I_A}, \quad C_B = \frac{N}{I_B} \tag{5.2}$$

These two cycle counts can differ by a huge margin if the difference between I_A and I_B is large. If I_A is twice I_B, C_B will be twice C_A and hence, the two cycle counts can differ by 100%. Therefore, the number of instructions by itself

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

may not provide the full picture and the IPC is required to obtain the total cycle count of an application running on a processor.

Ideally, the IPC should be 1 for a RISC processor, as there should be one instruction getting executed every clock cycle due to the pipelining. But in actual execution the IPC is less than 1. This is mainly due to the following reasons which lead to pipeline stalls

- *Cache misses*: During a cache miss, the pipeline has to be stalled until the missed data arrive.
- *Structural Hazards*: These arise when the instructions in the pipeline attempt to access the same resource, thus leading to conflicts. [77] lists the common reasons for structural hazards :
  - *Some functional unit is not fully pipelined.* Then a sequence of instructions using that un-pipelined unit cannot proceed at the rate of one per clock cycle
  - *Some resource has not been duplicated enough* to allow all combinations of instructions in the pipeline to execute.
- *Control Hazards*: Control hazards arise whenever any control instruction is executed and the wrong branch taken resulting in the flushing of the pipeline.
- *Data Hazards*: Data hazards arise when an instruction depends on the results of a previous instruction in the pipeline and is not available yet.

In this work a large enough cache size were assumed, so that cache size ceases to be a limiting factor. The details of arriving at the cache size is described in section 5.3.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

The other factors which affect the IPC are control hazards and data hazards. In a pipelined execution, the pipeline would need to be flushed at the control breaks whenever the control jumps from the normal sequence of execution. Hence, applications with longer basic blocks are expected to have higher IPC, as the endpoints of basic blocks are the points at which control breaks occur. To incorporate the data dependency factors, in the estimation process, test programs of different average basic block lengths were written. One set was written in such a way as to minimize any chance of data hazards. Another set of test programs were written which are expected to result in pipeline stalls.

These programs were simulated on the SimpleScalar. The first set of programs has a higher IPC since it was written in a way which minimizes the chances of data hazard. This value of IPC will be the upper limit of IPC for the given average basic block length and the cache size. Similarly, the second set of programs will have a lower value of IPC. As the second set of program were written to create maximum data dependency, their IPC values will be the lower limit of the IPC. Later it will be shown that this lower limit is not accurate. The test programs were written in high level language and the compiler was able to optimize the data dependencies. Hence, a few of the observed IPCs are less than the lower limit. The upper limit and lower limit of the IPCs for different basic block length was plotted. The average of the two limits was used as the estimate for the IPC.

The basic block length used was obtained by taking the weighted average of all the basic blocks in the program. The weights used was the number of time the particular basic block is executed. Eq. 5.3 gives the formula how average basic block length was computed, where bb_len_{avg} is the average basic block length, B is the number of basic blocks in the application, n_i is the number of times basic block i is executed and bb_len_{avg} is the length of the basic block i.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$bb_len_{avg} = \frac{1}{B} \sum n_i \cdot bb_len_i \qquad (5.3)$$

5.3 Estimation Results

First of all the cache size to be used for the simulations needs to be calculated. The benchmarks from the MiBench Benchmark suite were simulated for different cache sizes ranging from 1K to 128K. Direct-mapped cache was used for the experiments. LRU (Least recently used) block replacement policy was used. The block size used was 32 bytes. Fig. 5.1 shows the variation of IPC with cache size for ARM. Fig. 5.2, shows the same plot for the PISA architecture.

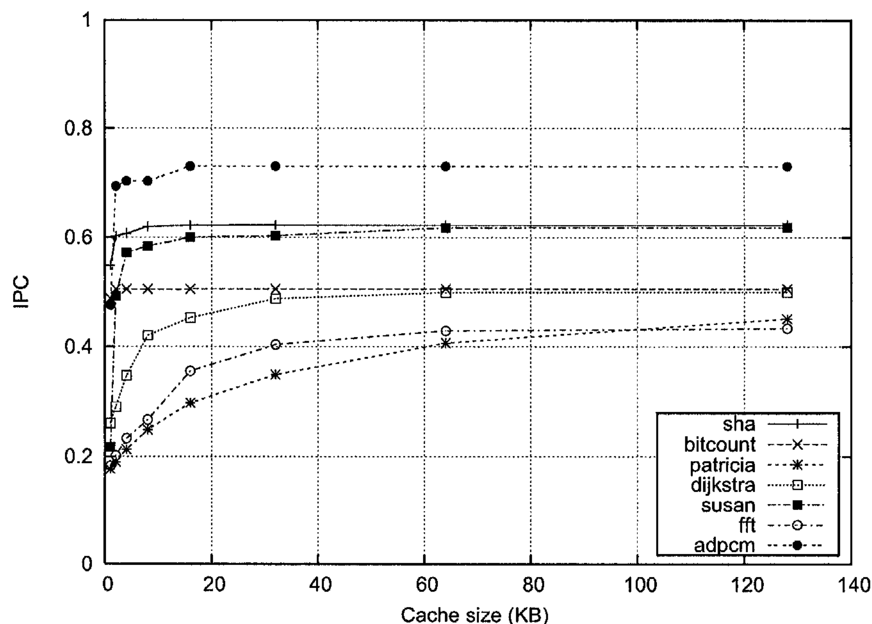

Figure 5.1: IPC vs cache size for ARM

As expected, IPC increases initially as cache size increases. But beyond a certain size the plot flattens out, there is no or very little improvement in IPC

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

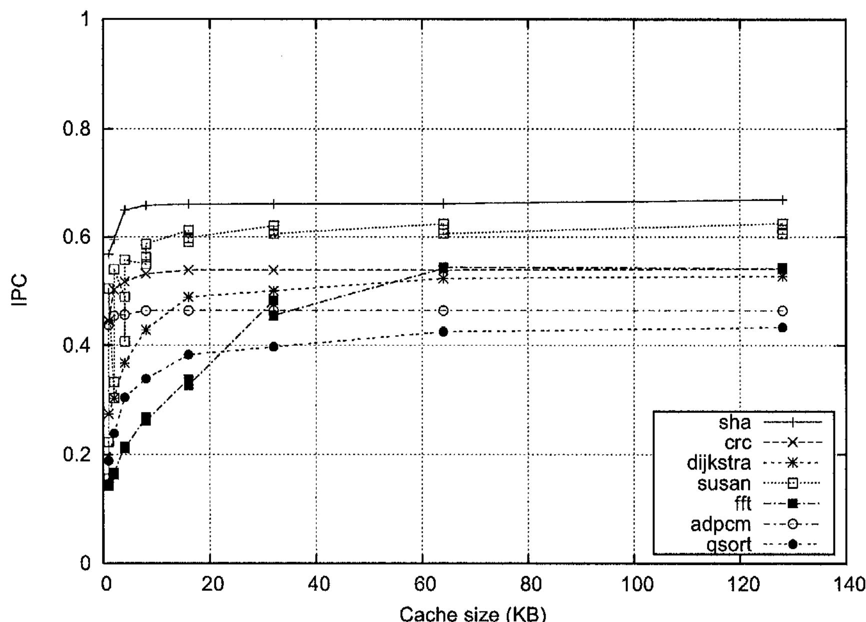

Figure 5.2: IPC vs cache size for PISA

if the cache size is increased. This can be explained by the fact that, increasing the cache size beyond a certain value does not lead to much increase in the cache hit rate. The *principle of locality* states that programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code [78]. Hence having a cache size of more than 10% of the code size should not lead to a much increase in hit rate. Thus, increasing the cache size beyond a certain value does not lead to much increase in IPC. From Figures 5.1 and 5.2, it can be seen that, there is not much increase in IPC, for cache sizes above 32 KB. The miss rates for different caches sizes were also plotted for the different benchmark programs. Fig. 5.3, shows the miss rates for ARM. The miss rate is the ratio of number of cache misses and the total number of references. The same plot for PISA is shown in Fig. 5.4. It can be seen that the cache miss rates are quite low for

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

cache sizes larger than 32KB. Hence, for both ARM and the PISA architecture, beyond 32KB the cache size ceases to be a limiting factor for the IPC.

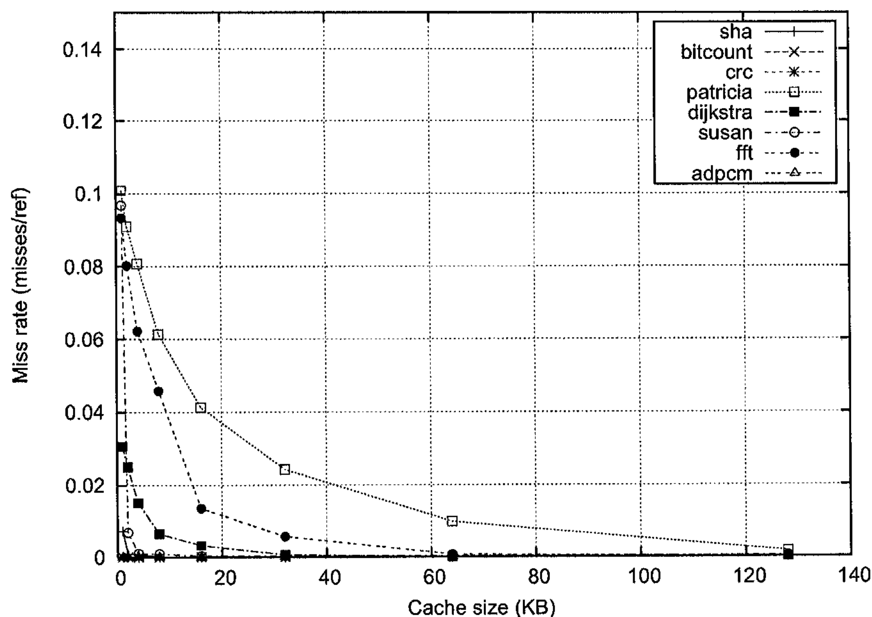

Figure 5.3: Cache miss rate vs cache size for ARM

Test programs were written for average basic block lengths varying from 4 to 25. One set of programs were written with no data dependency between the different instructions. The IPC of these programs was plotted against the average basic block length. The curve gives the upper limit of IPC for a given average basic block length. Similarly, the set of programs with data dependency gives the lower limit. Fig. 5.5 shows the upper limit and the lower limit of IPC for ARM, and Fig. 5.6 shows the same for PISA architecture. In these graphs, actual IPC for the different benchmark programs of MiBench were plotted against their basic block length. To obtain the average basic block length, the application was compiled using the LLVM compiler [69]. LLVM has a profiler, which was used to execute the compiled bytecode file and the execution count of each basic blocks

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

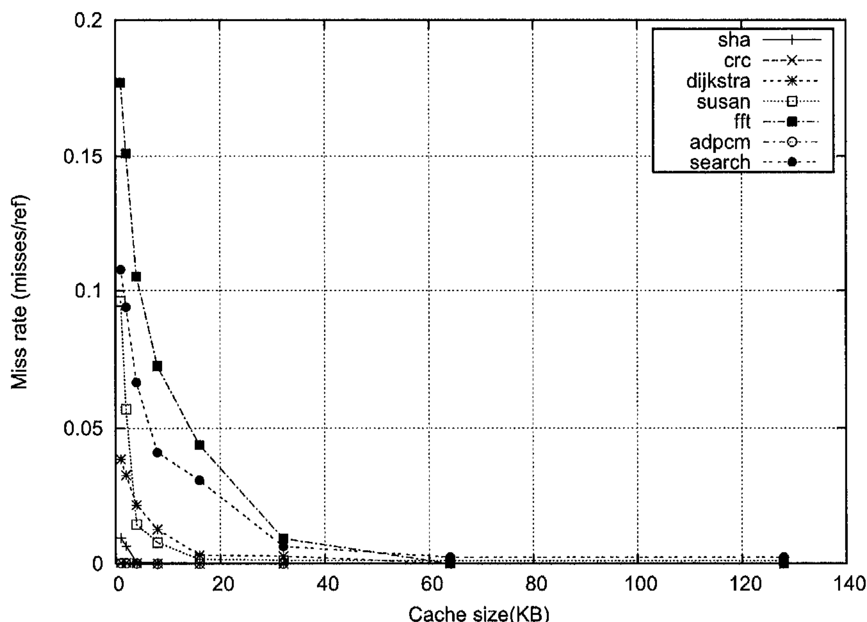

Figure 5.4: Cache miss rate vs cache size for PISA

in the bytecode was obtained from the profiler output. The average basic block length was calculated from the execution count of each basic black using Eq. 5.3.

From the Figures 5.5 and 5.6, it can be seen that some of the actual IPC values do not lie within the upper limit and the lower limit. This is because the test programs were written in C, i.e., a high level language and the compiler can optimize away some of the data dependency inserted into the test programs. So the actual lower limit may be slightly less than the one shown by the plot. This inaccuracy can be reduced if the test programs are written in the assembly language.

These limits were used to obtain the IPC estimates for the different average basic block lengths. The estimates used are the midpoint of the upper limit and the lower limit. Figures 5.7 and 5.8 show the IPC estimates and also the actual IPC for the MiBench Benchmark programs for ARM and PISA architectures,

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

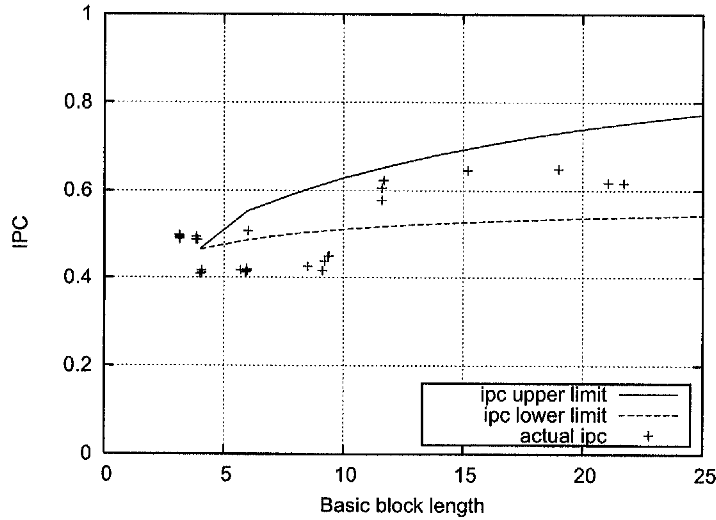

Figure 5.5: Upper and lower limits of IPC for ARM

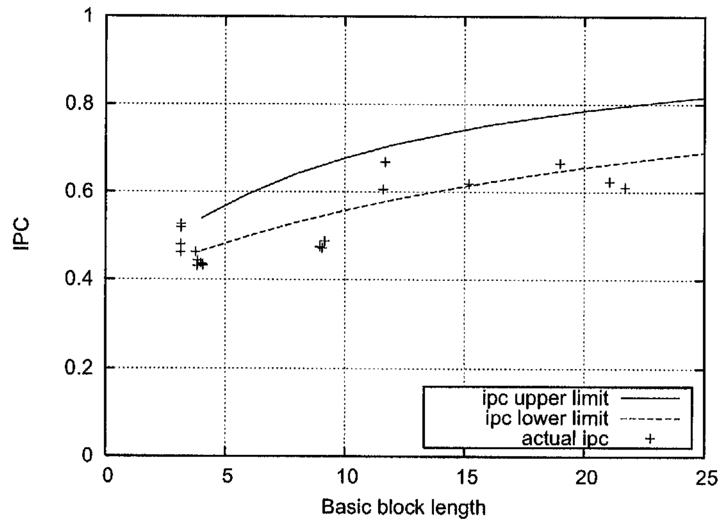

Figure 5.6: Upper and lower limits of IPC for PISA

respectively.

The estimates were also compared with the actual IPC values after running the benchmark programs on the SimpleScalar simulator. The percentage errors

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

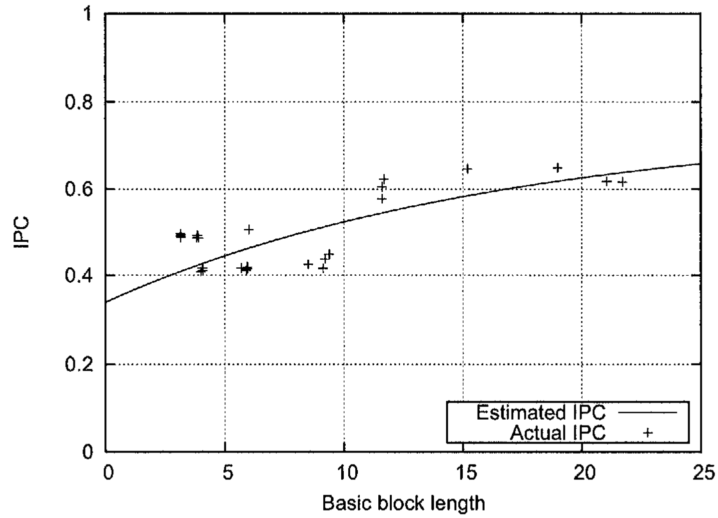

Figure 5.7: IPC Estimates for ARM

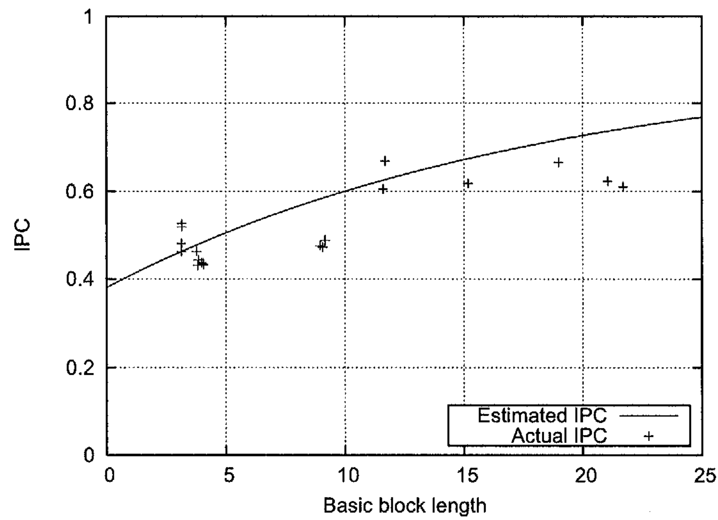

Figure 5.8: IPC Estimates for PISA

of the estimates are given in table 5.1 and 5.2. For ARM, the maximum error is −22% and for PISA the maximum error is −23%.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 5.1: IPC Estimates for ARM

Benchmark	Basic Block length	IPC simulated	IPC Estimated	error(%)
rawcaudio_large	3.13	0.4972	0.4089	17.77
rawcaudio_small	3.13	0.4917	0.4089	16.85
dijkstra_large	3.15	0.4944	0.4093	17.22
dijkstra_small	3.15	0.4884	0.4093	16.20
rawdaudio_small	3.82	0.4872	0.4227	13.24
rawdaudio_large	3.84	0.4933	0.4231	14.23
qsort_large	3.89	0.4866	0.4241	12.85
qsort_small	4.00	0.4090	0.4262	4.21
search_large	4.06	0.4174	0.4274	2.39
search_small	4.07	0.4112	0.4276	3.98
basicmath_large	5.69	0.4174	0.4574	9.59
patricia_small	5.91	0.4120	0.4613	11.96
basicmath_small	5.95	0.4193	0.4620	10.18
patricia_large	5.96	0.4153	0.4621	11.28
bitcount_large	6.01	0.5063	0.4630	8.55
bitcount_small	6.01	0.5061	0.4630	8.52
fft_small1	9.11	0.4164	0.5117	22.89
fft_small2	9.21	0.4376	0.5131	17.26
fft_large2	9.36	0.4491	0.5153	14.73
fft_large1	9.38	0.4494	0.5155	14.72
susan_large_s	11.60	0.6057	0.5448	10.06
susan_small_s	11.60	0.5777	0.5448	5.70

Continued on next page

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 5.1 – continued from previous page

Benchmark	Basic Block length	IPC simulated	IPC Estimated	error(%)
sha_large	11.68	0.6234	0.5458	12.45
sha_small	11.68	0.6227	0.5458	12.36
susan_large_e	15.19	0.6463	0.5845	9.55
susan_large_c	18.98	0.6493	0.6182	4.79
susan_small_c	21.05	0.6179	0.6336	2.54
susan_small_e	21.70	0.6166	0.6380	3.48

Table 5.2: IPC Estimates for PISA

Benchmark	Basic Block length	IPC simulated	IPC Estimated	error(%)
dijkstra_large	3.1	0.5269	0.4633	12.06
dijkstra_small	3.1	0.5189	0.4633	10.71
fft_large1	9.2	0.4888	0.5864	19.97
fft_large2	9.2	0.4886	0.5861	19.95
fft_small1	8.9	0.4760	0.5826	22.40
fft_small2	9.1	0.4732	0.5845	23.52
qsort_large	3.8	0.4631	0.4780	3.21
qsort_small	4.0	0.4374	0.4831	10.44
rawcaudio_large	3.1	0.4811	0.4629	3.79
rawcaudio_small	3.1	0.4628	0.4629	0.01
rawdaudio_large	3.8	0.4440	0.4796	8.02

Continued on next page

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 5.2 – continued from previous page

Benchmark	Basic Block length	IPC simulated	IPC Estimated	error(%)
rawdaudio_small	3.8	0.4316	0.4791	11.02
search_large	4.1	0.4354	0.4847	11.31
search_small	4.1	0.4319	0.4847	12.22
sha_large	11.7	0.6698	0.6269	6.41
sha_small	11.7	0.6688	0.6269	6.27
susan_large_c	19.0	0.6668	0.7172	7.56
susan_large_e	15.2	0.6187	0.6747	9.05
susan_large_s	11.6	0.6070	0.6257	3.07
susan_small_c	21.0	0.6243	0.7368	18.01
susan_small_e	21.7	0.6113	0.7425	21.46
susan_small_s	11.6	0.6060	0.6257	3.24

As mentioned earlier, one reason for the errors was that a high level language was used for writing the test programs while obtaining the upper and lower limits of the IPC. Second reason for the errors in the estimation is that the basic block length was counted from the intermediate format code. The actual basic block length might be slightly different in the machine level code. The effect of the first reason can be reduced by hand coding the test programs used to calculate the upper and lower limits on the IPC, in assembly. For the second case on the other hand, it is not possible to use assembly level code to obtain the basic block lengths, as it would go against the aim of not using compilation for the estimation process.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

5.4 Summary

Performance estimate in terms of the number of instructions executed may not be sufficient for judging a processor's performance. For a realistic comparison, the IPC is also an important metric to be taken into consideration. Estimation of IPC is difficult as the IPC depends on data dependency of the program and the cache size. A technique to estimate the IPC of an application running on a processor was presented. The average basic block length is used to arrive at rough upper and lower limits for the IPCs. The estimations results are within $0 - 21\%$ of actual values as measured by an instruction set simulator. The mean error is 11.07%.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 6

Framework for Processor Selection and HW-SW Partitioning

6.1 Introduction

Previous chapters described the proposed techniques for performance estimation of processors which do not require the compilation of the application for each of the processor being evaluated. In this chapter, a framework for processor selection is presented which is based on estimating the performance of processors under consideration. The processor selection framework helps in identifying a suitable processor for a given application. A hardware-software partitioning framework is also proposed which utilizes the processor performance estimates. The processor selection framework and the hardware-software partitioning framework was combined to form an embedded system design framework. The embedded system design framework is based on selection of a processor using the proposed performance estimation technique and then tightening or relaxing the constraints to

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

obtain a near optimal design.

The outline of this chapter is as follows: section 6.2 explains the processor selection framework. And in section 6.3 the hardware-software partitioning framework and experimental results of the partitioning algorithm based on the knapsack problem is provided. Finally a constraints-aware embedded system design framework is provided in section 6.4.

6.2 Processor Selection

As explained in section 3.1, it is important to select a proper processor. This ensures that the processor is utilized as much as possible. It is not preferable to select an over-performing processor as they would be more expensive and a decrease in cost by a even few cents can lead to large cost savings due to the high volume involved in embedded systems. Hence it is better to select a processor which is performing at the maximum of its potential.

Towards this end it is proposed that a processor which just falls short of the performance requirements be selected. The reasoning is better explained by Fig 6.1. Performance deficiency, $\mathcal{P}.\mathcal{D}$, in the figure is defined as

$$\mathcal{P}.\mathcal{D} = \mathcal{D}.\mathcal{P} - \mathcal{A}.\mathcal{P} \tag{6.1}$$

where $\mathcal{D}.\mathcal{P}$ is the desired performance and $\mathcal{A}.\mathcal{P}$ is the actual performance.

In the figure, processor P_1 and P_2 have a performance below the desired performance. Processor P_3 has a performance higher than the desired performance. If processor P_3 is selected the desired performance is achieved but the processor might be expensive and hence can lead to the unit cost being prohibitive. If processor P_1 or P_2 is selected, the desired performance has to be achieved using custom hardware. But since the performance deficiency of P_1 is higher the size

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

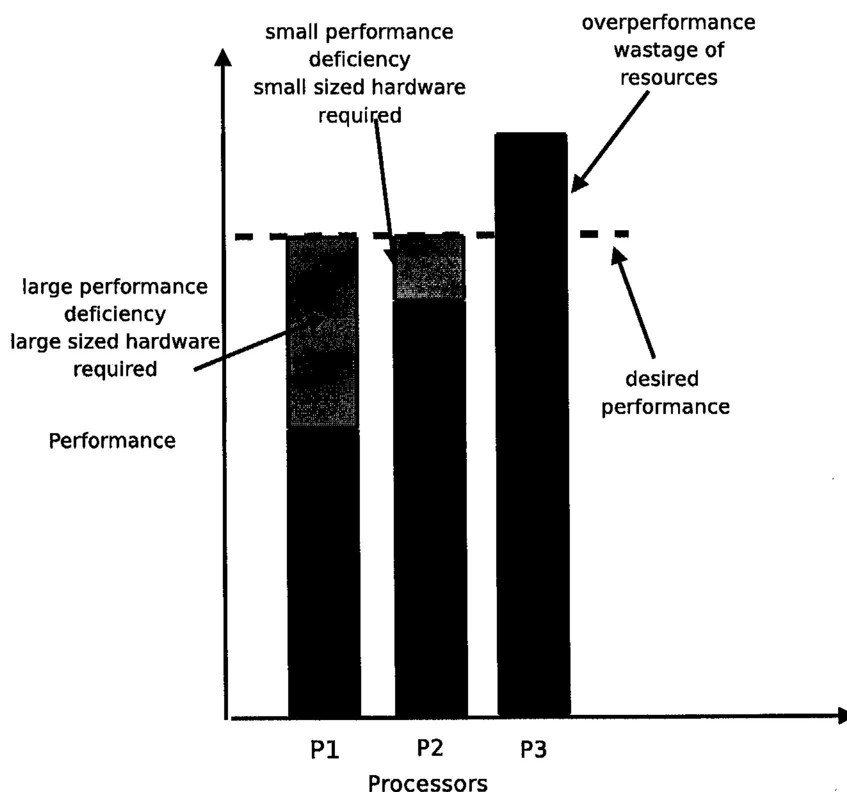

Figure 6.1: Processor Selection

of the custom hardware required to bring the performance to the desired level is bigger and thus this may also result in higher unit cost. Hence it is propose to choose processor P_2 which just falls short of the desired performance level.

It is desirable to use as small custom hardware as possible for reasons mentioned below. Moreover an over-performing processor is better avoided due to its high cost. These two requirements can only be satisfied by using a processor which has the least performance deficiency

Also it would be preferable to keep as much of the application running in software. In addition to the added flexibility, following are the reasons for doing

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

so [79]

- **Availability of Optimizing Compilers** : Optimising compilers are available for most processors in the market. These produce very efficient and optimized code. The code produced is better than the code produced by retargetable compilers as the latter are designed to generate code for a range of processors. Since optimizing compilers are available, it is better to keep as much of the application running on the processor.

- **Careful Verification and Field Testing of Standard Cores** : The processors undergo a very careful verification process. A major fraction of the cost (60-70%) is in fact spent on verification and validation at various stages of the design process [80]. This is also another reason for using a commercially off the shelf processor as there is no need to bother with verifying a new one. As a result the verification time is reduced. This in effect helps cut down the time to market.

- **Increase in the Average Code Size**: The average code size has increased from 16-64KB in 1992 to 64K-512KB in 1996 [81]. As the code size increases, it is easier to implement in software as it requires a minimal change compared to if it is implemented in hardware.

- **Upgradability** : As the system matures, it might need to be upgraded or sometimes the application needs to be modified for correcting errors. And it is easier to modify software than to make changes in the hardware [82], hence, it is desirable to keep the hardware parts to minimum.

- **Flexibility of Software** : Software is easier to develop than hardware and some basic programming constructs like recursion are usually not easy to implement in hardware. Design failure in hardware requires new chip design

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[83]. With small variations in the software and using the same hardware, it is possible to release many similar product targeted at different market segments [10].

- **Ease of Debugging**: Debugging software is made easier due to the availability of debuggers for the different processors. Also GDB (the GNU debugger) has been ported to many processors. Currently it supports the ARM, Renesas H8/300, Motorola M68k, OpenRISC 1000, Fujitsu Sparclite, PowerPC and many more. Debugging software is also far easier than debugging hardware. While debugging software it is possible to put in diagnostics to inspect the execution [84]. This may not be possible in hardware.

- **Minimise Compiler Effects**: When part of the application is moved to hardware, the compiler optimisations are sometimes invalidated. For example, if a part of a function is moved to hardware, the assumptions about the number of accesses to a variable or even its live range may change if a variable is moved to a hardware. Hence, the original register allocation done by the compiler may not be the optimum.

- **Minimize Communication Overheads**: As more and more portions of the application is moved to hardware, the communication between the hardware and software will increase. This may increase the latency to unacceptable level and may not negate any improvement to performance by moving application parts to hardware.

- **Hardware Cost**: Usually application specific hardware is much faster than software and also more power efficient, but expensive at the some time. Software, on the other hand, is cheaper, but slow and consumes much power when implemented on a general purpose processor [85]. Hence to keep costs low only the minimum required hardware to achieve the desired performance

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

is used. Hence the drive to keep as much as possible running in the software i.e., on the microprocessor.

Keeping the above in mind a processor selection framework was devised which is shown in Fig. 6.2. It begins with calculating the performance deficiency for all the processors under consideration and then selecting the processor with the least performance deficiency. Processor with negative performance deficiencies are rejected as those are the over-performing processors and hence likely to be expensive. The processor with the smallest performance deficiency is selected as that would require a smaller hardware area to meet the required performance.

6.3 HW-SW Partitioning Framework

Efficient hardware/software partitioning is crucial towards realizing optimal solutions for constraint driven embedded systems. The size of the total solution space is typically quite large for this problem. This section shows that the Knapsack model could be employed for the rapid identification of hardware components that provide for time efficient implementations. In particular, a method to split the problem into standard 0-1 knapsack problems is proposed in order to leverage on the classical approaches. The proposed method relies on the tight lower and upper bounds for each of these knapsack problems for the rapid elimination of the sub-problems, which are guaranteed not to give optimal results. Experimental results show that, for problem sizes ranging from 30 to 3000, the solution of the whole problem can be obtained by solving only 1 sub-problem except for one case where it required the solution of 3 sub-problems.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

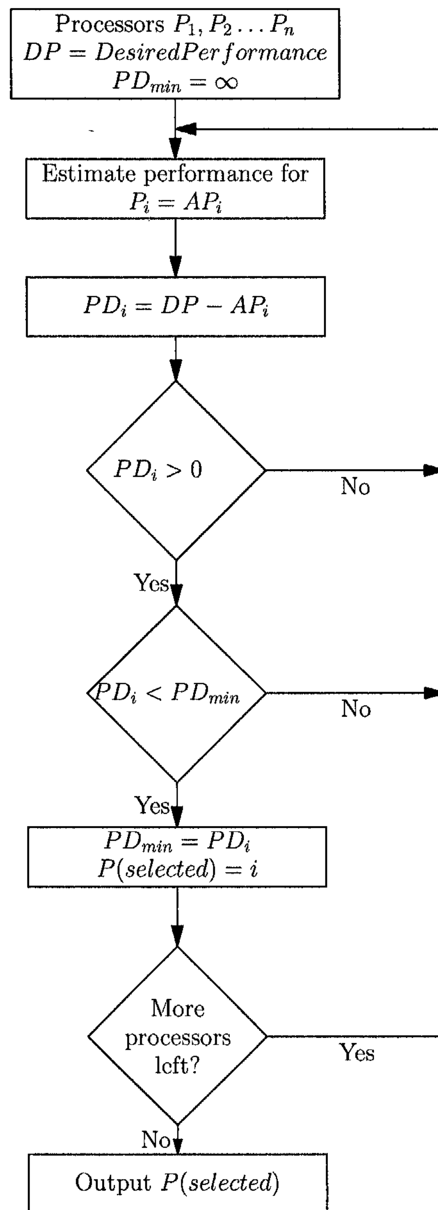

Figure 6.2: Processor selection flow

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

6.3.1 Hardware Software Partitioning

Hardware/software partitioning (HSP) problem is the problem of deciding for each subsystem, whether the required functionality is to be implemented in hardware or software to get the desired performance in terms of running time and power while maintaining least cost. HSP is one of the crucial steps in embedded system design. Satisfaction of performance requirements for embedded systems can frequently be achieved only by hardware implementation of some parts of the application. Selection of the appropriate parts of the system for hardware and software implementation respectively has a crucial impact both on the cost and overall performance of the final product. At the same time, hardware area minimization and latency constraints present contradictory objectives to be achieved through hardware/software partitioning.

Most of the existing approaches to HSP are based on either hardware oriented partitioning or software oriented partitioning. A *software-oriented* approach means that initially the whole application is allotted to software and during partitioning system parts are moved to hardware until constraints are met. In a *hardware-oriented* approach on the other hand, the whole application is implemented in hardware and during partitioning the parts are moved to software until constraints are violated. A software oriented approach has been proposed by Ernst *et al.* [79], Vahid *et al.* [86]. Hardware oriented approach has been proposed in Gupta *et al.* [87], Niemann *et al.* [88]. In [89], the authors proposed a flexible granularity approach for hardware software partitioning. Karam *et al.* [90], proposes partitioning schemes for transformative applications i.e., multimedia and digital signal processing applications. The authors try to optimize the number of pipeline stages and memory required for pipelining. The partitioning is done in an iterative manner. Rakhmatov *et al.* [91] modeled the hardware/software partitioning as a unconstrained bipartitioning problem. Cost

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

functions were used to model the computation and the communication costs. The disadvantage in a hardware oriented partitioning is that the partitioning process is stopped as soon as constraints are met. This can easily result in a non-optimal solution. Similarly for software oriented partitioning the algorithm can stop in a local minimum.

The proposed method is different in the sense that it does not start with a all hardware or all software solution; rather the partitioning problem is modeled as some standard knapsack problem, which can be solved independently to arrive at the solution. Also for every subproblem the lower bound and its upper bound is calculated, this helps in rejecting subproblems, which are not expected to give optimal results. Hence not all subproblems need to be solved. This is the first work to solve hardware/software partitioning problem using the knapsack problem. The advantage of this work is that many problems are rejected based on their lower bound and upper bound, and this reduces the number of subproblems that need to be solved; hence the algorithm is quite fast.

6.3.2 Model of the physical problem

A basic case is considered first, which can be later extended. In the case taken into consideration the application can be broken down into parts such that each of them can be run simultaneously or in other words the parts do not have any sort of data dependency between them. So there is a set of items $S = \{p_1, p_2, \ldots, p_n\}$ to be partitioned into hardware and software. Let h_i and s_i be the time required for the part p_i to be run in hardware and software respectively. Also let a_i be the area required for hardware implementation of part p_i. And let A be the total area available for hardware implementation. The goal is to allot each part into hardware and software so that the combined running time of the whole application is minimized while the area constraint is satisfied. Let us denote the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

solution of the problems as a vector $X = [x_1, x_2, \ldots, x_n]$ such that $x_i \in \{0,1\}$, where $x_i = 0\,(1)$ implies that the part p_i is implemented in software (hardware). Since the hardware and software can be run in parallel, the total running time of the application is given by

$$T(X) = max\{H(X), S(X)\} \tag{6.2}$$

where $H(X)$ is the total running time of the parts running in hardware and $S(X)$ is the total running time of the parts in software. Since all the parts that are implemented in hardware can be run in parallel to each other and all the software parts has to be run in serial,

$$H(X) = \max_{1 \le i \le n}\{x_i \cdot h_i\} \quad and \quad S(X) = \sum_{i=1}^{n}(1 - x_i) \cdot s_i.$$

Hence, the problem can be modeled into

$$\mathcal{P} \begin{cases} \text{minimize} & T(X) \\ \text{subject to} & \sum_{i=1}^{n} x_i \cdot a_i \le A \end{cases} \tag{6.3}$$

6.3.3 Problem Splitting

First of all, the knapsack problem is reviewed. Given a knapsack capacity C and set of items $S = \{1, \ldots, n\}$, where each item has a weight w_i and a benefit b_i. The problem is to find a subset $S' \subset S$, that maximizes the total profit $\sum_{i \in S'} b_i$ under the constraint that $\sum_{i \in S'} w_i \le C$, i.e., all the items fit in a knapsack of carrying capacity C. This problem is called the knapsack problem. The 0-1 knapsack problem is a special case of the general knapsack problem defined above, where each item can either be selected or not selected, but cannot be selected

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

fractionally. Mathematically, it can be described as follows

$$\text{0-1 KP} \begin{cases} \text{maximize} & \sum_{i=1}^{n} p_i \cdot x_i \\ \text{subject to} & \sum_{i=1}^{n} w_i \cdot x_i \leq C, \\ & x_i \in \{0,1\}, i = 1, \dots n \end{cases} \tag{6.4}$$

where x_i is a binary variable equalling 1 if item i should be included in the knapsack and 0 otherwise. It is well known that this problem is NP-complete [92]. However, several large scaled instances could be solved optimally in fractions of a second in spite of the exponential worst-case solution time of all Knapsack algorithms [92, 93, 94].

It is assumed that the items are listed in order of their efficiencies in a non-increasing manner. The efficiency is defined as

$$e_j = \frac{b_j}{w_j} \tag{6.5}$$

therefore $e_i \geq e_j$ for all i, j such that $i < j$.

Let

$$\overline{b_j} = \sum_{i=1}^{j} b_i \quad j = 1, 2, \dots, n \tag{6.6}$$

$$\overline{w_j} = \sum_{i=1}^{j} w_i \quad j = 1, 2, \dots, n \tag{6.7}$$

Packing a knapsack in a greedy way means to put the items in the decreasing order of their efficiency as long as $w_j \leq C - \overline{w}_{j-1}$ i.e., as long as the next item fits in the unused capacity of the knapsack. According to the definition given in [92], the break item is the first item, which cannot be included in the knapsack. Thus the break item t satisfies

$$\overline{w}_{t-1} \leq C \leq \overline{w}_t \tag{6.8}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Let the residual capacity r be defined as

$$r = C - \overline{w}_{t-1} \tag{6.9}$$

By linear relaxation, [92] showed that an upper bound on the total benefit of 0-1 KP is

$$u = \lceil \overline{b}_{t-1} + r \cdot \frac{b_t}{w_t} \rceil \tag{6.10}$$

and the lower bound is given by,

$$l = \overline{b}_{t-1} \tag{6.11}$$

In HSP problem, sort all the items $p_1, p_2, \ldots, p_n$ in decreasing order of their hardware running time. Sorting is done so that if the item with highest hardware running time is allocated, the running time of the whole hardware part is fixed to its hardware running time. This is because all the hardware parts can be run in parallel. After sorting the items are ordered as $p'_1, p'_1, \ldots, p'_n$ and let $h'_i(s'_i)$ be the time required for the part p'_i to be run in hardware(software). So after sorting the following condition is satisfied

$$h'_i \geq h'_j \qquad \text{for all} \quad i \leq j \quad \text{and} \quad 0 \leq i, j \leq n. \tag{6.12}$$

Let us define $S_T = \sum_{i=1}^{n} s'_i$ and $R_i = S_T - s'_i$. Now the problem $\mathcal{P}$ is split into the following n subproblems $\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n$.

Subproblem $\mathcal{P}_1$:

Let p'_1 be implemented in hardware, i.e., $x_1 = 1$, so the total time T that is to be minimised becomes

$$T(X) \;=\; max\,\{h'_1, S(X)\}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$
\begin{aligned}
&= max\left\{h'_1, \sum_{i=1}^{n}(1-x_i)\cdot s'_i\right\} \\
&= max\left\{h'_1, S_T - \sum_{i=1}^{n} x_i\cdot s'_i\right\} \\
&= max\left\{h'_1, S_T - x_1\cdot s'_1 - \sum_{i=2}^{n} x_i\cdot s'_i\right\} \\
&= max\left\{h'_1, S_T - s'_1 - \sum_{i=2}^{n} x_i\cdot s'_i\right\} \\
&= max\left\{h'_1, R_1 - \sum_{i=2}^{n} x_i\cdot s'_i\right\}
\end{aligned}
$$

The objective is to minimize the total running time $T(X)$, i.e.,

$$
\begin{aligned}
minimize \quad T(X) &\Longleftrightarrow minimize\left\{R_1 - \sum_{i=2}^{n} x_i\cdot s'_i\right\} \\
&\Longleftrightarrow maximize\left\{\sum_{i=2}^{n} x_i\cdot s'_i\right\}
\end{aligned}
$$

subject to the constraint $x_i \in \{0,1\}$ and the area constraint

$$\sum_{i=2}^{n} x_i\cdot a_i \le A - a_1 \tag{6.13}$$

Formally, the subproblem $\mathcal{P}_1$ is described as

$$\mathcal{P}_1 \begin{cases} \text{maximize} & \sum_{i=2}^{n} x_i\cdot s'_i \\ \text{subject to} & \sum_{i=2}^{n} x_i\cdot a_i \le A - a_1 \end{cases} \tag{6.14}$$

It is clear that $\mathcal{P}_1$ is the standard 0-1 Knapsack problem, and the solution of $\mathcal{P}_1$ is a feasible solution of the problem $\mathcal{P}$. Let

$$L_1 = \max\{h'_1, R_1 - u_1\},$$

$$U_1 = \max\{h'_1, R_1 - l_1\},$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

where l_1 and u_1 are the lower bound and the upper bound on $\mathcal{P}_1$, respectively. $[L_1, U_1]$ is the bounded interval of $\mathcal{P}_1$ in the sense that the optimal solution of $\mathcal{P}_1$ would lie in the range $[L_1, U_1]$.

Similarly there is subproblem $\mathcal{P}_k$ for $k > 1$

Subproblem $\mathcal{P}_k$:

p'_k is fixed to be implemented in hardware i.e., $x_k = 1$, and all the items $1, 2, \ldots, k-1$ are in software, because if any of them, say j is in hardware then any subproblem $\mathcal{P}_l$ such that $l > j$ is a subset of subproblem $\mathcal{P}_j$. That is, $x_1 = 0, x_2 = 0, \cdots, x_{k-1} = 0$. The total time is

$$T(X) \quad = \quad max\left\{h'_k, R_k - \sum_{i=k+1}^{n} x_i \cdot s'_i\right\}$$

The total running time $T(X)$ is to be minimised, i.e.,

$$\begin{aligned} minimize \quad T(X) \quad &\Longleftrightarrow \quad minimize\left\{R_k - \sum_{i=k+1}^{n} x_i \cdot s'_i\right\} \\ &\Longleftrightarrow \quad maximize\left\{\sum_{i=k+1}^{n} x_i \cdot s'_i\right\} \end{aligned}$$

subject to the constraint $x_i \in \{0, 1\}$ and the area constraint

$$\sum_{i=k+1}^{n} x_i \cdot a_i \leq A - a_k \tag{6.15}$$

Formally, the subproblem $\mathcal{P}_k$ is described as

$$\mathcal{P}_k \begin{cases} \text{maximize} & \sum_{i=k+1}^{n} x_i \cdot s'_i \\ \text{subject to} & \sum_{i=k+1}^{n} x_i \cdot a_i \leq A - a_k \end{cases} \tag{6.16}$$

The bounded interval of subproblem $\mathcal{P}_k$ are

$$L_k = \max\{h'_k, R_k - u_k\},$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$U_k = \max\{h'_k, R_k - l_k\},$$

and the optimal solution of $\mathcal{P}_k$ would lie in the range $[L_k, U_k]$ where l_k and u_k are the lower bound and the upper bound of total benefit of $\mathcal{P}_k$, respectively.

Theorem 6.3.1 *The optimal solution of $\mathcal{P}$ is the solution X_k of $\mathcal{P}_k$, which gives the maximum benefit amongst all the solutions of subproblems $\mathcal{P}_i$, $1 \leq i \leq n$*

To prove the above, let X_i be the solution of the subproblem $\mathcal{P}_i$, $i = 1, 2, \ldots, n$. Now the following needs to be proved

1. Any X_i is a feasible solution of the problem.
2. Any optimal solution of the problem $\mathcal{P}$ belongs to $\{X_1, X_2, \ldots, X_n\}$

Proof

1. Each of $\mathcal{P}_i$, $i = 1, 2, \ldots, n$ is formed from the original problem $\mathcal{P}$ by fixing the part i as being included in the knapsack. And for each $\mathcal{P}_i$, the capacity is $A_i = A - a_i$. Hence every optimal solution of $\mathcal{P}_i$ is a feasible solution of $\mathcal{P}$ with part i in the knapsack.
2. Let X be any feasible solution of $\mathcal{P}$, and let h_i be the part with the highest hardware running amongst the parts included in the knapsack. Since part i is included in the knapsack, it is one of the solutions of the subproblem $\mathcal{P}_i$.

Theoretically, it is possible to create as many subproblems as there are items to be partitioned. But a closer look at the derivations indicate that it is not required to create n subproblems if there are n items to be partitioned. For example, if after creating subproblem i, it is found that

$$\sum_{j=i+1}^{n} a_j \leq A - a_i \tag{6.17}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

This means that after items a_i is fixed to be implemented in hardware and $a_1, a_2, \ldots, a_{i-1}$ into software, the rest of the items left to be partitioned can be implemented easily in hardware as there are enough hardware space left. Most of the cases, hardware runs faster than software and also the items in hardware can run in parallel. So the objective is to implement as much as possible in hardware. Since enough space is available, there is no need to solve the subproblem to find a partition as all the remaining items can now be implemented in hardware. And no more subproblems need to be created as soon as eq[6.17] is satisfied.

A point to be noted is that all subproblems are not of the same size. This implies from the fact that for the first problem item 1 was fixed to be implemented in hardware and the rest of the items are not known whether they are in hardware or software. But in the second problem item 2 was fixed to be implemented in hardware and item 1 is no longer implemented in hardware. This is because if in the second problem, item 1 is implemented in hardware then it becomes a subset of subproblem one. Similarly for subproblem i all the items $1, 2, \ldots, i-1$ are automatically fixed to be in software. Hence, the size of the problem decreases from subproblem 1 to n.

Similarly it is possible to create subproblems for each of the items $p_1, p_2, \ldots, p_n$. The optimal solutions obtained for each of the subproblems will be different from each other, but the optimal solution for the whole problem is the optimal solution from only one of the subproblems. The best solution of all the subproblems in this case will be the optimal solution for the original problem.

6.3.4 Algorithm Description

Following is a brief description of the proposed algorithm. The first step is to sort all the items that are to be partitioned in decreasing order of their hardware running time. Then, create a subproblem corresponding to each of the items

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

to be partitioned, i.e., there are n subproblems to be solved. For each of the subproblems calculate the upper bound and lower bound on the benefits. Find the subproblem with the highest lower bound, lb_{max}. All subproblems whose upper bound is smaller than lb_{max} is guaranteed to give a solution whose benefit is less than lb_{max}. Hence all such subproblems can be ignored and need not be solved. The subproblem with the maximum lower bound should be solved first. Then the subproblems whose upper bound is smaller than the solution should be rejected. These steps should be repeated until all subproblems have been solved or rejected. The outline of the algorithm for solving the HSP problem is given below:

```
1:  BOUND := 0;
2:  sort all the items to be partitioned in decreasing order of
    their hardware running time;
3:  form the subproblems P(i), i=1, 2,...,n;
4:  for(i:=1 ; i <= n ; i++ ){
5:      calculate the upper bound U(i)
          and the lower bound L(i) for P(i);
6:      if( L(i) > BOUND){
7:          BOUND := L(i);
8:      }
9:  }

10:  while( there are subproblems left to be solved){
11:     select the subproblem with the highest lower bound;
12:     if( U(i) < BOUND ){
13:          reject this subproblem;
14:     }else {
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library


```
15:          solve this subproblem;
16:          B(i): = benifit of the above solution;
17:          if ( B(i) > BOUND ){
18:              BOUND := B(i);
19:          }
20:      }
21: }
```

6.3.5 Experimental Works

The proposed algorithm was implemented on a Pentium, 500MHz system, running on Linux. Random data were used for partitioning. For solving the individual 0-1 knapsack problems, the algorithm for 0-1 knapsack problem, given in [93] was used. All running time results are the average of 1000 executions of the algorithm. The experiment was performed for different problem sizes and area constraints.

Table 6.1 shows the execution time of the algorithm for different problem sizes. In this table, the columns denote the different values for the area constraint. Specifically, the valued used were $10, 20, \ldots, 90$ percent of the total area required if all the items were to be implemented in hardware. The different rows are for different problem sizes. For example, for a problem size of 60 and an area constraint of 30 percent of total area required when all the 60 parts are implemented in hardware, the running time was found to be 0.195 ms (from Table 6.1).

Table 6.2 gives a count of the number of subproblems that needed to be solved to arrive at the optimal solution for the whole problem. For example, for a problem size of 30 and area constraint of 20 percent of total area required if all the 30 parts were to be implemented in hardware, the algorithm needed to solve 3 subproblems to arrive at the optimal solution. The result shows that very few

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 6.1: Running time (ms) of the algorithm for different problem sizes

size	fraction of area put as constraint								
n	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
30	0.156	0.461	0.180	0.180	0.180	0.102	0.117	0.102	0.117
60	0.203	0.211	0.195	0.195	0.188	0.117	0.195	0.164	0.109
90	0.125	0.195	0.211	0.188	0.203	0.203	0.188	0.203	0.180
300	0.312	0.336	0.375	0.281	0.352	0.383	0.344	0.359	0.289
500	0.555	0.578	0.594	0.547	0.617	0.508	0.594	0.516	0.516
700	0.906	0.906	0.891	0.945	0.906	0.930	0.859	0.852	0.883
1000	1.688	1.852	1.695	1.633	1.625	1.688	1.758	1.680	1.672
2000	8.078	8.008	8.086	8.078	8.133	7.969	7.977	8.000	7.914
3000	22.102	22.172	22.406	22.141	22.688	22.203	22.195	22.281	22.242

subproblems were needed to be solved to arrive at the solution. In fact only for one instance three subproblems had to be solved to arrive at the solution, while the rest needed only one subproblem to be solved.

A plot of running time of the algorithm for the different problem sizes is shown in Fig 6.3. The area constraint used was 50 percent of the area required if all parts were to be implemented in hardware. Y-axis denotes the running time in milliseconds. X-axis denotes the problem size.

From the results, it can be seen that, for small problem size (e.g. 30, 60, 90), the execution time fluctuates. With increase in size the running time increases. This correctly reflects the property of the knapsack problem. This can also be seen from the graph (Fig 6.3). Also for the same problem size the execution time is stable for different area constraint. This is because the number of subproblems solved is 1 for almost all the cases (see Table 6.2). Hence since the same numbers of same sized subproblems are being solved for all the cases, the execution time is

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 6.2: Number of subproblems solved for different problem size and area constraints.

size	fraction of area put as constraint								
n	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
30	1	3	1	1	1	1	1	1	1
60	1	1	1	1	1	1	1	1	1
90	1	1	1	1	1	1	1	1	1
300	1	1	1	1	1	1	1	1	1
500	1	1	1	1	1	1	1	1	1
700	1	1	1	1	1	1	1	1	1
1000	1	1	1	1	1	1	1	1	1
2000	1	1	1	1	1	1	1	1	1
3000	1	1	1	1	1	1	1	1	1

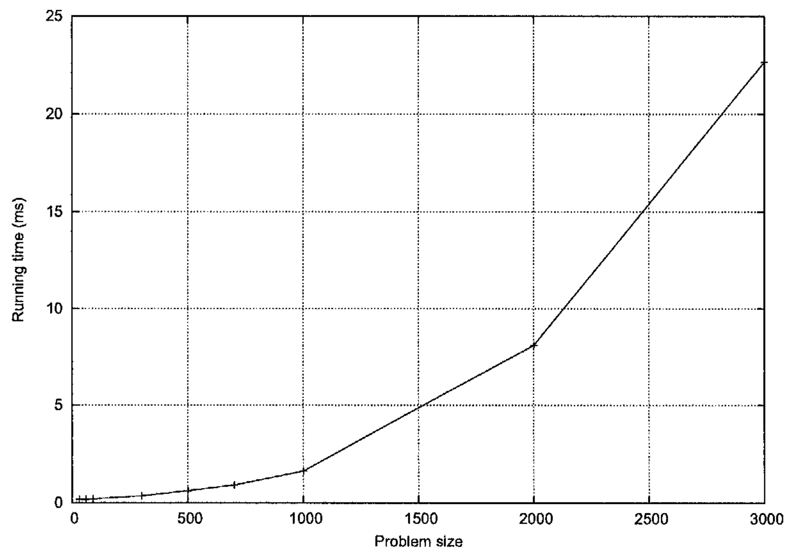

Figure 6.3: Running time vs. Problem size

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

similar. It has to be understood that since, other parameters like communication cost, power, etc were not taken into consideration, this solution may not be the optimal solution to the whole hardware software partitioning problem.

6.4 Design Framework

The processor selection framework and the partitioning framework can be integrated into an embedded system design framework. The design framework uses processor selection to obtain a first choice for a processor for the system. The proposed framework is shown in Fig. 6.4.

In this initially the processor selection framework described in section 6.2 is used to select a processor which just under-performs. Once a processor is selected, the application needs to be partitioned as the processor by itself is unable to meet the performance constraints. But before partitioning potential code segments are removed from the application so that the truncated application can be executed by the processor while still meeting the performance constraints. This process can be iterative, removing code segments until the performance constraints are met. This gives a rough estimate of the size of code that may need to be moved to hardware during the partitioning process and hence also provides a rough estimate of the hardware area required.

The application is then partitioned into hardware and software, using the estimated area obtained in the previous step, to obtain the best performance. If the performance obtained after partitioning is better than the required performance the area can be further reduced while still meeting the performance constraint. If on the other hand the partitioned system fails to meet the performance constraint then the hardware area has to be increased so that the performance constraint is met. Hence the flow is iterative, with the area being decreased if performance

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

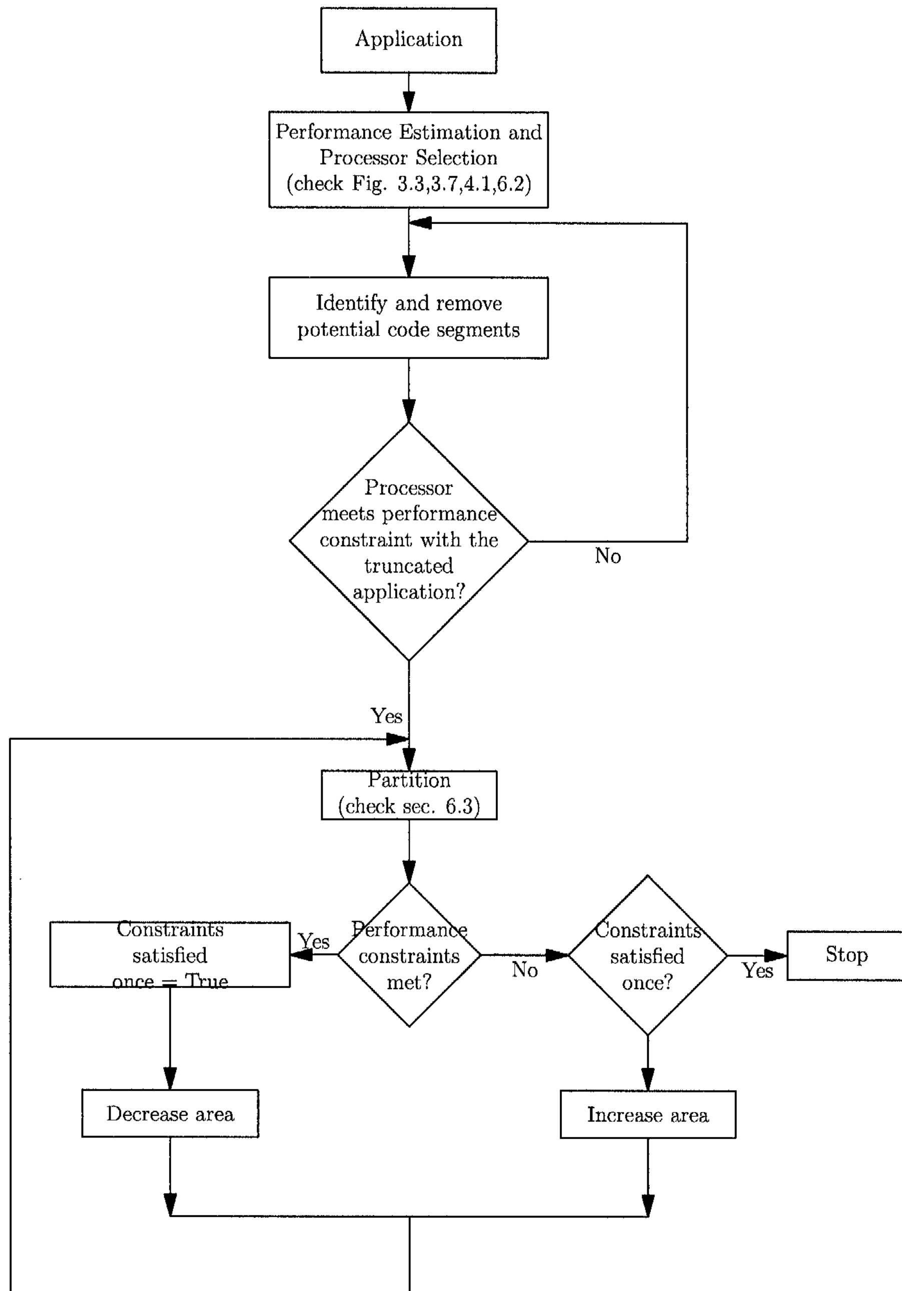

Figure 6.4: Design Flow

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

constraint is met and increased if not.

The idea behind this is that if the performance requirements are met then probably there are more resources than necessary and hence the resource usage is cut down and similarly if requirements are not met then more resources are added. That way the design should finally settle to some near optimum usage of all resources.

6.5 Summary

To maximize the utilization of a processor, a processor selection framework is proposed which is based on selection of a processor which just under-performs with respect to the performance constraints. An over-performing processor means wastage of computing resources besides being expensive. An efficient hardware software partitioning technique based on the Knapsack model was proposed. It was shown that by examining the upper and lower bounds of the sub problems, it is possible to rapidly eliminate the large number of sub-problems that do not contribute to optimal solutions. The investigations demonstrate that a substantial reduction in the number of sub problems that require processing is possible, thereby providing an efficient means for hardware-software partitioning. This is particularly of significance when the problem size is large. Simulations based on problems size ranging from 30 to 3000 confirm that the number of sub-problems that require solution is one except for one case where it was increased to just 3 in order to compute the optimal solution. An embedded system design framework based on processor selection is also presented. Constraints are tightened or relaxed depending on whether the constraints are met or not to achieve a near optimal design.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 7

Conclusion

7.1 Dissertation Summary

Performance estimation can help in reducing the time taken for processor selection. This time can be considerable if a detailed processor performance evaluation is undertaken using instruction set simulators. A shorter time to market is especially important for embedded system design as shown in chapter 1. Whereas instruction set simulators are attractive because of their high accuracy and the ability to model the detailed architecture like the underlying pipelines etc., the high accuracy comes at the cost of long simulation time and the complexity of the simulator development. This problem is further aggravated by the availability of the large number of processors in the market. Hence, a lot of options need to be evaluated before deciding on a particular processor. There is also the additional cost of development of the compiler tool-chains for each processor under consideration. This becomes an expensive process in terms of time and money when a large number of processors need to be evaluated.

Due to the above limitations of detailed performance evaluation, a simpler methodology for performance estimation was suggested in this thesis. The work

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

is based on converting all applications into a target independent intermediate format and then estimating the instructions of the compiled code from the intermediate format and the instruction set of the processor under consideration. The code estimate combined with the profiling output from execution of the intermediate code is used to arrive at the final performance estimates of a processor. As the intermediate format code is target independent, the proposed framework does not need the development of compiler development tools for all the processors under consideration. The proposed framework allows for consideration of a large number of processor during the processor selection stage. Additionally the proposed framework can be used to short-list a few processors which can then be evaluated in detail using the simulator based approaches. Application written in all the high level languages supported by the LLVM front-end can be estimated using the proposed framework.

Techniques were also developed to estimate the performance of standard library functions, which cannot be estimated using the previously mentioned technique in the absence of source codes. The technique is based on obtaining performance characteristics of the standard library functions through experiments with a varied set of inputs. The estimation technique for library functions requires that the performance characteristics of each function must be obtained for each processor under consideration beforehand. However, this needs to be done only once for each processor. Table 4.4 and 4.5 shows that an error of less than ± 2.5 can be achieved for library functions using the proposed technique. Estimates of IPC were also obtained. The upper limit and the lower limit of IPC was used to arrive at an estimate of the IPC.It was found that around 20% accuracy on an average can be achieved using a non-compilation based strategy for performance estimation. The accuracy is better using a combination of compilation and simulation based techniques. But the latter was found to be slower than the proposed

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

techniques.

Based on the performance estimation techniques, a processor selection strategy was proposed. This utilizes the performance estimation techniques to obtain the performance of different processors for a given application. The processor selection framework uses these estimates to select the processor with the least shortfall with respect to the required performance. This ensures that an over-performing (and hence expensive) processor is not selected and that the size of the hardware area required to bring the performance to the required level is the least. There is currently no systematic framework for processor selection. Processor selection in the industry is done by considering the different aspects of the processor like programming skills of the team in any particular processor, benchmark performance, memory (on-chip, extended), DMA (direct memory access) etc. [95]. This research proposes a systematic well defined method for processor selection which can be a significant factor in a good embedded design, given that software development cost is around *to* to 75% of the embedded system project cost [68].

A hardware-software partitioning framework was also proposed based on the 0-1 Knapsack problem. The partitioning framework is based on splitting the hardware-software partitioning problem into many smaller sized 0-1 Knapsack sub-problems and the elimination of sub-problems which are not guaranteed to give optimal solutions and hence need not be solved. As a result only a few problems need to be actually solved using classical approaches towards 0-1 Knapsack problems to arrive at a good partition. In the experimental works, for number of sub-problems ranging from 30 to 3000, for only one case 3 sub-problems had to be solved. In all the other cases solution of only one sub-problem was required to arrive at the partition.

A design framework for constraints-aware embedded system was also pro-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

posed. This involves using the processor selection framework (which in turn uses the performance estimation techniques developed) to identify a suitable processor. Processor selection is followed by hardware-software partitioning in an iterative manner, increasing or decreasing the area depending on whether the performance constraints are satisfied or not. This allows the design to use the optimum processor and the optimum hardware size while still satisfying the performance constraints.

7.2 Further Enhancements

The framework described in this thesis can be extended to provide more accuracy or to allow more processors to be estimated. Listed below are few of the areas which can be future work for this research.

The methodology can be improved to implement a simple register allocator to estimate the register pressure of an application. It will help in obtaining an estimate of the spill code. Currently the methodology does not incorporate the number of registers. Usually RISC processors have a large number of registers and hence the spill code generated may not be much but estimation of the amount of spill code can help to expand the framework to include non RISC processors too.

Incorporation of the memory hierarchy into the estimation framework would be an interesting addition as it would allow for performance estimation under varying cache sizes. The application performance can be studied with different cache sizes and this can be used to obtain a cache size vs. performance relationship which should make possible to estimate processor performance depending on the cache size. Memory accesses can also be classified into the accesses to the stack (usually local variables) and accesses to the heap (usually these consists of

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

dynamically allocated data). The number of accesses of each type along-with the type of memory hierarchy should help in estimating the affect of memory accesses on the overall performance.

Processor performance estimation under multitasking environment would be a natural extension to the current work. Usually a processor works under a multi-tasking environment and hence there is the interaction of multiple tasks competing for the processor resource and sometimes an application can get pre-empted. The multi-tasking estimation can be further extended to multi-processor systems. There has been some work related to WCET analysis of multi-tasking systems [96], but as pointed out earlier (Chapter 2), WCET may not be suitable always.

Hardware estimation i.e., the amount of hardware required for a given application or parts of application, along-with the software estimation framework should enable formulation of a broad-based hardware-software partitioning framework. For an effective partitioning framework, the communication overhead has to be estimated when a part of the software is moved to hardware. Then it can be used to estimate the overall performance of the application when different parts of the application are moved to hardware. This should make possible evaluation of different partitioning schemes.

In this work the performance estimation was obtained for the application. The same technique can be used for performance estimation of small code segments. But in a co-design environment a part of the application runs in hardware while a part of the application runs in software. This entails the need for communication between the two. The communication has a cost associated with it. And for an efficient partition this communication cost has to be taken into account. It would be interesting to find out the amount of data transfer that is needed when a particular code segment is moved from software to hardware. This would require

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

a study of the data variables which are accessed by both the hardware and the software and also those which are exclusively accessed by only one. The number of accesses will also affect the communication cost.

Some of the compiler optimizations can be invalidated by partitioning. For example a if code segment which exclusively accesses a variable is moved to hardware, the said variable is no longer required to be stored in the registers of the processors. Thus partitioning can affect register allocation. This can also lead to future work in the field of a partition aware compiler. After an application has been partitioned the partition-aware compiler can compile the software taking into consideration the partitioning decisions made.

Further work can also be done to allow more advanced processor architectures like out-of-order execution, branch prediction, speculative execution etc. More research can be done to investigate if the techniques developed as part of this thesis can be extended to cater for these features.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Appendix A

Estimation Results for Library Functions

We show the estimation results for some of the standard library functions for ARM. Similar results were obtained for PISA and PPC750.

1. **cos** : The simulation results for the function $cos()$ are provided in Table A.1. Initially the function $cos()$ was called with different random arguments for varying number of iterations. The number of instructions executed per invocation was calculated by diving the number of instructions executed by the number of invocation. The simulation results are shown in Table A.1. We show only the results for 5 different arguments. The table shows the number of instructions executed per invocation of the function $cos()$ for 5 different random arguments (rounded to the nearest integer). We find that the number of instructions executed per invocation stabilizes after about 50000 iterations.

   Next the same experiment was repeated with 100 different random arguments (from 0 to 2π) to $cos()$ for 50000 iterations each. The mean value measured was 159 with a standard deviation was 17 (both the mean and

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table A.1: Simulation results for *cos*()

# of invocations	arg1	arg2	arg3	arg4	arg5
10	1550	1568	1607	1598	1581
50	455	465	492	501	489
100	297	306	324	338	331
500	167	173	191	204	199
1000	151	157	174	187	182
5000	138	143	160	173	169
10000	136	141	158	171	167
50000	135	140	157	170	166
100000	135	140	157	170	166
500000	135	140	157	170	166
1000000	135	140	157	170	166

the standard deviation were rounded to the nearest integer). This shows that the the number of instructions executed does not vary strongly with the argument and hence the function belongs to the first category i.e., the independent category (see Chapter 4). Hence for the function *cos*() the number of instructions executed was estimated to be 159.

2. **sin** : Experiments similar to *cos*() above were performed and the results The simulation results are provided in Table A.2. As with *cos*, we find that the number of instructions executed per invocation stabilizes after about 50000 iterations. And for 100 different arguments iterated 50000 times the mean was found to be 167 and the standard deviation was found to be 24 (both rounded off to the neared integer). Hence for the function *sin*() the number of instructions executed was estimated to be 167.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table A.2: Simulation results for *sin*()

# of invocations	arg1	arg2	arg3	arg4	arg5
10	1557	1559	1590	1594	1612
50	449	455	456	494	494
100	283	288	335	332	328
500	154	156	206	198	193
1000	138	139	190	182	177
5000	124	126	177	168	163
10000	122	124	175	166	161
50000	121	123	174	165	160
100000	121	123	174	165	160
500000	121	123	174	165	160
1000000	121	123	174	165	160

3. **exp** : The function was invoked for varied number of iterations with random arguments. The results for 5 of those arguments are shown in Table A.3. The number of instructions executed per invocation seems to stabilize after 50000 iterations. Then the experiment was repeated with 100 different random numbers as arguments to *exp*() for 50000 iterations each. The mean value of number of instructions executed was found to be 153 with a standard deviation of 0 (rounded off to the nearest integer).

4. **pow** : The function was invoked for varied number of iterations with random arguments. The results for 5 of those arguments are shown in Table A.4. The number of instructions executed per invocation seems to stabilize after 50000 iterations. Then the experiment was repeated with 100 different random numbers as arguments to *exp*() for 50000 iterations each. The mean value of number of instructions executed was found to be 404 with a

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table A.3: Simulation results for *exp*()

# of invocations	arg1	arg2	arg3	arg4	arg5
10	3049	3040	3043	3051	3052
50	732	730	731	732	732
100	442	441	442	443	443
500	210	210	210	211	211
1000	182	181	181	182	182
5000	158	158	158	158	158
10000	155	155	155	155	155
50000	153	153	153	153	153
100000	153	153	153	153	153
500000	153	153	153	153	153
1000000	153	153	153	153	153

standard deviation of 0 (rounded off to the nearest integer).

5. **sqrt**: The function was invoked for varied number of iterations with random arguments. The results for 5 of those arguments are shown in Table A.4. The number of instructions executed per invocation seems to stabilize after 50000 iterations. Then the experiment was repeated with 100 different random numbers as arguments to *exp*() for 50000 iterations each. The mean value of number of instructions executed was found to be 126 with a standard deviation of 0 (rounded off to the nearest integer).

6. **rand**: The function was invoked for varied number of iterations with random arguments. The results for 5 of those arguments are shown in Table A.6. The number of instructions executed per invocation seems to stabilize after 500000 iterations. Then the experiment was repeated with 100 differ-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table A.4: Simulation results for *pow*()

# of invocations	arg1	arg2	arg3	arg4	arg5
10	3338	3329	3332	3340	3341
50	990	989	989	991	991
100	697	696	697	697	697
500	462	462	462	462	462
1000	433	433	433	433	433
5000	409	409	409	409	409
10000	406	406	406	406	406
50000	404	404	404	404	404
100000	404	404	404	404	404
500000	404	404	404	404	404
1000000	404	404	404	404	404

Table A.5: Simulation results for *sqrt*()

# of invocations	arg1	arg2	arg3	arg4	arg5
10	3623	3616	3603	3609	3607
50	826	829	822	821	817
100	474	474	471	473	472
500	196	196	196	196	196
1000	161	161	161	161	161
5000	133	133	133	133	133
10000	129	129	129	129	129
50000	126	126	126	126	126
100000	126	126	126	126	126
500000	126	126	126	126	126
1000000	126	126	126	126	126

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table A.6: Simulation results for *rand*()

# of invocations	arg1	arg2	arg3	arg4	arg5
10	3687	3659	3666	3683	3641
50	794	791	792	785	785
100	432	429	431	425	433
500	138	137	137	136	137
1000	101	100	100	100	101
5000	71	71	71	71	71
10000	67	67	67	67	67
50000	64	64	64	64	64
100000	64	64	64	64	64
500000	63	63	63	63	63
1000000	63	63	63	63	63

ent random numbers as arguments to *exp*() for 50000 iterations each. The mean value of number of instructions executed was found to be 63 with a standard deviation of 0 (rounded off to the nearest integer).

7. **fscanf**: For *fscanf*(), the function was invoked with varying number of characters that were read from a file. The result of the simulation experiments are shown in Table A.7. As can be seen the number of instructions executed per invocation varies with the input. This variation is shown in Fig. A.1. This plot was later curve fitted to obtain the number of instructions executed for different number of characters read.

8. **fprintf**: For *fprintf*(), the function was similarly invoked with varying number of characters to be printed to the file. The result of the simulation experiments are shown in Table A.8. As can be seen the number of in-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table A.7: Simulation results for $fscanf()$

# of characters read	# of instructions executed per invocation
1	849
2	1088
3	1328
5	1808
7	2288
10	3008

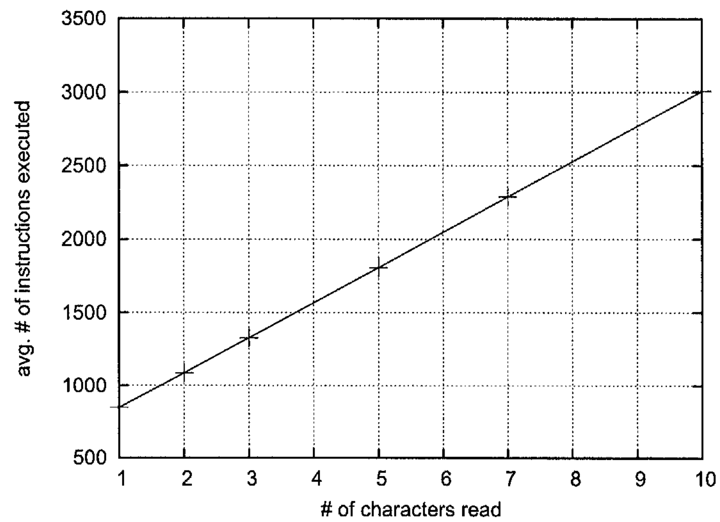

Figure A.1: Simulation results for $fscanf()$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table A.8: Simulation results for $fprintf()$

# of characters printed	# of instructions executed per invocation
1	324
3	332
5	347
10	374
15	401
20	428

structions executed per invocation varies with the function argument. This variation is shown in Fig. A.2. This plot was later curve fitted to obtain the number of instructions executed for different number of characters printed to the file.

9. **printf**: For $printf()$, the function was similarly invoked with varying number of characters to be printed to *stdout*. The result of the simulation experiments are shown in Table A.9. As can be seen the number of instructions executed per invocation varies with the function argument. This variation is shown in Fig. A.3. This plot was later curve fitted to obtain the number of instructions executed for different number of characters printed to *stdout*.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

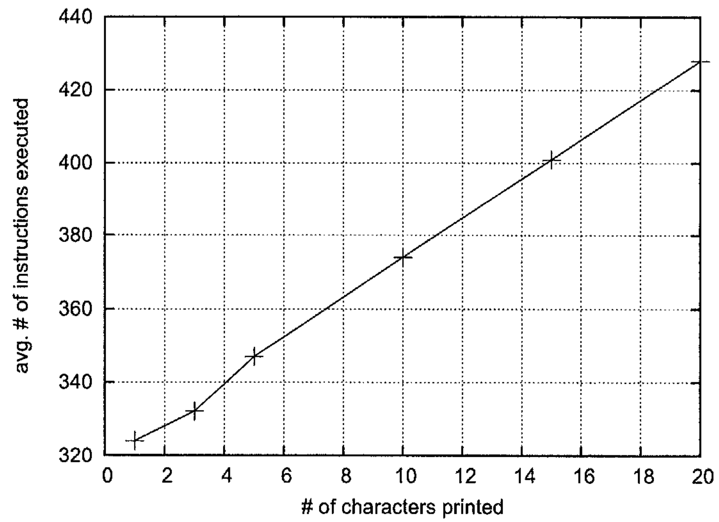

Figure A.2: Simulation results for $fprintf()$

Table A.9: Simulation results for $printf()$

# of chars printed	# of instructions executed per invocation
1	339
3	358
5	384
10	438
15	494
20	552
25	594
30	642

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

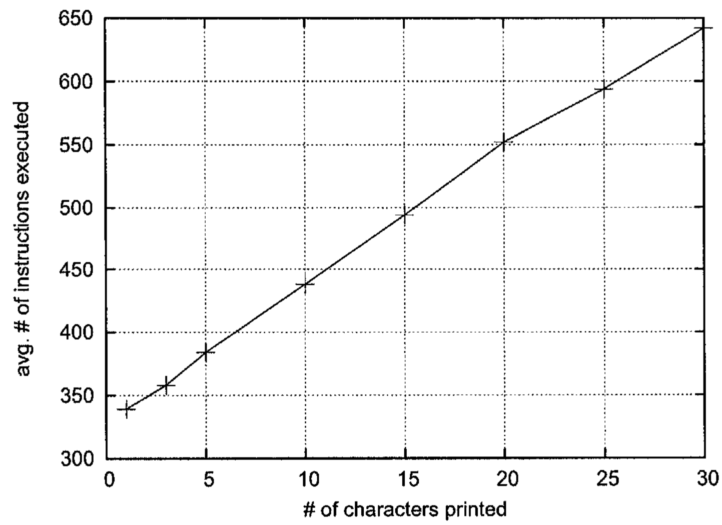

Figure A.3: Simulation results for *printf*()

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Appendix B

MiBench Benchmark Suite

- **Automotive and Industrial Control**: In this category the programs are related to embedded control systems. The programs included in this category are a) *basicmath*, which performs simple calculations that often do not have dedicated hardware support in embedded processors, b) *bitcount* counts the number of bits in an array of integers, c) *qsort* sorts an array of strings using the quick sort algorithm and d) *susan*, is an image recognition package.

- **Network**: The network category represents the processors in network devices like switches and routers. The programs included in this category are a) *dijkstra* is a shortest path algorithm. The benchmark program finds the shortest path between every pair of nodes in the given input graph, b) *patricia* is a program to create and search through a patricia trie data structure.

- **Security**: The security category contains includes programs for data encryption, decryption and hashing. The programs included in this category are a) *blowfish* is a keyed, symmetric block cipher, designed by Bruce

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Schneier. The benchmark program contains encrypts and decrypts ASCII files given as input, b) *sha* or Secure Hash Algorithm is considered to be the successor to MD5, an earlier, widely-used hash function.

- **Consumer Devices**: The consumer category, contains programs related to the consumer devices like digital cameras. The programs included in this category are a) *jpeg*, contains programs for jpeg encoding and decoding. It is a commonly used standard method of lossy compression for photographic images. b) *tiff* contains programs to convert a colour TIFF image to either a black and white image or a RGB colour formatted TIFF image, to dither a black and white image to reduce the resolution and size of an image, or to convert an image to a reduced colour palette, c) *lame*, is a open source MP3 encoder, d) *mad* is a opensource MPEG audio decoder, d) *typeset* is a typesetting tool, that has a front-end processor for HTML.

- **Office Automation**: This category primarily involves programs which do text manipulation. The programs included in this category are a) *ghostscript* is a postscript interpreter. This program is important for many postscript enabled embedded devices like printers, b) *stringsearch*, searches for given words in a text, c) *ispell*, is a fast spelling checker, d) *rsynth* is a text to speech synthesizer, d) *sphinx* is a speech decoder.

- **Telecommunication**: This telecommunication category contains voice processing programs like voice encoding and decoding. The programs included in this category are a) *FFT*, performs fast fourier transforms and its inverse on an array of data, b) *GSM* is a commonly used standard for encoding and decoding voice, c) *adpcm* is a digital representation of an analog signal where the magnitude of the signal is sampled regularly at uniform intervals, then quantized to a series of symbols in a digital (usually binary) code. It

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

is used in digital telephone systems and is also the standard form for digital audio in computers and various compact disc formats, d) *crc* or cyclic redundancy check is a type of hash function used to produce a checksum - which is a small, fixed number of bits - against a block of data, such as a packet of network traffic or a block of a computer file. The checksum is used to detect and correct errors after transmission or storage.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Appendix C

The LLVM Compiler Infrastructure

Low Level Virtual Machine (LLVM) is:

1. A compilation strategy designed to enable effective program optimization across the entire lifetime of a program. LLVM supports effective optimization at compile time, link-time (particularly interprocedural), run-time and offline (i.e., after software is installed), while remaining transparent to developers and maintaining compatibility with existing build scripts.

2. A virtual instruction set - LLVM is a low-level object code representation that uses simple RISC-like instructions, but provides rich, language-independent, type information and dataflow (SSA) information about operands. This combination enables sophisticated transformations on object code, while remaining light-weight enough to be attached to the executable. This combination is key to allowing link-time, run-time, and offline transformations.

3. A compiler infrastructure - LLVM is also a collection of source code that

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

implements the language and compilation strategy. The primary components of the LLVM infrastructure are a GCC-based C & C++ front-end, a link-time optimization framework with a growing set of global and interprocedural analyses and transformations, static back-ends for the X86, PowerPC, IA-64, Alpha, & SPARC V9 architectures, a back-end which emits portable C code, and a Just-In-Time compiler for X86, PowerPC, and SPARC V9 processors.

4. LLVM does not imply things that you would expect from a high-level virtual machine. It does not require garbage collection or run-time code generation (In fact, LLVM makes a great static compiler!). Note that optional LLVM components can be used to build high-level virtual machines and other systems that need these services.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Appendix D

SimpleScalar Tool-set

The SimpleScalar tool set is a system software infrastructure used to build modeling applications for program performance analysis, detailed microarchitectural modeling, and hardware-software co-verification. Using the SimpleScalar tools, users can build modeling applications that simulate real programs running on a range of modern processors and systems. The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. The SimpleScalar tools are used widely for research and instruction, for example, in 2000 more than one third of all papers published in top computer architecture conferences used the SimpleScalar tools to evaluate their designs. In addition to simulators, the SimpleScalar tool set includes performance visualization tools, statistical analysis resources, and debug and verification infrastructure.

SimpleScalar simulators can emulate the Alpha, PISA, ARM, and x86 instruction sets. The tool set includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations. All of the simulators distributed with the current release of SimpleScalar can run

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

programs from any of the above listed instruction sets. Complex instruction set emulation (e.g., x86) can be implemented with or without microcode, making the SimpleScalar tools particularly useful for modeling CISC instruction sets.

The PISA instruction set (a.k.a. the portable instruction set architecture) is a simple MIPS-like instruction set maintained primarily for instructional use. A GNU GCC-based cross-compiler and pre-built libraries are also available for this target. The PISA target is particularly useful for computer engineering instruction as the tools can be built on a wide range of host platforms, including Linux/x86, Win2000, SPARC Solaris, and others.

SimpleScalar builds on most 32-bit and 64-bit flavors of UNIX and Windows NT-based operating systems. The internal software architecture of the tool set includes a host interface module, permitting fast porting to other host platforms. The host interface module permits cross-endian emulation, thus it is possible to use emulate a target on a host platform with a different endian, e.g., running Alpha ISA emulation on a SPARC Solaris host platform. Most SimpleScalar users and developers (including SimpleScalar LLC) use SimpleScalar on Linux/x86.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Appendix E

Publications

International Journal Papers

- Abhijit Ray, Wu Jigang, Thambipillai Srikanthan, "Knapsack Model and Algorithm for Hardware/Software Partitioning Problem", Computing and Informatics, Volume 23, 2004, No. 5-6.

International Conference Papers

- Abhijit Ray, Thambipillai Srikanthan, Wu Jigang, "Practical Techniques for Performance Estimation of Processors", The 5th International Workshop on System-on-Chip for Real-Time Applications (IWSOC) 2005: pp: 308-311

- Abhijit Ray, Thambipillai Srikanthan, Wu Jigang, "Estimating Processor Performance of Standard Library Function", The 2nd International Conference on Embedded Software and Systems, December 2005, pp: 181-186.

- Abhijit Ray, Wu Jigang, Thambipillai Srikanthan, "Knapsack Model and Algorithm for HW/SW Partitioning Problem", International Conference on Computational Science 2004, pp: 200-205.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

References

[1] Mauricio Varea. *Modelling and Verification of Embedded Systems Based on Petri Net Oriented Representations.* PhD thesis, University of Southampton, 2003.

[2] Philip Koopman. Embedded system design issues (the rest of the story). In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, page 310, Washington, DC, USA, 1996. IEEE Computer Society.

[3] Nasser Jazdi. Component-based and distributed web application for embedded systems. In *International Conference on Intelligent Agents, Web Technology and Internet Commerce - IAWTIC'2001*, Las Vegas, USA, November 2001.

[4] Marcello Lajolo, Mihai Lazarescu, and Alberto Sangiovanni-Vincentelli. A compilation-based software estimation scheme for hardware/software co-simulation. In *CODES '99: Proceedings of the seventh international workshop on Hardware/software codesign*, pages 85–89, New York, NY, USA, 1999. ACM Press.

[5] Texas Instruments. Modern 32bit processors make processor selection an easytask.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[6] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, April 1965.

[7] http://www.ampro.com/html/overview.html. Ampro website, September 2005.

[8] Jim Turley. Ten lies about microprocessors. *Embedded Systems Design*, June 2003.

[9] http://www.giichinese.com.tw/chinese/bc31319-embedded systems_toc.html. Future of embedded systems technology, June 2005.

[10] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *Design Automation Conference*, pages 147–152, 1997.

[11] Norbert Imlig and Akihiro Tsutsui. Performance estimation of embedded software with pipeline and cache hazard modeling. In *ISHPC '97: Proceedings of the International Symposium on High Performance Computing*, pages 131–142, London, UK, 1997. Springer-Verlag.

[12] Sriram Subramanian. Software performane estimation techniques in a co-design environment. Master's thesis, University of Cincinnati, 2003.

[13] P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[14] Robert Nilsson. Software performance estimation by static analysis.

[15] Antoine Colin and Stefan M. Petters. Experimental evaluation of code properties for wcet analysis. In *RTSS*, pages 190–199, 2003.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[16] David I. August Kaiyu Chen, Sharad Malik. Retargetable static timing analysis for embedded software. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, October 2001.

[17] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 456–461, New York, NY, USA, 1995. ACM Press.

[18] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of c programs. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 98–103, New York, NY, USA, 2001. ACM Press.

[19] Jwahar R. Bammi, Wido Kruijtzer, Luciano Lavagno, Edwin Harcourt, and Mihai T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*, pages 82–86, New York, NY, USA, 2000. ACM Press.

[20] T. Vinod Kumar Gupta, Purvesh Sharma, M Balakrishnan, and Sharad Malik. Processor evaluation in an embedded systems design environment. In *VLSID '00: Proceedings of the 13th International Conference on VLSI Design*, pages 98–103, Washington, DC, USA, 2000. IEEE Computer Society.

[21] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 598–604, Washington, DC, USA, 1997. IEEE Computer Society.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[22] A. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating tight execution time bounds of programs by annotations. *IEEE Real-Time Syst. Newsl.*, 5(2-3):81–86, 1989.

[23] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5):48–57, 1991.

[24] Prahat Mishra and Nikil Dutt. *Customizable and Configurable Embedded Processors(to appear)*. Morgan Kaufmann Publishers, 2006.

[25] W. Qin and S. Malik. *The Compiler Design Handbook: Optimizations & Machine Code Generation*, chapter Architecture Description Languages for Retargetable Compilation. CRC Press, 2002.

[26] Subrata Dasgupta. *WIE Computer Architecture: A Modern Synthesis*. John Wiley & Sons, Inc., New York, NY, USA, 1988.

[27] Frank Engel, Johannes Nhrenberg, and Gerhard P. Fettweis. A generic tool set for application specific processor architectures. In *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*, pages 126–130, New York, NY, USA, 2000. ACM Press.

[28] Peter Marwedel. The mimola design system: Tools for the design of digital processors. In *DAC '84: Proceedings of the 21st conference on Design automation*, pages 587–593, Piscataway, NJ, USA, 1984. IEEE Press.

[29] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The mimola language - version, 1994.

[30] A. Halambi and P. Grun. Expression: A language for architecture exploration through compiler/simulator retargetability, 1999.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[31] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, pages 503–507, Washington, DC, USA, 1995. IEEE Computer Society.

[32] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. Chess : retargetable code generation for embedded dsp processors, 1995.

[33] Rajat Moona. Processor models for retargetable tools. In *RSP '00: Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, pages 34–39, Washington, DC, USA, 2000. IEEE Computer Society.

[34] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *VLSID '99: Proceedings of the 12th International Conference on VLSI Design - 'VLSI for the Information Appliance'*, pages 132–137, Washington, DC, USA, 1999. IEEE Computer Society.

[35] Subhash Chandra and Rajat Moona. Retargetable functional simulator using high level processor models. In *VLSI Design*, pages 424–429, 2000.

[36] Y Subhash Chandra. Retargetable functional simulator. Master's thesis, ndian Institute of Technology, Kanpur, June 1999.

[37] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl: an instruction set description language for retargetability. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 299–302, New York, NY, USA, 1997. ACM Press.

[38] Bhuvan Middha, Anup Gangwar, Anshul Kumar, M. Balakrishnan, and Paolo Ienne. A trimaran based framework for exploring the design space

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

of vliw asips with coarse grain functional units. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 2–7, New York, NY, USA, 2002. ACM Press.

[39] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. Lisa - machine description language and generic machine model for hw/sw co-design, 1996.

[40] S.P. Seng, K.V. Palem, R.M. Rabbah, W.F. Wong, W. Luk, and P.Y.K. Cheung. Pd-xml: Extensible markup language for processor description. In *FPT '02: Proceedings of IEEE International Conference on Field-Programmable Technology*, pages 437–440, Dec 2002.

[41] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 758–763, New York, NY, USA, 2003. ACM Press.

[42] Todd M. Austin. A user's and hacker's guide to the simplescalar architectural research tool set, January 1997.

[43] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[44] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12, UWCSE 93-06-06, Sun Microsystems Laboratories,Inc., 1993.

[45] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137, New York, NY, USA, 1994. ACM Press.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[46] http://simit arm.sourceforge.net/. Simit-arm, October 2005.

[47] http://microlib.org/projects/ppc750sim/. Microlib: Ppc750 sim, October 2005.

[48] ARM. *RVDS 2.1 Introductory Tutorial.*

[49] Mehrdad Reshadi and Prabhat Mishra. Memory access optimizations in instruction-set simulators. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 237–242, New York, NY, USA, 2005. ACM Press.

[50] C. Mills, S. C. Ahalt, and J. Fowler. Compiled instruction set simulation. *Software, Practice and Experience*, 21(8):877–889, 1991.

[51] Ho Young KIM and Tag Gon KIM. Trace-Driven Performance Simulation Modeling for Fast Evaluation of Multimedia Processor by Simulation Reuse. *IEICE Trans Fundamentals*, E88-A(12):3306–3314, 2005.

[52] C. May. Mimic: a fast system/370 simulator. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques*, pages 1–13, New York, NY, USA, 1987. ACM Press.

[53] Bryan Black, Andrew S. Huang, Mikko H. Lipasti, and John Paul Shen. Can trace-driven simulators accurately predict superscalar performance? In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 478–485, Washington, DC, USA, 1996. IEEE Computer Society.

[54] Charlton D. Rose and J. Kelly Flanagan. Constructing instruction traces

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

from cache-filtered address traces (citcat). *SIGARCH Comput. Archit. News*, 24(5):1–8, 1996.

[55] Bryan Black, Andrew S. Huang, Mikko H. Lipasti, and John Paul Shen. Can trace-driven simulators accurately predict superscalar performance? In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 478–485, Washington, DC, USA, 1996. IEEE Computer Society.

[56] Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz, and Mikko H. Lipasti. Precise and accurate processor simulation. In *caec2002*, pages 437–440, Dec 2002.

[57] Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, Rainer Leupers, and Heinrich Meyr. A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 468–473, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[58] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *RTSS 2000: Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 163–174, 2000.

[59] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code. In *Proceedings of the 10th EuroMicro Workshop on Real-Time Systems*, June 1998.

[60] Sung-Soo Lim, Jung Hee Han, Jihong Kim, and Sang Lyul Min. A worst case timing analysis technique for multiple-issue machines. In *RTSS*, pages 334–345, 1998.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[61] Peter Puschner. Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In *Proc. 2nd Euromicro International Workshop on WCET Analysis*, Technical Report, York YO10 5DD, United Kingdom, Jun. 2002. Department of Computer Science, University of York.

[62] Christian Ferdinand. Worst case execution time prediction by static program analysis. In *IPDPS: Proceedings of the 19th International Parallel and Distributed Processing Symposium*, April 2004.

[63] Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[64] Amer Baghdadi, Nacer-Eddine Zergainoh, Wander O. Cesrio, and Ahmed Amine Jerraya. Combining a performance estimation methodology with a hardware/software codesign flow supporting multiprocessor systems. *IEEE Trans. Softw. Eng.*, 28(9):822–831, 2002.

[65] Naji Sami Ghazal. *Evaluation and Guidance in Processor Architecture Selection for DSP.* PhD thesis, University of California at Berkeley, 2000.

[66] Kelvin D. Nilsen and Bernt Rygg. Worst-case execution time analysis on modern processors. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 20–30, New York, NY, USA, 1995. ACM Press.

[67] Greg Stitt and Frank Vahid. Hardware/software partitioning of software binaries. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international*

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

conference on Computer-aided design, pages 164–170, New York, NY, USA, 2002. ACM Press.

[68] Peter S. Gilmour. How to select tools for microcontroller software. *IEEE Spectr.*, 28(2):37–39, 1991.

[69] Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[70] http://www.eecs.umich.edu/mibench/. Mibench, October 2005.

[71] Doug Burger and Todd M. Austin. *The SimpleScalar Tool Set, Version 2.0.*

[72] http://llvm.cs.uiuc.edu. The llvm compiler infrastructure project, June 2004.

[73] http://www.gnu.org/software/libc/libc.html. Gnu c library, May 2005.

[74] http://www.corpit.ru/mjt/qsort.html. Inline qsort() implementation, May 2005.

[75] David James Ofelt. *Efficient Performance Prediction For Modern Microprocessors.* PhD thesis, Stanford University, 1999.

[76] http://www.simplescalar.com/. Simplescalar, October 2005.

[77] http://www.cs.iastate.edu/ prabhu/Tutorial/PIPELINE/structHaz.html. Computer architecture tutorial, January 2006.

[78] John L. Hennessy and David A. Patterson. *Computer Architecture; A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[79] Rolf Ernst, Jörg Henkel, and Thomas Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, 10(4):64–75, 1993.

[80] David H. Albonesi Pradip Bose and Diana Marculescu. Complexity-effective design. In *Workshop on Complexity-Effective Design*, Anchorage, Alaska, USA, 2002.

[81] Rajesh K. Gupta. Ics 212: Introduction to embedded systems, 2002. Lecture Notes for Course Code: 36605, Winter 2002.

[82] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test*, 18(6):23–33, 2001.

[83] http://www.sysgo.com/products/portable-afdx/software-vs-hardware solution/. Software vs. hardware solution, Feb 2008.

[84] John Catsoulis. *Designing Embedded Hardware*, pages 154–155. O'Reilly, May 2005.

[85] Alakananda Bhattacharya, Amit Konar, Swagatam Das, Crina Grosan, and Ajith Abraham. Hardware software partitioning problem in embedded system design using particle swarm optimization algorithm. In *Second International Conference on Complex, Intelligent and Software Intensive Systems (CISIS-2008)*. IEEE Computer Society Press, 2008.

[86] Frank Vahid, Daniel D. Gajski, and Jie Gong. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. In *EURO-DAC '94: Proceedings of the conference on European design automation*, pages 214–219, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[87] R. K. Gupta and G. De Micheli. System-level synthesis using re-programmable components. In *EDAC '92: Proceedings of the European Conference on Design Automation (EDAC)*, pages 2–7, 1992.

[88] Ralf Niemann and Peter Marwedel. Hardware/software partitioning using integer programming. In *EDTC '96: Proceedings of the 1996 European conference on Design and Test*, pages 473–479, Washington, DC, USA, 1996. IEEE Computer Society.

[89] Jorg Henkel and Rolf Ernst. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(2):273–290, 2001.

[90] Karam S. Chatha and Ranga Vemuri. Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):193–208, 2002.

[91] Daler N. Rakhmatov and Sarma B. K. Vrudhula. Hardware-software bipartitioning for dynamically reconfigurable systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 145–150, New York, NY, USA, 2002. ACM Press.

[92] David Pisinger. *Algorithms for knapsack problems.* PhD thesis, University of Copenhagen, 1995.

[93] D. Pisinger. A minimal algorithm for the 0-1 knapsack problem, 1997.

[94] Wu Jigang, Lei Yunfei, and Heiko Schroder. A minimal reduction approach for the collapsing knapsack problem. *Comput. Inform.*, 20(4):359–369, 2001.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[95] Alexandre Jose da Silva and Romulo Silva de Oliveira. Methods and guidelines for embedded system processor selection. In *VII Induscon - Conferencia Internacional de Aplicacoes Industriais.*

[96] Matteo Corti, Roberto Brega, and Thomas Gross. Approximation of worst-case execution time for preemptive multitasking systems. In *LCTES '00: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 178–198, London, UK, 2001. Springer-Verlag.