Investigations on CPI Centric Worst Case Execution Time Analysis

A Thesis

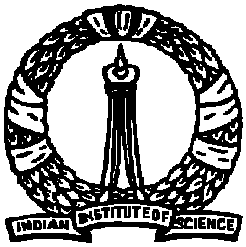
Submitted For the Degree of

Doctor of Philosophy

in the Faculty of Engineering

by

Archana Ravindar



Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

June 2013

Dedicated to the fond memory of

Prof. Priti Shankar

(September 20, 1947 - October 17, 2011)

Acknowledgements

The journey up till here has been a wonderful experience and I have been supported by many people in various ways to reach here. Firstly, I wish to thank my advisor Prof. Y. N. Srikant, who agreed to be my advisor. It has been an enriching experience working under him, discussing ideas, exchanging feedback and evaluating several possibilities. He has been a source of constant support and encouragement throughout my research and I am deeply grateful to him for that. It was the support of my family, especially my parents, which gave me the courage of embarking on this journey at the time I was about to welcome my son into this world. My son, Vamshidhar, has been a source of unending delight ever since and has equally supported me in his own way to carry out my academic responsibilities. I wish to express my deep gratitude to Prof. Matthew Jacob T, who gave me vital feedback on the architectural and experimental aspects of this work. I wish to thank Kapil Vaswani, Rupesh Nasre, Meghana Mande, Indrajit Bhattacharya, Chiranjib Bhattacharya, Jan Reineke, Reinhard Wilhelm, Sibin Mohan, Tulika Mitra, Rathijit Sen, Vinayak Puranik and Ananda Vardhan who have provided feedback on several aspects of this work. I thank Niklas Holsti for providing access to the DEBIE-1 benchmark and answering several of my queries regarding the benchmark and its usage. I thank Guillem Bernat for providing the license of *RapiTime* and his excellent support team at *Rapita Systems* who have helped clarify several aspects of *RapiTime* and its usage. I also thank Antoine Colin for providing me access to some of the benchmarks. I thank T. V. Ananthapadmanabha for his strong support and encouragement throughout the course of this research. Both the former chairman, Prof M. Narasimha murthy and the current chairman, Prof Y. Narahari have been very supportive during the course of my research. Thanks to Prof. P. Balaram, I could stay within the campus with my family and carry out research without any major logistical problems. I thank Jagadish N and B K Pushparaj for helping me maintain my machines well

and obtain timely software and backups. I thank the office staff of CSA who have treated me as their own over the years. I am very grateful to IMPECS for funding a part of my research. I dedicate this work to Prof Priti Shankar, whom I had first met as a nervous undergraduate student in 1998. Ever since then, life was never the same again. She has influenced my life in more ways than I have ever known. Knowing her has been a highest privilege. Her passing away has left a deep void in my life, but memories of her are a constant everyday companion and a source of inspiration.

Publications based on this Thesis

1. Archana Ravindar and Y. N. Srikant, *Implications of Program Phase Behavior on Timing Analysis*, In Proceedings of the 15th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-15), HPCA 2011, pages 71 – 79.

2. Archana Ravindar and Y. N. Srikant, *Relative Roles of Instruction Count and Cycles Per Instruction in WCET Estimation*, In Proceedings of the second ACM SPEC International Conference on Performance Engineering (ICPE), 2011, pages 55 – 60.

3. Archana Ravindar and Y. N. Srikant, *Estimation of Probabilistic Bounds of Phase CPI and Relevance in WCET Analysis*, In Proceedings of ACM International Conference on Embedded Software (EMSOFT) 2012, pages 165 – 174.

Abstract

Estimating program worst case execution time or WCET is an important problem in the domain of real-time and embedded systems that are associated with deadlines. In such systems, it is vital that a part or whole of the program executes within a specified time limit. If WCET of a program is greater than the specified time limit then the program is either recoded or the architecture is redesigned to meet the specified time limit. Knowledge of WCET guides effective scheduling of tasks ensuring optimum resource usage. Current state of the art techniques estimate WCET of a program by dividing the program into a number of smaller components. The cost of execution of these program components on the target are either statically obtained by a static WCET analyzer or obtained by direct measurement by a measurements based WCET analyzer before they are combined in an orderly manner using well known techniques such as integer linear programming, timing schema or graph algorithms, to give the final WCET estimate. Statistical WCET analyzers fit end to end measured execution times into a model usually based on extreme value theory and extrapolate the curve up to the desired probability to estimate WCET.

Static WCET analysis methods estimate WCET without running the program on the target system and are hence constrained to make conservative assumptions about dynamic program behavior, potentially leading to pessimistic WCET estimates. A static WCET analyzer is complex to build and is not easily retargetable. A measurements based WCET analyzer has access to runtime behavior, as a result, the pessimism of the estimate can be reduced. The process of measurement and the amount of instrumentation should not affect the very timing of the program which is being analyzed. Achieving accurate WCET estimates with sparse instrumentation is not easy. In the case of statistical WCET analyzers, the model should be chosen such that it is close to the real world. In case of both measurements and statistical

WCET analyzers, the choice of test inputs exercised to build the samples should cover those paths that most likely contribute to WCET.

The thesis proposes a hybrid WCET analyzer that consists of strong aspects of both static WCET analysis (theoretical upper bound) and measurement based WCET analysis (accurate information about runtime behavior). The thesis proposes to estimate program WCET as a product of maximum instruction count (IC), where IC is the number of instructions executed and maximum cycles per instruction (CPI). The idea of estimating WCET as a product of IC and CPI instead of estimating it as a function of processor cycles, arises from the way RISC architectures which comprise real-time and embedded systems, are built. RISC machines consist of an instruction pipeline wherein multiple instructions are in execution at the same time. In such systems, it is more meaningful to talk about the average cycles per instruction than the number of cycles taken by each instruction. If there is low confidence on the coverage aspect of the test input set, maximum IC is taken as the theoretical upper bound on IC, computed by static structural analysis. If the test input set adequately covers the program, maximum IC could be the measured maximum IC observed across different runs when the program is run with the test input set. CPI is the measured parameter. On advanced architectures, this simple timing equation is observed to give 10-50% improvement in accuracy of WCET compared to *Chronos*, a static WCET analyzer.

Factorizing execution time as a product of IC and CPI, reveals the existence of a correlation between CPI and IC in many programs. Either a direct or an inverse correlation is observed. In some programs, the correlation is mixed. In some straight line programs, irrespective of the input, there is negligible variation seen in IC or CPI. Using these observations, a scatter plot of CPI versus IC is generated using large number of CPI, IC samples. A curve is fit over these points and extrapolated up to theoretical upper bound on IC. The product of theoretical upper bound on IC and the corresponding CPI is found to be more accurate than the product of maximum IC and maximum CPI in many cases. On advanced architectures, correlation is observed to give 50-60% improvement in accuracy of WCET compared to *Chronos*.

The prime advantage of viewing execution time in terms of CPI is that it enables us to make use of program phase behavior that refers to a phase like variation of program CPI during execution. On close observation, this kind of CPI variation is determined by the way in which instructions are executed in a program. We use existing algorithms which statically decompose

the program into regions that exhibit homogeneous phase behavior. The worst case execution time of each phase is estimated by a product of the phase worst case IC and phase worst case CPI. The individual worst case execution times of the phases are combined together with the information about the worst case occurrence of these phases to yield the overall program worst case execution time. Variation of CPI within a phase is repetitive and homogeneous. On an average, the coefficient of variation of CPI within a phase is within 10% of mean CPI. As a result, one can capture CPI information of a phase with very less instrumentation (1 instrumentation point in every 100-1000 instructions). Less instrumentation is desirable in any measurement based WCET analysis technique as it implies minimum intrusion in the measurement process. With the instrumentation ratio being low, we can resort to even source code level instrumentation, which will result in negligible overhead(up to 2.2% using performance API such as PAPI).

The homogeneous variation of CPI within a phase implies that we can obtain tight confidence intervals of CPI associated with a probability using a simple probabilistic inequality like Chebyshev inequality. We shall see that using the theoretical upper bound on IC and the probabilistic upper bound on CPI, we can derive a probabilistic bound on the program WCET. In some programs, there are points where CPI variation is quite high. Using Chebyshev inequality as is, results in highly pessimistic upper bounds on CPI. We hence propose a mechanism to isolate such points of high CPI variation and divide phases into smaller sub-phases by defining a PC signature that codifies executed paths concisely and is obtained using profiling. This process of refining phases is observed to bring down the variance of CPI within a sub-phase and hence tighten the CPI bounds(9-33%). In some programs, refinement based on signature is not successful in bringing down the CPI variance. For such programs we describe a method wherein sub-phases are further refined based on allowable CPI variance within a sub-phase which is user controlled leading to further improvement in accuracy of WCET(13-52%).

The proposed probabilistic WCET analyzer is compared with *RapiTime*, a commercial probabilistic measurement based WCET analyzer. At a probability of 0.99, *RapiTime* with the highest level of instrumentation(FULL), estimates WCET with 10.6% more accuracy compared to our WCET estimate obtained by refinement based on PC signatures. However, with further refinement based on controlling CPI variance of a subphase to 50% and 10% of its original value, the accuracy of our WCET estimate improves by 18% and 32% compared to *RapiTime*. Use of

program phase behavior enables us to achieve this result with only 12% of the instrumentation points used by *RapiTime*. WCET analysis based on signatures takes only half the time taken by *RapiTime* using FULL instrumentation. Further refinement based on controlling CPI variance takes about 3/4ths of the time taken by RapiTime.

Since theoretical worst case IC computation is independent of worst case CPI computation, the two processes can be parallelized, reducing overall analysis time significantly, unlike many state of the art methods that carry out structural analysis and architecture modeling/measurement of execution time together. The phases can themselves be analyzed in parallel as the results of analysis of one phases is independent of the results of analysis of the other phases. Running a parallelized version of our technique on one of our programs, the serial version of which is 4 times slower than RapiTime, we observe a speedup of a factor of 5.5 with 8 threads.

The homogeneity of CPI variation within a phase also helps in estimating worst case remaining execution time well before time for a particular (program, input) pair. This information is very useful in a situation where the time between the availability of the result of a program and the usage of the result is quite high. Energy consumption can be reduced by reducing the processor frequency in such a case. Early availability of an accurate estimate of the remaining execution time prevents hoarding of resources for longer than needed and helps in better resource utilization.

Contents

List of Tables

List of Figures

Keywords

WCET Analysis, Measurements, Cycles Per Instruction, Profiling, Control Flow Graph, Integer Linear Programming, Bounds Estimation, Soft Real-Time Systems, Program Phase Behavior, Worst Case Remaining Execution Time, Hardware Performance Counters

C.3[Special-Purpose and Application-Based Systems]:Real-time and embedded systems; C.4[Performance of Systems]:Measurement techniques

Notation and Abbreviations

API Application program interface

CFG Control Flow Graph

CPI Cycles per instruction

COV Coefficient of variation

ET Execution Time

ETP Execution Time Profile

EVT Extreme Value Theory

IC Instruction count

IID Independently and identically distributed

ILP Integer linear programming

IPET Implicit Path Enumeration Technique

IPG Instrumentation Point Graph

MC/DC Modified condition/Decision coverage criterion

MIC Observed maximum instruction count

PC Program Counter

RISC Reduced Instruction Set Computers

SRS Simple Random Sampling

SWIC Theoretical upper bound on IC, computed statically

TLB Translation lookaside buffer

WCET Worst Case Execution Time

WIC Worst case IC

WCPI Worst case CPI

Chapter 1

Introduction

Real-time systems pervade several aspects of modern life such as household appliances, air traffic controllers, medical systems, robotics, ticket reservation systems, video games and defense systems. These systems operate within the constraints of time. In such systems, the availability of results within the allotted time is as important as the logical correctness of the results themselves. In real-time systems and embedded systems, where time is a critical resource, estimating program worst case execution time (WCET) is an important problem. In such systems, it is vital that a part or whole of the program executes within a specified time limit. If a program is not able to produce the result within the allotted time, it is said to miss its deadline. If the program WCET is greater than the specified time limit then the program has to be recoded or the architecture has to be redesigned to meet the specified time limit.

Realtime systems are broadly classified into two kinds. Hard real-time systems are systems where all programs must strictly meet its deadlines. Failure to do so can compromise the integrity of the system itself and can cause grave damage to life and property. Examples of such systems are autopilot, navigation control systems built into aircrafts, air traffic controllers and automobile control programs. On the other hand, in soft real-time systems, the ability for a program to meet its deadline is only a desired property and such systems can tolerate a few deadline misses. Typical examples are multimedia and telecommunication systems. Soft real-time systems are generally driven by human perception. As a result a few misses do not cause the user to observe a significant change in the behavior of the system. In the case of soft real-time systems, performance is important. As a result, accuracy of the WCET estimate assumes higher priority than safety. If WCET is estimated to be much higher than the actual

WCET, hoarding of resources ensue even when there is no real need, thus causing ineffective resource utilization leading the system to perform poorly.

The execution time of a program is dependent on its input and the underlying system architecture on which the program runs. Each program is associated with a theoretical upper bound on its execution time on a given architecture, which is the worst case execution time, popularly referred to as WCET. Information about WCET also helps in an efficient scheduling of resources in a situation where several programs are executed in sequel. The worst case input is defined as the hypothetical input that causes the program to execute for the theoretical upperlimit of time. Since building the set of all possible inputs is computationally hard, the earliest attempts of estimating WCET involved working with a representative set of test inputs, executing the program with each one of them and multiplying the maximum observed execution time by a predetermined factor. However such an estimate can get too pessimistic to be even usable and hence much more informed methods are needed to obtain a reasonably accurate estimate.

The WCET of a program is influenced by two main factors.

1. The number of instructions executed, determined by the static program structure.
2. The time taken by instructions to execute, determined by the underlying system architecture.

In early microprocessors, the cost of executing all instructions take the same time. But with the introduction of the pipeline mode of execution, each instruction takes a variable amount of time depending on the preceeding and succeeding instructions. Processor complexity further increases with the introduction of components like cache memories that are introduced to mitigate the delay in fetching instructions and data from the main memory. The presence and absence of the instructions or data in the cache have a significant influence on the estimated WCET of the program[65]. This is further exacerbated by components such as branch predictors that track program history to predict branch targets early. As a result, one can no longer analyze an instruction in isolation.

Estimating WCET accurately is hence a computationally hard problem owing to the large size of the possible inputs and the complexity of the underlying system architecture. There is a

possibility that the estimated WCET is much larger than the actual WCET. Such an estimate is said to be pessimistic. During the course of exhaustive testing, if one encounters a test input where the maximum measured execution time is greater than the estimated WCET then the estimated WCET is said to be unsafe. A safe estimate is always greater or equal to the actual WCET. The confidence in a WCET estimate being safe increases as the program is used for a longer time. An estimate closest to the maximum measured execution time during exhaustive testing is said to be tight and ensures optimal resource allocation.

1.1 Traditional Ways of Estimating WCET

A tool or method or algorithm that estimates WCET is called a WCET analyzer. A WCET analyzer is developed for a given system architecture. There are two main schools of thought for estimating WCET- Static analysis and Measurement based analysis. The static method estimates WCET of a program without actually running the program on the particular hardware architecture. The measurement based method executes the program on a simulator or a real system architecture for large number of inputs and uses these measurements for analysis and estimation of WCET.

1.1.1 Static WCET Analysis

In static method, instead of analyzing the program as a whole, the program is split into smaller components. The components can be basic blocks or groups of basic blocks. The execution time of each component, irrespective of the input, is computed based on an analytical model of the underlying architecture developed specifically for this purpose. The component execution times are combined appropriately, using well known techniques like integer linear programming (ILP), tree based schema, graph algorithms, to give the overall program WCET[65]. Since static analysis does not have runtime information and since details about the architecture components might not be available, the analysis has to make certain conservative assumptions about the architectural state at various points in the program which can result in a pessimistic WCET estimate.

If static analysis is performed on a sound architectural model, they are theoretically guaranteed to be safe. However they are not guaranteed to be tight. The static WCET estimate

suits hard real-time systems where emphasis is more on safety than tightness. Popular static WCET analyzers include the commercially successful tool aiT[111] and several research tools developed by universities such as BoundT[112], SWEET[6, 8] and Chronos[94]. Static analysis will be described in detail in Chapter 2.

1.1.2 Measurement-Based WCET Analysis

Measurement based analysis usually involves measuring the cost of executing the parts of the program either on a system simulator or directly on the system architecture. The measurements can be carried out at the level of paths[66], basic blocks[29] or groups of basic blocks[55]. A popular way of combining these costs has been through of the use of an ILP framework, tree based schema or graph algorithms, similar to static WCET analysis to give the final estimated WCET. Measurement based WCET analysis carried out thus are also known as hybrid WCET analysis as they make use of both measurements and static analysis. A major concern in any measurement based method is the coverage provided by the test input set. The set of inputs selected should be such that all the likely paths that may contribute to WCET are covered. Higher the path coverage, higher is the accuracy of the WCET estimate. RapiTime[102], a commercial timing analysis tool is an example of a popular measurements based WCET analyzer. Due to the availability of information at runtime, no conservative assumptions need be made and the possibility of the WCET estimate being closer to the actual WCET is higher. The other concern in measurement based WCET analyzers is that the process of measurement and the amount of instrumentation should be such that it should not alter the very timing of the program that is being analyzed[3].

Statistical measurement based methods generally try to fit a model over measured execution times obtained by running the program with a large number of inputs. The curve is extrapolated to achieve estimates of WCET depending on the probability at which the estimate is desired[38, 97, 98, 71]. It should be noted that only a statistical estimate of WCET can be derived in this case but not the theoretical upper bound. However, if the tail of the distribution is heavy, the extrapolated WCET can asymptotically tend to infinity, leading to a very pessimistic estimate of WCET especially if tail end estimates are required. Further, the parameters derived from the distribution CDF need to be themselves validated as any set of measured samples could have missed "the worst case input". For this reason, both measurement based WCET analyzers

and statistical WCET analyzers are more suited for soft real-time systems where the emphasis is more on tightness than on safety.

The estimates obtained both by any kind of WCET analysis need to be validated before they are used. Since actual WCET is unknown, the estimated WCET needs to be ultimately compared with maximum observed WCET obtained through measurement on that system, which actually forms the lower bound of the actual WCET.

1.2 Objectives of this Research

Using a purely static approach to estimate WCET is associated with certain issues. The absence of runtime information compels us to make certain conservative assumptions about runtime behavior which can lead to pessimistic WCET estimates. The effort to model the underlying system architecture is a complex task and porting the model on to a different architecture is not a trivial task. One of the major concerns in measurement based techniques is instrumentation overhead. Deciding where to place instrumentation points and having minimum number of instrumentation points is a challenging task[2]. The instrumentation should be nonintrusive and should not affect the very timing of the program which is being measured. Statistical techniques need to validate the model which is fitted over the measured execution points.

Earlier microprocessors were based on Von neumann architecture where instructions were stored in memory. The processor would fetch instructions one by one sequentially and execute them. Each instruction would take a certain amount of cycles to execute. Computing WCET of a program assuming a Von Neumann architecture is straightforward. One has to simply multiply the maximum occurrence of each instruction with the cycles it takes to execute and do this for all the instructions in the program to obtain WCET.

However today's microprocessors are far more complex than Von Neumann machines. Most real-time systems and embedded microprocessors are dominated by RISC (Reduced instruction set computing) type of machines[46] that involve a pipeline of a number of stages ranging typically from 5 to 10. Each instruction goes through these pipeline stages before it completes execution. Due to the presence of a pipeline, several instructions could be executing at a given point of time. Hence in such systems, it is more meaningful to talk of the average number of cycles an instruction can take (also known as CPI) rather than individual cycles taken by an

instruction. The execution time of a program is a product of the number of instructions it executes (instruction count or IC) and the average CPI of the program.

In this thesis, we propose a WCET analyzer that treats program execution time as a product of the instruction count(IC) and cycles per instruction(CPI). This factorization uncovers an inherent correlation between IC and CPI that can be used to improve the accuracy of WCET estimate. For a specific class of programs, that exhibit phase behavior, we can fine tune the accuracy of WCET estimate further by estimating WCET of the program in terms of its individual phases. Phases also help in reducing instrumentation required. The technique is modified to also provide a probabilistic WCET estimate, as a result, one can obtain WCET estimates at the desired probability value depending on the criticality of the application.

We target our research towards soft real-time systems. Three representative architectures of varying complexity are studied (Simplest which has only an instruction cache and no data cache, Inorder_complex which is an inorder pipeline and has both instruction and data cache and complex which is an out of order pipeline and has both instruction and data cache). For evaluation, the standard benchmarks taken from the WCET project suite[108] and embedded benchmarks[109] are used. In addition to this, we also evaluate our technique by applying it to DEBIE-1, a real-life space application developed by Space Systems Ltd, Finland[106]. The proposed method is evaluated by comparing the results with the static WCET analyzer, Chronos[94] and the commercial measurement based WCET analyzer, RapiTime[102].

1.3 Our Contributions

1. Our first contribution is that we present a fundamental timing model that estimates program WCET as a product of a maximal function of IC and maximal function of CPI. All further enhancements presented eventually build on this model. By static structural analysis, we compute the theoretical upper bound on IC. If adequate coverage is achieved, we could use maximum observed instruction count in place of theoretical upper bound on IC. CPI forms our measurement parameter. We measure average CPI of a program using several representative test inputs. Employing several analytical and statistical functions on these CPI samples, we estimate WCET. When using an analytical function of maximal CPI (maximum of average CPI), Chronos[94] estimates WCET with greater

accuracy (by 8.9%) on Simplest architecture. While on Inorder_complex and complex architectures, the WCET obtained using analytical function of maximal CPI is 38% and 51.7% more accurate than Chronos. When using a statistical function of the maximal CPI (99th percentile CPI), Chronos estimates WCET with greater accuracy (by 36.5%) on Simplest architecture. While on Inorder_complex and complex architectures, the WCET obtained using analytical function of maximal CPI is 10.6% and 29.3% more accurate than Chronos.

2. It is observed that the IC and CPI values, that have been collected over runs of a program with several inputs, are correlated. We find five kinds of correlations- direct correlation, inverse correlation, programs where irrespective of the input, there is no variation in IC and CPI, programs where with increasing IC, CPI saturates to a particular value and finally, programs that show a random correlation between IC and CPI. Our second contribution is that we show how this correlation helps us optimize our previous WCET estimate which is a product of maximal IC and maximal CPI. Using the correlation, we can estimate an optimal CPI corresponding to maximal IC and use that instead to estimate WCET. On Simplest architecture, Chronos estimates WCET with 4.7% more accuracy than WCET estimated using a product of maximal IC and optimal CPI. On Inorder_complex and complex architectures, the product of maximal IC and optimal CPI is 49% and 62.3% more accurate than Chronos. Apart from increasing accuracy in estimated WCET, correlation information helps reduce test resources in case of programs where IC and CPI are not found to significantly change with different inputs. Depending on the kind of correlation, benchmarks can be classified into groups such that one benchmark from each group can be studied in detail.

3. Our third contribution is that we use phase behavior observed in many programs to build a measurement based WCET analyzer that estimates WCET with greater accuracy with less instrumentation. The basic timing equation is modified to estimate WCET of a program in terms of its phases. Considering phase wise CPI makes the WCET more accurate than considering overall program CPI. Phase behavior manifests itself in two ways. Firstly, the variation of CPI within a phase is homogeneous and repetitive. This behavior is seen due to the manner in which instructions execute. It is observed that

in most programs, some instructions are executed more number of times than the rest. This is due to the presence of programming language structures like loops. If we examine the instructions in the pipeline belonging to a loop, we see repeatedly similar patterns of instructions occurring over time. This results in the CPI changing in a repetitive pattern while a loop is being executed, as the same set of instructions is being executed in every loop iteration. As a result, there is little variation in the individual CPIs across various iterations comprising the same loop. Secondly, CPI varies in a phaselike pattern with each phase exhibiting a distinct pattern. This occurs in situations where programs are made up of several sub-tasks. A sub-task can be either a loop or a procedure or even a large set of functionally related instructions. A program proceeds to execute instructions pertaining to each sub-task in an orderly manner. CPI varies in such a way that the coefficient of variation of CPI within each phase is quite less compared to the coefficient of variation of CPI across phases. Using average CPI per phase instead of average CPI of the whole program in the timing equation increases the accuracy of WCET estimate. Hence the problem of estimating WCET can be thus sub-divided into problems of estimating phase WCET and then combining the individual phase WCETs, factoring in their maximum occurrence frequency to give overall program WCET.

On Simplest architecture, phase information brings the WCET estimate very close to Chronos (1.73% higher than Chronos). On Inorder_complex and complex architectures, using phase information the WCET estimates are 43% and 55.3% more accurate than Chronos. Phases have important implications on the instrumentation aspect of WCET analysis. Other standard measurement based methods employ instrumentation at the level of basic blocks or a group of basic blocks. The homogeneity in the variation of CPI allows us to instrument the program at the granularity of thousands of instructions without causing a significant impact on the accuracy of WCET.

4. The homogeneity of CPI within a phase allows us to use simple probabilistic inequalities to bound phase CPI. Using this concept, we modify the hybrid WCET analyzer to estimate WCET probabilistically. A probabilistic WCET estimate is much more useful than an absolute WCET estimate as depending on the criticality of the application, one can choose the corresponding WCET estimate at the desired probability level. In this thesis, we estimate WCET at three probability values p=0.9, 0.95, 0.99. Across all benchmarks

considered in this thesis, on an average we have found the variation of CPI to be within 10% of the mean. Using this fact, we estimate probabilistic bounds of CPI within a phase using a very basic probabilistic inequality the Chebyshev inequality, which gives us tight upper bounds when variance is small. We prove that a probabilistic WCET for the whole program can be obtained using theoretical upper bound of phase IC and probabilistically bounded phase CPI. This forms our fourth contribution.

5. Our fifth contribution is that we introduce a PC signature, that detects phases at a much finer level than conventional phase detection techniques. PC signatures codify paths in a compressed manner. We describe a way to collect them using profiling. There exists programs in which CPI variation is high at certain points, as a result, the Chebyshev bounds computed as is, are quite large compared to the mean CPI. This results in pessimistic WCET estimates. The PC signatures help isolate the points of high variation of CPI bringing down the variance of CPI within a sub-phase and hence tightening the corresponding CPI bounds. We also describe a method to refine such sub-phases into smaller sub-phases based on allowable CPI variance within a sub-phase that can be specified by the user. At p=0.99, using signatures, the average pessimism of WCET estimates across all benchmarks improves by 9%, 23% and 33% compared to estimates obtained by Chronos on Simplest, Inorder_complex and complex respectively. Further refinement based on controlling CPI variance within a sub-phase to 50%, 10%, 5% and 1% of its original value yields 12.9%, 13.1%, 13.1%, 13.1% improvement on Simplest architecture. On Inorder_complex and complex architectures, the corresponding improvements are 38%, 40%, 41%, 43% and 46%, 47%, 50%, 52%.

   Compared to RapiTime, the average pessimism of WCET obtained by our technique based on PC signatures across all benchmarks at p=0.99 improves by 7% when programs are instrumented at START_OF_SCOPES granularity. Program phase behavior helps us to achieve this with only 10% of instrumentation points used by RapiTime. Further refinement based on controlling CPI variance within a sub-phase to 50%, 10% and 5% of its original value yields an improvement of 37%, 49% and 51% respectively. Any further refinement yields marginal improvement. WCET analysis based on signatures takes about 3/4ths of the time taken by RapiTime using START_OF_SCOPES. Further refinement based on controlling CPI variance takes 30% more time than RapiTime.

When RapiTime instruments at FULL granularity, the average pessimism obtained by our technique based on signatures is more pessimistic by 10.6%. However further refinement based on controlling CPI variance of a sub-phase to 50% and 10% of its original value yields 18% and 32% improvement. Any further refinement yields marginal improvement. Use of program phase behavior enables us to acheive this result with only 12% of the instrumentation points used by RapiTime. WCET analysis based on signatures takes half the time taken by RapiTime using FULL instrumentation. Further refinement based on controlling CPI variance takes about 3/4ths of the time taken by RapiTime.

6. We also present an implementation of this technique on a native platform. A simulation is atleast 10 times slower than native execution. Hence gathering CPI traces takes time for programs that execute for longer time. In such cases, we can benefit from native execution wherein large traces can be generated within a few seconds. Since CPI is a very important performance parameter of a system, there exists hardware support in most machines to measure CPI with minimal intrusion. We use PAPI, the popular performance API to access hardware performance counters. On an average, the overhead due to measurement of CPI using PAPI is found to be 2.2%.

7. Lastly, we demonstrate that apart from requiring minimal instrumentation, phases offer several advantages. The time to estimate WCET using the phase based technique can be easily parallelized by analyzing different phases in parallel. Applying this technique to Dijkstra, we found an improvement in the time taken for WCET analysis based on refinement with respect to PC signature by a factor of {1.98, 3.68, 4.7, 5.5} with {2 threads, 4 threads, 6 threads, 8 threads} respectively. The homogeneity of CPI within a phase can be used in estimating the worst case remaining execution time of a program run with a specific input well before the program finishes execution. Predicting execution time early prevents holding onto resources for a longer time and leads to better resource utilization.

1.4 Organization of this Thesis

This thesis is organized as follows. In Chapter 2, we begin with a brief background on WCET

analysis. This includes the various factors that affect program WCET, the desirable traits of a WCET analyzer and the various challenges one faces in estimating WCET. This is followed by a detailed survey of existing WCET analysis methods and sets the context of the proposed work. In Chapter 3, we begin by describing our experimental framework in terms of the architectures studied, the details of the benchmarks used and their input configurations. We describe our fundamental timing model which forms the basis for all the forthcoming chapters. We also describe how we compute the theoretical upper bound on the number of instructions executed (IC) using integer linear programming. This bound is combined with several analytical and statistical functions of measured CPI to give various WCET estimates.

In Chapter 4, we shall see that in many programs, there exists a correlation between overall IC and CPI. Five classes of correlation are observed. The correlation information is used to improve upon the WCET estimated in Chapter 3. In Chapter 5, we shall examine program phase behavior in detail and how phase information improves worst case execution time estimates in programs exhibiting phase behavior. The basic timing equation that estimates whole program WCET described in chapter 3, is now modified to estimate phase WCET, which are combined appropriately factoring in the maximum frequency of occurrence of phases to give us overall program WCET.

In Chapter 6, we shall describe a probabilistic model using which we obtain probabilistic bounds of phase CPI using Chebyshev inequality. We introduce the PC signature and describe the method of classifying a phase into smaller sub-phases based on PC signatures. We also describe how one can refine a sub-phase further into smaller sub-phases based on allowable CPI variance within a sub-phase. Using probabilistic phase CPI bound, we describe a derivation of the probabilistic WCET of a program. We estimate WCET at three different probability values p=0.9, 0.95, 0.99. In Chapter 7, we describe the implementation of a phase based WCET on a native machine using performance API or PAPI, that allows us to access hardware performance counters lodged within the processor to obtain CPI measurement with least intrusion.

In Chapter 8, we describe other applications of phases in timing analysis. The first advantage is that the process of timing analysis in itself can be parallelized as each phase can be analyzed in parallel. Secondly, the homogeneity of phase CPI can be used to predict well in advance, the worst case remaining execution time of a run of a particular (program, input) pair. This information can be used in preventing hoarding of resources for a longer time. We

summarize our work in Chapter 9 and indicate a few key directions in which this work can be extended.

Chapter 2

Background and Literature Survey

In this chapter, we first present a brief background on worst case execution time (WCET) analysis and the challenges in WCET estimation and several aspects with respect to its usage. Then we shall review various approaches to WCET analysis and solutions that have been proposed to deal with the issues present in each one of them. We review measurement based WCET analyzers in greater detail as the thesis also proposes a measurement based WCET analysis method.

2.1 Background

When a given program is run with the worst case input, the program executes for the theoretical upper limit of execution time or the worst case execution time (WCET). If the worst case input were to be known *apriori*, estimating WCET is trivial and involves running the program with that worst case input. In general, it is difficult to guess the worst case input as it depends on both structural properties of the program and the underlying system architecture. Hence the standard method of evaluating any WCET analyzer has been as follows. Lets term the WCET estimate made by a WCET analyzer as 'W'. The program is executed with an exhaustive set of inputs that satisfy standard criteria such as MC/DC coverage criteria commonly used in real-time and embedded system testing[47] and cover the widest possible range of data. In some programs the likely worst case inputs are easy to determine, for example, bubble sort executes maximum instructions if the input elements are in reverse sorted order. Such inputs are also included into the test input set. The observed maximum execution time in cycles, 'M'

is noted. Ideally M should be equal to W. If W >= M, the estimate is said to be safe. If W < M, it is an unsafe estimate. If W is much greater than M, the WCET estimate is said to be pessimistic. The closer W is to M, the more accurate the WCET estimate is said to be. In the coming sections we shall review literature that discusses several aspects of test input set generation.

2.1.1 Desirable Features of a WCET Analyzer

Typically a WCET analyzer is evaluated only with respect to the accuracy of the estimate it produces. However, there are several other desirable features of a WCET analyzer that are beneficial to the user and they are enumerated as follows:

1. Accuracy: It is important that an estimate made by a WCET analyzer is accurate. If a WCET estimate is too pessimistic, this results in over-provisioning of resources. In a system where there are multiple tasks dependent on the timing of each other, a pessimistic WCET estimate of one task has a cascading effect on the component dependent on it. Accurate estimates are also called as tight estimates in literature.

2. Safety: A safe estimate is a necessity in hard real-time systems. If a component within a hard real-time system is designed with the help of an unsafe estimate, the possibility of a deadline miss increases which can result in catastrophic consequences to life and property. However in case of soft real-time systems, where a few deadline misses can be tolerated without compromising on the functionality of the system, tightness assumes more priority than safety.

3. Non-Intrusive Instrumentation: A measurement based WCET analyzer typically instruments the program inorder to measure the time taken by its components and creates a trace which will be analyzed further to estimate WCET. In such WCET analyzers, it is important that the process of instrumentation and the number of instrumentation points are both non-intrusive, so that it does not affect the very timing of the program which is being analyzed.

4. Time taken to estimate WCET: The need for fast WCET analysis depends on the application domain, where the WCET analyzer is going to be used. In systems where the architecture is already finalized, the emphasis is on the accuracy of the WCET estimate

rather than the time taken to come up with an estimate. However if the target system is yet to be finalized, quick WCET estimates are required to find the best architecture to run the program on, analysis time takes on more importance than accuracy.

5. Retargertability: If a WCET analyzer can be modified with ease to analyze programs for another architecture then the WCET analyzer is said to be retargetable. Retargetability is a desirable feature in systems that are in the design phase and the final architecture is yet to be decided and in architecture exploration studies. Such systems can benefit from retargetable WCET analyzers which can quickly provide WCET estimates on a range of architectures, so that the designer can weigh the trade-offs and make a well informed decision.

6. Scalability: A static WCET analyzer is said to be scale well if the analysis time is preferably a sub linear or or logarithmic function or linear or atleast quadratic function of the size of the code being analyzed. A measurements based WCET analyzer is said to scale well if the analysis time if preferably a sub linear or logarithmic or linear or atleast quadratic function of the execution length of the program and the trace size.

7. Computing other related information: Apart from stating only the WCET, the analyzer can provide several other valuable information regarding the program as it studies the program in depth anyway. Information such as the bottle necks lying in the code, the WCET of individual functions, the longest path that contributes to the WCET, the list of critical variables or functions that heavily influence the WCET of the program can help the designer in improving his program significantly. In certain class of applications, it is desirable to be able to compute complementary statistics such as BCET, ACET with ease. The difference between the program WCET and its BCET is termed as program jitter. In industrial applications, there has been a lot of interest in knowing the program jitter apart from just its WCET[34].

There are many other considerations to be kept in mind while designing a WCET analyzer[35]. The dependence on the availability of the program source code is one factor. If the WCET analyzer doesn't need the program source code, the program binary has to be read by a decoder and a basic control flow graph of the program has to be constructed for the purpose of WCET analysis. Introducing annotations becomes easier with the availability of source code

than rather just the binary. If the WCET analyzer needs program debug information, then it is said to have a dependency on the compiler. Many a times, the library code is often not available at the source level. In such cases, the WCET analyzer should be able to work with the library code as a black box with the provision of assigning pre-determined estimates for such code.

2.1.2 Challenges in WCET analysis

There are several challenges in structural analysis and architecture modeling while building an accurate WCET analyzer.

1. Program Structure

   WCET analysis requires certain conditions to be satisfied regarding program structure. It requires all the loops to be bounded and all the recursive routines to have a maximum depth. Failing which, the program is said to be unbounded in time. Most static tools require the user to annotate the loop bounds manually. However there are some industry standard tools like Absint[111] that can automatically derive loop bounds for many programs with simple loops. In some situations, the number of times a loop iterates depends on many factors such as input values, satisfaction of a particular condition or dynamically computed values. In these cases it is impossible to arrive at a loop bound. Hence deriving loop bounds in general, with the least manual intervention still remains a great challenge for WCET analyzers today. WCET analyzers typically cannot analyze code that contains dynamic data structures, since they are created on the fly during program execution and are difficult to model statically.

2. Infeasible Paths

   An infeasible path is a path in the program control flow graph that can never be traversed in any valid execution sequence of the program. Such paths have to be excluded during WCET analysis as they tend to inflate the estimated WCET. In addition to genuine infeasible paths, there might be certain portions of code like exception/error handling code which might not be visited during normal operation and hence need not occur in the worst case and which one might want to exclude from WCET analysis. The WCET analyzer should support excluding analysis of parts of code. More about this will be

discussed in coming sections.

3. Context Sensitivity

   An application program may have several procedures, some of them being called in more than one context. If the procedure takes a large amount of time in one context than the other, ignoring context sensitivity would again inflate the estimated WCET. Because the WCET analyzer would always assign an upper bound on the execution time of the procedure regardless of the context. Accounting for context sensitivity would distinguish the procedure calls occurring in different contexts during the analysis thus avoiding the problem of overestimation. Either procedure inlining or cloning techniques are employed to deal with contexts. Procedure call strings have also been used to deal with context sensitivity during WCET analysis[73]. More about will be discussed in coming sections.

   Another kind of context sensitivity exists in program loops. A program loop can be thought of as a series of non recursive subroutine calls of the same procedure, each call representing a different iteration. Since the program behavior can be different in different iterations, assigning the worst case behavior for all the iterations would inflate the WCET estimate. Hence this warrants separate analyses per iteration which requires loop unrolling to be performed in most of the cases[73]. A concept called VIVU or virtual inlining virtual unrolling is performed to mitigate the overhead of code expansion[18]. Healy et al[20] describes methods to detect whether branches are going to fall through or jump by analyzing assignments to variables and registers. Using this information the timing analyzer in [20] is able to detect in some programs that the longest path is taken in only half the time during loop execution and not every iteration.

4. Program Modes

   A program is said to execute in different modes if it executes different paths based on the value of certain input variables. A typical examples is the fast fourier transform popularly known as the FFT (section 3.2 of chapter 3). It has two primary modes of operation, the normal FFT and the inverse FFT. The inverse FFT executes an additional amount of code compared to the normal FFT. A mode specific WCET is more accurate than one that does not consider modes.

5. Architecture modeling

   Modeling the complexities of the underlying architecture statically is a great challenge for a static WCET analyzer. Typically vendors avoid publishing processor specifications in great detail as they want to retain some flexibility as unpublished specifications can be subject to future change, if needed[74]. Hence there is a possibility of error being introduced during translation from documentation to the model used for static analysis. Once the abstract processor model is developed, it is non-trivial to verify the correctness of the model. The workings of components like caches depend highly on the contents of program variables. Determining the exact range of values of a variable statically is almost impossible. A coarser range might be easier to derive but might result in a pessimistic WCET estimate[65].

6. Timing Anomalies

   Modern architectures comprise of several components that may interact each other in non-intuitive ways. Intuitively, it would be assumed that a locally faster execution entails a decrease of the overall program WCET. A timing anomaly occurs when a locally faster execution instead increases the overall program WCET[81, 36]. Since verifying the absence of timing anomalies is provably hard, timing analyzers are forced to consider all possible scenarios, that is, to follow execution through several successor states whenever a state with a nondeterministic choice between successor states is detected. This may lead to a state explosion. A model-checking-based automated timing anomaly identification method has been proposed [44] for a simplified processor. However, the scalability of this method for complex processors is not obvious.

2.2 Literature Review on WCET Analysis

In static and measurement based methods, the problem of estimating whole program WCET is divided into smaller problems of estimating WCET of smaller program components, which we term as units of analysis. Thus WCET analysis is comprised of three following steps.

- Structural analysis

  The structure of a program is studied by building a control flow graph out of the program binary. Depending on the method, the unit of analysis could be a simple basic block[111,

29] or a collection of basic blocks like segments[55] or scopes[6]. It could also be the individual paths in the program[66]. Loops in the program are identified and bounds are ascertained by analysis or can be provided by the user in the form of annotations. Paths that can never be executed in any valid execution (infeasible paths) could be singled out from analysis.

- Low Level Analysis

  Structural analysis only depends on the program and is totally independent of the underlying architecture on which the program is run. Inorder to estimate WCET, the effect of the underlying architecture has to be accounted for. Hence in this step, the cost of executing the analysis unit determined in the previous step is estimated either through architectural modeling or direct measurement. Architecture modeling involves statically estimating the cost of executing the analysis units taking into account, the effects of caches, pipelines etc. Alternatively, the time taken by an analysis unit can be measured either directly on the target hardware using either software calls or special hardware[73]. If the final system is not available, a simulator could be used.

- Final Estimate Calculation

  The costs of the analysis units thus computed have to be combined in accordance with the program structure to give us the overall program WCET. There are multiple ways of obtaining the combined WCET estimate. The tree-based approach [4] traverses the abstract syntax tree of a program at source code level in a bottom-up manner and computes upper timing bounds for connected code constructs. The computation of the timing bounds is steered by predefined combination rules which state how the execution times of parent constructs are derived based on the execution times of their child constructs. After the computation, the current construct is collapsed and its WCET is propagated to its parent constructs and hence ultimately to the root of the program to give the final WCET estimate.

  The path-based approach [66, 6] models each possible path in the control flow graph explicitly. For each of these paths, the cost of executing the path is computed. The length of the longest path together with the cost of executing the instructions appearing in the path together contributes to the execution of the path. That path which has the

maximum execution time is the worst case path.

The implicit path enumeration technique (IPET) [95] does not explicitly enumerate program paths but rather implicitly considers them in its solution. This is accomplished by converting the problem of determining the bounds to one of solving a set of integer linear programming (ILP) problems. The CFG is transformed into a set of linear constraints. The equations warrant that the traversal counts of all incoming edges of v are equal to the traversal counts of all outgoing edges of v. The problem to be solved is to maximize the overall flow through the CFG. The objective equation to be maximized is a sum of terms where each term is a product of frequency of execution of the analysis unit (variable which is to be maximized) and the cost of executing the analysis unit on the target.

2.2.1 Static WCET Analyzers

Static WCET analyzers analyze the timing properties of the program without actually running them on the target system. For this reason, it has to develop an abstract model of the system and analyze the effect of executing each instruction on the developed model. Once the cost of executing the components of the program on the given architecture, are derived, the final WCET estimate of the whole program can be evaluated using well known techniques like IPET[111], tree based schema[4] or explicit path enumeration[6]. We now discuss the steps involved in deriving the architectural cost at the component level.

1. Value Analysis

Caches are one of the most important components in the processor introduced to overcome the delays in fetching instructions and data from main memory. In order to capture the impact of the cache on the program timing, the variables in the program need to be mapped to a possible range of values which is the function of the step value analysis[16]. Measurement techniques work with real program data and hence need not carry out value analysis. But a static analyzer is expected to work for all valid inputs. Although the exact values of variables are available only at runtime, manyatimes the range of values can be determined in well written code statically[65]. Value analysis also helps in determining loop bounds and identifying infeasible paths in the program[33].

2. Control Flow Analysis

Static WCET analyzers work on the program binary representation directly and hence need to gather the control flow structure of the program themselves. For this reason, a control flow graph or the CFG is constructed. The nodes of the CFG are termed as basic blocks. The set of straight line sequence of instructions until a control flow instruction such as a branch or a procedure call or a return is encountered, forms a basic block. Hence more the number of branching edges in the CFG, more complex is the program control flow. The CFG forms the fundamental structure that will be used to derive the overall WCET estimate for the whole program. In order to identify the longest path the program could possibly take, one has to first build a control flow representation. Control flow analysis determines information about the possible flow of control through the task to increase the precision of the subsequent analyzes[65]. Often, value analysis precedes control flow analysis and assists control flow analysis by providing annotations regarding infeasible paths and loop bounds.

3. Architectural modeling

The execution time of an individual instruction, even a memory access depends on the execution history. To find precise execution-time bounds for a given task, it is necessary to analyze what the occupancy state of these processor components for all paths leading to the task's instructions is. Processor-behavior analysis determines invariants about these occupancy states for the given task[65]. Information about the processor states is derived by analyzing potential execution histories leading to this instruction. Different states in which the instruction can be executed may lead to widely varying execution times with disastrous effects on precision. For instance, if a loop iterates 100 times, but the worst case of the body, e_{body}, only really occurs during one of these iterations and the others are considerably faster (say twice as fast), the over-approximation is 99 * 0.5 *e_{body}[65]. Precision can be gained by regarding execution in classes of execution histories separately, which correspond to flow contexts. These flow contexts essentially express by which paths through loops and calls control can arrive at the instruction. Wherever information about the processors execution state is missing a conservative assumption has to be made or all possibilities have to be explored.

Most approaches use Data Flow Analysis, a static program-analysis technique based on the theory of Abstract Interpretation [21]. These methods are used to compute invariants, one per flow context, about the processors execution states at each program point[17]. If there is one

invariant for each program point, then it holds for all execution paths leading to this program point. Different ways to reach a basic block may lead to different invariants at the blocks program points. Thus, several invariants could be computed. Each holds for a set of execution paths, and the sets together form a partition of the set of all execution paths leading to this program point. Each set of such paths corresponds to what sometimes is called a calling context, context for short. The invariants express static knowledge about the contents of caches, the occupancy of functional units and processor queues, and of states of branch-prediction units. Knowledge about cache contents is then used to classify memory accesses as definite cache hits (or definite cache misses). Knowledge about the occupancy of pipeline queues and functional units is used to exclude pipeline stalls.

The different components of modern architectures like caches, pipelines, branch predictors, interacting together, can cause timing anomalies that either have a constructive influence or a destructive influence on the timing aspect of the overall program[81, 36]. Hence conservatively assuming a local worst case would not work in the presence of timing anomalies.

4. Final Estimate Calculation

Having determined the cost of executing components of a program, the final estimate is derived by combining them in a meaningful way to come up with an upper bound for the WCET. This bound will never be crossed by any execution of the program. The components differ depending on the different mechanisms used. For example, structure based methods evaluate the cost at the statement level. These statements are then combined according to the syntax rules and the cost of the program is determined in a bottom up fashion. Heptane[10] uses structure based analysis and incorporates tree based schema to combine the estimates of smaller units. Heptane models the architecture, specifically the I-cache, branch predictor and the pipeline in such a way that it is easily retargetable. However it doesn't support infeasible path detection which can cause a slight pessimistic estimate. Heptane doesn't model the data cache.

The components could be program paths. Each path could be analyzed separately and a bound could be calculated, wherein the upper bound would be the maximum of the bounds of all the paths. Paths are used in the research prototype tool developed by the FSU, North carolina and the Furman universities[24, 63]. The tool can also be used to obtain parametric bounds in terms of program parameters rather than an absolute numeric bound for a given

problem size. Modeling of data cache is limited in that, only a certain kind of access patterns are allowed. The mechanism can have problems in scaling to large systems. The parametric WCET for a program, for a given size can be got by substituting the corresponding program variables in the parametric equation[67, 23]. Having parametric bounds avoids high pessimism by using the actual values with which the program is going to run rather than some hypothetical value. Further, one can get WCET estimates for a range of input values without having to rework the entire WCET analysis.

AbsInt[111], a commercial static WCET analyzer makes use of the implicit path enumeration technique framework to derive the final estimate. The costs of each basic block on the target system are derived by abstract interpretation[21](Interpreting each instruction of the program by carrying out value analysis of the variables in use for each instruction). The modeling of in order and out of order pipelines are supported in addition to modeling of instruction and data caches[15].

Chronos[94] is an open source static WCET analyzer that uses IPET to estimate WCET. It models both instruction cache and data cache apart from a multilevel unified L2 cache, along with both inorder and out of order pipelines apart from dynamic branch prediction. The input to Chronos is a program written in C and the configuration of the target processor. The frontend of the tool performs data flow analysis of loops to derive bounds. For complex loops, the user is required to manually annotate loop bound information. The user may also provide annotations to compute infeasible paths although it is done automatically in Chronos. The binary is disassembled and a CFG is created. Chronos performs processor behavior analysis on the CFG. The core of the analyzer determines upper bounds of execution times of each basic block under various execution contexts such as correctly predicted or mispredicted jump of the preceding basic blocks and cache hits/misses within the basic block[94].

Chronos employs integer linear programming (ILP) to model branch prediction, instruction cache as well as their interactions. The analysis of branch prediction is generic and parameterizable with respect to commonly used dynamic branch prediction schemes. Using address analysis techniques, the presence of data in the cache is statically predicted at every point in the program using abstract interpretation techniques[17, 76]. Using these information, the cost of executing each basic block is statically determined. Finally using IPET, the WCET is estimated factoring in the infeasible path information and various flow constraints of CFG.

Bound-T[112], another commercial static WCET analyzer concentrates mainly on program path analysis and does not model cache, complex pipeline or branch prediction[72]. In path analysis, Bound-T focuses on inferring loop bounds for which it uses the well known Omega calculator[93].

SWEET[8, 37] uses a higher abstraction of the CFG namely the scope graph[6]. A scope can be informally defined as a group of basic blocks having variables in common. A scope can represent a statement, a loop, or even a small routine. A node in the scope graph represents multiple related basic blocks. Edges between nodes in different scopes in the original control flow now figure in the scope graph. The edges between the related basic blocks in the same scope are contained within the scope. Scope based WCET evaluation reduces the complexity of IPET by focussing only on the scopes and constructing the linear equations at the scope level instead of the usual basic block level. Further since scopes are clustered by considering control flows across scopes such that related scopes exist in the same cluster, the accuracy is maintained at a good level.

Flow facts[8], that are a set of rules denoting control flow information like the loop bounds, infeasible path information etc carry and communicate flow analyses between scopes. SWEET uses program slicing to analyze only those parts of the program that may affect the program flow. This is useful to incorporate modes if the program mode is known apriori. Loops are unrolled and each iteration is analyzed separately thereby making the analysis context sensitive. The architecture modeling is carried out separately by memory analysis that involves determining the memory addresses possibly accessed by each instruction. Pipelines are accounted for by simulating object code sequences through a trace-driven cycle-accurate CPU model. The final estimation in SWEET can proceed in either of the three ways-A fast path based technique or a global IPET technique or a hybrid clustered technique involving scope graphs. SWEET can do automatic flow analysis provided the program is compiled with the research compiler that SWEET is integrated with. Data caches are not modeled in the tool. The pipeline structure is assumed to be in-order without any associated timing anomalies.

The prototype static analyzer from TU Vienna[60] performs WCET analysis on programs written in special version of C, called as the wcetC language. wcetC is C extended to allow users to make annotations regarding infeasible paths. The compiler generates the object code which is analyzed to derive the WCET bound. IPET is used here as the basic mechanism.

Each methodology has the share of its own merits and demerits. Incorporating additional flow constraints is difficult in structure based methods while it is simple in the case of IPET and also path based approaches. However, path based approaches cannot efficiently model flow information extending across loop nesting levels. With the introduction of every consecutive branch condition, the number of paths can grow exponentially leading us to resort to heuristic search methods. With an increase in the number of flow facts, the number of constraints can also increase. Clustered WCET techniques can be employed to maintain the accuracy at the same time reducing the computational complexity of WCET analysis.

Cinderella[96] is an ILP-based research prototype developed at Princeton University. The main distinguishing feature of this tool is that it performs *both* program path analysis and micro-architectural modeling by solving an ILP problem which renders less scalability to this formulation[65]. Also, Cinderella mostly concentrates on program path analysis and cache modeling; it does not analyze timing effects of complex pipelines and branch prediction.

Alternate methods such as symbolic execution exist which combine both analysis and architectural modeling both in one step by simulating the program on the abstract model without any input. The disadvantage is that simulation is orders of magnitude slower than execution and one has to consider possibly large set of states in the absence of input informa- tion. The prototype tool from the Chalmers university of Technology employs the method of symbolic execution[80].

2.2.2 Measurement Based WCET Analyzers

Measurement based tools work by measuring the components of the program on the actual target hardware. The maximum of the observed measurements of each of the components are then combined to derive the WCET of the whole program. The process of measuring is equivalent to the step of architectural modeling in static analysis. The WCET derived by a measurement based tool cannot be called as an upper bound as it is not guaranteed to be safe. The main reasons are unknown worst case architectural state for each instruction and no knowledge of worst case input. If the inputs that are used to carry out the measurements exclude the worst case input, the resulting estimate is clearly not safe. Estimates can also be unsafe because only a subset of all the possible contexts (initial processor states) can be used before measuring each program component. This could be remedied by testing exhaustively.

But it is well known that testing all the paths of a program is a hard problem. To come up with a worst case initial processor state is not easy for complicated processors especially if timing anomalies are present. And the empty state need not imply the worst case initial state always. Static methods on the other hand are equipped in dealing with the absence of information and hence can analyze large state sets.

The method used for measuring the components of the program should be accurate and specially non-intrusive as it will affect the very timing of the program it wants to estimate. For this reason, logical analyzers or oscilloscopes could be used. However, the issue with these devices is that it is very difficult to map measurements to program paths. Most of the measurement based tools first capture the time of the various components by collecting an execution trace that contains timestamps for the execution of every component[73, 3]. This is followed by a mapping of these times to the paths in the program. Using this and the control flow information, the worst case path and hence the worst case execution time is estimated. In our work, we use hardware performance counters that form a part of current day processors to track the time of the program in terms of cycles per instruction. We shall review the various measurement based WCET analyzers in detail in this section.

Program Segments

The hybrid timing analysis tool described in Zolda et al[55] combines static program analysis techniques and execution time measurements to calculate an estimate of the WCET. The unit of analysis is a program segment that is chosen to be small enough to have limited number of paths to make test generation easier and big enough to weed out any infeasible paths and have lesser instrumentation points. The timing analyzer is designed to be adaptable to the resources that the user is willing to invest. The final WCET estimate is obtained using IPET that uses maximum execution times of program segments. Function calls are either inlined or treated as a black box. Automatic test-data generation is used to derive the required input data for the execution-time measurements. Since not every basic block gets a timestamp during measurement, the measurements are coarse. The tool has been designed with a special focus on analyzing automatically generated code, e.g., code generated from Matlab/Simulink models. The technique which we propose in this thesis is also a hybrid technique. We specifically target programs that exhibit phase behavior. Within each phase, the behavior of architectural

parameters like CPI is observed to be homogeneous and varies closely around the mean. So we factorize execution time as a product of instruction count (IC) and CPI. We present various functions to calculate worst case IC and worst case CPI. Phases are composed of hundreds to thousands of instructions and form our analysis unit. Phases are bound to static code regions and are demarcated by call loop boundaries. We use code structural analysis[39] which is detailed in Chapter 5 to detect phases.

Hybrid Clustered Technique

Apart from fast path and global IPET approach, SWEET uses a hybrid clustered technique which handles complex flows with low computational complexity but still generate safe and tight WCET estimate[6]. The cluster based technique achieves global precision at local efficiency and also considers possible interaction of these flow facts, provided flow fact units are used to construct units where all included flow facts are either directly or indirectly dependent. The technique is demand driven, WCET of a program region is calculated only when it is needed. To carry out low level timing analysis, pipeline timing analysis is performed on the scope graph that uses a cycle accurate simulator of the target to generate times. Additional annotations can give information about cache misses and so on. A mix of path based or IPET depending on the characteristics of the cluster. With addition of new flow facts, computation time increases linearly for path based analysis and non-linearly in case of IPET. Path based is not very accurate. IPET based analysis is very accurate but less scalable compared to path based technique[6].

The technique that we propose in this thesis has been implemented using IPET. Instead of scopes, we plug in phases. The objective function maximizes the number of instructions executed in a phase as we measure CPI of phases and compute worst case CPI using several functions. Our technique can be easily adapted to use path based approaches or tree based schema to compute a theoretical bound on IC.

Approximate WCET estimate

The technique presented in Corti et al[52] is also a hybrid technique that uses graph longest path search algorithm to estimate WCET. The nodes of the graph are basic blocks. The weight of an edge represents the cost of executing the preceding basic block in processor cycles. This

cost is computed by constructing an analytical equation that plugs in the values of various architectural parameters such as cache misses, stalls in the pipeline et cetera, that determine the actual cost of execution. These parameters are measured using hardware performance monitors. The technique yields only approximate estimates as event counting is not precise as they cannot be attributed to specific instructions, more so for out of order architectures. Moreover many events are not disjoint, hardware performance monitors do not report their intersection. The proposed technique measures only CPI of a phase at many points in the program using hardware performance counters. We exploit program phase behavior to accurately capture the time of a phase with less instrumentation. Since CPI of a phase varies closely to the mean, it accurately characterizes the time of a phase.

Hybrid Technique Based on Timing Schema

Colin et al [9] propose a hybrid scheme that uses simplescalar[113] to measure time of each basic block and use timing schema to compute WCET bottom up. They experiment with various architectures as it is easy to configure parameters in Simplescalar. The work identifies that the cache and the branch predictor are the two factors that influence the accuracy of the resultant WCET estimate, the most. They also look at the number of tests that need to be performed before the results start converging[9]. We share many of the observations in this work in our experiments. We also observe that, with the architecture becoming more complex, the possibility of overestimation in WCET increases. Measurement approaches work well with programs that show less variability across inputs. For such programs, it is very easy to obtain tight bounds on CPI and hence the resulting WCET estimate is guaranteed to be accurate.

Instrumentation Point Graphs

Betts et al [2] observe that when instrumentation points exist in only a few basic blocks (sparse instrumentation), these points are not sufficient to ensure coverage and lead to overly pessimistic estimates. Hence Betts advocates the use of an instrumentation point graph or IPG constructed from CFGs whose edges contain path information between instrumentation points or ipoints. The ipoints are such that they are path reconstructible. The ipoints are measured and trace is generated. The trace is then parsed and using either an ITree which is based on tree based schema and is specially designed for dealing with IPGs or the conventional IPET

applied to IPG, the final WCET is estimated. ITree is senstive to location of ipoints especially in loops. It can so happen that ipoints placed in certain locations can produce more pessimistic estimates than others. IPET is insensitive to location of ipoints. In both cases the common factor is that the ipoints are sparse. The limited memory available for trace buffers (hardware support for instrumentation) is one of the main motivations for this work. Context information is also easily integrated into the IPG as it models the transitions between basic blocks. The call graph along with IPG is used for trace analysis. Loop bounds are extracted automatically. For detecting infeasible paths, additional information has to be integrated into the analysis. The WCET calculation for procedures that are near leaves of call graphs have good precision but overestimation accumulates as the analysis moves upto the root. Hence if a procedure has too many calls that have high call depth the pessimism will be very high. Addition of frequency information to the graph helps tightens the estimate.

The work that we propose in this thesis shares the same motivation of having to deal with sparse instrumentation. We exhibit that the homogeneity and repeatability of the manner in which CPI changes in a phase can be used to reduce the number of instrumentation points within a phase without affecting the accuracy of WCET estimate. A technique which is less intrusive is highly desirable in the context of measurement based WCET analysis. Our technique integrates infeasible path analysis while computing worst case IC. Due to the phase property, instrumentation points can be placed arbitrarily anywhere within a phase unlike the technique proposed in Betts[2] where ipoint placement influences accuracy of WCET estimate. Our WCET estimate is not affected by the location of the procedure in a call graph. The technique performs uniformly for all procedures.

In [3], Betts et al describe a mechanism by which hybrid WCET analysis can still be performed at the source level when the timestamped trace has been collected at the object level by state-of-the-art hardware. This allows existing, commercial tools, such as RapiTime, to operate without the need for intrusive instrumentation and thus without the probe effect. Object level tracing eliminates need for software level instrumentation seen exceedingly in embedded processors that support hardware debugging. Hardware trace support usually can monitor only jumps as the debugger must keep pace with the rate at which trace data is produced otherwise there could buffer overflows and blackouts, as a result part of trace can be lost.

Inorder to achieve the goal stated in [3], two things need to be carried out. Firstly the analysis needs to have the ability for each inserted ipoint to use an address as its identifier. This can be done using global assembly labels which are generated for each inserted ipoint using a macro as described in [3]. The resulting addresses of the labels are then passed back to the hybrid measurement based analysis tool so that it can replace the original identifiers of ipoints. With this small step, the measurement based tool is able to parse a trace in (address, timestamp) format and therefore perform WCET analysis. The second issue is that hardware debug interfaces only record a sequence of branch destinations, and not every address which corresponds to an ipoint label. The solution is to analyse the disassembly for branch instructions and then interpolate the missing instructions from the trace, recording only those that correspond to ipoint addresses. Evidently, the technique is limited to processors that provide tracing facilities. Optimising compilers often unroll loops [3], in effect replicating the body of a loop a particular number of times and adjusting the control logic as necessary. This would generate multiple copies of a global label, which is illegal assembly, and consequently such optimisations have to be disabled. Tracing at the object-level typically results in very rapid generation of data. For a processor running at 200MHz, with an average of one branch every 10 instructions and using 64 bits to record each branch timestamp, would generate 160MB in one second.

Hence it is observed that on the one hand, source-level instrumentation provides greater flexibility and is often the most convenient, but it is handicapped by the probe effect. On the other hand, less intrusive instrumentation normally demands some type of hardware support. This forms one of the main issues addressed by this thesis. For programs that display phase behavior, our work describes mechanisms using which we can make use of homogeneity and repeatability seen in the variation of CPI within a phase to have sparse instrumentation points within a phase at the same time, accurately characterizes the time of a phase accurately. With sparse instrumentation, we can afford using source level instrumentation that introduces only 1-2% of instrumentation overhead thereby not affecting the timing of the program. In chapter 5, we shall describe more about program phases and how they help us in building a less intrusive measurement based WCET analyzer.

Independent Paths for ARM programs

The technique proposed in Liangliang et al[51] is specifically targeted towards simple ARM architectures. The analysis first removes all infeasible paths in the CFG and partitions a CFG into a set of program segments which is different from segments considered in [55]. A segment in this work is defined as a sequence of contiguous basic blocks. A new directed graph is created whose nodes represent segments. The set of independent paths or basis paths are calculated. The speciality of independent paths is that they are fewer in number compared to the total set of paths possible. Using linear combinations of independent paths of the directed graph, a set of feasible paths can be generated that gives complete coverage in terms of the program paths considered. Their timing measurements and execution counts of program segments are derived from a limited number of measurements of an instrumented version of the program. After the timing measurement of the feasible paths are linearly expressed by the execution times of program segments, a system of equations is derived as a constraint problem, from which we can obtain the execution times of program segments. By assigning the execution times of program segments to weights of edges in the directed graph, the WCET estimate can be calculated on the basis of graph-theoretical techniques. If there are acceleration features such as caches in the architecture, this equation will compute a bloated value. Hence data caches are not considered in [51]. Our technique can work with instruction and data caches seamlessly.

SymTA/P (SYMbolic Timing Analysis for Processors)

SymTA/P tool from TU Braunschweig, Germany, performs symbolic analysis on the abstract syntax tree to identify a single feasible path (SFP) which is a sequence of basic blocks where the execution is invariant of input data values[91, 92]. An SFP can reach beyond basic block boundaries in for example, a fast fourier transform program (FFT) or a finite impulse response filter program (FIR). The result of SFP analysis is a CFG with nodes containing a single feasible path or basic blocks that are a part of multiple feasible paths. Each SFP is represented as a node and is instrumented and measured. As the safe initial state is not known, an additional time delay is added to cover up a potential underestimation during measurements. The memory access trace for each node is generated and annotated to the corresponding node in the CFG. A data flow analysis is used to propagate information about the cache lines that are available at each node. Pipeline effects are not modeled directly. But a conservative overhead

corresponding to starting with an empty pipeline is assumed. The cost of execution of each node is given by the measured time and the statically analyzed cache behavior. Loop bounds have to be specified by the user if it determines control flow. A combination of symbolic execution and ILP is used to bound the worst case data cache behavior for input dependent memory accesses[65].

The measurements are more accurate if the path lengths between measurement points are longer. In simple programs such as FIR, FFT, an SFP can span multiple basic blocks resulting in fewer instrumentation points. We use phase behavior to identify repeated execution sequences to mark phases and can find such sequences in programs composed of complex structures too. Further we present a technique to refine a phase into sub-phases based on paths taken to execute to isolate points of high CPI variation within a phase. Phases are composed of thousands of instructions and likewise reduce the number of instrumentation points. In SymTA/P, if many basic blocks are measured individually, the added time delays to account for pipeline effects would lead to an overestimation of WCET[65]. If the architecture is based on an out of order pipeline, a simple added time delay might render estimates unsafe due to the presence of timing anomalies. The same holds for a simple added time delay due to unknown initial architectural state. In our work, the pipeline effects, cache effects, branch predictor effects and all other components of the architecture that can influence time is taken into account by considering a single parameter that covers them all, which is the CPI. Unknown worst case initial state is modeled in the proposed technique by running the test inputs multiple times to obtain measurements of the same input with multiple starting states[9].

Context Sensitivity and Measurement Based Analysis

In the IPET model, where the execution time for each basic block is determined either by architectural model or by measurement, accommodating context sensitivity requires additional efforts, the execution time of basic block can vary depending on its context. Zolda[56] observes that backward dependency is the more prominent case, where the execution time of a block depends on the concrete execution history. This is easily exemplified by the distinction of execution times of a block in the presence of a cold versus a warm instruction cache: In a simple setting, a certain block might be absent from the instruction cache during the first iteration of a loop, but present during all subsequent iterations. A distinction of execution

times of the block can then be based on whether the loop body is entered via the back edge, or from outside. In a different work[54], Zolda et al considers the objective function as a sum of terms. Each term is a product of the frequency of the basic block (variable) and cost of executing the block on hardware. Depending on the past execution context, the frequency variable is split into different other variables associated with a different cost that is dependent on its execution history. For lower cost paths, there might be other inputs that take higher times along those paths, hence can introduce optimism. For this reason, model checking is used to increase coverage for each path.

Stattelmann[73] investigates the influence of the execution history on the precision of measurement. By partitioning the analyzed programs into easily traceable segments and by precisely controlling run-time measurements with on-chip tracing facilities, the new method is able to preserve information about the execution context of measured execution times. Recent developments in debug hardware technology allow the creation of cycle-accurate traces with a fully programmable on-chip event logic which is used in [73] to gather accurate traces. The method works on the interprocedural control flow graph (ICFG) of a program executable and requires measurement hardware that can be controlled by complex trigger conditions. The approach in [73] was motivated by the limited size of trace buffer memory which is available in current hardware for on-chip execution time measurements. Due to bandwidth constraints these traces only store certain instructions, for example taken branches. Additionally, timestamps for these instructions are often only created when a partial trace is transferred from a small on-chip buffer to the large external memory. Hence deriving the execution time of every single instruction is difficult.

The ICFG is created and partitioned into program segments in such a way that every possible run through the segments can be measured with the available trace buffer memory. Information gathered during the partitioning phase is used to generate trace automata that will control the measurements. After the measurements have been taken, the context-sensitive timing information for each basic block of the program can be extracted and annotated to the ICFG. Further computations then yield the worst-case path through the ICFG and an estimate of the worst-case execution time. Virtual inlining virtual unrolling applies function inlining and loop unrolling on the level of the ICFG. Thus the ICFG can contain several copies of the same basic block for which different execution times can be annotated.

A call string is used to model a routine's execution history. Call strings can be seen as an abstraction of the call stack that would occur during an execution of the program. In this work, a call string will be represented as a sequence of call string elements. The intuition behind this representation of an execution context is that whenever a routine is called, the call string is extended with another element to describe the context of the function body. Therefore extending the call string works similar to extending the call stack during program execution. Since the execution history of a routine can be very complex, its call string representation might become very long. In order to achieve a more compact representation of execution contexts, the maximal length of call strings will be bounded by a constant. For call strings which describe a valid execution but exceed the maximal length, only the last k call string elements will be used.

Additional precision is gained by extracting loops from their parent routine and treating them like recursive routines. This allows a more precise classification of the execution history than a simple calling context when searching for the WCET path through the program, since varying execution times in different loop iterations can be represented independently from the parent routine. As the context in which a trace is generated is preserved while creating the measurements, basic block execution times from the trace are only annotated to those nodes with matching context. In case of virtual inlining, this means that execution times are only annotated to those nodes in the ICFG at the correct call site, but not to the nodes in other contexts. The design of the method allows an easy integration of static analyses to make the measurement phase more efficient. Cache analysis has been adapted to classify the instruction cache behavior of different execution contexts[73]. Nonetheless, the prototype implementation reported some WCET estimates which were smaller than the maximal execution time observed during the end-to-end measurements. One reason for this is that the measurement hardware sometimes did not start the traces immediately after they were triggered. As a result of these delays, some basic blocks were never completely covered by the measurements and thus the execution time was underestimated. The accuracy of WCET estimates will improve with exhaustive measurement and better coverage.

Predicated WCET Analysis

Marref et al[5] consider all different execution times of a basic block possible and express them as a outcome of executing some other basic blocks in the past using constraint logic

programming (CLP). The cost of execution of basic blocks is dependent on the satisfaction of certain conditions. For each basic block, at the most execution time can be constrained by five other blocks. In programs where more than five blocks determine execution time of a basic block, estimates may not be accurate. The work in [5] models the effect of only instruction caches and do not consider infeasible paths, which can be added if needed. The work proposed in this thesis can work with instruction and data caches, inorder and out of order pipelines and any kind of branch prediction scheme.

Our work deals with context sensitivity in the following way. If a procedure can be called in two different contexts, we perform procedure cloning and treat the two calls separately. We also distinguish loop iterations and avoid assigning a blanket value for all iterations together leading to a bloated estimate. We shall describe further details on this in Chapter 6.

Probabilistic WCET Analyzers

pWCET: Based on Probabilistic Hard Real Time Systems

The work by Bernat et al[29] was one of the first attempts to estimate WCET probabilistically. Instead of an absolute WCET that can occur very rarely, soft real time systems might benefit from a probabilistic WCET estimate associated with a degree of probability. Depending on the criticality of the application, WCET can estimated with the appropriate probability required. Bernat et al[29] apply timing schema and combine the worst case effects seen in basic blocks probabilistically using three different operators that are determined by the availability of dependence information among basic blocks. They define an execution time profile (ETP) for a basic block which is the range of different execution times a basic block might take to execute on the target. The ETPs of basic blocks are combined bottom up and carried upward to yield the overall combined effect of all basic blocks of the program. If the basic blocks are known to be independent, simple convolution on the ETPs are carried out. If there is a dependency between the basic blocks, a joint execution profile(JEP) is calculated based on the two dependent basic blocks. If the dependence cannot be ascertained, a worst JEP (biased convolution) which is a modified version of JEP that is consistent with the two basic blocks and that is safe, is used to combine the worst case effects probabilistically. Depending on the number of traces generated for each basic block, the time to compute joint profiles grows very quickly. Simple convolution is observed to yield optimistic estimates while

biased convolution yields very pessimistic estimates. The thesis also proposes a probabilistic WCET analysis method that probabilistically bounds phase CPI and uses these bounds to compute a probabilistic WCET estimate for the program. We shall compare our technique with RapiTime[102] which is based on the work by Bernat et al[29] in more detail in Chapter 6.

Hybrid WCET Analysis Based on Game Theory

Seshia et al[66] take a game-theoretic approach to analyzing quantitative properties that is based on performing systematic measurements to automatically learn a model of the environment. The problem of estimating WCET is modeled as a game between the algorithm (player) and the environment of the program (adversary), where the player seeks to accurately predict the property of interest while the adversary sets environment states and parameters to thwart the player. Over several rounds, the player or algorithm learns enough about the environment to be able to accurately predict path lengths with high probability, where the probability increases with the number of rounds. To solve this problem, [66] employs a randomized strategy that repeatedly tests the program along a linear-sized set of program paths called basis paths, using the resulting measurements to infer a weighted-graph model of the environment, from which quantitative properties can be predicted. Test cases are automatically generated using satisfiability modulo theories (SMT) solving. It is proved that their algorithm can, under certain assumptions and with arbitrarily high probability, accurately predict properties such as worst-case execution time or estimate the distribution of execution times. Based on this idea, a tool GAMETIME has been developed[66]. GAMETIME is measurement based but uses certain static characteristics for loop bound analysis and using symbolic execution and satisfiability solvers to compute input to derive the program down a specific path of interest. This happens to be the first work that uses game theory in the context of timing analysis. GAMETIME inlines all functions and unrolls all loops to a safe upper bound and uses a binary vector to represent each path.

GAMETIME operates in four stages. First it builds the program CFG. Second, it computes the basis paths and also ensures feasibility of the paths using integer linear programming and SMT solving. This is a slightly time consuming phase. Third, using constraint based test generation, it generates a set of inputs to program that will drive the program's execution down that path. Following this, for each input, GAMETIME creates a different program embedding

the data within the program. Fourth, it predicts the estimated weight vector or longest path using an algorithm that works on a representative set of basis paths called as barycentric spanner as described in [66]. The number of simulations performed for measurement is equal to the number of basis paths. In particular, [66] can predict the longest path, and its corresponding length. Given the predicted longest path, its feasibility can be checked with an SMT solver. If it is feasible, a test case is computed for that path, but if it is not feasible, the next longest path is checked. The process is repeated until a feasible longest path is predicted. The predicted longest path can be executed (or simulated) several times to calculate the desired timing estimate. GAMETIME also gives probabilistic WCET estimates. The test cases that drive execution through basis paths are valuable for hard and soft real time testing. GAMETIME can also be integrated with a static analysis tool instead of measurement if a tool that models the desired architecture accurately is available. One of the unique aspects of GAMETIME is the ability to predict the execution time profile of a program and the distribution of execution times over program paths by only measuring times for a linear number of basis paths and is described in detail in [66].

GAMETIME also assumes that the timings of a program can only depend on control flow and does not consider instances where even data can determine control flow. This aspect is considered as future work in [66]. However, our technique works seamlessly with any uniprocessor architecture and is not dependent on the presence / absence of any particular component. GAMETIME works on program paths, the analysis unit of our technique is a phase. One thing that is common between our work and GAMETIME and that is we do not assume any prior distribution of observed data. We use a simple probabilistic inequality that only requires mean and variance to be finite to obtain probabilistic bounds on phase CPI.

Role of Testing in Measurement Based WCET Analysis

Testing is an integral part of WCET analysis. Static methods carry out WCET analysis by considering any valid input by assigning abstract values to the variables. However, it evaluates the obtained WCET estimate by comparing it with the maximum observed value by running the program exhaustively with a good test input set that has adequately exercised the program. Testing is important in the case of measurement based WCET analyzers as measurements are used directly inorder to obtain the WCET estimate. If the test input set has not adequately

covered the program, the resultant estimate can be unsafe.

Practical WCET Analysis Approach Based on Testing

Lundqvist et al[82] observe that the execution time of a single program path can vary even when testing with the same input data. The reason is that the number of cache misses depends on the initial state of the cache. This initial state can be hard to control and observe which is fundamental in creating reliable tests. The work in [82] proposes to handle this situation by adding a safety margin to the WCET estimate for single path programs determined in three possible ways - constant bound, dynamic cache foot print, static cache foot print. Another reason for uncertainty is that the execution of certain program paths can trigger conflict misses that leads to a large nonintuitive increase in the execution time. These program paths might not appear to be interesting from a manual inspection point of view but may still be the paths that cause the longest execution time due to the extra conflict misses. The tester should be assisted in finding untested dangerous program paths. If the onus is on the user to provide information about such paths, there is a chance that the user overlooks certain cases that might perform worst on complex architectures say with respect to caches. The work in [82] gives certain recommendations using the assistance of the linker to identify potentially conflicting regions, so that it can be given control to place functions in an appropriate way to reduce conflicts. The kind of dangerous paths highlighted in [82] figure as phases with a high variation in CPI. In chapter 6, we shall describe how we deal with phases that exhibit high variation in CPI.

Less Optimism in Measurement Based Analysis

Bünte et al[78] uses a combination of genetic algorithms and model checking to heuristically optimize the set of measurements in terms of safety. In other work[90], genetic algorithms are used to find higher end to end execution times. In contrast, the work in [78] aims to maximize local WCET estimates of basic blocks. Genetic algorithms are used to provide inputs that will increase maximum observed execution time of all basic blocks jointly. Model checking generates test suites that satisfies basic block coverage. The initial seed population is 200 inputs. Those inputs that satisfy basic block coverage are kept alive and pursued. We work around this problem by computing a theoretical upper bound on instruction count that assures us maximum coverage of basic blocks for any valid execution, to circumvent the problem of inadequate test

input coverage. If the test input set provides adequate coverage, we can use maximum observed instruction count in place of theoretical upper bound on IC. The thesis uses a genetic algorithm to generate test inputs that maximizes end to end execution times.

Improving Confidence in Measurements Based Timing Analysis (MBTA)

Bünte et al[79] introduces extended pair coverage to evaluate quality of new MBTA test suites. This coverage criteria is quite effective for exercising the worst case temporal behavior of a program while requiring only a tractable number of test vectors. Enhanced pair coverage is better both with respect to hardware coverage and scalability. Since [79] considers segments and each segment can be a basic block, considering them pair wise instead of considering them in an isolated way, includes some amount of context information and results in more accurate WCET estimates. The thesis presents several ways of estimating WCET for programs exhibiting phase behavior. In our case the analysis unit is a phase in which most activity is clustered and contained within a phase. Hence the phase abstraction is a better abstraction than basic blocks in this respect.

Verifying Timing Constraints of Real Time Systems By Evolutionary Testing

Wegener et al[90] proposes a combination of evolutionary testing, random testing and systemic testing. Each of these techniques when performed in isolation might not be successful in capturing the complete range of possible inputs. Random testing can miss out on certain inputs that clearly exercise the program the most. A simple example is the Bubble sort program. Random inputs might fail to come up with the reverse sorted vector which is observed to be the worst case input for the program. Such cases are better caught with systemic testing. Similarly evolutionary techniques have the risk of getting stuck at certain local maximal cases but might fail to capture the overall worst case input[90]. The thesis uses a similar mix of test inputs. For simple programs, if the worst case input can be identified, it is included in the test input set. Together with genetic algorithms, we also use random input generation methods along with test harnesses that are already available for many programs developed by the designers.

Multicriteria Optimization

Khan et al[85] show that test criteria that generates inputs based on only execution time is very limited in finding all range of inputs. Using hardware metrics like instruction and

data cache misses, branch predictor misses can be useful in finding a complete range of test input. The reason is that some programs can be memory intensive in which data cache misses might have more influence. In some control flow intensive programs, instruction cache and branch predictor misses might influence execution time. In some other programs, loops may be highly influenced by values of data variables. In certain other programs, multiple of these parameters can be of equal importance. Hence multiple search criteria are employed to build the test input set. Sometimes these very criteria can conflict with each other. For example in janne_complex (section 3.2 of chapter 3), the loops are highly influenced by the values of variables and the inner loop entry is difficult due to the way the program is designed. This may lead to higher cache misses whenever the inner loop is entered thus creating an illusion of higher fitness inputs. Hence the criteria should be carefully evaluated before finalizing the test input set. Some amount of pre-analysis on the program will help the optimum mix of criteria to be considered.

Khan et al[85] give a list of recommendations to decide which criteria is to be used depending on the kind of programs. For single path programs, execution time is to be used as the criterion. If the program input space is large, data cache misses and execution time are used as criteria. If the program consists of a large number of conditions or loops, a mix of instruction cache misses and execution time is used as criteria. If the basic blocks of a program are large, execution time is used as the sole criteria for test input generation, which is also the default criterion. In chapter 4, we shall see that we present similar information in the form of a correlation between instruction count(IC) and CPI. For certain single path programs, who do not exhibit a significant variation in CPI, extensive testing is not required. For single path programs, who exhibit a variation in CPI, architectural parameters are said to influence the WCET estimate more than program structure. In certain programs, we shall see, CPI variation is stable but IC varies a lot across inputs. For such programs, the program structure is said to have more influence on WCET estimate. For other programs, we shall see that both IC and CPI vary, as a result both structural analysis and architectural parameters equally influence the WCET estimate.

Deriving Worst Case Input

Ermedahl et al[7] study the problem of worst case execution time analysis from the other end. They advocate a combination of input sensitive worst case static WCET analysis and systemic search over the value space of input variables to derive the input value combination that causes WCET and also present methods to speed up search. Measurement based WCET analyzers can then estimate WCET with more accuracy using worst case inputs than those obtained by using all inputs. Such worst case inputs forces program to run for long execution times. The complexity of systemic search over the value space of input variables gives a best running time proportional to $\log_2$ of size of input space. The work in [7] assumes the availability of an input sensitive WCET analyzer that can take constraints on input variables into account while calculating a WCET estimate. The algorithm systematically divides input space into smaller input space partitions each with a subset of the input value space. Then it calculates WCET estimates for each partition of program input value space. In each iteration, the partition with the largest WCET estimate is selected and divided into two or more smaller portions for which WCET calculations are made. The process continues until selected partition holds only one input value combination which is then returned. This combination is the desired WCET input value combination.

For a given program, there can be several qualifying combinations. There might be situations when the algorithm needs to back track. This happens when WCET of both the smaller partitions is smaller than the WCET estimate found higher up in the tree. Hence in the worst case, each value space partition has to get its WCET estimated. The work in [7] also uses a set of heuristics to speed up analysis. All partitions that produce smaller WCET values can be pruned away from further analysis. If there is no significant difference between best case execution time and worst case execution time, very likely, the program is insensitive to its inputs with respect to execution time and the search process can be terminated much earlier. We share a similar observation in chapter 4, that some programs do not exhibit any variation in IC and CPI irrespective of the input presented to them. Such programs need not be exhaustively tested with respect to WCET analysis. Exhaustive measurements could replace static analysis in [7], if it is more economical in the situation.

Role of Static Analysis in Measurement Based WCET Analysis

Schaefer et al[74] presents a new approach to enhance measurement based WCET analysis by deploying static analysis to ensure test coverage at the basic block level and reduce pessimism of WCET. In addition to test inputs that exercise the programs and generate execution times, static analysis is also carried out. If the difference between static WCET estimate and the maximum observed cycles is large, testing is repeated with a larger test input set. Deverge et al[43] explores issues to be addressed inorder for measurement based approaches to be safe and use of compiler techniques to reduce timing variability of program segments and to make the execution time of program segments independent of each other. The key assumption made in [43] is that execution of the same program path with different data yields the same timing, and hardware modified accordingly to make this possible. Some of the techniques that can help achieve this are cache conscious data placement, cache locking, static branch prediction, to account for variable latency of segments, add difference between BCET and WCET of all operations of the program path (also known as jitter), avoiding usage of instructions that take variable latency, usage of more predictable instructions (Eg: add instead of mul).

The results in [43] show that with such hardware control, the variability of execution time across executions reduce and the overall WCET estimate is much tighter. Our technique also would benefit from hardware control as advocated in [43]. If we were to run our technique on such an architecture, CPI would be very stable leading to tight CPI-IC clusters as we shall see in Chapter 4 which will result in tight WCET estimate. Alternatively, applying phase based WCET analysis, very tight bounds on CPI can be obtained if variance is small and this will also result in tight WCET estimates.

Most research done in probabilistic schedulability analysis assumes a known distribution of execution times for each task of a real time application. It is a non trivial task to determine the actual distribution with high confidence. Methods based on measurement do not exhaustively test all paths. David et al[50] uses static analysis to obtain probabilistic distributions of execution times. Given the source code of a task and for any execution path, their objective is to compute on the one hand the probability to go through this path and on the other hand, its associated execution time. Two sets of variables are considered. The first set of variables are global external environment variables and the second set of variables are the internal program

variables. Both these variables are treated as random variables. Each such variable is associated with probability functions such as, $f_E(x)$ is the probability that the random variable P takes the value x. The work in [50] assumes that the external variables are independent of each other. A set of formulae to compute probabilistic execution times of basic blocks, compound statements along with the conditions on the satisfaction of which, these will be executed, are defined. These formulae are applied bottom up upwards the root of the tree thereby computing probabilities for executing paths consisting of these basic blocks. The values are plotted and compared with the set of values obtained using stochastic simulation of the source code execution. At a given node of a tree, an *if-condition* produces 2 leaves and a loop that iterates for k times, produces k leaves. Hence complexity can be exponential and depends on relation between I and E (internal and external variables) and the domain width of external variables. Probabilistic evaluation can get exponential. In certain cases, complexity can be reduced by using specific probabilistic algebras.

The method in [50] does not take into account hardware considerations yet; the execution time of any instruction is intentionally assumed to be constant and context independent. Furthermore, it assumes no optimization mechanism in the compilation process. Assumption of mutual independence of external variables is also controversial. Several WCET analyzers place constraints on the level of compiler optimizations as they can interfere with analysis in unpredictable ways. Code optimizations can introduce additional control flow decisions that are not covered by test data suite providing a cause for WCET underestimation. Kirner et al[61] describes compiler support for measurement based timing analysis that can provide optimizations while preserving code coverage achieved by the input test suite at source code level. They work with a table that indicates with the help of a flag for each code optimization and each code coverage metrics, whether the corresponding metric is preserved by performing code optimization. Additional compiler optimizations that indicate to the compiler that coverage has to be preserved is implemented in [61]. By introducing a coverage preserving mode for the compiler, [61] guarantees that if a concrete structural code coverage has been achieved in the original code, it will also be fulfilled in the transformed code. We could also benefit from a similar compiler and benefit from generating test data automatically and ensuring better coverage provided by test inputs.

Keim et al[59] describes an approach to measure the execution time distribution by a hardware simulator and a path-based timing analysis approach to derive a static estimation of this same distribution. The technique can find a soft WCET for loops having any number of paths. Probability distributions are continuous functions but this is quantized by creating an array of buckets in which one bucket represents one possible execution time. 100 buckets store percentiles of execution time. The last bucket containing the highest value contains soft WCET. Two kinds of data are experimented with. Fair data where each path is given equal preference. Unfair data where certain paths are favored more than others. The work in [59] outlines an algorithm that statically generate distribution of execution times into an array. For an example code that has two paths within a loop, the technique in [59] uses binomial probability to populate the distribution array. For some programs results indicate that soft WCET is more or less as pessimistic as the hard WCET. This can be prevented if the analysis has some prior knowledge of the probability of a certain branch to be taken.

2.2.3 Statistical WCET Analyzers

It can be understood that measurements cannot be exhaustive always and hence measurement based estimates are only associated with a probability of success. This raises the need for statistical modeling where computation time is represented as by a probabilistic estimation and WCET estimates are associated with a level of confidence. For statistical techniques, data has to be collected and collated using mathematical formulae or a suitable model. The model is used to generalize the behavior of a system beyond the existing data. Typical methods use a particular probability distribution and an associated math function assigning probability values to a set of events. For each distribution there will be a corresponding density function representing density of data samples against magnitude of data. Random data samples tend to be bell shaped if a gaussian distribution is assumed. According to central limit theorem a random sample of n data items that are independent and identically distributed (IID) follows gaussian distribution as the sample size increases. But this is suitable if the values concerned are close to the mean but WCET analysis is much closer to near extreme values. The distribution that accurately model the maximum values are selected. Such distributions are known as extreme value distributions. Typical examples are Gumbel, Frehet and Weibull. Edgar et al[71] proposes to use one such extreme value distribution and fits it over a large set of measured end

to end execution times and calculates the parameters to extrapolate the execution time at the desired probability. In chapter 3, we plot CPI samples collected over a large number of runs and use percentiles to estimate worst case CPI. In Chapter 6, we exploit the homogeneity of CPI within a phase and describe a model to bound CPI within a phase probabilistically.

The Gumbel and other extreme value theory (EVT) distributions are intended to model random variables that are the maximum (or minimum) of a large number of other random variables. Hansen et al[38] hence groups sample execution times into blocks and fit only block maxima values to gumbel distribution, calculate parameters for the probability distribution so that an estimate of WCET can be obtained by extrapolation at the required probability. Chi-squared test is used to ensure sample data is correctly fitted to the distribution. There are no data dependent loops in any of the tasks measured. Huge traces are collected by running tasks by two different teams in experiments performed in [38]. This trace is divided into two parts- estimation part and validation part. However, in [38] there are no data dependent loops in any of the tasks measured.

Griffin et al[22] considers EVT that has been the most commonly used model to obtain worst case execution time estimates and how statistical models sacrifice realism inorder to provide generality and precision and how the sacrifice can compromise safety of the model. Gumbel distribution is examined specifically in its need for IID data. The issues concerned with statistical techniques are generality: which is the degree to which the model is applicable to general situations, realism: how accurately the model represents the real world and precision: the degree to which the error can be bounded. Firstly, Gumbel distribution assumes the execution times can take any arbitrary value which is not true in the real world, as a program can terminate only at certain execution points and not at any point. EVT assumes data samples are IID. However, execution times of programs neither are independent nor identically distributed state-changing programs. Several reasons exist for this, namely, the presence of an onchip cache, global state variables shared by more than one instance of the program running, each program run has the potential to change the world, each mode has its own distribution, each path could have a different distribution thus leading to explosion in the number of measurements and tests to be performed. An example is bubble sort where with every iteration, the timing changes as the array gets more and more sorted. Hence results of applying gumbel distribution are poor for this program[22].

Griffin et al[22] gives several suggestions to work around this problem. One of the suggestions is to add an error to prediction, break dependence among several runs by periodically resetting the system or at least a part of it. Every path will have different set of hazards leading to a different probability distribution. Hence proving IID is hard. A solution is to ensure sufficient MC/DC coverage and each path has sufficient samples to form a distribution. Avoid interaction of programs with each other, as this too can violate IID. One way to make IID assumption hold would be to use external measurements such as number of instructions executed, cache misses to create a categorization scheme. If the chosen measurements lead to categories whose members are sampled from the same or similar distributions, the IID assumption should hold. WCET can then apply statistical analysis to each category and pick the maximum. We also carry out a similar categorization where we split time into instructions executed and CPI and estimate WCET by estimating maximum IC and CPI.

Lu et al[98] uses bootstrapping sampling theory and generates traces and also satisfies requirements given by statistics and probability theory. The work in [98] is comprised of a novel sampling mechanism (which tackles some of the problems raised when statistics is used in WCET analysis[22]) and a statistical inference about computation of a WCET estimate of the program under analysis, with a certain predictable probability. The novel sampling mechanism is a simple random sampling technique with bootstrapping to collect qualified and robust analysis samples (that is, timing traces), which can be used by to estimate statistical WCET. Samples collected plainly using non IID program using simple random sampling are termed as SRS ET (execution time) samples. In [98], a program is executed n times using SRS technique, each time there are m non IID SRS et samples. Among each m, the maximum is chosen as a new sample to construct a new set which will be IID. This sample set is termed as IID SRS ET samples. Bootstrapping selects randomly with replacement samples out of n IID SRS ET samples to create N bootstrap ET samples. Using N bootstrap ET samples and EVT statistical WCET is estimated. In [97], Lu et al tries to fit IID SRS ET samples into a normal distribution. If it cannot be done, bootstrapping is carried out. Our work neither assumes any probability distribution of CPI samples nor tries to fit these samples into any distribution. We use Chebyshev inequality that is applicable to any distribution, to compute bounds on CPI. The precision of our results will definitely improve if information regarding true probability distribution of CPI samples is available.

2.2.4 New Trends in WCET Analysis

Conventionally WCET analysis has been limited to simple real time and embedded architectures. Recently there has been a lot of interest in WCET analysis being carried out for different processor architectures like the multicore, multithreaded and distributed systems[75]. The program analysis portions of a compiler and a WCET analyzer have several aspects in common. This has led to studies that look at the possibility of the WCET analyzer and the compiler to work together in a closely integrated manner. Since the compiler knows enough of the program to help resolve issues like targets of function pointers, it can go a long way in passing on this information to the WCET analyzer and help improve its precision. Similarly the WCET analyzer can offer hints regarding the temporal properties of the program regions and variables and help guide some of the compiler optimizations[35]. Several of the analyses made by the WCET analyzer regarding the code properties could help a program energy analyzer (WCEE) to estimate the worst case energy estimation of the program[72].

If the program can be transformed in some way to another program which has a lesser degree of path complexity[32], the WCET analysis time would be faster. The WCET analyzer works fastest for straight line code, code which has only one path of execution. Traditional compiler optimizations always work for bettering the average case execution time. However recently there has been focus on developing optimizations that can decrease the worst case execution time of the program. These could be using a mix of powerful instructions on the worst case path and use less powerful instructions on the non critical path that can effectively reduce the code size[69], developing customized static branch prediction schemes[27], placing of basic blocks along the worst case path in such a way so as to avoid penalties regarding jumps[89], optimally allocating variables to scratch pad allocation memory[86] etc. There have been efforts also at the architecture level to make the design much simpler enough, to make WCET analysis more predictable[1, 19].

2.3 Chapter Summary

In this chapter, we described the basic process of WCET analysis and various challenges in estimating WCET and issues concerning usability and applicability of WCET analysis. We also presented a survey of various WCET analysis techniques by classifying it into different

categories and briefly discussed the work in each category and also touched upon briefly the recent trends in the field of WCET analysis.

Chapter 3

Preliminaries: Base Timing Model and Experimental Setup

In this chapter, we present our basic timing model that considers a program as one single unit and estimates its WCET in terms of whole program instruction count(IC) and cycles per instruction(CPI). As the thesis progresses, we shall break the program into phases and apply the same formulation at the phase level and estimate program WCET in terms of its phases. We also describe our experimental setup that will be used throughout the thesis which includes the simulators, benchmarks and test input formation procedures.

The unit of analysis used by existing timing analysis methods is *cycles*. We propose to use cycles per instruction or CPI instead. Most processors of today are pipeline based. Hence CPI is a commonly quoted performance metric in today's processors. A processor with low CPI is said to have higher performance. Moreover, most of these processors are embedded with performance counters that enable accurate measurement of CPI with the least intrusion. When we speak of time in terms of CPI, execution time of a program then becomes a product of number of instructions executed or instruction count, IC and cycles per instruction, CPI. WCET is hence formulated as a product of worst case IC, WIC and worst case CPI, WCPI as shown in Eq(3.1).

$$WCET = WIC \times WCPI \tag{3.1}$$

The worst case number of instructions, WIC, is the maximum number of instructions that a program can execute. If the test input set covers the program adequately, we can use maximum

observed instruction count as WIC. However, if there is low confidence on the coverage aspect of the test input set, we compute the theoretical upper bound on IC for the program using static structural analysis and use it as WIC. The worst case CPI, WCPI depends on the worst case input, often unknown *a priori*. Hence one has to depend on certain functions of CPI samples obtained by measurement over fixed number of instructions termed as intervals. Over the CPI samples thus collected, different functions are defined. These functions of CPI are then combined with the worst case instruction count, WIC to give the estimate of program WCET. In this chapter, we shall evaluate various combinations to decide which one of them is best suited for purposes of WCET estimation.

3.1 Worst Case IC (WIC)

WIC has two clear candidates as we saw earlier, the maximum observed instruction count and the theoretical upper bound on instruction count. Maximum observed instruction count is available upon direct measurement of the program by running it on all inputs in the test input set. However, if it is expensive to obtain inputs that achieve adequate coverage, we obtain a theoretical upper bound on IC, also known as SWIC (static WIC). Now, we shall describe how SWIC can be derived.

3.1.1 Derivation of SWIC.

Several static WCET analyzers including Chronos use integer linear programming (ILP) to estimate static worst case execution time. The theoretical upper bound on execution time (ET) is derived by maximizing a linear objective equation which is essentially a sum of products. The product term consists of a constant term and a variable term. The constant term denotes the maximum cost of execution of a basic block on a given architecture. The variable term denotes the frequency of execution of the basic block. This product is summed over all basic blocks of the program and maximized to obtain WCET.

We use a similar formulation to estimate the theoretical upper bound on IC, SWIC. The constant term in our case is the number of instructions that comprise the basic block. The variable term remains the same and represents the frequency of execution of the basic block. The basic blocks are the components of the control flow graph (CFG) which is constructed from

the program binary. Each basic block, B is associated with an integer variable N_B that indicates B's execution count and an integer constant W_B that indicates the number of instructions in basic block B. The linear objective function is given by,

$$Maximize(\Sigma_{\forall B}, N_B * W_B) \tag{3.2}$$

The linear constraints on N_B are developed from the flow equations based on the control flow graph. Thus for basic block B,

$$\Sigma_{B' \to B}(E_{B' \to B}) = N_B = \Sigma_{B \to B"}(E_{B \to B"}) \tag{3.3}$$

Where, $E_{B' \to B}$ ($E_{B \to B"}$) is an (integer linear programming) ILP variable denoting the number of times control flows through the control flow graph edge $B' \to B$ ($B \to B$"). Additional linear constraints are also provided to capture loop bounds as follows. The parameter E_B is bounded by the maximum number of times the loop can iterate, L, if it happens to reside inside a loop else it takes the value *1* by default.

$$E_{i \to j} <= L \tag{3.4}$$

In order to simplify our analysis, we assume availability of loop iteration bounds for all the loops in the CFG. Any instances of recursion are converted to iteration whenever possible.

If infeasible paths are known *a priori*, they can be indicated in terms of additional linear constraints as follows. If an edge $E_{i \to j}$ cannot execute with an edge $E_{p \to q}$ together in any execution, we can specify the following constraint,

$$E_{i \to j} + E_{p \to q} = 1 \tag{3.5}$$

IPET is one of the mechanisms by which, SWIC can be derived. However, SWIC can also be derived using enumerating paths explicitly by create a directed graph out of the CFG. With each node representing a basic block and edges representing control flow, the weights of edges representing the number of instructions in the preceding basic block, one can use graph theoretical algorithms to find the longest path between the source and the sink node to give SWIC. Similarly, tree based schema can be used to derive SWIC bottom up. For a straight

sequence of basic blocks, the upper bound on IC is just the sum of the number of instructions of each basic block. For an if-then-else statement, the upper bound on IC is the maximum of the number of instructions that can get executed in either the if set of statements or the else set of statements. For a loop, the upper bound on IC is the maximum number of instructions that can get executed in each iteration multiplied by the loop bound.

Implementation

ChronosV3.0[94] uses an ILP based framework to compute WCET for a program by constructing a linear objective equation. Inorder to compute SWIC, we modify *ChronosV3.0* in the following ways. We disable code that carries out microarchitectural modeling and set the cost of executing every basic block on a given architecture as 1 unit. We set the constant factor of the variables corresponding to each basic block as the number of instructions in that basic block. We retain code that carries out infeasible path analysis and forms constraints based on CFG structure. Executing the modified version of *ChronosV3.0*, gives us a set of ILP equations along with only the structural constraints imposed by the graph. Solving the ILP equation gives us SWIC. Before we propose and evaluate possible candidates for worst case CPI, we describe our experimental setup and architectures targeted as CPI is the measured quantity and is hence associated with an architecture.

3.2 Experimental Framework

In this section, we describe the basic framework used for all our experiments carried out in the thesis.

3.2.1 Benchmarks

The Mälardalen WCET research group maintains a large number of WCET benchmarks[108], used to evaluate and compare different types of WCET analysis tools and methods. Several of these benchmarks are used in the WCET Tool Challenge[64], an event that occurs once every 2 years, involving numerous WCET tool developers around the world. These benchmarks have been developed from various sources like research groups and tool vendors from around the world. We select 18 benchmarks from this suite and they are,

- **Bubble Sort (Bub):** Sorts a set of numbers using the bubble sort algorithm.
- **Binary Search (Bs):** Searches for an element in a vector using the binary search algorithm.
- **Cyclic Redundancy Check (Crc):** Performs cyclic redundancy check on a character array.
- **Count (Cnt):** Counts positive and negative numbers in a N X N matrix.
- **Edn:** Implements Jpegdct algorithm together with other signal processing algorithms.
- **Finite Impulse Response filter (Fir):** Performs the finite impulse response filter algorithm on a sample of N items.
- **Insertion Sort (Ins):** Sorts a set of numbers using the insertion sort algorithm.
- **JANNE_COMPLEX (Jan):** Program with a complex nested loop whose inner MAX iterations depend on the outer loop.
- **Lms:** Adaptive signal enhancement on an array of N coefficients.
- **L U Decomposition (Lud):** L U Decomposition algorithm on a N X N matrix.
- **Matrix inversion (Minv):** Matrix inversion on a N X N matrix.
- **Matrix multiplication (Mat):** Matrix multiplication of two N X N matrices.
- **N-dimensional search (Ndes):** Search for an element in a N X N X N vector using linear search algorithm.
- **Extended petrinet simulator (Nsch):** Simulate an extended petrinet for N iterations.

Three benchmarks are taken from *Mibench* embedded benchmark suite[109]. They are,

- **Bitcount (Bit):** Performs bit operations on an array of numbers having various distributions.
- **Dijkstra Algorithm (Dij):** Finds M shortest paths in a graph of N vertices using Dijkstra's algorithm.

- **Fast fourier transform (Fft):** Fast fourier transform on a wave of size N.

The last benchmark is taken from Antoine Colin[9].

- **Bezier curve drawing (Bez):** Bezier draws a set of N lines of 4 reference points on a 800 X 600 image.

In the thesis, we present estimation of WCET with increasing levels of accuracy in each chapter. In Chapter 6, we shall describe design and implementation of a probabilistic WCET analyzer and present ways to tune the acccuracy of the estimate. To evaluate this analyzer, we choose an additional real world benchmark DEBIE-1[106] (DEBris In Orbit Evaluator) used for monitoring space debris and metereoids in near earth orbit. Details of this benchamark are described in the evaluation section of Chapter 6.

3.2.2 Input Set Formation

The proposed method is a hybrid WCET estimation method which involves measurements. Hence selection of inputs used to obtain measurements of CPI is important. Ideally, inputs should be selected such that they exercise *all* paths in a program for any measurement based or hybrid WCET estimation method. But the presence of complex if-conditions in a program can exponentially increase the number of paths possible, making enumeration of these paths and execution of a program along all paths, a computationally hard problem. We hence make use of MC/DC (Modified condition/Decision coverage) criteria[47], used in testing of real-time avionics systems to approximate complete path coverage. This criterion is also used to select inputs in other well known hybrid WCET estimation methods like Adam Betts's instrumentation point graphs[2]. Many of the benchmarks are memory intensive and are largely dependent on the distribution of data. Hence in addition to structural coverage, we also form test inputs that cover all possible distributions of data.

Structural Coverage

Inorder to satisfy MC/DC criterion during testing, all of the following conditions must be true atleast once.

- Each decision has tried every possible outcome.

Major clause: x<XSIZE	*x>0*	*y<YSIZE*	*y>0*
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	TRUE	TRUE

Table 3.1: Truth values of major clause: x<XSIZE and minor clauses: x>0, y<YSIZE and y>0.

- Each condition in a decision has taken on every possible outcome.
- Each entry and exit point has been invoked.
- Each condition in a decision has been shown to affect the outcome of the decision independently.

Most of the inputs for the benchmarks are vectors of either numbers or characters. Hence they can be generated automatically with ease. For simple programs with small number of conditions, random input generation is sufficient to ensure that inputs satisfy the MC/DC criterion. For complicated programs involving hundreds of conditions, we use genetic algorithms to generate data that are fit enough to satisfy MC/DC criterion. The condition clauses present in a benchmark are analyzed. If the condition contains only a single clause, benchmark inputs that result in the clause being TRUE and FALSE are generated. If the condition contains multiple clauses, each clause is considered as a *major clause* at a time. The remaining clauses are considered as minor clauses[47]. If the benchmark input demonstrates the effect of the major clause independently of the minor clauses, the input is considered fit. The structure of the condition determines the truth values of the minor clauses inorder for them to have no impact on the final truth value of the condition. Consider the benchmark, *bezier* that has the following condition clause, containing four logical conditions.
if((x < XSIZE) && (x > 0) && (y < YSIZE) && (y > 0))
Considering, (x < XSIZE) as the major clause, the truth values for the minor clauses to demonstrate the effect of the major clause independently is shown in Table 3.1

Similarly every other condition is deemed as a major clause and truth tables are generated for the other remaining minor clauses. Appropriate data values that cause the conditions to evaluate to TRUE or FALSE are determined. In most of the cases this results in a range of values that the data can take inorder to satisfy these truth values. It should be noted that there might be minor and major clauses in a benchmark whose truth values can never be attained

together. Hence in some programs, we cannot always guarantee 100% coverage of MC/DC criterion as it depends on the inherent structural makeup of the program.

To begin with, a set of 500 inputs are randomly generated for each benchmark. The fitness of these inputs are determined by examining the truth values of clauses obtained by applying these inputs. Inputs that are unfit are left out of consideration. Genetic computing requires that fit inputs are selected and mutated to produce newer inputs that continue fit. The mutation is done in such a way that their fitness is not affected. The inputs are crossed to produce new inputs. The crossover operation depends on the structure of program inputs. This process continues until we have 500 fit inputs or we reach a million iterations.

For some programs, both random input generation and genetic algorithms might not succeed in finding the complete range of test inputs. If the program structure is simple, we systematically analyze the program manually and form inputs that satisfy condition coverage.

Data Coverage

If a program is highly data dependent, merely covering it based on control flow would not suffice. Hence we need to test such a program with all kinds of data. Random data generation is clearly the simplest choice and is carried out. However, we are interested in generating data that most likely exercises the program for a long time and is a likely candidate to drive the worst case path. Hence we apply genetic algorithms with the fitness criteria centered around the execution time of the program. If an input makes the program execute for a long time (greater than the observed mean execution time over all inputs till now), it is considered as a fit candidate. Two pairs of fit inputs are mutated and crossed over at fixed points and the offspring is evaluated for fitness. Again, for some programs, random input generation and genetic algorithms might not capture the worst case candidate inputs. For such programs, manual intervention is necessary to craft special inputs. The most common example is Bubble sort that sorts an array of numbers. The worst case input for this program has been observed to be an array that is reverse sorted. This is detected using a systemic analysis on the program.

3.2.3 Simulation Tools

The proposed method is a hybrid WCET estimation method that uses measured CPI inorder to estimate WCET. Hence we need to measure CPI of the programs on the target architecture. For

simplest	Issue, decode and commit width=1, RUU size=8, Perfect data cache, Perfect branch prediction Inorder-issue, Instruction cache 8KB direct mapped
inorder_complex	Issue, decode and commit width=1, RUU size=8, 2 level branch predictor, Fetch Queue size=4, In-order issue, L1 Instruction cache 8KB direct mapped L1 Data cache 8KB 2-way set associative, L2 Unified 64KB 8-way associative cache
complex	Issue and decode width=4, commit width=1, RUU size=8, 2 level branch predictor, Fetch Queue size=4, Out-of-order issue, L1 Instruction cache 8KB direct mapped L1 Data cache 8KB 2-way set associative, L2 Unified 64KB 8-way set associative cache

Table 3.2: Architectural configurations used for experimentation.

this purpose, we choose the cycle accurate simulator, *Simplescalar V3.0*[113], which is a highly tweakable and reliable tool used by most architectural studies for over a decade. Another reason for this choice is that *Simplescalar V3.0* is used by Chronos, a popular static WCET analysis tool that is publicly available, with which we evaluate our WCET analyzer. Simplescalar V3.0 works with PISA binaries and is designed based on the MIPS R10K pipeline. The simulator allows users to tweak parameters of the instruction and data cache, branch predictor, pipeline and instruction and data TLB. In Chapter 6, we present a probabilistic WCET analyzer which we will compare with a commercial probabilistic WCET analyzer, *RapiTime* on ARM architecture. For that purpose, we use a cycle accurate ARM simulator *SimIt-ARM-2.1*[114] which will be discussed in Chapter 6.

3.2.4 Architectures

We perform our experiments on three different architectures as shown in Table 3.2. The three SimplescalarV3.0 based architectures are designed on the lines of architectures used for evaluation in previous WCET Tool challenge studies[64]. The presence of a data cache in embedded processors introduce variability and unpredictability in the system. Hence we begin with a *simplest* architecture with a perfect data cache and a perfect branch predictor. We increase the complexity by introducing a 2 level branch predictor and a data cache at L1 and a unified cache at L2 to form *inorder_complex*. We increase the complexity further by making the pipeline out-of-order and superscalar to form *complex*.

3.3 Candidates for Worst Case CPI (WCPI)

Once we have obtained worst case IC, WIC, for a given program, we need to multiply it by the appropriate worst case CPI, WCPI of that program, inorder to estimate WCET. We have seen that worst case IC can either be obtained by static analysis of the program or by measurement. Inorder to compute WCPI, we collect CPI samples by measuring CPI over fixed number of instructions termed as a interval. An interval is a contiguous slice of dynamic instructions executed. For our experiment, we choose a default interval size of 1000. For benchmarks that execute totally a few thousands of instructions, we choose a much smaller interval size (of about a 100 instructions). We run the benchmarks with the set of test inputs generated as described in the previous sections to generate a CPI sample set, each corresponding to a particular input, i, indicated by CPI_i. We repeat the process for all architectures under consideration. Various functions are defined on these CPI samples to estimate worst case CPI as follows. We shall refer to the following four possible candidates for worst case CPI as analytical CPI candidates:

1. Maximum of maximum CPI observed across all n test inputs with m_i intervals for each input, i:

$$Max\_Max(CPI) \ = Maximum(Max(CPI_1), Max(CPI_2), ..., Max(CPI_n)) \tag{3.6}$$

$$Max(CPI_i) \ = Maximum(CPI_{i,1}, CPI_{i,2}, ..., CPI_{i,m_i}), \ \forall i \in \{1, .., n\} \tag{3.7}$$

   $\text{Max}(\text{CPI}_i)$ is the peak CPI observed at any point during execution of the program with input i. Hence Max_Max(CPI) represents the highest peak CPI observed at any point during execution of the program with any input.

2. Average of maximum CPI observed across all n test inputs:

$$Avg\_Max(CPI) \ = \ \frac{\Sigma(Max(CPI_i))}{n} \ , \ \forall i \in \{1, .., n\} \tag{3.8}$$

   $\text{Max}(\text{CPI}_i)$ is computed using Eq(3.7). Avg_Max(CPI) represents the average of all peak CPI observed across n test inputs.

3. Maximum of average CPI observed across all n test inputs with m_i intervals for each

input, i:

$$Max\_Avg(CPI) \ = Maximum(Avg(CPI_1), Avg(CPI_2), ..., Avg(CPI_n)) \tag{3.9}$$

$$Avg(CPI_i) \ = \ \frac{\Sigma(CPI_{i,1})}{m_i} \ \forall i \in \{1, .., m_i\} \tag{3.10}$$

Avg(CPI_i) is the average CPI observed during execution of the program with input i, which is the same as overall CPI observed when the program is executed with input i. Max_Avg(CPI) is the maximum of average CPI observed across n test inputs.

4. Average of average CPI observed across all n test inputs:

$$Avg\_Avg(CPI) \ = \ \frac{\Sigma(Avg(CPI_i))}{n} \tag{3.11}$$

Avg(CPI_i) is computed using Eq(3.10). Avg_Avg(CPI) represents the overall average of the average CPI observed across n test inputs.

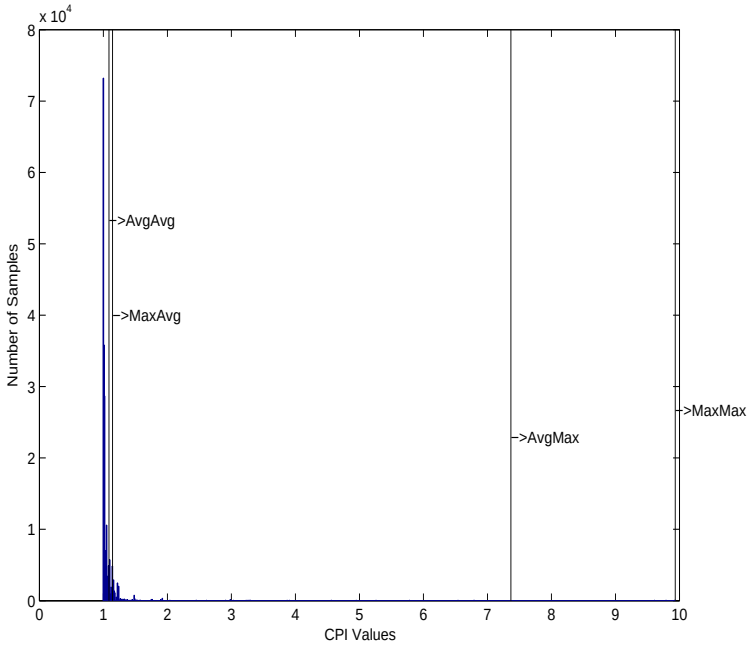


Figure 3.1: CPI distribution seen across 500 runs of *Bit* on *Complex* architecture with analytical candidates superimposed.

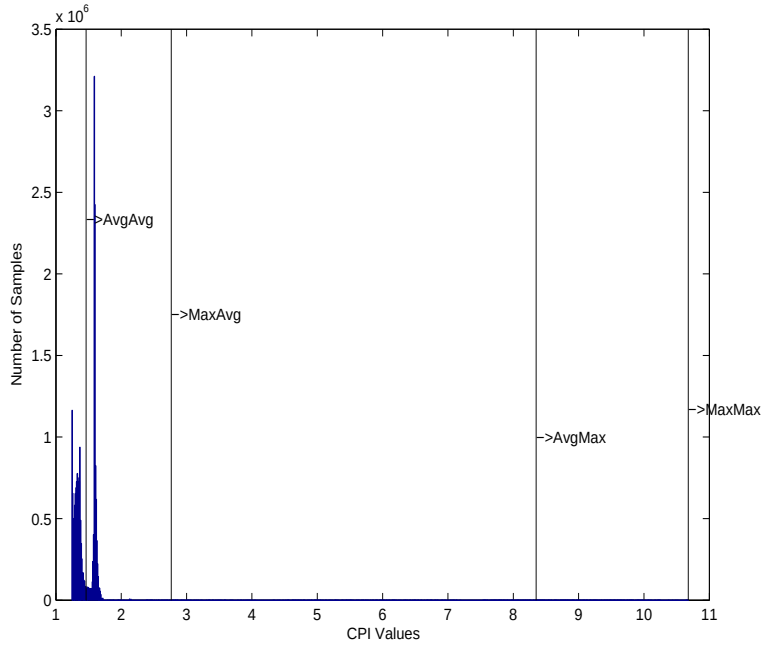


Figure 3.2: CPI distribution seen across 500 runs of *Bub* on *Inorder_complex* architecture with analytical candidates superimposed.

Amongst these four possible candidates, we need to find out the one that most accurately represents worst case CPI. For this purpose, we look at the distribution of CPI values by creating histograms of the CPI samples obtained as mentioned above. Figures 3.1, 3.2 and 3.3 show some of the distributions of programs on various architectures. These figures also superimpose the above mentioned candidates, Max_Max(CPI), Max_Avg(CPI), Avg_Max(CPI) and Avg_Avg(CPI) to give an idea about their location of occurrence in the distribution. From the figures, it is clear that Max_Max(CPI) and Avg_Max(CPI) represent values that occur very rarely and are located at the tail end of the distribution. The same trend is observed in other programs on all architectures. Considering Max_Max(CPI) and Avg_Max(CPI) as worst case CPI would imply that we expect every interval of the program to exhibit high CPI, which is not true. Further, we shall see in upcoming chapters, that programs are composed of phases, each phase having a different average CPI. Hence considering the tail end CPI values would imply using a high blanket CPI value for all phases which will only overestimate actual worst case CPI. Since we do not consider phases yet, Max_Avg(CPI) and Avg_Avg(CPI) appear to be more suitable than Max_Max(CPI) and Avg_Max(CPI).

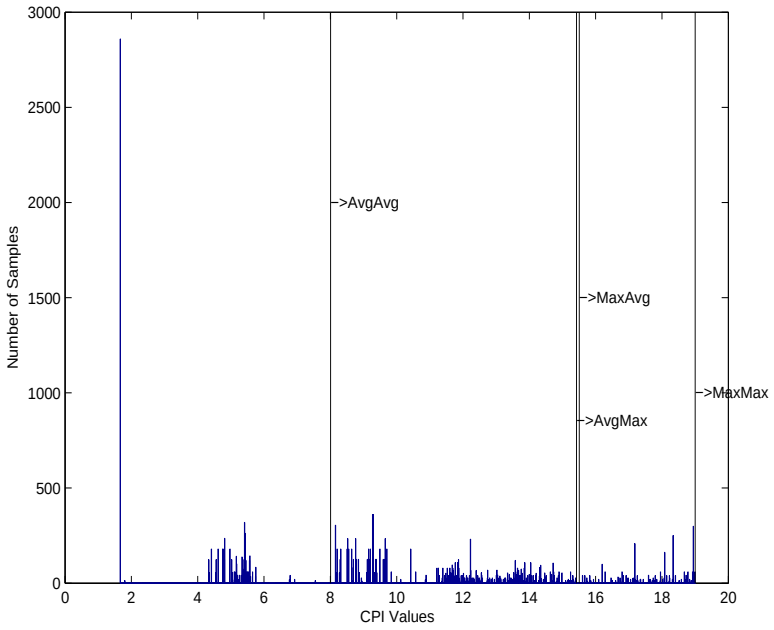


Figure 3.3: CPI distribution seen across 500 runs of *Nsch* on *Simplest* architecture with analytical candidates superimposed.

Alternatively, we can use statistical parameters such as 90th percentile, 95th percentile and 99th percentile CPI values of the distribution as candidates for worst case CPI. We shall term these as statistical candidates. A 90th percentile value is computed by dividing the distribution of CPI samples into a hundred groups of equal frequencies such that 90 percent of the values lie below the 90th percentile and ten percent of values lie above it[31]. To compute percentile values, we use the `prctile` function of *Matlab V7.1*[110]. To give an idea of the location of these parameters on the distributions, we plot these parameters on the distributions as shown in Figures 3.4, 3.5 and 3.6. We can infer from the figures that percentile values are highly dependent on the distribution. In Figure 3.6, the 90th percentile CPI value is greater than Max_Avg(CPI), whereas in Figure 3.5, it is much smaller than Max_Avg(CPI).

To give an overall picture of pessimism that results in the usage of these various candidates, we compare these candidates with CPI that occurs when the benchmark is run with that input which is observed to be worst among the set of test inputs, referred to as WCPI. The comparisons for *Simplest*, *Inorder_complex* and *Complex* can be found in Figures 3.7, 3.8 and 3.9 respectively.

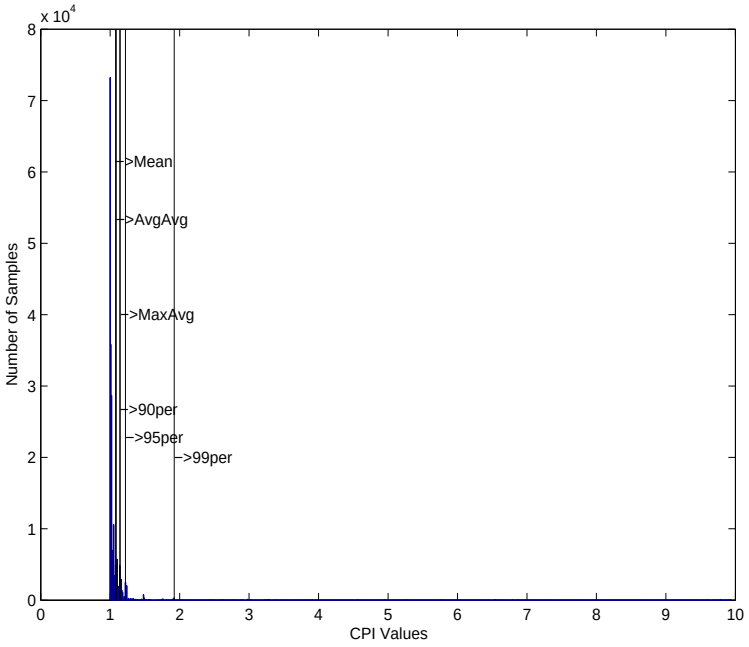


Figure 3.4: CPI distribution seen across 500 runs of *Bit* on *Complex* architecture with statistical candidates superimposed.

From these figures, we can infer that on an average, Max_Avg(CPI) comes very close to 90th percentile CPI value. While Max_Avg(CPI) is dependent on trend of CPI observed across inputs, the 90th percentile CPI value is dependent on the actual distribution. Hence the percentile values are highly sensitive to the choice of inputs used to form the test input set. Using the 99th percentile CPI value would still make WCET very pessimistic as can be seen by comparing it with WCPI. Avg_Avg(CPI) just about manages to touch WCPI. Max_Avg(CPI) is slightly more than WCPI in all cases. Max_Avg(CPI) and Avg_Avg(CPI) need only two values per input to compute- Cycles elapsed and instructions executed. But to compute the 90th percentile CPI value, we need all CPI samples collected over 1000 instruction intervals. Next, we shall evaluate pessimism in WCET estimate using analytical candidates Max_Avg(CPI), Avg_Avg(CPI) and statistical candidates 90th and 99th percentile CPI values.

With two possible candidates for WIC- SWIC and MIC and two possible analytical candidates for WCPI- Max_Avg(CPI) and Avg_Avg(CPI), we can have a total of four possible combinations of IC and CPI and hence four possible WCET estimates obtained by evaluating the following equations:

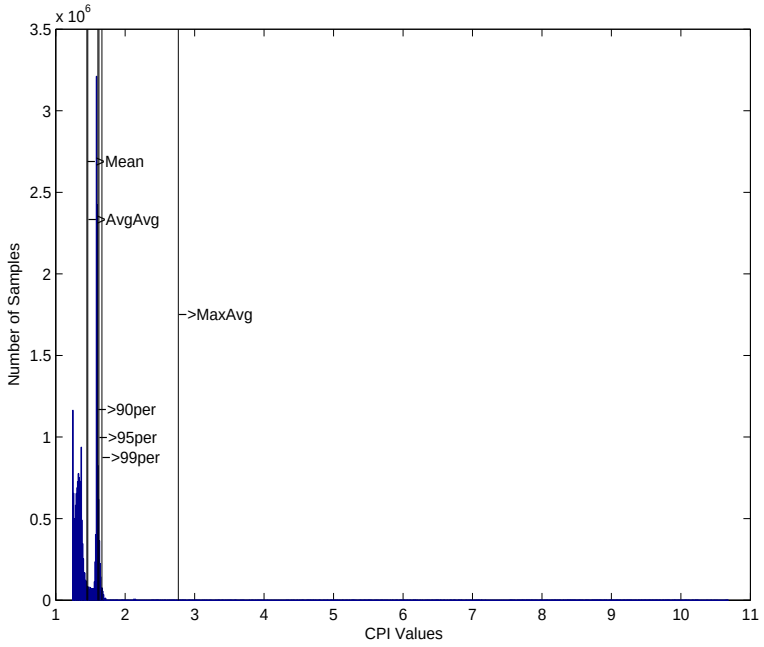


Figure 3.5: CPI distribution seen across 500 runs of *Bub* on *Inorder_complex* architecture with statistical candidates superimposed.

1. $\text{WCET}_1 = \text{SWIC} \times \text{Max\_Avg(CPI)}$
2. $\text{WCET}_2 = \text{SWIC} \times \text{Avg\_Avg(CPI)}$
3. $\text{WCET}_3 = \text{MIC} \times \text{Max\_Avg(CPI)}$
4. $\text{WCET}_4 = \text{MIC} \times \text{Avg\_Avg(CPI)}$

For simple straight line programs, SWIC is equivalent to MIC. For programs with more complex *if* conditions, SWIC might be much larger than MIC. As a result, WCET_1 and WCET_2 are more pessimistic than WCET_3 and WCET_4 for such programs. For programs that display stable CPI behavior across inputs, Max_Avg(CPI) is not very distant from Avg_Avg(CPI). For programs which display more varied CPI behavior across inputs, Max_Avg(CPI) >> Avg_Avg(CPI). As a result, WCET_1 is more pessimistic than WCET_2 and WCET_3 is more pessimistic than WCET_4 for such programs. WCET_1 is the most *safest* estimate of all four estimates. WCET_4 is the most optimistic estimate of all and hence can be unsafe. Theoretically, WCET_1 and WCET_2 are not guaranteed to be safe as they are computed using measured CPI

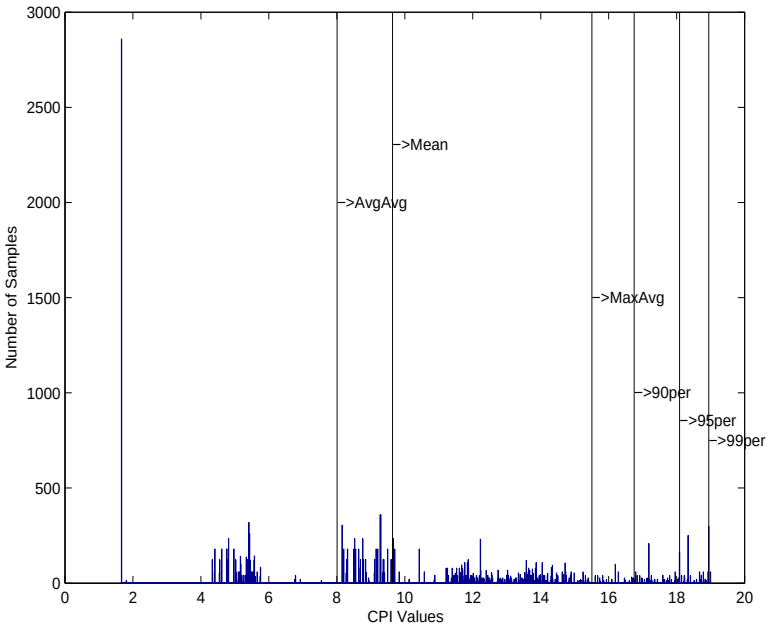


Figure 3.6: CPI distribution seen across 500 runs of *Nsch* on *Simplest* architecture with statistical candidates superimposed.

values and so are WCET_3 and WCET_4, that are computed using measured CPI and IC values.

Likewise, we compute WCET using statistical candidates - 90per(CPI) (90th percentile CPI value) and 99per(CPI) (99th percentile CPI value) to obtain four more possible combinations of IC and CPI as follows. We shall evaluate the resulting pessimism in these WCET estimates in the next section.

1. $\text{WCET}_5 = \text{SWIC} \times 90\text{per(CPI)}$
2. $\text{WCET}_6 = \text{SWIC} \times 99\text{per(CPI)}$
3. $\text{WCET}_7 = \text{MIC} \times 90\text{per(CPI)}$
4. $\text{WCET}_8 = \text{MIC} \times 99\text{per(CPI)}$

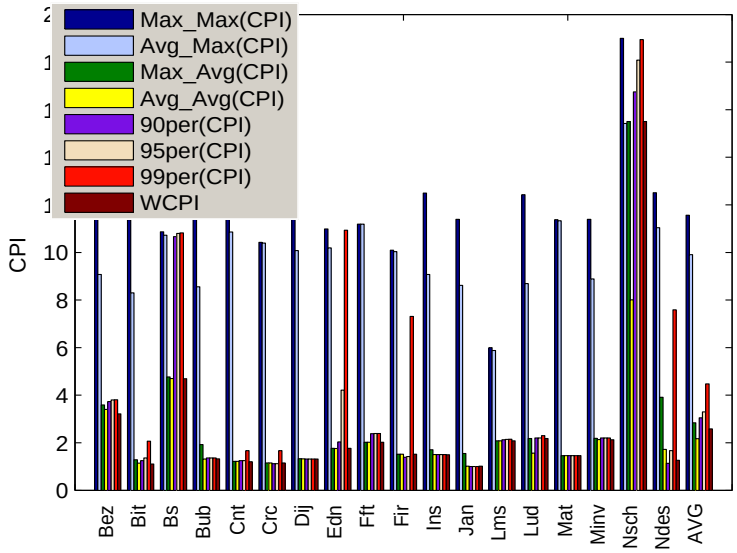


Figure 3.7: Comparison of various CPI candidates with WCPI on *Simplest* architecture.

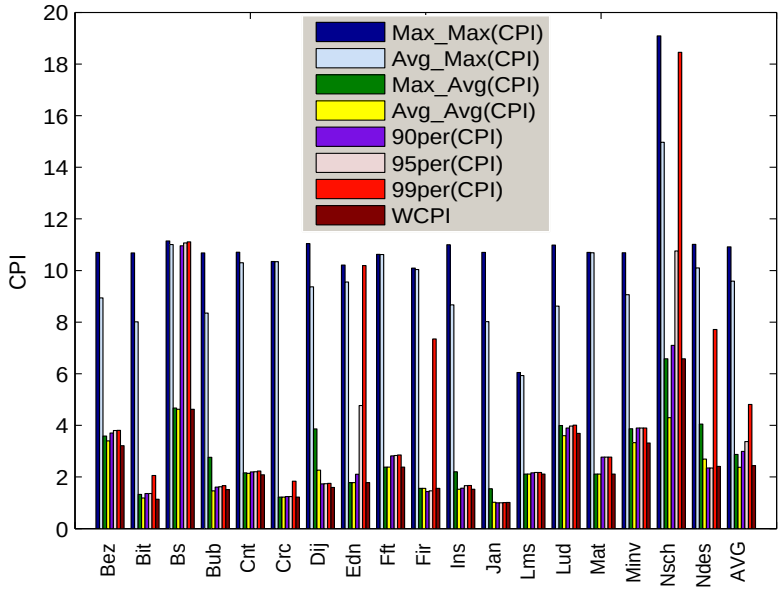


Figure 3.8: Comparison of various CPI candidates with WCPI on *Inorder_complex* architecture.

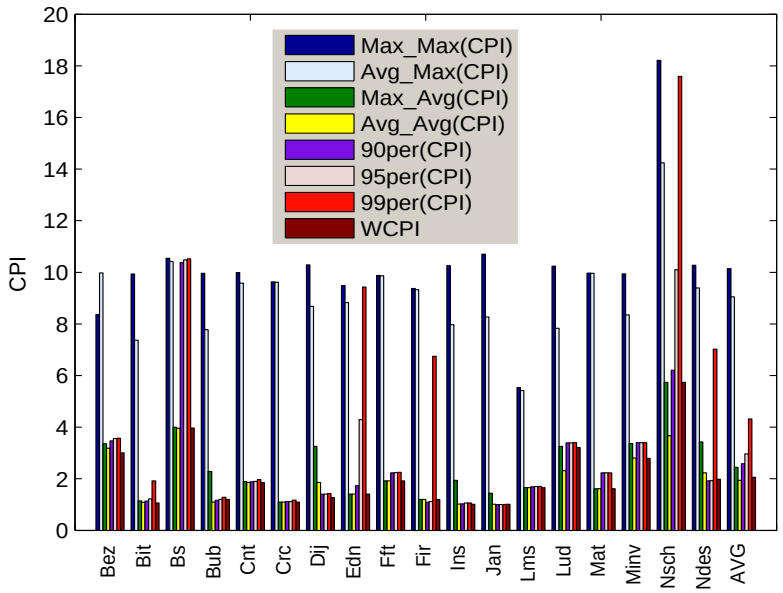


Figure 3.9: Comparison of various CPI candidates with WCPI on *Complex* architecture.

3.4 Evaluation

3.4.1 SWIC versus MIC

In this section, we shall compare the theoretical upper bound IC value of every benchmark with the corresponding maximum measured IC value. It should be noted that the number of instructions executed is determined by the ISA which is the same for all PISA architectures considered in this paper- *Simplest*, *Inorder_complex* and *Complex*. Figure 3.10 plots the ratio of computed SWIC to observed MIC for all benchmarks on PISA architectures. These ratios quantify the structural complexity of the benchmark to some extent. *Edn*, *Lms* and *Mat* are straight-line programs. Hence they execute the same number of instructions for any valid input. The structure of programs, *Bs*, *Cnt* and *Ndes* is such that, it is easy to guess the algorithmic worst case input. Hence we can guess inputs that execute SWIC number of instructions causing SWIC/MIC to be 1 for such programs. However, even after exercising the algorithmic worst case inputs for programs like *Ins*, MIC is lesser than SWIC. *Ins* contains a triangular loop nest which is not obvious by the usage of *while* loop. Hence the analysis assumes both inner and outer loop iterate for the same number of times which is not true. This can be remedied by adding manual annotations specifying a lower bound owing to the triangular loop nest. The worst case inputs for structurally complex programs such as *Dij*, *Lud*, *Minv* and *Nsch* are not known. Hence for these programs, one cannot isolate the cause for a high SWIC/MIC ratio to in-adequate coverage or structural complexity.

3.4.2 Time for SWIC Computation

In this section, we shall evaluate the time taken to compute SWIC for all benchmarks. As SWIC is computed by solving an ILP problem built by modeling the program structure in the form of flow constraints, computation of SWIC takes longer time for complex programs that are either large in size or consist of complex *if*-conditions. Figure 3.11 describes the time taken for computation of SWIC in seconds for all benchmarks. The X-axis plots benchmarks arranged in the increasing order of their structural constraints, constants and variables figuring in the ILP equation. We can observe that for most programs, SWIC is computed within a few seconds. Amongst all programs, computation of SWIC takes longest time (105 seconds) for *Nsch* which is a structurally complex program with hundreds of *if*-conditions. The number of

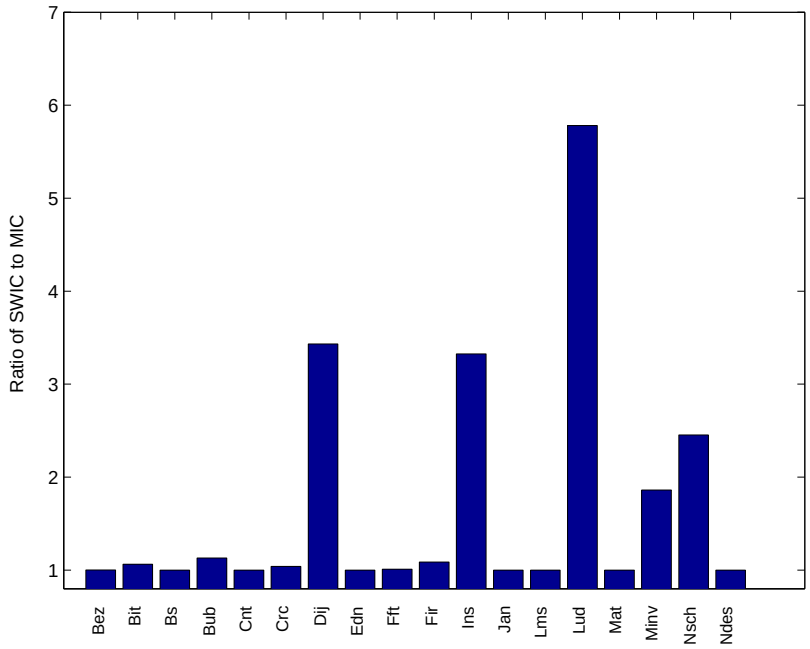


Figure 3.10: Comparison of SWIC to MIC for all benchmarks on PISA architecture.

structural constraints is 29260. As we described in section 3.1.1, SWIC can also be computed using alternate methods such as graph theoretical algorithms or tree based schema.

3.4.3 Max_Avg(CPI) versus Avg_Avg(CPI)

In this section, we shall compare for every benchmark on every architecture, ratio of Max_Avg(CPI) to Avg_Avg(CPI), observed across all inputs (Figure 3.12).[1] Benchmarks that exhibit a high ratio are said to exhibit highly varied CPI behavior across inputs. Some of the straight-line benchmarks like *Crc*, *Edn*, *Fft*, *Lms* and *Mat* exhibit the same CPI irrespective of input and this holds true across architectures. Hence the ratio of Max(CPI) to Avg(CPI) for these programs is 1. For benchmarks *Bs*, *Cnt* and *Fir* that have very few *if* conditions, the ratio of observed Max(CPI) and Avg(CPI) is less than 1.02. The closeness between observed Max(CPI) and Avg(CPI) across inputs implies that CPI behavior is uniform across inputs. This justifies our factorization of execution time into IC and CPI and also the use of overall program CPI in estimating WCET. For programs which display stable CPI behavior during the program run

[1]In the figure, Max_Avg(CPI) is referred to as Max(CPI) and Avg_Avg(CPI) as Avg(CPI) for ease of notation

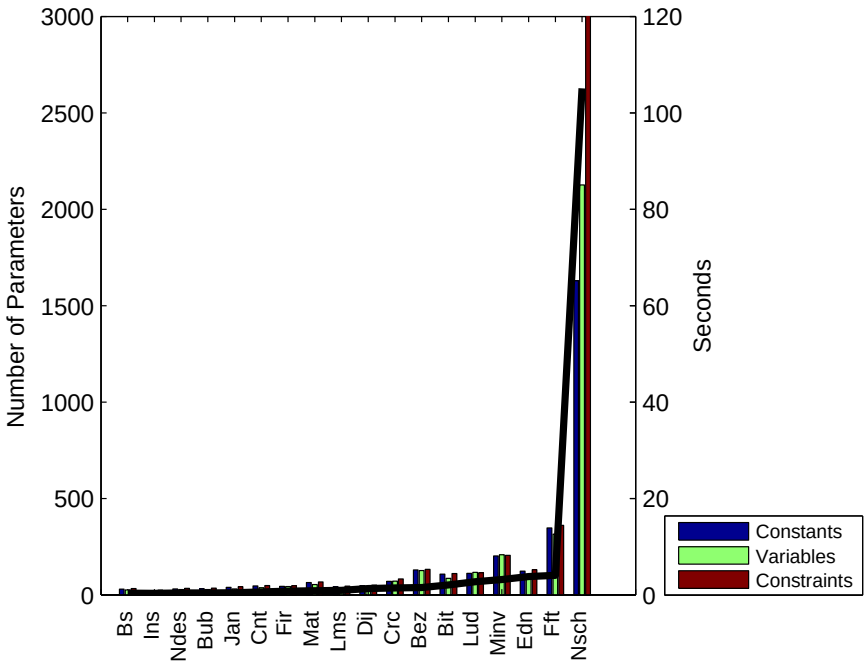


Figure 3.11: Time taken to compute SWIC on PISA architecture. Benchmarks are ordered with respect to structural complexity.

and also across runs with different inputs, the basic timing model presented in this chapter gives accurate WCET estimates.

A greater difference between Max(CPI) and Avg(CPI) implies CPI varies across inputs. Examples are *Bub*, *Ins*, *Janne*, *Lud*, *Minv*, *Nsch* and *Ndes*. One of the causes is the variation in the number of instructions executed across different inputs due to the presence of *if* conditions. Sometimes if a program execution with input i finishes earlier than input j, the average CPI observed with input i will be higher than input j. Hence Max(CPI) can pick those CPIs corresponding to executions that terminated sooner causing WCET estimates to be pessimistic compared to maximum observed cycles. In the next chapter, we shall see how we remedy this by making use of correlation between IC and CPI to optimize our timing model.

3.4.4 Comparison with Chronos

In this section, we shall compare the estimates of WCET obtained by our proposed method that uses eight combinations out of {SWIC, MIC}, analytical candidates of CPI, {Max_Avg(CPI),

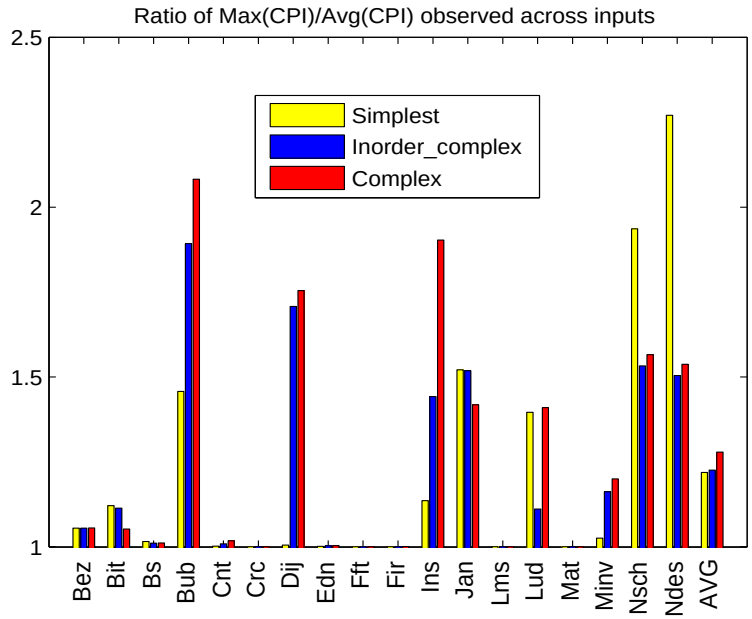


Figure 3.12: Ratio of maximum CPI to average CPI observed across inputs on all PISA architectures.

Avg_avg(CPI)} and statistical candidates of CPI {90per(CPI), 99per(CPI)} with the corresponding estimate given by Chronos[94]. We describe more details on Chronos and it's usage in the Appendix. The WCET estimate on *Simplest* architecture is obtained using *ChronosV3.0* and that on *Inorder_complex* and *Complex* is obtained using *ChronosV4.0*[94]. Chronos is a static WCET analyzer. We shall compare our results with Chronos in terms of pessimism in the WCET estimate and safety.

Figures 3.13, 3.14 and 3.15 plot the pessimism observed in WCET estimate using the four formulae proposed in this chapter using analytical CPI candidates along with the corresponding pessimism observed in WCET estimated by Chronos for PISA architectures *Simplest*, *Inorder_complex* and *Complex* respectively. The WCET estimate which has a pessimism of 1 is said to be most accurate. WCET estimates that have a pessimism greater than 1 are safe estimates as indicated in the figures. WCET estimates obtained using SWIC are closer to estimates obtained by Chronos as they are computed by static structural analysis of the benchmark. Some estimates obtained using MIC in the proposed method are observed to be unsafe.

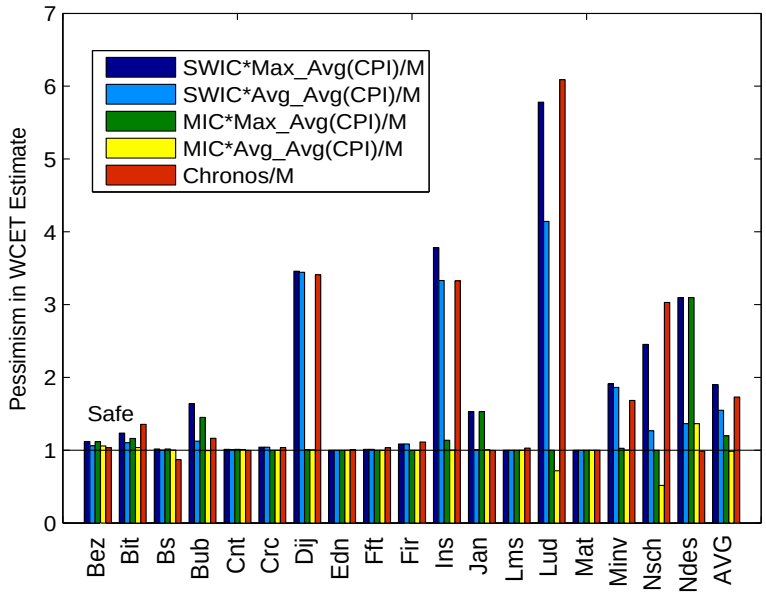


Figure 3.13: Pessimism of proposed method and Chronos on *Simplest* architecture using analytical CPI candidates.

Since the proposed method is measurement based, unless 100% coverage is assured, it is difficult to guarantee a safe WCET estimate. Chronos being a static WCET analyzer, produces a WCET estimate that is *always* safe. However, the proposed method gives safer estimates with SWIC than MIC. If safety is not a concern, but tightness is, MIC may be used.

Figures 3.16, 3.17 and 3.18 plot the pessimism observed in WCET estimate using statistical candidates. It can be observed that the pessimism in WCET estimates computed using statistical candidates is more compared to ones computed using analytical candidates. The prime reason being analytical candidates are average values and statistical candidates are tail end values. Table 5.2 describes the average pessimism in WCET estimates obtained using both analytical and statistical candidates of CPI and pessimism obtained using Chronos, observed for all PISA architectures. It can be observed that the average pessimism in WCET estimates obtained using SWIC and Max_Avg(CPI) is around twice the maximum observed cycles. The average pessimism in WCET obtained using SWIC and 99per(CPI) is around thrice the maximum observed cycles. The model proposed in this chapter is very basic as it considers the whole program as a single unit. The model will be refined further in the coming chapters to

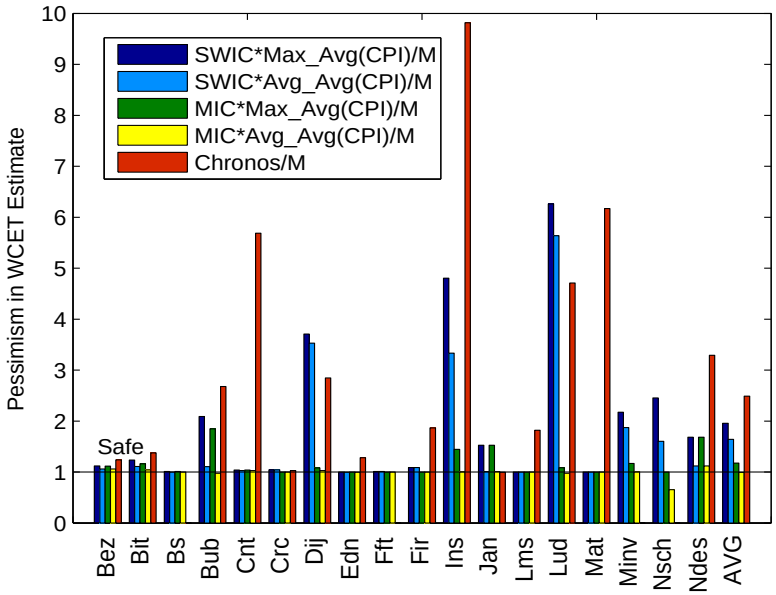


Figure 3.14: Pessimism of proposed method and Chronos on *Inorder_complex* architecture using analytical CPI candidates.

give tighter estimates.

Sensitivity to Architecture

In this section, we shall evaluate the influence of the underlying architecture on the resultant WCET estimate. Figures 3.19 and 3.20 plot for each benchmark, the variation in pessimism of WCET estimates over PISA architectures *Simplest*, *Inorder_complex* and *Complex* obtained using the proposed method and Chronos respectively. For plotting Figure 3.19, we use a combination of SWIC and Max_Avg(CPI). The proposed method gives WCET estimates for all benchmarks on all architectures. The pessimism observed in WCET estimate obtained by the proposed method for benchmarks *Bub*, *Dij*, *Ins*, *Lud*, *Minv*, *Ndes* changes with change in architecture complexity, the pessimism in all other remaining benchmarks remains more or less the same across architectures. For most benchmarks, the pessimism in WCET estimate is found to increase with increasing architecture complexity in case of Chronos. *ChronosV4.0* does not run to completion for the following programs *Bs*, *Fft*, *Minv*, *Nsch* on both *Inorder_complex* and *Complex* architectures and *Crc*, *Edn* on the *Complex* architecture. In this respect, we can say

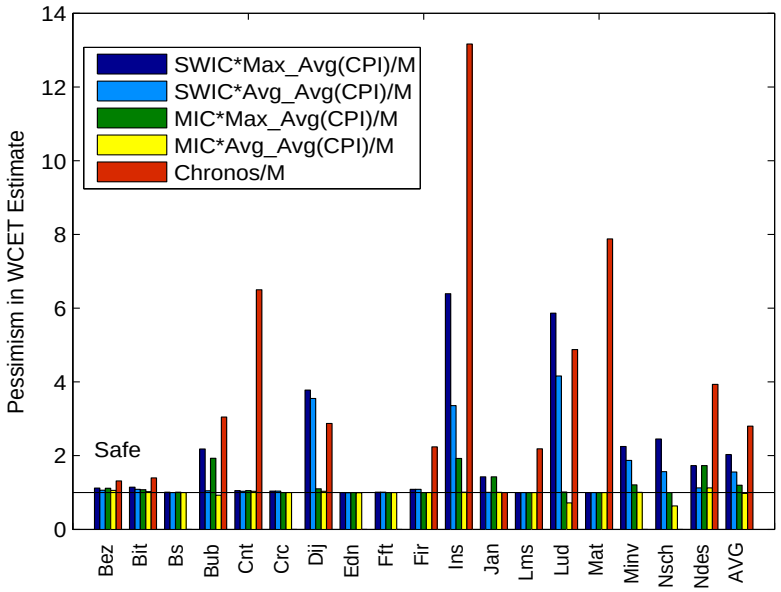


Figure 3.15: Pessimism of proposed method and Chronos on *Complex* architecture using analytical CPI candidates.

that the proposed method is more tolerant to changes in architecture.

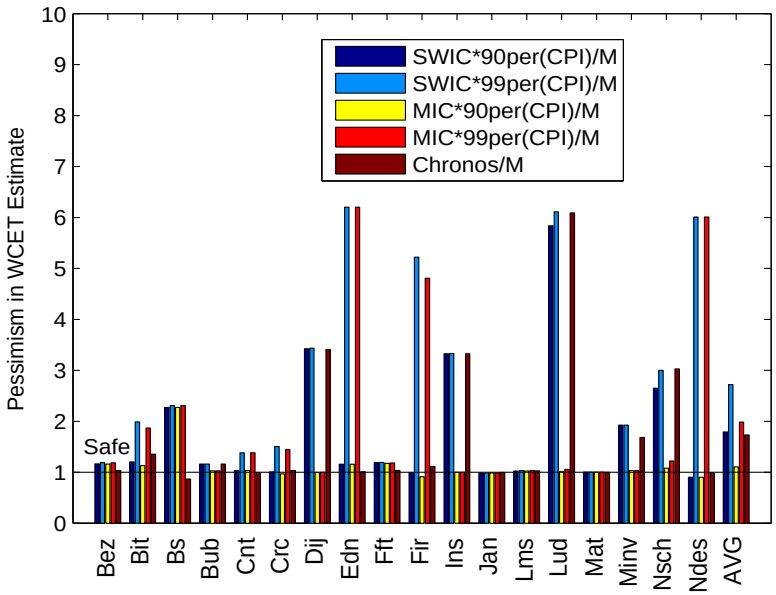


Figure 3.16: Pessimism of proposed method and Chronos on *Simplest* architecture using statistical CPI candidates.

WCET	*Simplest*	*Inorder_complex*	*Complex*
$\frac{SWIC \times Max\_Avg(CPI)}{M}$	1.89824	1.95765	2.02981
$\frac{SWIC \times Avg\_Avg(CPI)}{M}$	1.54734	1.64062	1.55611
$\frac{MIC \times Max\_Avg(CPI)}{M}$	1.19725	1.17536	1.19929
$\frac{MIC \times Avg\_Avg(CPI)}{M}$	0.983739	0.99387	0.976724
$\frac{SWIC \times 90per(CPI)}{M}$	1.79203	1.88219	1.8945
$\frac{SWIC \times 99per(CPI)}{M}$	2.72016	2.81922	2.96792
$\frac{MIC \times 90per(CPI)}{M}$	1.10422	1.15887	1.1689
$\frac{MIC \times 99per(CPI)}{M}$	1.98565	1.91699	2.04842
$\frac{Chronos}{M}$	1.7308	3.15256	4.1998

Table 3.3: Average pessimism of WCET on all PISA architectures using the proposed method and Chronos.

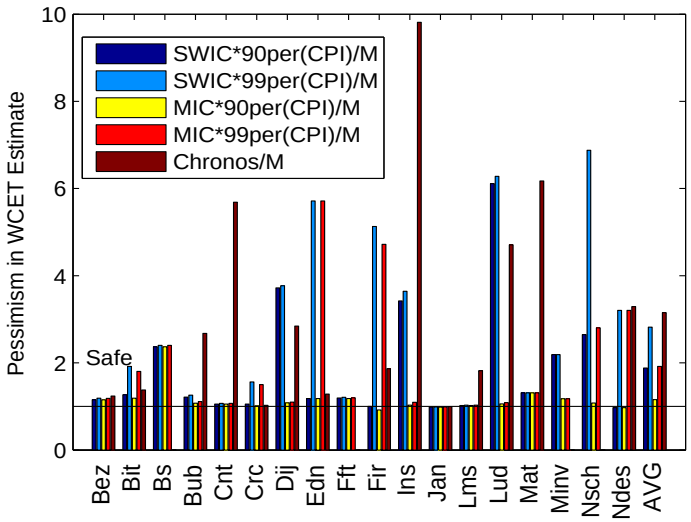


Figure 3.17: Pessimism of proposed method and Chronos on *Inorder_complex* architecture using statistical CPI candidates.

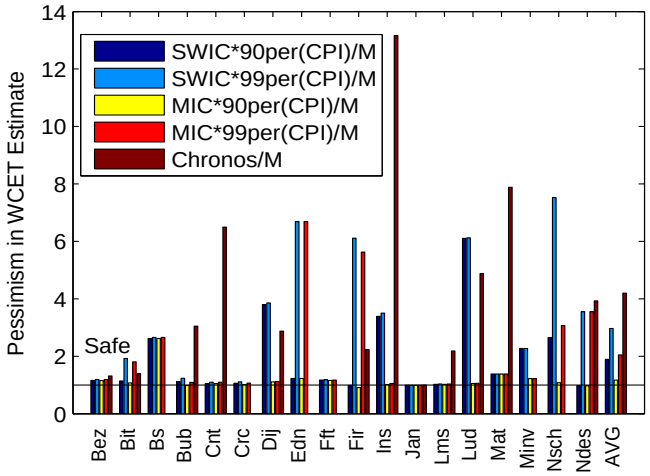


Figure 3.18: Pessimism of proposed method and Chronos on *Complex* architecture using statistical CPI candidates.

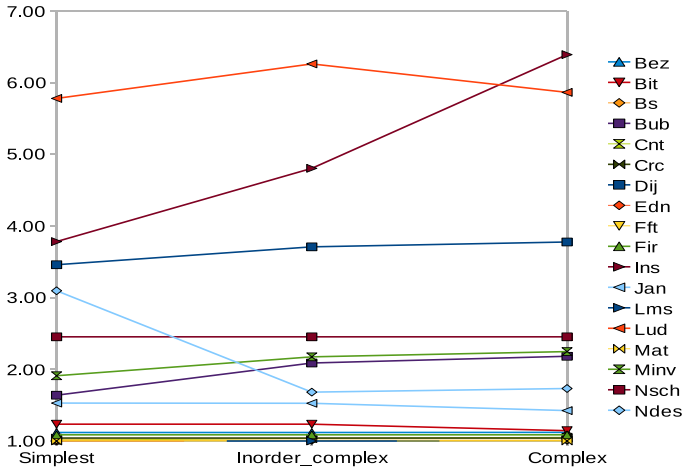


Figure 3.19: Variation in pessimism of WCET estimates obtained by proposed method over PISA architectures.

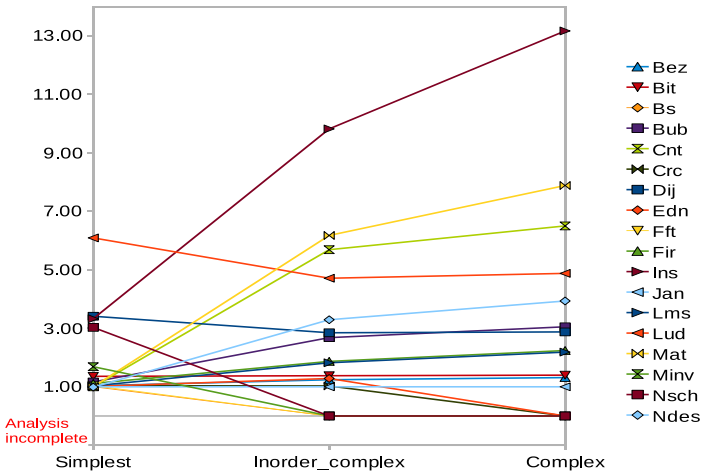


Figure 3.20: Variation in pessimism of WCET estimates obtained by Chronos over PISA architectures.

3.5 Related Work

Corti et al [52] presents a hybrid measurement based WCET analysis technique that uses graph longest path search algorithm to estimate WCET. The nodes of the graph are basic blocks. The weight of an edge represents the cost of executing the preceding basic block in processor cycles which is computed in terms of CPI. CPI is broken down into busy cycles, stall cycles and cycles taken by cache misses. These components are further broken down so that they can be represented by available hardware performance counters. The equation is evaluated by plugging in the values of the hardware performance counters. The CPI along with the number of instructions in the basic block gives the cost of executing the basic block. The technique yields only approximate estimates as event counting is not precise as events cannot be attributed to specific instructions more so for out of order architectures. Moreover many events are not disjoint, hardware performance monitors do not report their intersection. Our technique uses only one parameter that has to be measured which is the CPI. So we do not face the problem of mapping events to instructions as we measure average CPI over intervals of 1000 instructions. We estimate WCET as a product of maximum IC and maximum CPI.

Compared to other statistical methods that use extreme value theory to fit a curve on measured end to end execution times[38, 97, 98], we plot measured CPI samples collected over a large number of runs and calculate percentile values to approximate worst case CPI. Our estimates are only approximate and can be further improved. The reasons are- firstly we assume that the samples we have is exhaustive which might not be true. Percentiles only determine the relative standing of a value in a population, a more accurate measure would be the probability of a particular unknown CPI being greater than the mean. This aspect will be explored in Chapter 6. Factorization of execution time into components such as number of instructions, cache misses et cetera, has been suggested by Griffin et al[22] in the context of statistical WCET analysis. Post factorization, it is easier to collect independently and identically distributed samples from each of the components and estimate WCET by applying statistical analysis to each component and picking the maximum. Our work builds on such a factorization of execution time into IC and CPI. We shall see in coming chapters, that this factorization has many benefits to offer with respect to timing analysis.

3.6 Summary

This chapter proposes a basic formula to estimate WCET of a program in terms of whole program instruction count (IC) and cycles per instruction (CPI). Both static (theoretical upper bound on IC) and dynamic estimates (maximum observed IC) of worst case IC, WIC are examined along with various analytical and statistical candidates for worst case CPI, WCPI are examined. If adequate coverage by test inputs is assured, maximum observed instruction count (MIC) may be used as WIC. If coverage is an issue, the theoretical upper bound on IC, SWIC, may be used as WIC. Among the analytical candidates, Max_Avg(CPI) is the most suitable choice for WCPI. If a softer estimate is desired, Avg_Avg(CPI) may be used. Among the statistical candidates, the 99th percentile CPI value can render pessimism in the resultant WCET estimate. The 90th percentile CPI comes closest to the Max_Avg(CPI) value, but it is very much dependent on the CPI distribution and hence inputs that form the test input set. In the next chapter, we shall see that in many programs, correlation exists between instruction count(IC) and cycles per instruction(CPI) for a given program on a given architecture. We shall use this correlation to obtain an improvised WCET estimate.

Chapter 4

Relative Roles of IC and CPI in WCET Estimation

The previous chapter introduced the basic framework to estimate program WCET as a product of worst case instruction count and worst case cycles per instruction, considering the program as a whole. Two candidates for worst case IC- theoretical upper bound on IC (SWIC) and maximum observed instruction count (MIC) were proposed and evaluated. Several analytical and statistical candidates for worst case CPI were also put forth and evaluated.

Considering maximal values for both IC and CPI is understandable if IC and CPI are totally independent variables. However, we shall see in this chapter that in many programs, there exists a relation between measured IC and CPI values such that, CPI = f(IC). Note that such a functional relationship is derived from measurements and hence reliable. This implies that CPI can be predicted for any given IC. If there are multiple CPI points for a given IC, we consider the maximum CPI to determine the functional relation.

Hence, estimated execution time,

$$\widehat{ET} \;=\; IC \times f(IC) \tag{4.1}$$

Since we are interested in determining Max $\widehat{ET}$, assuming f(IC) to be continuous and differentiable, this occurs at IC=IC$_{max}$ obtained by solving the equation,

$$\frac{d(\widehat{ET})}{d(IC)} = 0 \tag{4.2}$$

thereby setting, $\frac{d(IC*f(IC))}{d(IC)}$=0, we get,

$$Max\ \widehat{ET} = IC_{max} * f(IC_{max}) \tag{4.3}$$

Since SWIC is a theoretical upper bound on IC, we consider SWIC as a candidate for IC_{max}. Hence estimated execution time based on SWIC is computed as,

$$\widehat{ET} = SWIC * f(SWIC) \tag{4.4}$$

However there may be a function f(IC) such that Max $\widehat{ET}$ computed by Eq(5) is greater than that computed by Eq(6) in which case Max $\widehat{ET}$ can be taken as $\widehat{WCET}$. If IC_{max} corresponds to one of the measured points then this estimate coincides with measured maximum cycles, *M*. If there is no functional relationship between IC and CPI then $\widehat{WCET}$ is estimated as the maximum of

a) *SWIC * Maximum of measured CPI* and

b) *Measured maximum cycles,* M

4.1 Relationship between IC and CPI

In this section, we shall study how IC and CPI co-vary for each benchmark on the architectures considered in this work. The benchmarks are run with a large number of inputs generated that satisfy structural coverage and cover the whole range of possible data as described in the previous chapter. In addition to such inputs, we include inputs that execute maximum number of instructions which is easy to derive for some well known benchmarks. For example, *bub*(Bubble sort) executes maximum number of instructions when the input array is reverse sorted. Execution of the benchmarks with the test input set generates IC and CPI vectors that consist of observed instruction counts, observed average CPI for each test input. These IC and CPI vectors are analyzed by generating scatter plots of IC versus CPI and computing the covariance between IC and CPI.

4.1.1 Scatter Plots of IC versus CPI

After obtaining the IC and CPI vectors for each (benchmark, architecture) pair, we generate a scatter plot of IC versus CPI. Each co-ordinate point on the scatter plot is defined as an ordered pair (x_i, y_i) with the X-axis representing IC and Y-axis representing CPI. The input which causes the program to execute the maximum number of cycles, represented by an (ic, cpi) value is depicted as a '▷' and is superimposed upon the scatter plot. Distinctive patterns in the scatter plot signify a definite relationship between IC and CPI which are considered as two independent variables at present. Absence of any predominant pattern indicates that IC and CPI do not have any predictable relation. Inputs for which IC and CPI are very close appear to lie on the same point in the scatter plot. Some of the interesting patterns are shown in the following figures.

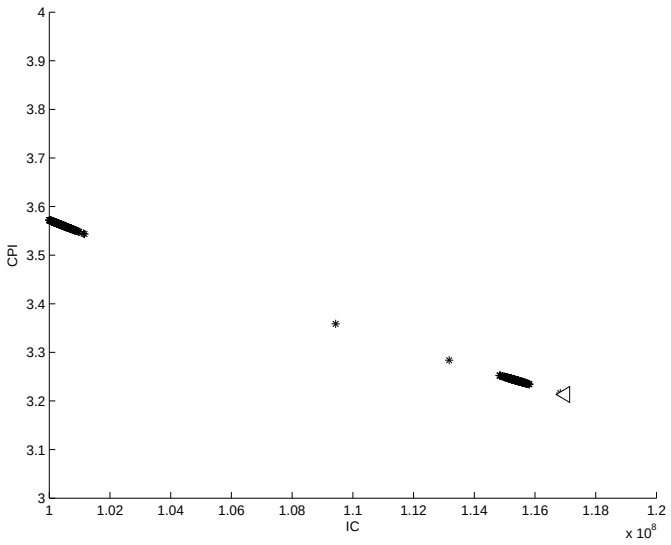


Figure 4.1: Scatter plot of CPI versus IC for *bez* on *inorder_complex* architecture.

In case of benchmark *bez*, on *inorder_complex* architecture, we see that CPI is inversely related to IC as shown in Figure 4.1. Statistical WCET analysis techniques[59] suggest the 90th or the 95th percentile point as a representative WCET. This implicitly assumes that 90% or 95% of the inputs lie below the input that takes the longest time. However it can be seen that there are inputs that can cause the program to execute longer lying outside this set as illustrated by the symbol '▷' in Figure 4.1. CPI and IC are inversely related for benchmarks

like *Bs*, *Cnt*, *minv*. In case of benchmark *ndes*, CPI rapidly decreases with increase in IC at first but eventually saturates as shown in Figure 4.2 on *inorder_complex* architecture. *ndes* shows a similar trend on other architectures as well. Considering only the maximal points in Figure 4.3 that plots correlation for *Bub* on *inorder_complex* architecture, CPI changes very slowly with increase in IC.

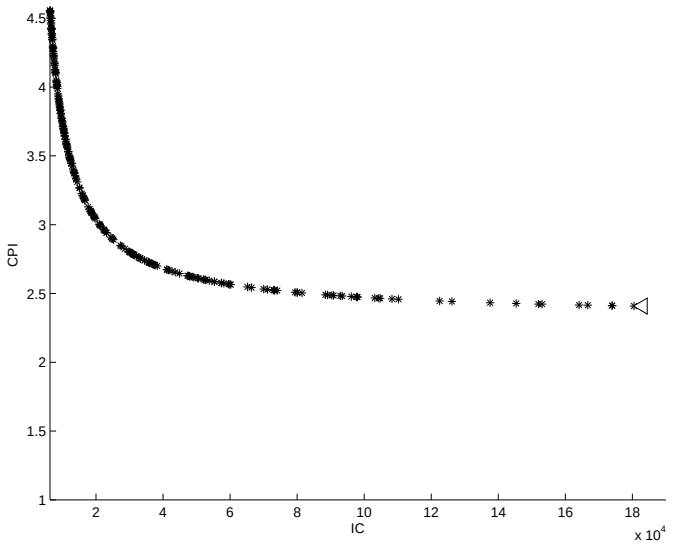


Figure 4.2: Scatter plot of CPI versus IC for *ndes* on *inorder_complex* architecture.

Programs such as *nsch* exhibit a direct relationship between IC and CPI on *complex* architecture (Figure 4.4). With increasing IC, CPI is also observed to increase. *nsch* behaves similarly on other architectures considered in this thesis. Another benchmark that exhibits positive correlation is *lud* on *simplest* architecture (Figure 4.5).

Some benchmarks are observed to exhibit negligible correlation between IC and CPI irrespective of the inputs presented. The scatter plot for such benchmarks reduces to a single dense point as in the case of *Fft* on *inorder_complex* architecture, shown in Figure 4.6. Such independence on input is observed in other benchmarks like *Crc*, *Edn*, *Fir Lms* and *Mat*.

Certain benchmarks like *ins*(Insertion Sort) exhibit a near-constant CPI with increasing IC. Figure 4.7 shows the scatter plot for *ins* on *complex* architecture. A similar trend is observed in *dij*(Dijkstra) (Figure 4.9). However there are benchmarks where the relationship between IC and CPI is not clear. In the scatter plot of such benchmarks, IC and CPI may appear to be directly related at some points while at other points they may be inversely related or not

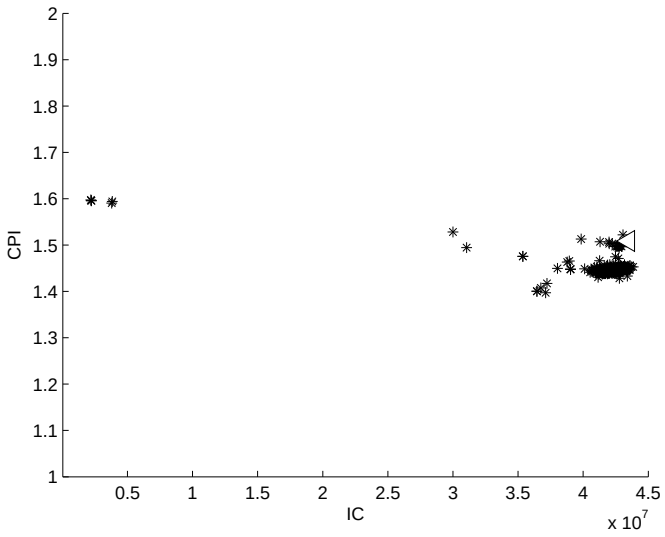


Figure 4.3: Scatter plot of CPI versus IC for *bub* on *inorder_complex* architecture.

related at all. An example is *lud* on *inorder_complex* architecture(Figure 4.8).

4.1.2 Quantifying Cross Correlation by Covariance Matrix

Let A and B be two random vectors with the corresponding mean values, μ=$\{\mu_a, \mu_b\}$. The dispersion of $A_i(B_i)$ around its mean $\mu_a(\mu_b)$ is measured by its variance that form the diagonal elements of the covariance matrix. For ease of notation, lets denote variance of elements of A and B by σ_{11} and σ_{22} respectively. The cross variance $\sigma_{12}(=\sigma_{21})$ represents the mutual dependency between A and B. In our case A denotes the IC vector and B denotes the CPI vector. μ_a represents the mean IC or average instruction count observed across all inputs. Hence σ_{11} denotes variance in IC. μ_b denotes the global average CPI observed across all inputs(termed as Average_Average(CPI) in the previous chapter). Hence σ_{22} denotes variance in CPI. The instruction count values are orders of magnitude greater than the corresponding CPI values. Hence we normalize IC and CPI vectors with respect to their respective measured maximum values before computing the covariance matrix. The elements of the covariance matrix for all the programs when run on PISA architectures *simplest*, *inorder_complex* and *complex* are shown in Table 4.1 respectively. The σ_{11} column values (variance in IC) for all PISA architectures are the same as the instruction set for all PISA architectures considered is identical.

We observe that the covariance matrix quantifies the scatter plots in terms of individual

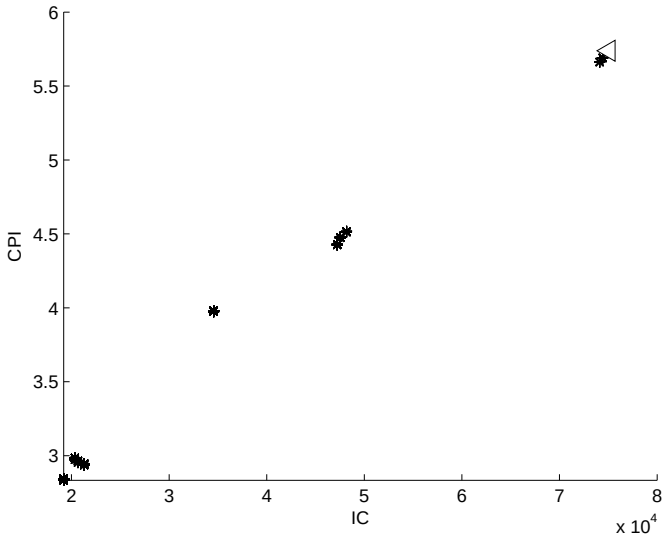


Figure 4.4: Scatter plot of CPI versus IC for *nsch* on *complex* architecture.

variance in IC, CPI and the cross correlation between IC and CPI. Benchmarks that exhibit inverse relation between IC and CPI in the scatter plot have negative values for $\sigma_{12}(\sigma_{21})$. Benchmarks that show a direct relationship between IC and CPI have positive values. Benchmarks that have very less σ_{11}, $\sigma_{12}(\sigma_{21})$, σ_{22} values exhibit a narrow cluster in the scatter plot. Zero values in the covariance matrix indicates absence of any variance across inputs. For example, *edn*, *mat*, *fft*, *lms*, *fir* and *crc* has all its variances and cross correlation values as zeroes and hence appears as a dense dot in the scatter plot. The benchmarks are grouped accordingly based on the covariance matrix values and scatter plots in the tables.

We also observe from the covariance matrix values that the σ_{11} values(variance in CPI) are much lesser than the corresponding σ_{22} values (variance in IC). The reason is average CPI is relatively more stable compared to IC. Depending on program structural complexity, instruction count(IC) can vary widely across inputs. But CPI is more dependent on the underlying architecture and is generally found to vary in small degrees in comparison to IC.

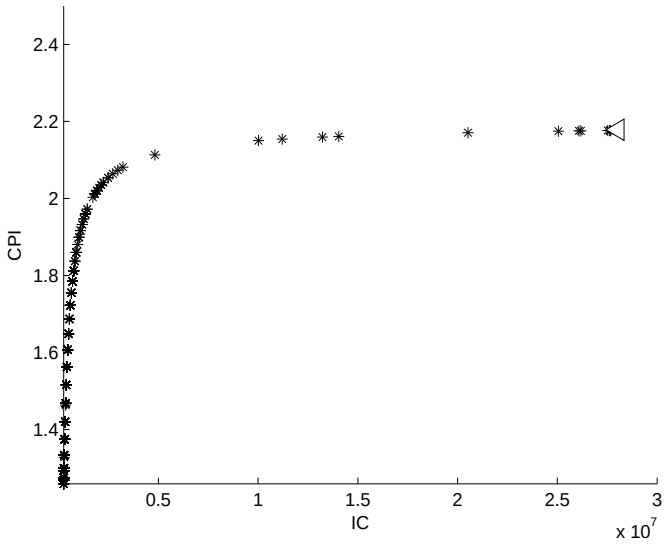


Figure 4.5: Scatter plot of CPI versus IC for *lud* on *simple* architecture.

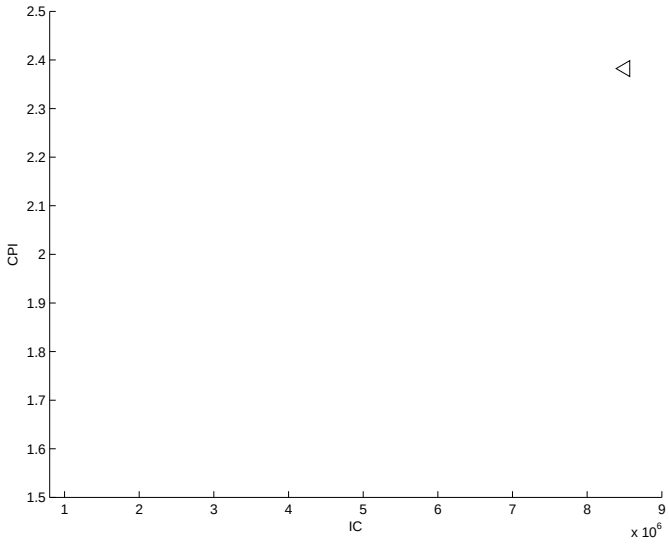


Figure 4.6: Scatter plot of CPI versus IC for *fft* on *inorder_complex* architecture.

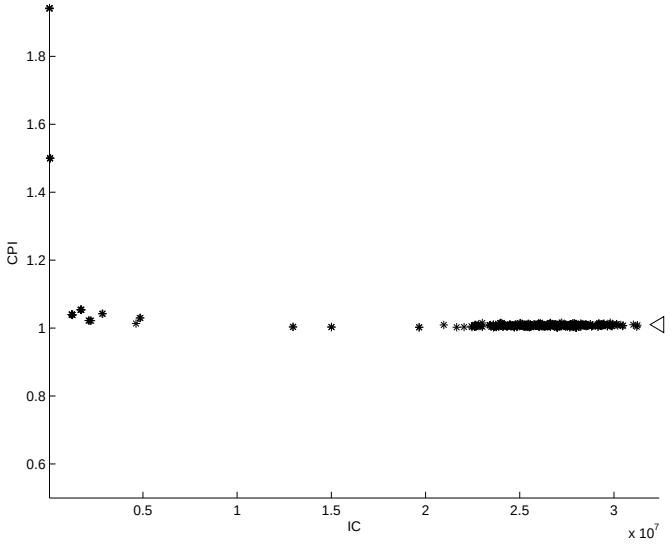


Figure 4.7: Scatter plot of CPI versus IC for *ins* on *complex* architecture.

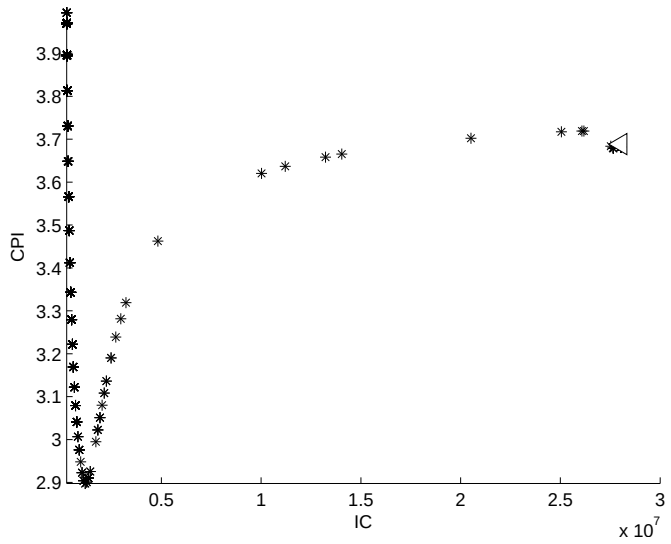


Figure 4.8: Scatter plot of CPI versus IC for *lud* on *inorder_complex* architecture.

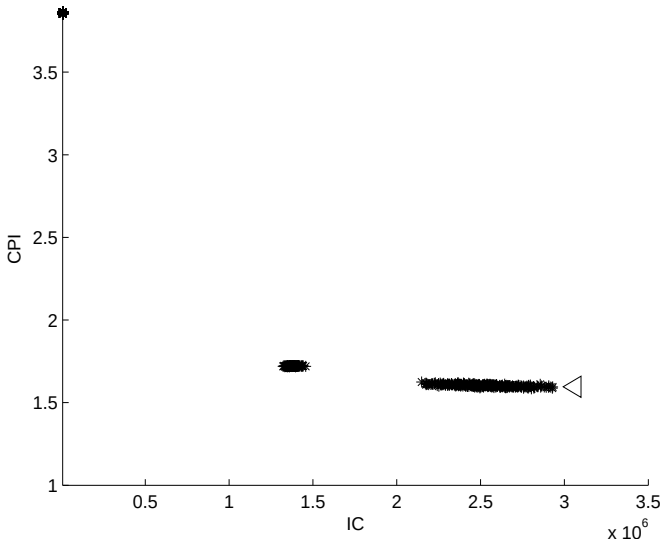


Figure 4.9: Scatter plot of CPI versus IC for *Dijkstra* on *inorder_complex* architecture.

Benchmark	σ_{11}	σ_{22}			$\sigma_{12}(\sigma_{21})$		
		sim	inc	com	sim	inc	com
Class I : Negligible variance in IC and CPI							
crc	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
edn	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
fft	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
fir	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
lms	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
mat	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Class II : Negative correlation							
bez	2.0699	1.0043	1.0052	1.0313	-1.4418	-1.4424	-1.4610
bit	1.5748	0.0496	0.0530	0.0144	-0.2627	-0.2757	-0.1444
bs	0.0071	0.0055	0.0037	0.0040	-0.0044	-0.0030	-0.0026
bub	7.9896	0.4027	0.9193	1.1373	-1.1594	-2.0284	-2.0952
cnt	0.1430	0.0046	0.0386	0.0729	-0.0256	-0.0722	-0.0935
minv	1.6859	0.0030	0.0975	0.1391	-0.0697	-0.3945	-0.4711
Class III : Near-constant CPI with increasing IC							
dij	61.5444	45.0631	33.3215	35.9115	-47.3924	-41.5692	-43.2699
ins	18.2686	0.0809	0.5083	1.1672	-0.6416	-1.6774	-2.4700
jan	62.9813	34.0591	34.0359	30.5467	-41.0942	-41.0786	-38.9158
ndes	93.0716	24.0540	9.9804	10.7526	-38.9250	-25.0814	-26.0349
Class IV : Positive correlation							
lud	41.4837	9.4686			14.2630		
nsch	33.8862	32.4919	15.5365	16.8486	32.7998	22.7565	23.5978
Class V : Mixed correlation							
lud	41.4837		3.2984	4.6620		0.6662	13.4533

Table 4.1: Elements of Covariance Matrix for PISA architectures- *simplest*, *inorder_complex* and *complex*. Grouping is based on values of covariance matrix and scatter plots.

4.2 Implications of IC-CPI Relationship

We began by assuming that the two parameters IC and CPI are random independent variables. We sought to find out whether the two variables are related in any way by analyzing scatter plots. By looking at the scatter plots and the covariance matrix values, it is clear that in most cases, there exists a definite relationship between IC and CPI. Based on this relationship, we classify benchmarks as follows.

4.2.1 Benchmark Classification

- Class I: Negligible Variance and Cross-correlation:
  Many benchmarks such as *Crc*, *Edn*, *Fir*, *Fft*, *Lms*, *Mat* show very little variance in both IC as well as CPI irrespective of input and architecture as we have seen. These benchmarks are composed of straight line code with negligible number of simple if-conditions leading to a highly predictable instruction execution pattern. They access the same set of data in a repetitive manner which might be the reason for negligible variance in IC and CPI across all inputs. Their scatter plots are composed of a small dense region which covers all the inputs. For such benchmarks, one need not carry out extensive testing for gathering experimental data required for measurement based WCET analysis techniques. Estimating WCET for such benchmarks is trivial and involves computing the product of IC and CPI corresponding to the single point in the scatter plot.

- Class II: Negative Cross-correlation:
  Benchmarks such as *Bez*, *Bit*, *Bs*, *Cnt*, *Jan*, *Minv*, *Ndes* show an inverse relationship between IC and CPI for all architectures considered in the thesis. These benchmarks are composed of a small number of if-conditions and loops that repeatedly work on the same set of data in a well-defined sequence causing a lot of hits in the data cache thereby bringing down the overall execution time. This could be the reason program cycles do not increase in the same rate as the rate of increase of in IC leading to a curve of negative slope. The inverse relationship makes it easy to estimate WCET for such benchmarks.

- Class III: Near-constant CPI with increasing IC:
  Benchmarks such as *Dij*, *Ins*, *Jan* and *Ndes* exhibit near constant CPI with increase in IC. The same constant CPI along with the theoretical upper bound of IC, SWIC, is used

to estimate WCET for such benchmarks.

- Class IV: Positive Cross-correlation:

  Benchmarks such as *Nsch*, exhibit a high degree of positive cross correlation between IC and CPI. *Nsch* is dominated by complex if-conditions difficult to predict. Hence with execution of more instructions, more branch prediction misses might lead to increase in CPI. *Lud*(*simplest*) exhibits a positive cross-correlation which seems to saturate with increase in IC. Such benchmarks need to be investigated further as there might be always an input that can cause a greater CPI and IC value which will result in a higher WCET. Such benchmarks are good candidates for WCET benchmarks and would need extensive testing for experimental data collection in measurement based WCET analysis techniques.

- Class V: Mixed Cross-correlation:

  In the case of *lud* on *inorder_complex* and *complex* architectures, initially, there appears to be an inverse relationship between IC and CPI. However with increase in IC, the relationship appears to be direct and later saturates before becoming inverse again as IC tends to maximum measured IC. The actual magnitude of cross-correlation can be positive or negative. Such benchmarks have to be investigated further to ascertain the relationship between IC and CPI. These benchmarks also need extensive testing for experimental data collection in measurement based WCET analysis techniques. These are also ideal candidates for WCET benchmarks. Programs which display a random correlation between IC and CPI may also be treated as belonging to this class.

4.2.2 Optimized WCET Estimation

In the previous chapter, we formulated program WCET as a product of maximal IC and maximal CPI. We treated IC and CPI as two independent random variables. During the course of this chapter, we saw that they are indeed correlated in most cases. This correlation gives us an opportunity to improve upon our original formulation of WCET estimation. For maximal IC, we use the theoretical upper bound on IC, SWIC, as it cannot be surpassed by any input. Instead of using maximal CPI, we fit a curve to the points in the scatter plot and predict CPI as a function of SWIC using the curve. The product of SWIC and f(SWIC) gives us a precise WCET.

In the following scatter plots, we continue to use a '▷' to indicate (IC, CPI) that caused the program to run for maximum number of cycles. A vertical dashed line is drawn at IC=SWIC. A horizontal dashed line is drawn at CPI used to estimate WCET by the proposed method and is denoted by f(SWIC) in the scatter plot. The point (SWIC, f(SWIC)) is denoted by a square. Although *Chronos* estimates execution time in terms of whole program cycles, for comparison purposes, we compute the inherent CPI that has been used by *Chronos* as $\frac{Cycles\ estimated\ by\ chronos}{SWIC}$, which is indicated by a horizontal dashed line indicated by CPI=$\text{CPI}_{chronos}$. If $\text{CPI}_{chronos}$ is lesser than f(SWIC), *Chronos* has performed better than the proposed method. On the other hand if it is greater than f(SWIC), the proposed method better than *Chronos*. The estimated WCET is validated by comparing it with measured maximum cycles, M. The product of SWIC and f(SWIC) is the WCET estimated by the proposed method, $\widehat{WCET}$. We compare $\widehat{WCET}$ with the estimate made by the static WCET analyzer *Chronos*. The results are described in Table 4.2. We now demonstrate the derivation of f(SWIC) using a few examples.

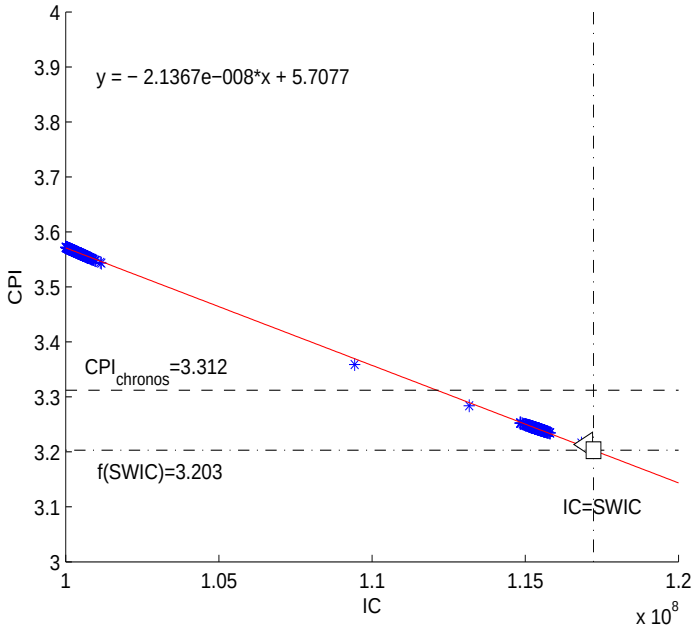


Figure 4.10: Computing f(SWIC) for *bez* on *inorder_complex* architecture.

In Figure 4.10, we revisit the scatter plot for *Bez* on *inorder_complex* architecture. The points suggest a linear relationship between IC and CPI. The negative slope indicates a negative

correlation. Hence we fit the points using a straight line as shown. The optimum CPI derived using the functional relationship, f(SWIC) is also depicted. The inherent CPI used by *Chronos* is also indicated, which is observed to be 3.4% higher than f(SWIC).

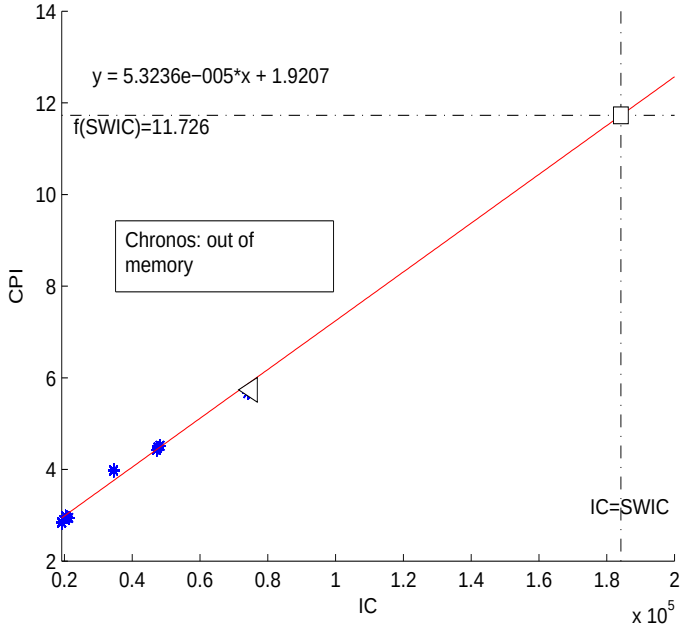


Figure 4.11: Computing f(SWIC) for *nsch* on *complex* architecture.

In Figure 4.11, we revisit the scatter plot for *nsch* on *complex* architecture. The points suggest a linear relationship between IC and CPI. The positive slope indicates a direct correlation. Hence we fit the points using a straight line as shown. The optimum CPI, f(SWIC) is almost twice the CPI corresponding to the input that executes for maximum number of cycles. The inherent CPI corresponding to *Chronos* is unknown as *Chronos* runs out of memory while analyzing *nsch* on *complex* architecture. Another interesting case occurs in benchmark *lud* on the *inorder_complex* architecture where at some points in the scatter plot, the IC and CPI are directly related while in the other parts they appear to be inversely correlated. In this case, we choose the maximal CPI point leaving out inputs that quickly terminate the program as shown in Figure 4.12. We notice that the resultant f(SWIC) is 23.8% pessimistic than $\text{CPI}_{chronos}$.

Considering, Fft (Figure 4.6) again, which shows the scatter plot on *inorder_complex* architecture, the IC and CPI values are concentrated around a dense cluster irrespective of the

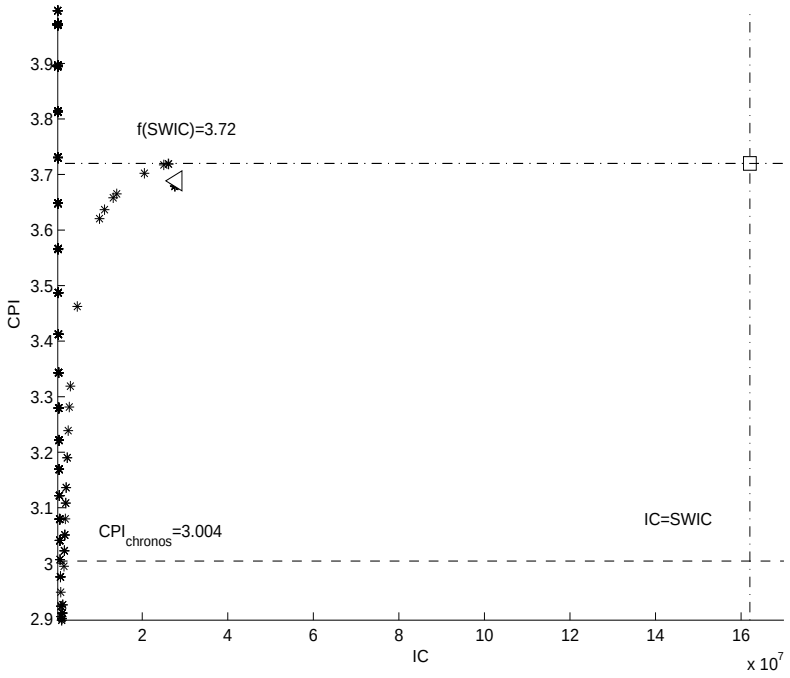


Figure 4.12: Computing f(SWIC) for *Lud* on *inorder_complex* architecture.

input. The difference between the theoretical upper bound on number of instructions executed, SWIC and the maximum number of instructions observed, MIC is very small in the case of *Fft* as it is composed of a very simple structure. Hence f(SWIC) is the same as the CPI observed for the input that caused the program to execute for the maximum number of cycles. WCET is estimated as the product of f(SWIC) and SWIC. *Chronos* does not complete analysis of *Fft* on *inorder_complex* architecture due to memory issues.

As a final example, we revisit the scatter plot for *Ins* on *complex* architecture. The CPI value saturates quite early and is unchanged with increase in IC. Hence we extrapolate and use the same CPI value for SWIC as shown in Figure 4.13. The inherent CPI used by *Chronos* is observed to be close to 5 times f(SWIC). f(SWIC) is likewise computed for rest of the programs and are compared with WCET estimates made by the static WCET analyzer *Chronos*. The results are tabulated in Table 4.2.

For each architecture, the WCET estimate derived by using the functional relationship between IC and CPI is validated first by comparing with the maximum observed cycles, *M*. The improvement in accuracy is noted by comparing the estimate with that obtained by *Chronos*.

The class of the benchmarks are also indicated alongside. It can be observed that in most cases the resultant WCET estimate is safe. The unsafe estimates occur in cases of benchmarks that exhibit a negative correlation between IC and CPI. For these benchmarks, the estimate is short of M by less than 0.2%.

In case of most benchmarks, the resultant WCET estimate is more accurate compared to estimates obtained by *Chronos*. Benchmarks that exhibit either mixed relationship or a direct correlation between IC and CPI lead to pessimistic WCET estimates. For *Lud* that exhibit a mixed relationship, the resultant WCET estimate is 23% and 20% pessimistic compared to *Chronos* on *inorder_complex* and *complex* respectively. Another example is *Nsch* that exhibits a positive relationship between IC and CPI, the resultant WCET estimate is twice that compared to *Chronos*.

In the case of *Dij*, in spite of CPI saturating with increasing IC, the resultant WCET estimate is pessimistic compared to *Chronos* by 20% and 19% on *inorder_complex* and *complex* respectively. Similarly, even though *Minv* exhibits a negative correlation between IC and CPI, the resultant estimate is 6% more pessimistic compared to *Chronos* on *simplest* architecture. An overestimation of SWIC could be the possible reason for the pessimism for classes of benchmarks that exhibit a negative or near-constant correlation between CPI and IC.

Figures 4.14, 4.15 and 4.16 compare the CPI derived on the basis of IC-CPI relationship with worst case CPI considered in the previous chapter, Max_Avg(CPI), which is the maximum of average CPI observed across inputs on *simplest*, *inorder_complex*, *complex* architectures respectively. For comparison purpose, we also plot the inherent CPI used by *Chronos* alongside the bars. In the case of *simplest* architecture (Figure 4.14), there is no major difference between f(SWIC) and Max_Avg(CPI) and CPI_chronos. That is because the architecture is very simple and has no data cache and branch predictor. *Nsch* is the only benchmark where f(SWIC) is more pessimistic than Max_Avg(CPI) due to the steep slope of IC-CPI curve and this trend is consistent across architectures. If we look at the other architectures, we find that for most benchmarks, f(SWIC) improves upon Max_Avg(CPI) and CPI_chronos.

In contrast to other WCET analysis methods, who work with execution time of program components in terms of cycles[29, 6, 66, 55], we factorize execution time into instruction count and CPI. We shall now see that *Chronos* estimates IC and CPI pessimistically for some programs as it considers execution time as one whole parameter. Splitting the WCET into two

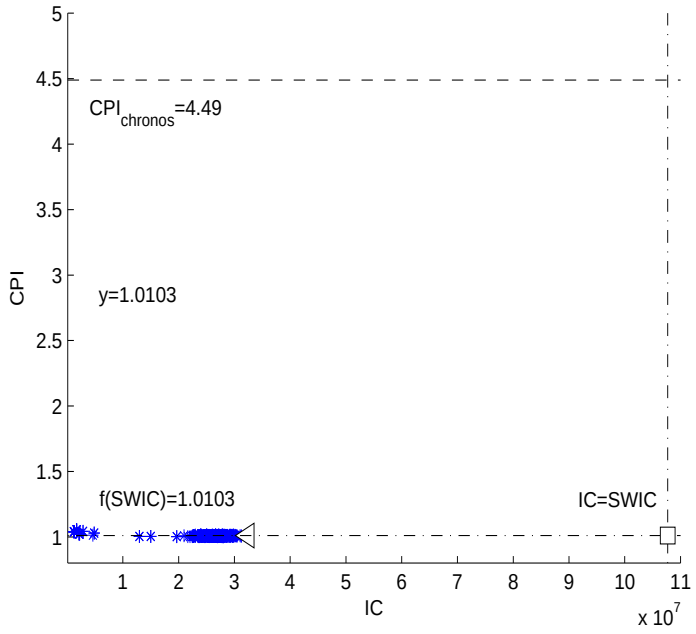


Figure 4.13: Computing f(SWIC) for *Ins* on *complex* architecture.

factors such as IC, CPI, gives us a fair idea as to which component is responsible for pessimism in WCET estimate. Depending on whether WCET is more sensitive to IC or CPI, additional efforts can be directed to either structural analysis or architectural modeling. Figures 4.17, 4.18 and 4.19 plot the factors responsible for a pessimistic estimate. Estimates that are accurate are close to 1. Overestimation in IC is plotted as the ratio of SWIC to maximum observed instruction count. Overestimation in CPI is derived as a ratio of overestimation in cycles to the overestimation in IC. It is interesting to see that in some benchmarks like *Matmul*(*inorder_complex, complex*), *Lms*(*inorder_complex, complex*), *Fir*(*inorder_complex, complex*, *Cnt*(*inorder_complex,complex*),*Bub*(*inorder_complex, complex*, *Ndes*(*complex*), overestimation in CPI is responsible for pessimism in WCET. For benchmarks *Nsch*, *Minv*, *Lud*, *Ins*, *Dij*, overestimation in both IC and CPI are responsible for pessimism in WCET. This result is inline with our observation that estimating WCET as a whole can make use of pessimistic IC and pessimistic CPI.

We summarize the results as follows.

1. Estimating WCET as a product of SWIC and f(SWIC) is found to be much optimal than

Benchmark	**simplest**		**inorder_complex**		**complex**		Class
	$\frac{\widehat{WCET}}{M}$	$\frac{\widehat{WCET}}{Chronos}$	$\frac{\widehat{WCET}}{M}$	$\frac{\widehat{WCET}}{Chronos}$	$\frac{\widehat{WCET}}{M}$	$\frac{\widehat{WCET}}{Chronos}$	
Bez	0.998778	0.966540	0.998803	0.806836	0.998821	0.761281	II
Bit	1.079158	0.796427	1.108225	0.817879	1.068661	0.766020	II
Bs	0.997989	1.149622	0.997479	N/A_1	0.997362	N/A_1	II
Bub	1.212706	1.043051	1.204614	0.449837	1.476607	0.484224	II
Cnt	0.999971	1.003278	1.001646	0.176184	0.994585	0.153073	II
Crc	0.999979	0.965672	1.040777	1.013983	1.040783	N/A_1	I
Dij	3.432624	1.006663	3.432698	1.206713	3.432723	1.194388	III
Edn	1.000025	0.991075	0.999976	0.780132	0.999970	N/A_1	I
Fft	1.010519	0.977760	1.010434	N/A_2	1.010571	N/A_2	I
Fir	1.086121	0.977863	1.086142	0.581876	1.086089	0.485524	I
Ins	3.325180	1.000067	3.325181	0.338777	3.325374	0.252605	III
Jan	1.000048	1.002281	1.000000	1.002312	0.999953	1.002116	III
Lms	0.999979	0.971079	1.000010	0.549564	0.999991	0.457846	I
Lud			5.831052	1.238244	5.864458	1.202899	V
Lud	5.780452	0.949420					IV
Mat	1.000022	1.000031	0.999993	0.162041	1.887018	0.387059	I
Min	1.793374	1.064778	2.123263	N/A_1	2.185113	N/A_1	II
Nsch	6.058287	2.000649	4.925038	N/A_2	5.013082	N/A_2	IV
Ndes	0.999991	1.013494	1.000007	0.303941	1.000022	0.254355	III

Table 4.2: Improvement in accuracy of WCET due to application of relationship between IC and CPI on *simplest*, *inorder_complex*, *complex* architectures. N/A_1 implies chronos gives a segmentation fault. N/A_2 implies chronos goes out of memory.

the product of maximal IC and CPI in most cases. The IC-CPI relationship is based on actual measurements and hence reliable. However, it is important to ensure that the test input set covers the program structurally and tests with all possible range of data values.

2. For benchmarks exhibiting negative correlation between IC and CPI, the estimate can be unsafe, if the slope of the curve is very steep and the difference between SWIC and MIC is large. If the slope is very less, CPI almost saturates with increasing IC. Amongst cases of negative correlation considered in this thesis, we found that in the worst case, the resulting WCET estimate was unsafe by less than 0.2%.

3. Similarly, for benchmarks exhibiting positive correlation between IC and CPI, the estimate can be pessimistic, if the slope of the curve is too steep and the difference between SWIC and MIC is large. We saw an instance of this occurring in the case of *Nsch*.

4. For benchmarks wherein the IC-CPI relationship is not very clear, we fall back on our original proposal of estimating WCET as a product of maximal IC and CPI. Hence points 3,4 form limitations of our proposed approach.

5. Our proposal of estimating WCET as a product of SWIC and f(SWIC) is ideal for two

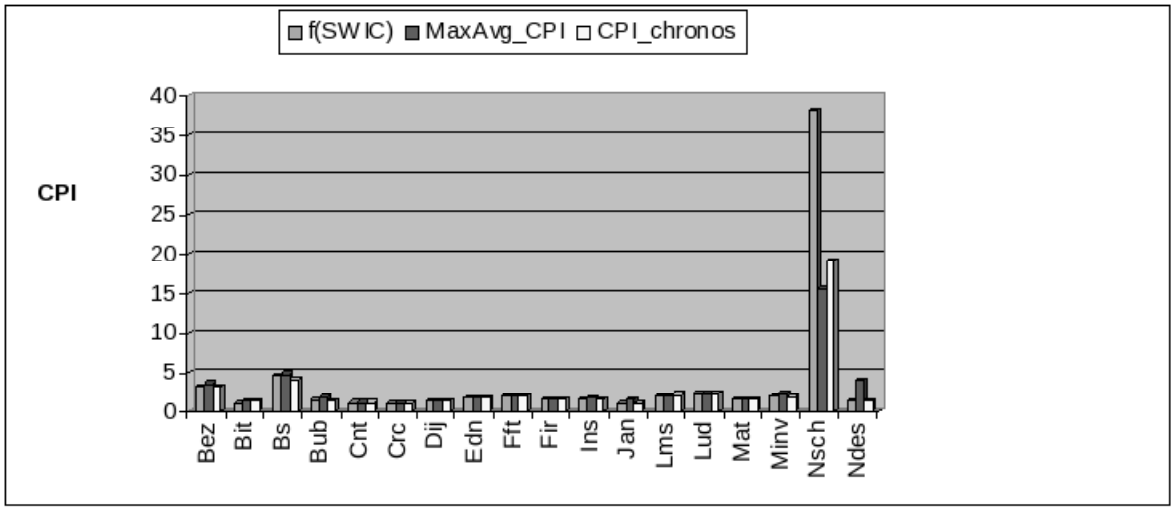


Figure 4.14: f(SWIC) versus Max_Avg(CPI) and $CPI_{chronos}$ on *simplest* architecture.

class of benchmarks. One is where, the measured IC and CPI is the same irrespective of the input. The other is where, CPI saturates with increase in IC.

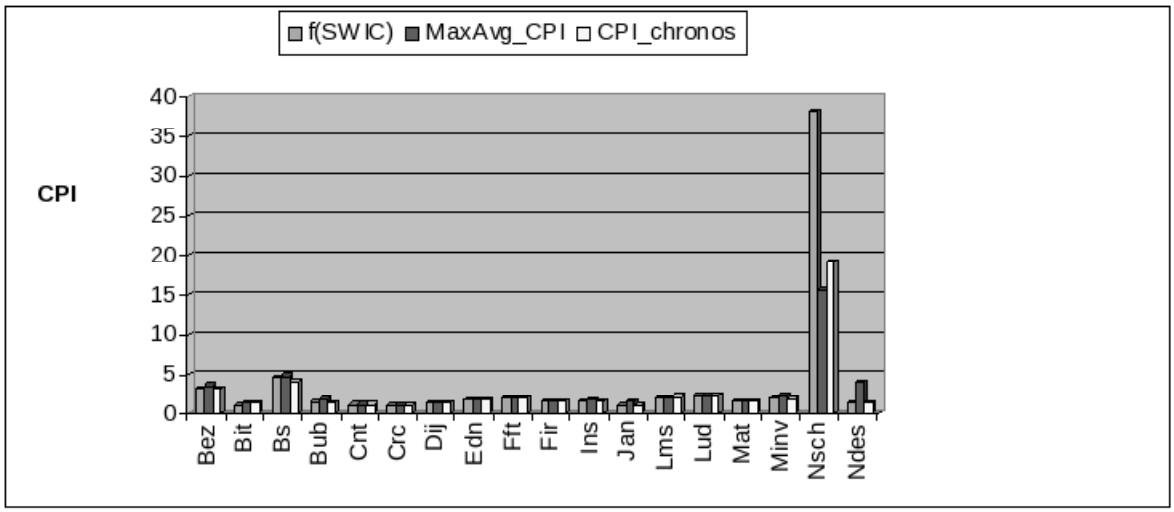


Figure 4.15: f(SWIC) versus Max_Avg(CPI) and $\text{CPI}_{chronos}$ on *inorder_complex* architecture.

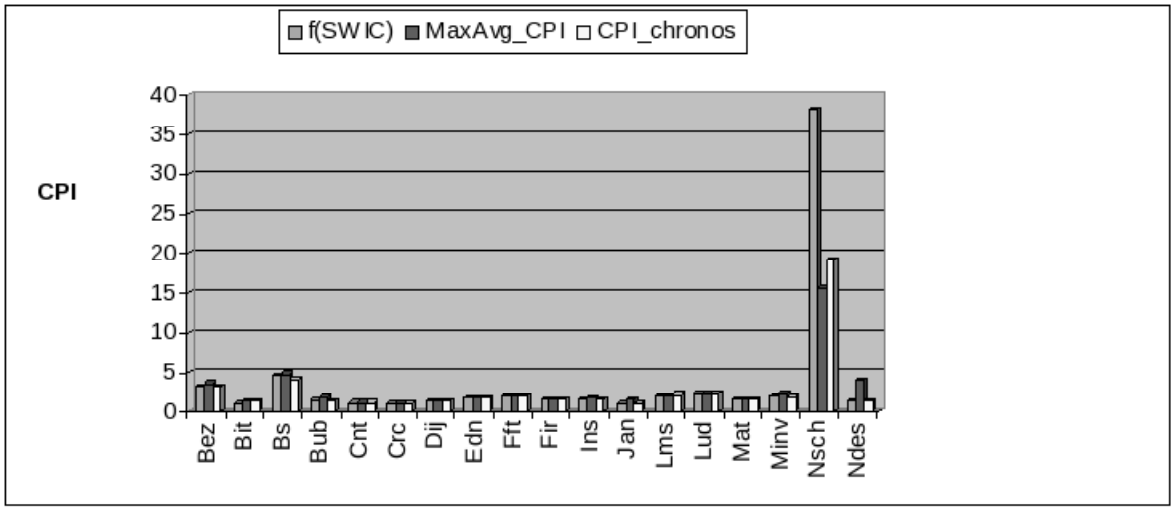


Figure 4.16: f(SWIC) versus Max_Avg(CPI) and $\text{CPI}_{chronos}$ on *complex* architecture.

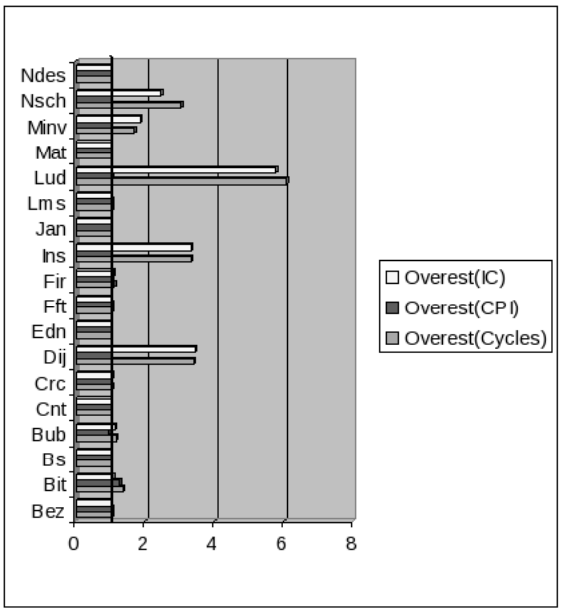


Figure 4.17: Factors responsible for overestimation of WCET by *Chronos* on *simplest* architecture.

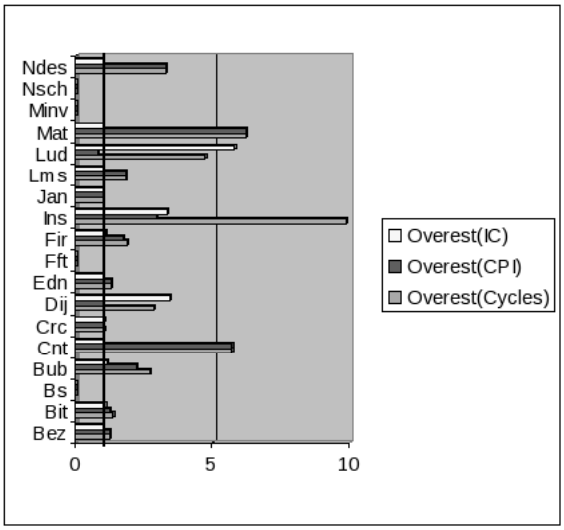


Figure 4.18: Factors responsible for overestimation of WCET by *Chronos* on *inorder_complex* architecture.

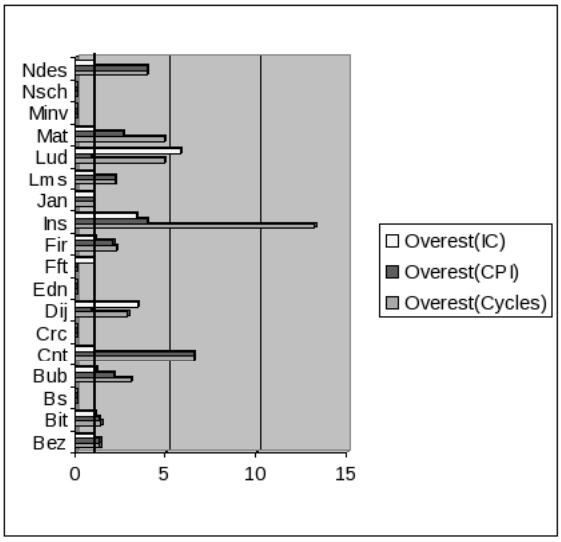


Figure 4.19: Factors responsible for overestimation of WCET by *Chronos* on *complex* architecture.

4.3 Related Work

There has been prior research in the study of factors influencing WCET. Colin et al[9] ascertain the influence of cache, branch predictor, pipeline etc on overall estimated WCET. Among various architectural components it identifies data and instruction caches and their properties like size, organization etc to be having the highest influence on WCET. In this work however, we investigate the relationship between IC and CPI of a program and how it influences WCET estimation. Bünte et al[77] describe the desirable features of a WCET benchmarking suite tool-set in the context of measurement based analysis. These features are at the program structure level. Our work tries to characterize the benchmarks in terms of variability in IC and CPI and how they correlate to each other. Benchmarks that exhibit more variability in IC and CPI across different inputs are more interesting and challenging for a WCET analyzer than benchmarks that display constant behavior in IC and CPI across inputs.

Deverge et al[43] suggests making hardware modifications to make a program execute for more or less the same time with different data thereby reducing the variability in timing. Some of the things that can be implemented to this effect are cache conscious data placement, cache locking, static branch prediction, to account for variable latency of segments, add difference between BCET and WCET of all operations of the program path (also known as jitter), avoiding usage of instructions that take variable latency, usage of more predictable instructions (Eg: add instead of mul) etc. This would amount to bringing a program to class I if the program has no if conditions or if both branches of the *if* condition execute the same number of instructions or to class III, if only the hardware timing can be controlled but not the variability in the number of instructions executed. For both these classes, the proposed technique is proven to give accurate WCET estimates. Statistical WCET analyzers[38, 97, 98, 71] fit models based on extreme value theory to end to end measurement based execution times and extrapolate the curve to give an estimate of WCET at the desired probability. This work however, fits a curve to the scatter plot of end to end measured IC versus CPI values. The curve is extrapolated upto the value corresponding to the theoretical upper bound on IC and used to obtain the corresponding CPI value. The product of these values gives us the optimal WCET.

4.4 Conclusions

In this chapter, we saw that program instruction count, IC and cycles per instruction, CPI, although seeming to be independent variables are actually correlated in most cases. Five predominant classes of benchmarks were seen to emerge based on the IC-CPI relationship. Based on this relationship, the optimal worst case CPI was derived by fitting a curve to the points in the scatter plot of IC versus CPI and extrapolating the curve up to SWIC and noting f(SWIC). The product of SWIC and f(SWIC) gives a much more precise WCET in three cases. The first case was when a negative correlation was present between CPI and IC. The second case, when IC and CPI did not change significantly irrespective of the input. The third case was when with increasing IC, CPI reached a saturation value. It was found that other static WCET analyzers could estimate both IC and CPI pessimistically and this can be avoided by factoring cycles into IC and CPI. The IC-CPI relationship also helps in benchmark classification. During this process, we found that for some simple benchmarks, irrespective of the input presented, the observed IC and CPI was the same. This has important ramifications in testing and input selection for evaluation of WCET analyzers. In the next chapter, we shall see a critical implication of considering CPI as our measurement parameter.

Chapter 5

Implications of Program Phase Behavior on Timing Analysis

The thesis begins by proposing a timing model that estimates WCET of a program considering it as a single unit. Execution time is factorized as a product of instruction count, IC and cycles per instruction, CPI and WCET is computed as a product of maximum IC and maximum CPI. In the previous chapters, we evaluated several candidates for maximum IC and CPI using analytical, statistical functions and also the fact that in many programs, there exists a correlation between IC and CPI which helps optimize the WCET estimate beyond the product of maximal IC and maximal CPI. Estimating WCET as a product of maximal CPI and IC is observed to be most accurate when CPI is most stable and is centered around the mean throughout execution. This kind of behavior is observed in small programs wherein execution centers around a single loop kernel. The CPI variation of one such benchmark, *Ins* is as shown in Figure 5.1. The axes are plotted by sampling CPI and program counter address (masking it's most significant bits), for every 1000 instructions executed. The execution of *Ins* is centered around the loop that sorts elements of the input vector. Such programs are termed as single-phase programs. But this seldom is always the case.

Some programs are composed of a small number of distinct *phases* of computation, where each phase represents a single simple task. Such programs are termed multi-phase programs. Benchmark *Bitcount* is one example (Figure 5.2). In such programs, variation in CPI is

markedly distinct across phases. The corresponding PC graph indicates that each phase corresponds to a different region of execution. This trend of repetitive, predictable and homogeneous CPI variation observed in the dynamic execution of programs is termed as *program phase behavior* and this trend is observed simultaneously in other architectural parameters like cache misses, branch predictor misses etc. Program phase behavior has been extensively used for architectural simulation effort reduction[28], power and energy optimizations[25], adaptive system reconfiguration[42], memory footprint optimizations[14] etc.

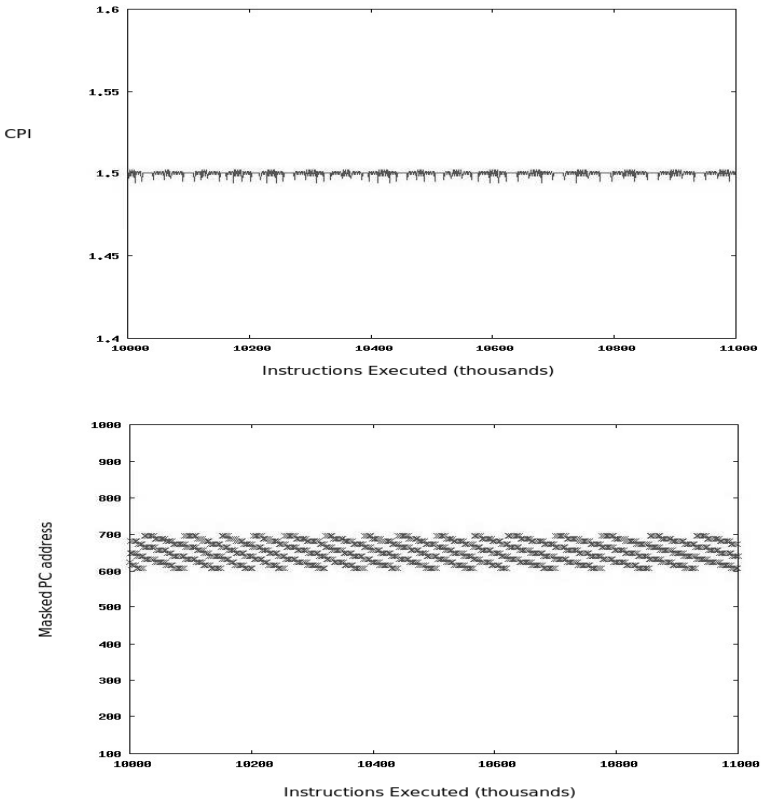


Figure 5.1: Variation of CPI and program counter address values with respect to time for a single run of Insertion sort PISA binary.

The theme of this chapter is to show that program phase behavior has important implications in measurement based WCET analysis. An important requirement in measurement-based WCET analyzers is that the process of measurement itself should be *least intrusive* in-order to avoid causing any impact on the accuracy of estimated WCET. Achieving an accurate estimate

with less instrumentation is a non-trivial task[2]. We propose the use of phases to estimate WCET with minimal instrumentation without compromising on the accuracy of the estimate.

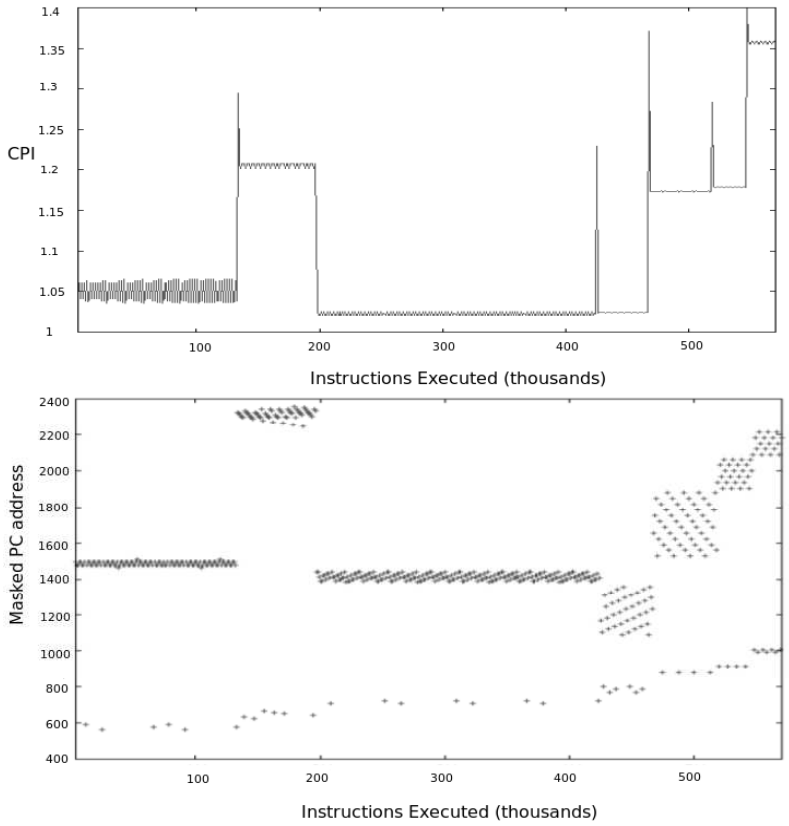


Figure 5.2: Variation of CPI and program counter address values with respect to time for a single run of Bitcount PISA binary.

A phase is defined as a set of non-overlapping intervals where each interval is a contiguous sequence of instructions from a program's dynamic execution stream. An important characteristic of a phase is that it exhibits similar CPI behavior irrespective of temporal adjacency. Moreover the coefficient of variation(COV) of CPI within a phase is very less as compared to the COV of CPI across phases. We build on these observations and measure CPI at the phase level to effectively characterize timing of program phases and hence the whole program. Accounting for phase behavior helps alleviate instrumentation overhead compared to other measurement based approaches as phases are typically composed of thousands of instructions.

5.1 Phase Detection Methods

Depending on the following parameters, phase detection methods can take various forms as shown[26].

- What makes up a phase?
    1. A phase can refer to a static code region.
    2. A phase could be a fixed set of dynamic instructions.
    3. A phase could be a set of dynamic instructions but variable in length.
    4. A phase could be long range and periodically occurring.
- Phase Granularity
    1. A phase can correspond to program structures such as a call or a loop.
    2. A phase could refer to a set of instructions.
- Time of Phase Analysis
    1. Phases can be detected by static structural analysis with a little help of profiling.
    2. A phase can be detected by analysis of offline data traces.
    3. A phase can be detected online during simulation or native execution.
    4. A phase can be detected by a mixture of online and offline data.
- Microarchitectural Dependence
    1. A phase can be dependent on microarchitecture. When intervals are classified into phases based on the values of microarchitecture parameters such as CPI, they are said to be microarchitecture dependent.
    2. When intervals are classified into phases based on their static location in code, they are said to be microarchitecture independent.

The complete list of phase detection methods that are built by a combination of the above mentioned characteristics is described in [26]. In this thesis, we divide the program static region into phases taking into account patterns of instruction execution[39]. This method is in better

sync with the natural period of the program and more dependable as code that is executed has an important influence on architectural behavior(Fig. 5.2). Mapping a phase to a code region gives us a handle to construct a timing model for that region. Further, such a classification makes the phase pattern repeat consistently across most architectures rendering the technique re-targetable[39].

We use code structural analysis [39] to mark phases in the binary. The CPI for every phase is measured by running the program with a large number of inputs. Measurements are taken using *Simplescalar V3.0*[113] for PISA binaries. The worst case CPI is defined as a function on measured CPI. The worst case number of instructions that can be executed within a phase is determined by static analysis of the program control flow graph (CFG). The WCET of a phase is then computed as a product of worst case CPI and worst case instruction count. The WCET of the whole program is computed as sum of WCETs of the individual phases. The resultant WCET will be much more precise in programs constituting of multiple phases such as *Bitcnt* (Figure 5.2). If the program has only one phase, WCET is simply a product of worst case instruction count and worst case CPI. The steps to estimate WCET using phases are formalized in the next section.

5.2 Phase Based Timing Model

5.2.1 Single-phase programs

For a program, exhibiting predominantly a *single phase*, WCET is computed as,

$$WCET = (WIC) * (WCPI) \tag{5.1}$$

Where,

- **WIC** or ***Worst case instruction count***

  The theoretical upper bound on IC that is statically determined by analyzing program CFG (section 3.1), SWIC is used. For a softer estimate, maximum observed instruction count, MIC is used.

- **WCPI** or ***Worst case CPI***

  The maximum of average CPI of a program observed across a large number of inputs,

Max_Avg(CPI) as described in section 3.3 is used. The CPI within a phase is expected to be fairly stable, hence average CPI is used in characterizing the execution time of a phase for each input and the maximum average CPI observed across inputs is taken to be worst case CPI. A softer estimate could be obtained by using overall average of average CPI observed for each input, Avg_Avg(CPI) as described in section 3.3.

5.2.2 Multi-phase programs

For a program, exhibiting multiple phases, WCET is computed as,

$$WCET = \Sigma_{(j \in 1 \,..\, p)} \, (T_j * WIC_j * WCPI_j) \tag{5.2}$$

Where,

- $\boldsymbol{p}$ is the number of phases occurring during program execution.
- $\boldsymbol{T_j}$ is the number of times phase j occurs in the worst case.
- $\boldsymbol{WIC_j}$ is the worst case instruction count of code region corresponding to phase *j*. WIC_j is substituted by either SWIC_j or MIC_j.
- $\boldsymbol{WCPI_j}$ is the worst case CPI of phase *j*. WCPI_j is substituted by either Max_Avg_j(CPI) or Avg_Avg_j(CPI).

The sequence of code regions (phases) visited by a multi-phase program during its execution forms the *phase sequence* for that program. If a program has two phases *P1* and *P2*, that are executed one after the other, phase sequence for the program is *P1 P2*. A simple structured program might have a single phase sequence irrespective of the input tested. A complex structured program that has plenty of *if-conditions* might have more than one phase sequence for all inputs tested. Using the above equation, Eq(5.2), we estimate WCET for each possible phase sequence, k, separately as WCET_k and finally take the maximum WCET over all possible sequences as the program WCET as follows.

$$WCET = Maximum \; (WCET_1, ..., WCET_k) \tag{5.3}$$

5.3 Methodology

The overall methodology is depicted as shown in Fig. 5.3. The number of phases occurring in the binary, p, are first identified using *code structural analysis*[39] by marking the instructions that indicate phase change in the binary. Static analysis is then performed for the code region corresponding to each phase to determine WIC_j and T_j. The worst case cycles per instruction ($WCPI_j$) for each phase j is estimated using direct measurement of the program CPI with a large number of test inputs on the target architecture. All these values are substituted in the timing equations Eq(5.1) and Eq(5.2) and if required, Eq(5.3), to give the program WCET. It is worthy to note that estimation of WIC and WCPI for each phase can be done in parallel thereby reducing the time to perform WCET analysis. We now describe each step in detail, beginning with phase identification.

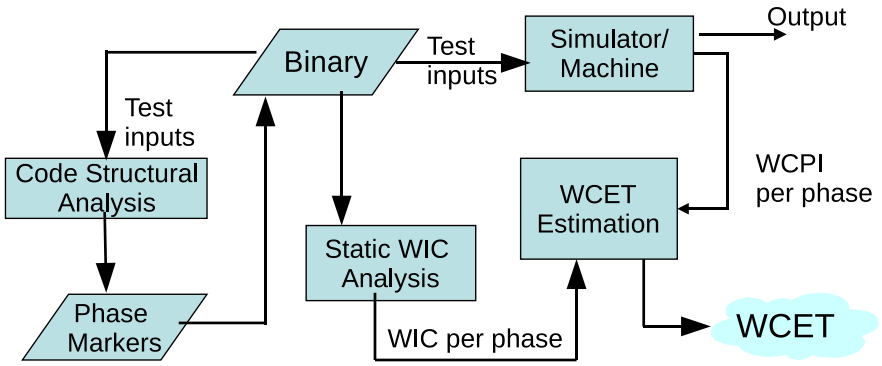


Figure 5.3: High level structure of the proposed solution.

5.3.1 Phase Identification

Code structure analysis[39] involves identifying instructions in the binary that accurately indicate start of unique stable behaviors seen across different inputs. Such instructions are termed as *software phase markers* and typically represent call-loop boundaries. The basic structure of analysis is a *dynamic hierarchical call-loop graph* which is built by using profile information gathered while instrumenting the application binary. A call-loop graph is a directed graph, whose nodes represent either a procedure call or a loop. It is termed hierarchical as the edges store hierarchical execution information along the path from call and loop nodes and are hence

said to abstract path information in some sense. Each loop is associated with two nodes- loop head and loop body, to differentiate between loop invocation and each iteration of the loop respectively. Each call is associated with one node for non-recursive calls, two nodes for recursive calls. In this work, we convert instances of recursion into iteration, wherever possible. Each edge stores average number of instructions executed along that path(A), coefficient of variation in instructions executed each time this edge was traversed (COV_{instn}), maximum number of instructions executed along that path (N_{max}) and total number of times the edge was traversed(C).

The call loop graph is annotated by instrumenting the program at a number of strategic places like calls, around loop branches, basic blocks as shown in Figure 5.4. The program is run with a test harness which supplies a minimal set of inputs that exercise all calls and loops in the program. The graph is constructed and the edges are annotated with C, A and COV_{instn}, when the instrumented program is run. Once all the edges are thus annotated, the minimum length of a phase, *ilower*, for a given program is determined. *ilower* depends on the average dynamic length of a program and the number of phases desired. For a given program, we compute *ilower* as a ratio of average instruction count observed across inputs to the number of phases required (`Compute_Threshold` in Figure 5.4). We shall see in chapter 6, that the number of phases also influences WCET analysis time. All edges whose average instruction count exceeds *ilower* are candidates for software phase markers. This is to ensure that each phase is long enough. Those candidate edges that also show minimum COV_{instn} qualify as the final software phase markers. The algorithm is outlined as shown in Figure 5.4. Each time, such a marker edge is traversed, the amount of instructions hierarchically executed is more or less the same. That proves our assumption that we are seeing a faithful repetition of a phase every time we enter this path making it a valid software phase marker edge. The running time of the phase marking algorithm is O(E+N log(N)) where N and E are the number of nodes and edges of the call loop graph. N log(N) is due to sort of all nodes to create a total call loop depth ordering of nodes during the first pass of the algorithm. E is for checking all edges for satisfiability of the two conditions.

The code region corresponding to a phase p is represented by the region between the phase marker edge of p and the phase marker edge of the following phase occurring in code. We modify the original algorithm that identifies instructions where a phase change is likely to occur, to

```
1. [ Annotate call loop graph with C, A, Cov(instns) information ]
Place instrumentation calls as follows:
- after program add ComputeVariance();
- For every loop  branch  instruction in every procedure
   Get source, target
   Loopid = GetLoopid(source, target);
   Get enclosing procedure id into encl_procid
   Before loop  branch  add CountLoopBody(loopid, encl_procid);
   After loop branch add ResetLoop(loopid);
   Before loop target add CountLoopHead(loopid, encl_procid);

- For every procedure call  get id of enclosing procedure and called
  procedure into encl_procid and call_proc_id
- Before every procedure call add CountCalls(encl_procid, call_proc_id);
- After every procedure call add CallSum(encl_procid, call_proc_id);
- For every Basic Block add CountInst(encl_procid);

/****
CountLoopBody(loopid, procid() : sets flag for loopid  indicating that it
should be counted
CountLoopHead(loopid, procid) : keeps track of number of instructions
executed in every loop invocation in a list
CountCalls(encl_procid, call_proc_id) : pushes call_proc_id on to the stack
CallSum(encl_procid, call_proc_id) : pops call_proc_id and records call
hieararchical information
CountInst(encl_procid) : updates counts associated with encl_procid and for
all loops whose flag is set to be true
ResetLoop() : reset flag for loopid indicating that it should no longer be
counted
ComputeVariance() : for each edge, compute C, A, Cov(instns) information
******/

2. ilower = Compute_Threshold(Avg_dyn_length, max_phases);
/******
Avg_dyn_length: average number of instructions executed by the program
max_phases: Maximum phases desired
ilower: minimum phase interval
*******/

3. For each node in call loop graph, compute max_depth(node)
[Ensure that children are processed first and then parents]
4. Place nodes in priority Queue sorted by decreasing max_depth(node)
5. For each node in the Queue
     For each incoming edge E,
      If E.A <= ilower and E.COV < COV_Threshold
        Add E to marker list;

/*******
COV_Threshold is decided to be a very low value of the order of 1-5% as low
COV is necessary for accurate timing estimation
If no such edge exists, the whole program is considered as a phase
*******/
```

Figure 5.4: Algorithm to annotate hierarchical Call-loop graph and compute phase markers.

also number phases as they occur. A program is said to be composed of a *single phase* if it's hierarchical call loop graph contains exactly one edge that satisfies these properties and that edge encompasses the whole program. Programs that cannot be classified into phases using this algorithm are also viewed as single-phase programs. However such programs depict a high degree of variance in their CPI throughout execution. *Nsch* (Section 3.2.1) is an example.

Fig. 5.5 depicts a part of the dynamic hierarchical call loop graph constructed for *Bitcount* that is run for 1000 iterations. *Bitcount* has an average dynamic length of 455157 instructions. With maximum number of phases set to 10, we obtain 45515 as ilower. By setting a minimum COV_*instn* as 0%, the edges that are marked with an asterisk are selected as a valid software

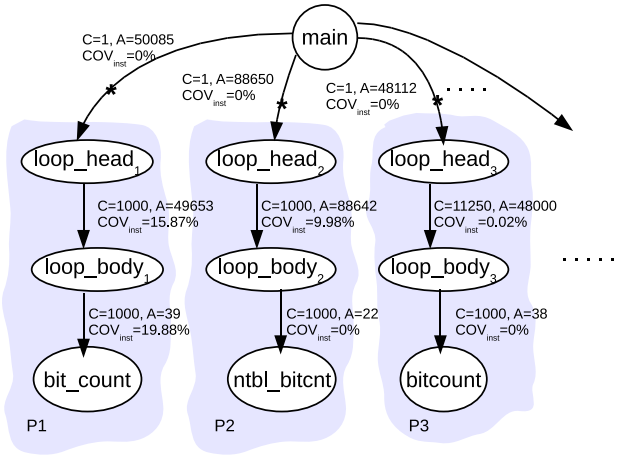


Figure 5.5: Hierarchical Call-loop graph for Bitcount: C is the number of times, each edge is traversed. A is the average number of hierarchical instructions executed each time the edge is traversed. COV_{inst} is the hierarchical instruction count coefficient of variation. P1, P2, P3.. are phase numbers.

phase marker edge. The phase marker edges picked by the algorithm for one input are observed to work well for other inputs as well[39]. The number of phase marker edges defines p in Eq.(5.2).

Code structure analysis marks phases based on instruction execution patterns. Hence we can see that *phase markers obtained by analyzing alpha binaries with ATOM[12] hold good for MIPS R3K PISA binaries* as shown in Fig. 5.6. Execution of a program thus marked produces a phase sequence indicating the order in which instructions belonging to different phases are executed. The phase sequence encountered for each program considered in this work is shown in Table 5.1. Benchmarks *Lud* and *Minv* exhibit multiple phase sequences featuring repeating phase patterns. The notation used to express the phase sequence is the same as the one used for expressing regular expressions.

5.3.2 Implementation

We build the hierarchical call loop (HCL) graph by analyzing binaries with the ATOM instrumentation framework. ATOM provides a convenient way to identify procedures and branches. ATOM works with two files- an instrumentation file, xxx_inst.c and an analysis file, xxx_anal.c.

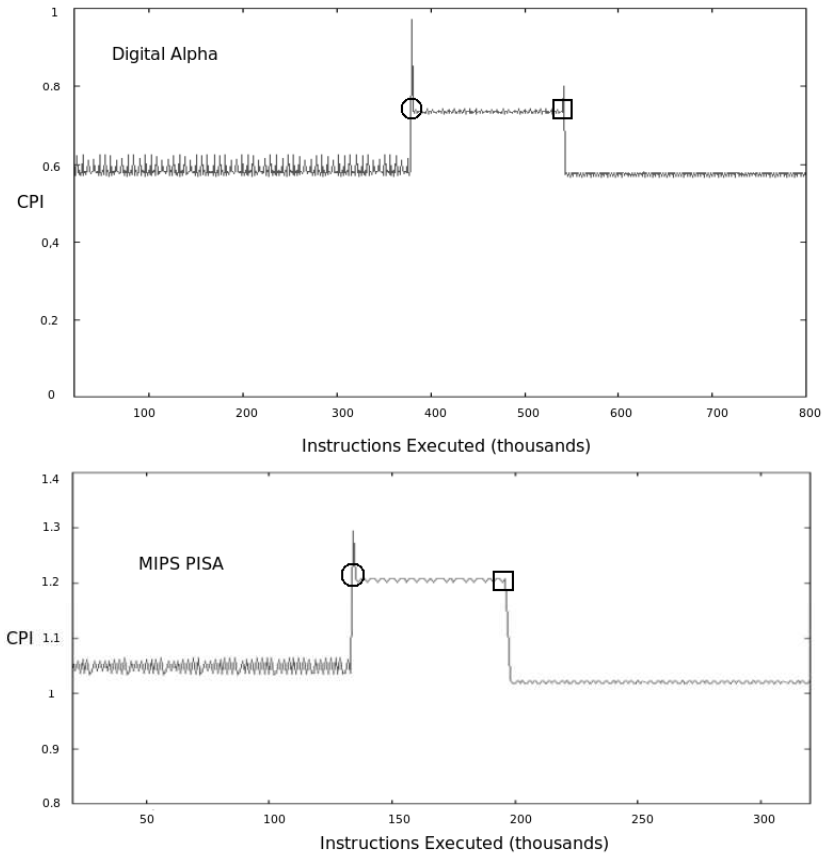


Figure 5.6: Time varying CPI graphs with phase markers of the Digital (Alpha) and a MIPS (PISA) binary for the program Bitcount. The phase markers were selected from the call loop profile graph from the Bitcount Alpha binary, were mapped back to source code level and then used to mark the Bitcount MIPS PISA binary.

The user can place calls to instrumentation routines around instructions, basic blocks and procedure calls in xxx_inst.c. The routines defined in xxx_anal.c are invoked when program control reaches these points.

For the purpose of phase detection, we need to count instructions executed per routine, per loop invocation, per loop iteration. Hence we place suitable calls to instrumentation routines around boundaries of procedures, loops and iterations respectively. The routines are coded in such a way that information is computed hierarchically and the call graph structure is maintained. When control reaches the end of the program, the hierarchical call graph is traversed starting from the main routine. Those call or loop edges that satisfy the condition of having an average count, A, of more than *ilower* and a coefficient of variation of less than minimum COV_{instn} are output as software phase markers along with the corresponding instruction PC

Benchmark	*Phase Sequence*
Bezier	P1 P2
Bitcount	P1 P2 P3 P4 P5 P6 P7
Bs	P1
Bub	P1
Cnt	P1 P2
Crc	P1 P2
Dij	P1
Edn	P1 P2 P3 P4 P5 P6 P7 P8
Fft	P1 P2
Fir	P1
Ins	P1
Jan	P1
Lms	P1
Lud	P1 P1 P2 P3 P1 P2 (P3 P4)+ P1 P2 (P3 P4)+ P5
Matmul	P1 P2
Minv	P1 P3 (P5 P6)+ P9 P8 P1 P2 P3 (P4 P5 P6)+ (P5 P6)+ P9 P8 P1 P2 P3 ((P4 P5 P6)+ |(P5 P6)+)+ P7 P9 P8
Nsch	P1
Ndes	P1

Table 5.1: Benchmarks and their phase sequences.

value. Since these instructions correspond to call-loop boundaries, they can easily map to a different binary like PISA or ARM. Assuming our instrumentation and analysis files are named swphasemarker.inst.c and swphasemarker.anal.c, if test.c is our benchmark program, we first compile it using the CC compiler as shown.

```
cc -Wl,-r -non_shared test.c -o test.rr
```

Following this, we create an instrumented executable, test.trace, by running ATOM with swphasemarker.inst.c and swphasemarker.anal.c as follows

```
atom -Wla,-lm test.rr swphasemarker.inst.c swphasemarker.anal.c -o test.trace
```

Running ./test.trace will give us a list of instruction PCs that serve as phase markers demarcating phases in our program. We create a test harness program for each benchmark that executes the benchmark with a set of different inputs to ensure the HCL graph is complete and has not omitted any procedure or loop.

5.3.3 Estimating WIC

Section 3.1.1 discusses estimation of WIC for the whole program in detail. In this section, we discuss the particulars of estimation of WIC for a phase. A phase as defined and described in the earlier sections refers to a code region demarcated by binary instructions known as phase

markers. The phase marker instructions typically lie at call and loop boundaries. The basic blocks that span between the start basic block corresponding to the start phase marker and the end basic block corresponding to the end phase marker constitutes the phase CFG. We then apply the method discussed in Section 3.1.1 on the phase CFG to estimate WIC of the phase.

5.3.4 Context Sensitivity

A program analysis is termed as context sensitive if it differentiates two instances of a procedure occurring at two different contexts. In this work, we perform procedure cloning and treat each call instance as a separate call. This might cause the algorithm to assign different phase numbers to two call instances of the same procedure even if their CPI behavior is similar. If the differing context does not impact variation in CPI, this has an effect of increasing the number of phases but has no bearing on correctness of the subsequent timing analysis. Example: *Crc* has two phases, one for each clone and average CPI for each phase is about the same despite the differing context. Context sensitivity has also been applied to loops by distinguishing iterations[54, 73]. In this chapter, we apply context sensitivity for only procedures. In the next chapter, we shall apply context sensitivity to loop iterations as well.

5.3.5 Infeasible Paths

An infeasible path is one which can never occur in any valid execution of the program. Weeding out infeasible paths helps compute a much tighter WCET estimate. We follow the approach used in Suhendra et al[87] and identify branch-branch conflict pairs and assignment-branch conflict pairs. A branch-branch(BB) conflict pair is a set of branch induced paths that can never occur together. Similarly an assignment-branch(AB) conflict pair is an assignment and a branch path that can never occur together. Fig. 5.7 shows a simple example of an AB and a BB conflict that can occur.

Infeasible paths are modeled as additional linear edge constraints and are added to our linear system of equations as described in section 3.1 of chapter 3. Two branch edges that figure in a BB pair, say, $E_{i \to j}$ and $E_{m \to n}$ have a linear constraint as shown in (5.4). Similarly an assignment (node) and a branch edge that figures in an AB pair have a linear constraint as

```
int test(int x, int z)
{
 int sum;
 if (x < 10)
    z = 10;
 else
    sum = x;
 if (z > 50)
    sum = z;
 else
    sum = sum + z;
 if (x > 20)
    sum = sum + x;
 else
    sum = sum – x;

}
```

BB Conflicts
{n1->n2, n7->n8}

AB Conflicts
{n2, n4->n5}

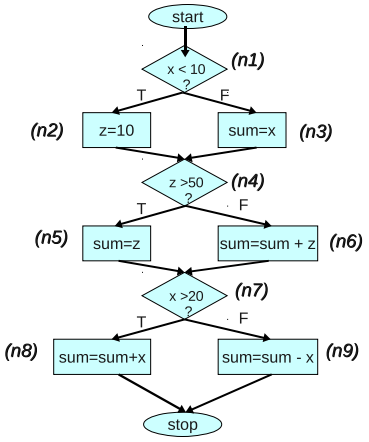


Figure 5.7: Illustration of Branch-Branch (BB) conflicts, Assignment-Branch (AB) conflicts.

shown in (5.5).

$$BB\ conflict: \ E_{i\to j} + E_{m\to n} = 1 \tag{5.4}$$

$$AB\ conflict: \ N_B + E_{i\to j} = 1 \tag{5.5}$$

We have used integer linear programming to estimate WIC statically. Alternatively, WIC can also be estimated statically by viewing the CFG as a weighted directed graph with basic blocks as nodes, W_B as edge weights and computing weighted longest path in the graph or tree based schema.

5.3.6 Estimating WCPI

This section describes estimation of worst case CPI of a phase that is carried out by measurement. The CPI of a phase is measured by sampling the phase at large intervals of instructions and averaging the samples. The samples for a phase are measured when the code region corresponding to that phase is executed. If the COV of CPI is very less within a phase, we can afford to take fewer samples without affecting the accuracy of phase CPI[88], which means, very less instrumentation is required within such a phase. Worst case CPI (WCPI) is defined as a function of measured per-phase CPI. We reapply analytical and statistical functions now on phase CPI to estimate WCPI of a phase.

Analytical

In chapter 3, we saw two possible analytical candidates that could be used to compute WCPI. One was the maximum of observed overall program CPI, Max_Avg(CPI) (described in section 3.3 in chapter 3) across a large number of inputs, i. Instead of computing Max_Avg(CPI) for the whole program, we now compute phase-wise Max_Avg as follows. If a program comprises of p phases, the CPI sample set for each (phase, input) pair, $\text{CPI}_{j,k}$ is averaged out to yield $\text{Avg\_CPI}_{j,k}$. WCPI_p is then computed as a maximum of the observed averages of CPI in phase p, across a large number of inputs, i-

$$WCPI_p = Max\_Avg_p(CPI) = Maximum(Avg\_CPI_{1,p}, Avg\_CPI_{2,p}, ..., Avg\_CPI_{i,p}) \quad (5.6)$$

The other candidate is the overall average program CPI (Avg_Avg(CPI)) across a large number of inputs, i. Instead of computing Avg_Avg(CPI) for the whole program, we now compute phase-wise Avg_Avg as follows. If a program comprises of p phases, WCPI_p is computed as an average of the observed average CPI in phase p, across a large number of inputs, i-

$$WCPI_p = Avg\_Avg_p(CPI) = \frac{Avg\_CPI_{1,p} + Avg\_CPI_{2,p} + ... + Avg\_CPI_{i,p}}{i} \quad (5.7)$$

Statistical

In chapter 3, we saw three possible statistical candidates that could be used to compute WCPI. They were the 90th percentile CPI value, 95th percentile CPI value and the 99th percentile CPI value. Instead of computing the tail end percentile values using CPI samples pertaining to the whole program, we combine CPI samples across all inputs for each phase, p. We substitute the percentile values considering phase wise CPI samples in place of WCPI_p as follows. Depending on the safety requirement, either 90per_p(CPI) or 99per_p(CPI) can be used.

$$WCPI_p = 90per_p(CPI_{1,p} \cup CPI_{2,p} \cup .... \cup CPI_{i,p}) \quad (5.8)$$

$$WCPI_p = 99per_p(CPI_{1,p} \cup CPI_{2,p} \cup .... \cup CPI_{i,p}) \quad (5.9)$$

For each phase, p, we evaluate $\{\text{SWIC}_p, \text{MIC}_p\}$ to represent WIC_p and $\{\text{Max\_Avg}_p, \text{Avg\_Avg}_p, 90\text{per}_p, 99\text{per}_p\}$ to represent WCPI_p, which gives us eight combinations to estimate WCET.

The safest analytical combination that can be used is $SWIC_p$ and Max_Avg_p. The safest statistical combination that can be used is $SWIC_p$ and $99per_p$. The softest and most approximate analytical and statistical combinations that can be unsafe are {MIC_p, Avg_Avg_p} and {MIC_p, $90per_p$}. If coverage by inputs is assured, MIC could be used instead of SWIC along with safer WCPI candidates such as Max_Avg_p and $99per_p$.

5.3.7 Warmup CPI

Warmup is an essential component of program execution that refers to the initial stage when all the architectural structures get filled in. The warmup CPI is typically higher than the stable program CPI. For programs executing millions of instructions, the effect of warmup can be ignored. The dynamic instruction count of the programs considered in this work range from a few thousand up to few millions. In this work, we consider the warmup as a special phase and add the warmup cycles separately to our estimated program execution time.

5.4 Experimental Methodology

We perform our experiments for the same set of benchmarks as defined in section 3.2.1 All programs are compiled to MIPS PISA binaries with -O2 -static flags. We use *Simplescalar v3.0*[113] for measuring CPI of programs across a large number of inputs. The inputs are chosen so as to satisfy structural coverage and to cover a wide range of data values as described in section 3.2 of chapter 3. Invalid inputs and inputs that produce very short sequence of instructions are pruned away from calculations. We test the WCET analyzer for architectures, described in Table 3.2 in chapter 3.

We sample benchmarks at every phase marker instruction in addition to sampling every 1K instructions within a phase to note CPI. For benchmarks of very small dynamic execution length, we sample every 100 instructions within a phase to note CPI. For each (phase, input) pair, we compute the corresponding CPI by taking the average of CPI samples for that input to generate CPI_1, ..., CPI_i for i inputs, indicated as in (Eq.(5.6) and Eq.(5.7)). We have experimentally verified that the width of the sampling interval within a phase can be varied arbitrarily without causing any impact on WCPI or ACPI of the phase. Here is where, the homogeneous nature of the phase lends itself to minimal instrumentation that is tapped

within the framework of measurement based WCET analysis. The resultant WCET estimate is compared with the estimate made by Chronos[94].

5.4.1 Phase Detection

Figure 5.8 describes the time taken to carry out hierarchical call loop graph analysis for all benchmarks under consideration. The time indicated in the figure includes time taken to create the instrumented executable using ATOM. The phase detection algorithm is directly dependent on the dynamic execution length of the program, as higher the number of instructions instrumented, higher will be the resulting analysis time. Hence the dynamic lengths are indicated to give an idea about scalability of the phase detection procedure. The maximum time taken to detect phases is about 12.65 seconds taken by *Bezier* which also has the highest dynamic length compared to all other benchmarks.

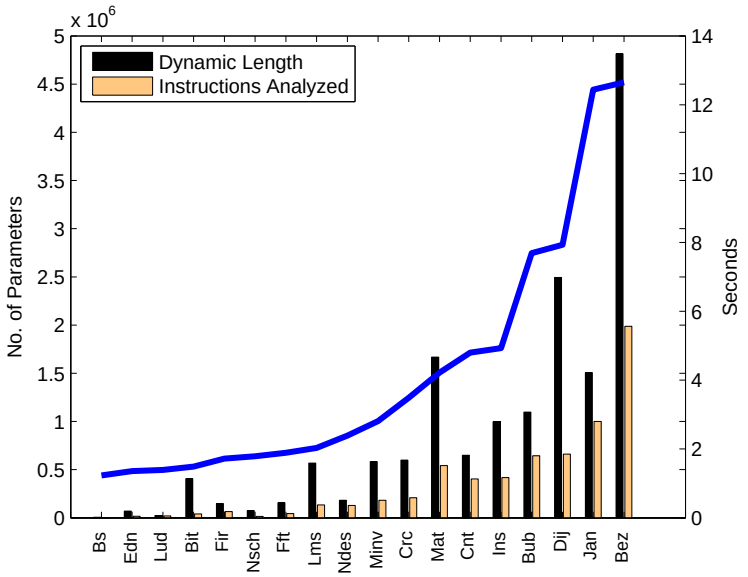


Figure 5.8: Plot of phase detection time versus program parameters.

5.4.2 Percentage COV of CPI

A good phase is said to exhibit minimum variance in CPI. The percentage COV of CPI for most of the single-phase programs is observed to be within 10% as shown in Figures 5.9 for *Simplest*, *Inorder_complex* and *Complex* architectures. These programs are mostly dominated by loops

that exhibit repetitive behavior resulting in the CPI becoming stable. It can be observed that the architecture significantly influences the COV of CPI of phases in the case of some programs. *Bub* and *Nsch* exhibit highly varying CPI as it is dominated by execution of branches within a loop that results in a large COV in instructions executed and hence makes phase identification difficult. The per-phase COV of CPI for most phases in multi-phase programs is observed to be much lesser than the overall COV of CPI of the program considering it as a single phase. Even if the COV of CPI of a phase continues to be high, the effect it has on the overall WCET is reduced as a phase constitutes only a fraction of the program. Figures 5.10, 5.11 and 5.12 depict the COV of CPI of individual phases versus that of the whole program for *Simplest*, *Inorder_complex* and *Complex* architectures. It is this property of low variance in phase CPI that reduces pessimism in estimated WCET using the proposed method.

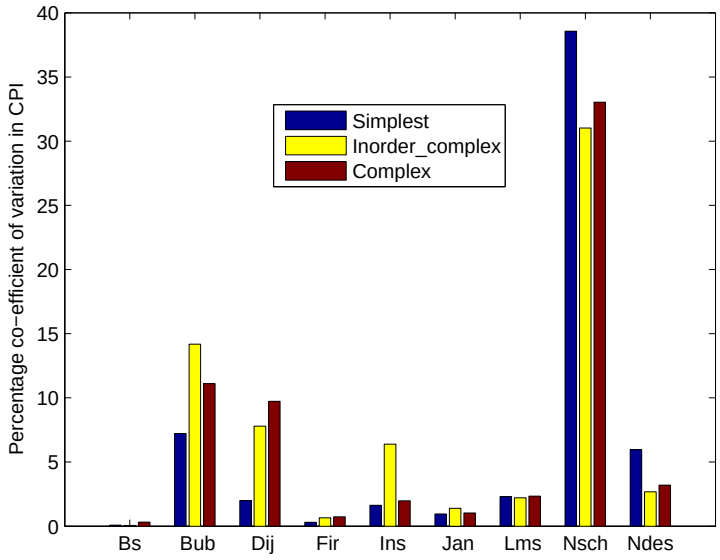


Figure 5.9: COV of CPI for single-phase programs on all architectures.

5.4.3 Accuracy of WCET Estimate

In this section, we shall evaluate the accuracy of WCET estimate obtained after detecting phases and computing phase-wise WCET. There exist benchmarks which do not display multiple phases during execution. The CPI variation in such benchmarks is seen to be either entirely stable (Coefficient of Variation of CPI $\leq 5\%$) or highly varying throughout execution

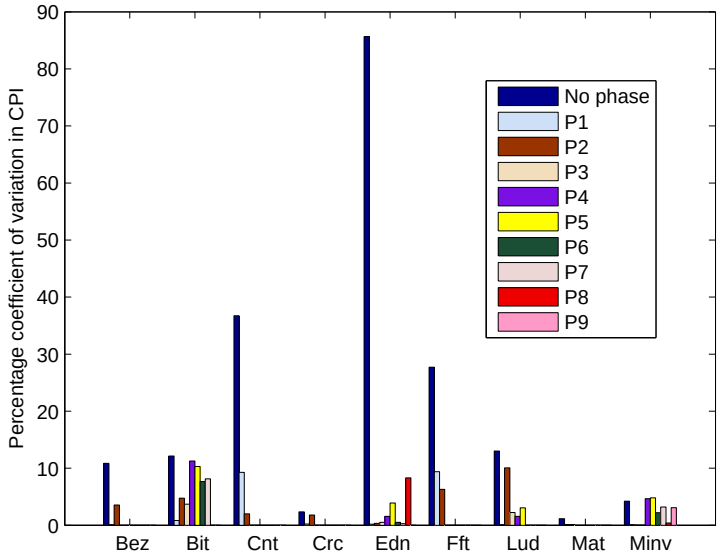


Figure 5.10: COV of CPI of individual phases versus whole program for multi-phase programs on *Simplest* architecture.

as can be observed from Figure 5.9. Examples of benchmarks with highly stable CPI behavior comprising of a single phase are *Bs*, *Fir*, *Jan* and *Lms*. The stability holds even across architectures. Examples of benchmarks with a single phase exhibiting highly varying CPI are *Bub*, *Dij* (Inorder_complex and Complex), *Ins* (Inorder_complex) and *Nsch*. For all single phase benchmarks, we retain the equation to estimate WCET as a product of overall theoretical upper bound on instruction count, SWIC and Max_Avg(CPI) as described in Chapter 3.

Analytical

The pessimism in WCET estimated using analytical functions of measured phase-wise CPI is depicted in Figures 5.13, 5.14 and 5.15 along with pessimism in WCET obtained using *Chronos* for *Simplest*, *Inorder_complex* and *Complex* architectures respectively. The average pessimism in WCET obtained using all the four analytical combinations applied phase-wise for all architectures is indicated in Table 5.2. On *Simplest* architecture, the safest analytical combination of SWIC_p and Max_Avg_p(CPI) yields about the same pessimism as *Chronos*. On *Inorder_complex* and *Complex* architectures, the same combination improves pessimism compared to *Chronos*, by 43% and 55% respectively.

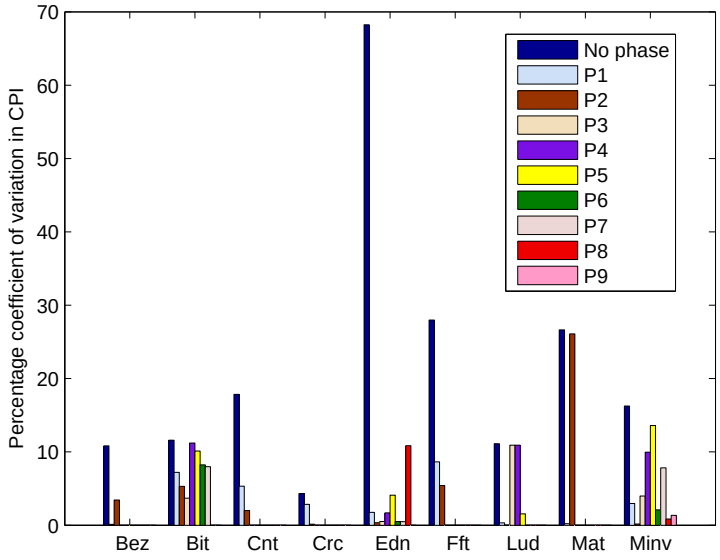


Figure 5.11: COV of CPI of individual phases versus whole program for multi-phase programs on *Inorder_complex* architecture.

Statistical

The pessimism in WCET estimated using statistical functions of measured phase-wise CPI is depicted in Figures 5.16, 5.17 and 5.18 along with pessimism in WCET obtained using *Chronos* for *Simplest*, *Inorder_complex* and *Complex* architectures respectively.

The average pessimism in WCET obtained using all the four statistical combinations applied phase-wise for all architectures is indicated in Table 5.2. The estimates computed using combination of SWIC and 99per(CPI) are very pessimistic as high end spikes get enlisted as 99th percentile CPI values, even though they occur at very few instances during program execution. For both single and multi-phase benchmarks that display similar CPI across runs with different inputs, the WCET estimate computed using both analytical and statistical candidates yield very accurate estimates. On an average, the combination of SWIC and 99per(CPI) is 32% more pessimistic than *Chronos*. For simple architectures, *Chronos* yields accurate WCET estimates. However, on *Inorder_complex* and *Complex* architectures, the combination of SWIC and 99per(CPI) improves pessimism by 40% and 65% compared to *Chronos*.

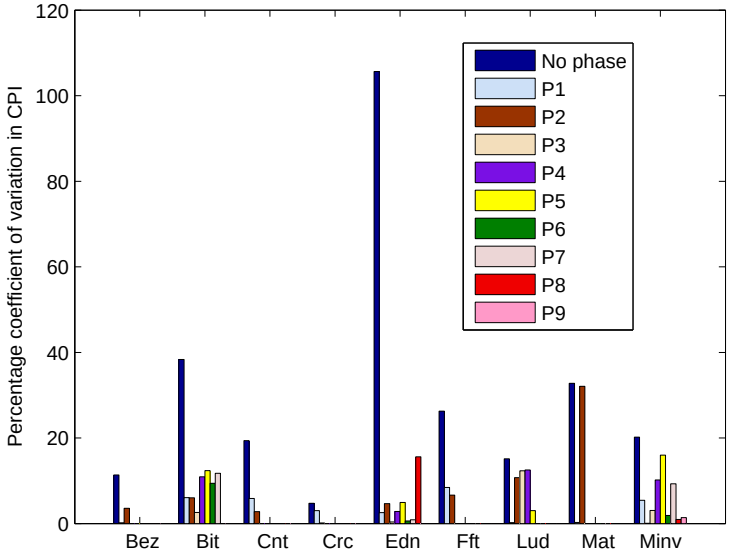


Figure 5.12: COV of CPI of individual phases versus whole program for multi-phase programs on *Complex* architecture.

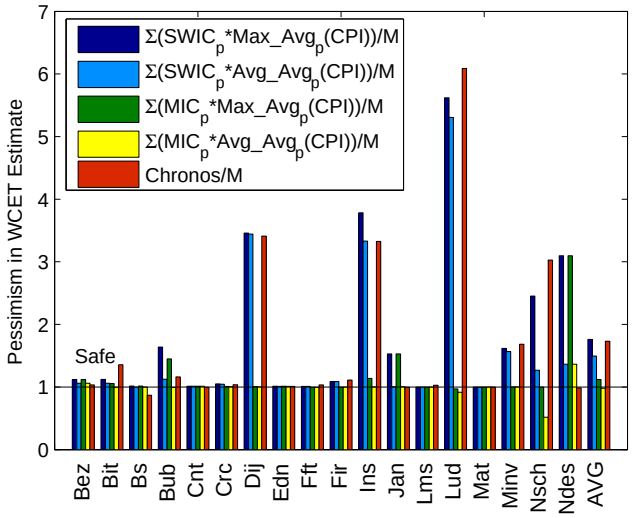


Figure 5.13: Pessimism in WCET estimate using analytical CPI candidates taking into account phase information on *Simplest* architecture.

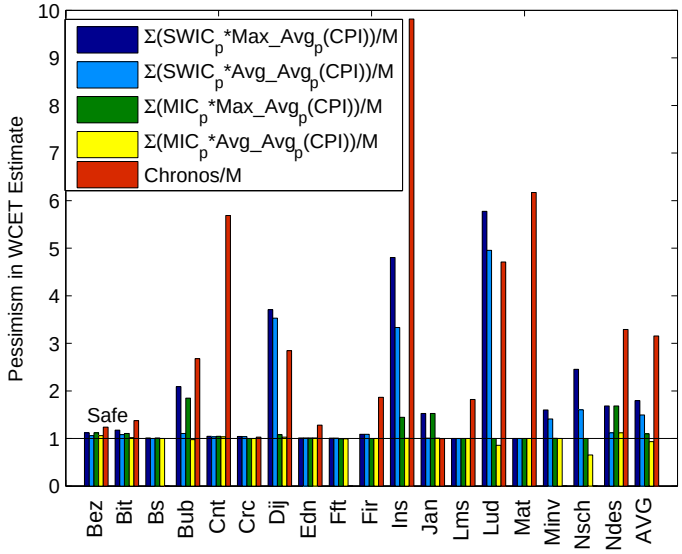


Figure 5.14: Pessimism in WCET estimate using analytical CPI candidates taking into account phase information on *Inorder_complex* architecture.

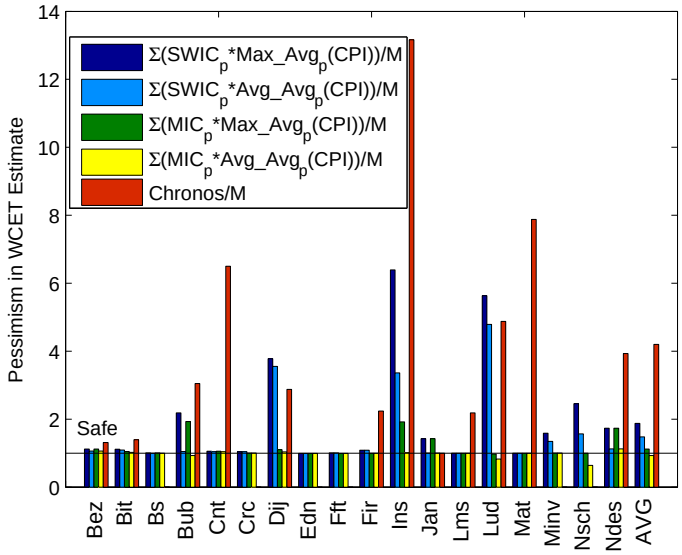


Figure 5.15: Pessimism in WCET estimate using analytical CPI candidates taking into account phase information on *Complex* architecture.

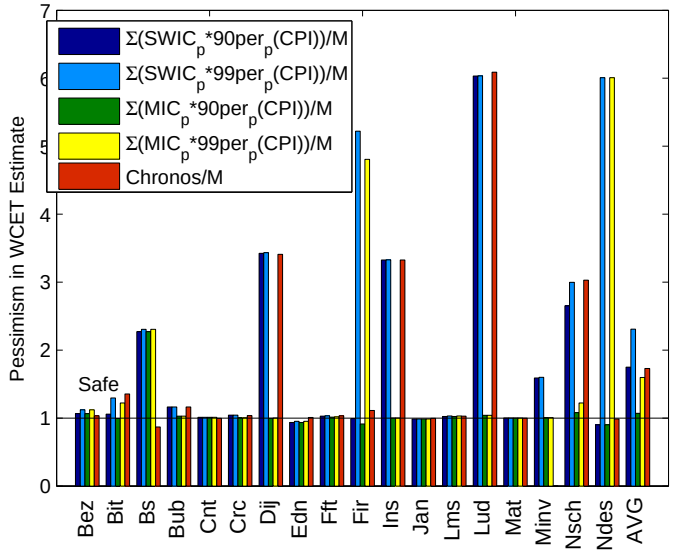


Figure 5.16: Pessimism in WCET estimate using statistical CPI candidates taking into account phase information on *Simplest* architecture.

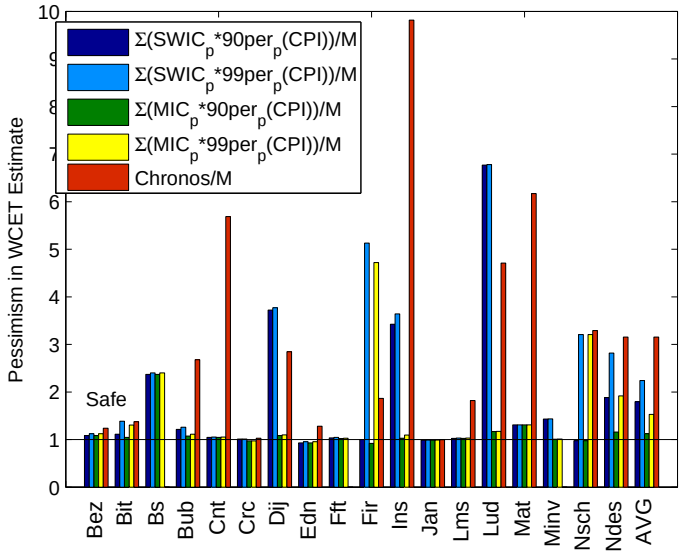


Figure 5.17: Pessimism in WCET estimate using statistical CPI candidates taking into account phase information on *Inorder_complex* architecture.

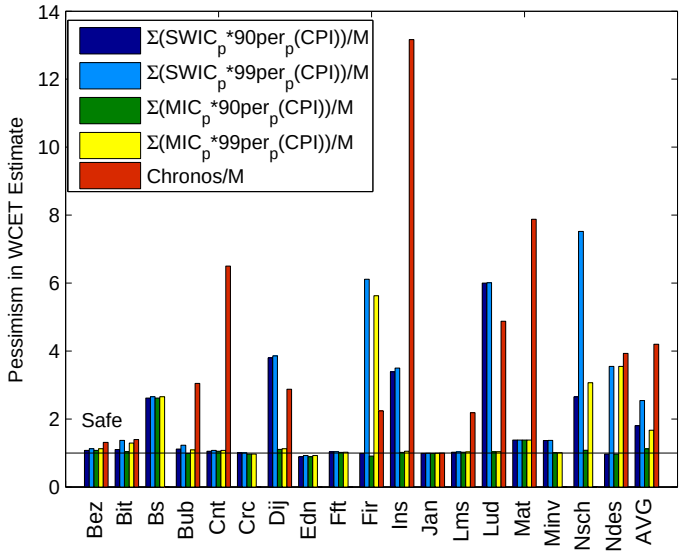


Figure 5.18: Pessimism in WCET estimate using statistical CPI candidates taking into account phase information on *Complex* architecture.

WCET	*Simplest*	*Inorder_complex*	*Complex*
$\frac{\Sigma(SWIC_p \times Max\_Avg_p(CPI))}{M}$	1.76099	1.79653	1.8748
$\frac{\Sigma(SWIC_p \times Avg\_Avg_p(CPI))}{M}$	1.49432	1.49341	1.47986
$\frac{\Sigma(MIC_p \times Max\_Avg_p(CPI))}{M}$	1.12051	1.09855	1.12196
$\frac{\Sigma(MIC_p \times Avg\_Avg_p(CPI))}{M}$	0.980906	0.934616	0.930796
$\frac{\Sigma(SWIC_p \times 90per_p(CPI))}{M}$	1.75027	1.79559	1.8021
$\frac{\Sigma(SWIC_p \times 99per_p(CPI))}{M}$	2.30933	2.2408	2.54154
$\frac{\Sigma(MIC_p \times 90per_p(CPI))}{M}$	1.07082	1.12226	1.1198
$\frac{\Sigma(MIC_p \times 99per_p(CPI))}{M}$	1.59818	1.52749	1.66775
$\frac{Chronos}{M}$	1.7308	3.15256	4.1998

Table 5.2: Average pessimism of WCET on all PISA architectures using the proposed method and *Chronos*.

5.5 Related Work

5.5.1 Worst Case Execution Time Analysis

Most existing WCET analyzers partition the program into smaller components like basic blocks[29], scopes[6], segments[55] or paths[66] to reduce complexity of WCET analysis. Each of these components are either statically analyzed or directly measured to compute the cost of executing the component on the target machine. Finally these execution times are combined using abstractions such as ILP, tree based schema or graph theoretical algorithms to give the final WCET estimate. The proposed method partitions the programs into smaller components based on observed instruction execution patterns. The nature of repetitive execution of instructions within a phase leads to a repetitive homogeneous pattern of CPI variation within a phase. The phase change boundaries are detected by an algorithm that builds a hierarchical call graph and annotates the graph with profile information. The phase change boundaries known as phase markers act as primary locations of instrumentation points. The homogeneity of a phase allows us to place instrumentation points at arbitrarily large intervals within the phase to measure CPI accurately thereby helping build a non intrusive measurement based WCET analyzer.

5.5.2 Phase Behavior

Phases can be detected in various ways with respect to a number of parameters as we saw in section 5.1. Phases can be detected by collecting a trace of the program and analyzing it offline. Phases detected based on basic block vectors is a good example of offline phase detection and is applied in architectural simulation effort reduction[28]. Phases can also be detected online while the program is executing. Such detection methods are generally used for dynamic configuration of the processor to obtain performance optimizations[42]. Phases can either be associated with instruction execution patterns or data access patterns. Phases that are based on data access patterns are generally associated with memory related optimizations and are not relevant to the work presented in this thesis. Phases that are associated with instruction patterns can either be composed of fixed intervals[28] or can be of variable intervals[41]. Variable length intervals have been shown to effectively characterize CPI of a phase more accurately as it aligns with the natural period of the program better than fixed length intervals. With fixed length intervals,

more samples would be needed to characterize the CPI of a phase with same accuracy as variable length intervals[41]. Hind et al [53] formalizes program phase classification by focusing on how to appropriately define granularity and similarity to perform phase detection. Phases can also be associated with code regions that could be static[39] or dynamic[28]. Since we perform a combination of structural analysis that is static and measurements that are dynamic, we need to have a handle on the program static code. Hence only those methods that statically divide a program into phases are relevant to us. We review some static phase classification methods that associate phases with code regions.

Sondag et al[84] propose a static analysis that identifies likely phase transition points where runtime characteristics are likely to change to help guide code region to core assignment in a performance asymmetric multicore processor(AMP). Such a phase-transition point is statically instrumented to insert a small code fragment which is termed as a phase mark in [84]. A phase mark contains information about the phase type for the current section, code for dynamic performance analysis, and code for making core switching decisions. At runtime, the dynamic analysis code in the phase marks analyzes the actual characteristics of a small number of representative sections of each phase type. These analysis results are used to determine a suitable core assignment for the phase type such that the resources provided by the core matches the expected resources for sections of that phase type. On determining a satisfactory assignment for a phase type, all future phase marks for that phase type reduce to simply making appropriate core switching decisions. The method in [84] involves block typing that looks at a combination of instruction types in a basic block as well as a rough estimate of cache behavior (computed based on reuse distances [48]). Information describing these two components are used to place blocks in a two dimensional space. The blocks are then grouped using the k -means clustering algorithm. The phases detected in [84] are very different from the phases that we detect in our work as the application of phases in [84] is maximizing resources of the processor by optimal code region to core assignment. Our objective is to accurately characterize CPI of a phase and hence timing of a phase to help estimate WCET more accurately and efficiently. The phase markers in [84] involve a lot of code executed at phase transitions, whereas phase markers in our case are mere instructions and serve as boundaries to help collect phase-specific CPI and compute phase-specific worst case instruction count.

5.6 Conclusions

This chapter throws light on how program phase behavior can have important applications in worst case timing analysis. The homogeneity of a phase allows us to model WCET of a phase in terms of its average CPI. This chapter also serves to justify and support our hypothesis of estimating WCET of a program in terms of its instruction count and CPI in the case of single phase programs. While the analytical and statistical expressions described in chapter 3 applied to the whole program, in this chapter, we modify these expressions to pertain to a phase. Phases are detected by code structural analysis that mark instructions in the binary as the start of a new phase. Phases obtained this way are observed to be independent of architecture as they rely on instruction execution patterns.

We recommend to compute WCET in terms of SWIC_p and $\text{Max\_Avg}_p(\text{CPI})$ which yields about the same pessimism as *Chronos* on *Simplest* architecture. On *Inorder_complex* and *Complex* architectures, the average pessimism reduces by 43% and 55.35%. Since CPI varies in a homogeneous and repetitive manner within a phase the average CPI of a phase can be computed using CPI samples collected at arbitrarily large intervals of instructions thus reducing the instrumentation overhead. Tail end percentile functions can be used to approximate worst case CPI, but the percentile CPI values are found to be highly sensitive to the distribution of CPI data and hence would depend on the mix of data provided in the test input set. Moreover, CPI spikes that occur very rarely within a phase end up as 99th percentile CPI values causing additional pessimism. In the next chapter we shall see how to obtain probabilistic bounds of CPI that work with any arbitrary distribution of data and this probabilistic bound of phase CPI is used to obtain a probabilistic bound on program WCET as well.

Chapter 6

Probabilistic Bounds of Phase CPI and Relevance in WCET Analysis

In the previous chapter, we saw that program phase behavior has important implications on worst case timing analysis. Based on homogeneous variation of CPI within a phase, we proposed a model that estimates program WCET in terms of its phases. The proposed model uses function of average CPI which results in the estimate being approximate. In this chapter, we describe a method to compute probabilistic bounds on phase CPI which will enable us to estimate WCET of each phase at a given probability value. We also describe how probabilistic WCET estimates of each phase can be combined to give the probabilistic WCET estimate of the whole program.

Probabilistic WCET estimates have two clear advantages. These estimates are more robust and associated with real guarantees which can be confidently used in real time systems. Instead of an absolute WCET estimate, one can estimate WCET at various probabilities, especially useful, when tasks with different priorities exist. Bernat et al[29, 30] probabilistically combine worst case effects of *execution profiles* (ETP) which are a combination of basic blocks and execution frequency profile information (gathered by running the program with various inputs) under three different scenarios and build the program worst case path to estimate probabilistic WCET. Bernat et al[29] use tree based schema to combine the local timing information of basic blocks into a global WCET. The three scenarios considered are: ETPs are independent, the dependencies between ETPs are known and no dependency information among ETPs are

known. This work forms the underlying model of the commercial measurement based WCET tool, *RapiTime*[102].

In our phase based WCET analyzer, CPI samples are collected by measurement at numerous points by running benchmarks with a large number of test inputs. The true probability distribution of these CPI samples is not known. But we do know that the samples have finite mean and variance. Hence we use Chebyshev's inequality[68] to bound CPI of a phase within a confidence interval for a probability, p. Applying Chebyshev inequality to benchmarks with stable CPI behavior (coefficient of variation or CoV of CPI $\leq 0.5\%$) results in accurate WCET estimates (that are within 1% of maximum observed cycles even at p=0.99).

Some benchmarks like Bubble sort (Table 2) exhibit high variation in CPI during execution(Figure 6.1). Applying Chebyshev inequality directly for such phases yields a wide confidence interval for CPI leading to highly pessimistic WCET estimates, as execution time is directly proportional to CPI. We observe that deviations in CPI actually correspond to deviations in the program counter even at a granularity of a few tens of instructions. Using this observation, we refine such phases into smaller sub-phases based on PC (program counter) signatures, collected using profiling. These signatures basically encode path information of loop iterations in a concise manner and are analyzed to isolate high deviations in CPI. Re-applying Chebyshev inequality on CPI samples for each sub-phase gives us a tight bound on CPI thereby resulting in an accurate WCET estimate. In some programs, where points of high CPI variation are not solely determined by control flow, we go one step further and make provision to modify the refinement process so as to also allow the user to control CPI variance within a sub-phase and hence accuracy of WCET.

We evaluate this technique for all PISA architectures described in Table 3.2 in chapter 3 by comparing it with the static WCET analyzer *Chronos*. We use the same set of benchmarks taken from Mälardalen WCET project and Mibench embedded benchmark suite described in chapter 3. Since the proposed technique yields a probabilistic WCET estimate, we also compare the results with the commercial probabilistic WCET analyzer, *Rapitime*. For these purposes, we use two cycle accurate simulators *Simplesim-3.0* and *SimIt-ARM-2.1*, used by Chronos and RapiTime respectively, to carry out measurement of CPI on PISA and ARM platforms. We compute bounds on CPI for unrefined and refined phases at three chosen probability values, p=0.9, 0.95, 0.99.

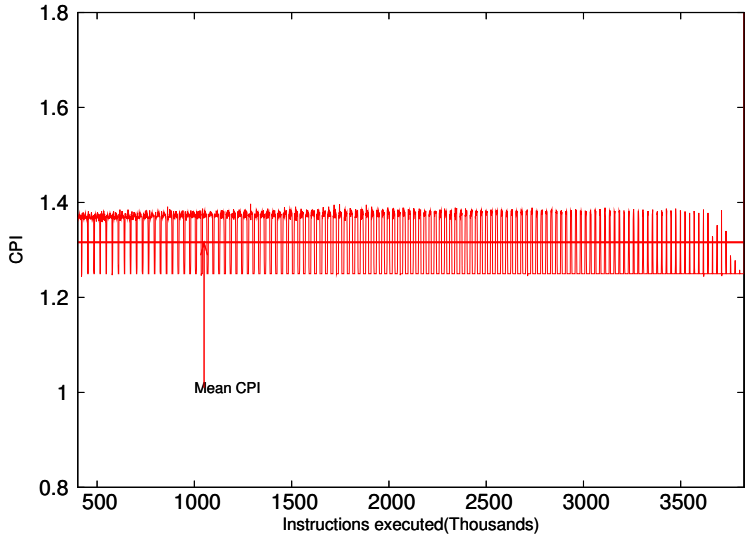


Figure 6.1: Deviation of CPI around the mean in Bubble sort on *Inorder_complex* architecture.

We address the following questions in this chapter.

1. How do we obtain robust WCET estimates that are also accurate in the phase based timing model?
2. How do we isolate points of high CPI variation within a phase?
3. How do we control CPI variation within a phase?
4. What is the impact of 2 and 3 on WCET accuracy?
5. How do we obtain probabilistic WCET for the whole program using probabilistic CPI bounds of phases?

6.1 Baseline Model

The phase-based timing model that we proposed in chapter 5 estimates program WCET as a sum of WCET of its phases. A phase corresponds to a static code region detected by code structure analysis[39]. The unit of analysis is a hierarchical call loop(HCL) graph, created out of the program binary. The program is executed with various inputs to ensure coverage of all

functions and conditions. Profile data is used to annotate the HCL graph with hierarchical information regarding number of calls, loop iteration counts, variance in instructions executed every time each call/loop is executed. The HCL graph is analyzed to pick phase marker edges. The code region lying between a marker edge *e1* and the following marker edge *e2* comprises the phase associated with *e1*.
The WCET of a program is estimated as,

$$WCET = \Sigma_{j \in \{1..p\}} WCET_j \tag{6.1}$$

where p is the number of phases of the program.
WCET of the j-th phase, is estimated as,

$$WCET_j = T_j \times WIC_j \times WCPI_j \tag{6.2}$$

where, $\mathbf{T}_j$**:** Maximum number of times phase j occurs during execution[1].
$\mathbf{WIC}_j$**:** Worst case instruction count(IC) of phase *j*. WIC_j is either the theoretical upper bound on IC derived using static analysis(SWIC_j) or the maximum observed IC of phase j(MIC_j).
$\mathbf{WCPI}_j$**:** Worst case CPI of phase *j*. CPI is measured within each phase at various points and maximum of mean CPI across all tested inputs is taken as $WCPI_j$. In this work, we use probabilistically bounded CPI instead of maximum of mean CPI to obtain a more robust WCET estimate.

A program depending on its structural complexity, can execute different code regions(phases) on execution with different inputs thereby exhibiting multiple phase sequences across inputs. In that case, WCET is estimated as the maximum among WCET of all possible sequences.

$$WCET = Max\ (\ WCET(1),\ .....\ , WCET(s)\ ) \tag{6.3}$$

6.2 Computing Probabilistic Bounds on Phase CPI

The phase based timing model defined by Eq(6.2) uses function of mean CPI values that have no probabilistic guarantees to compute phase WCET and, in turn, program WCET. Hence the

[1]From now, we factor in T_j into WIC_j for ease of notation in this chapter.

resultant WCET estimate is only approximate. We now describe how CPI of a phase is bounded for a given probability, p. To bound CPI, we collect n CPI samples for each phase(static code region) to form the sample set, $\widehat{S}_i$, by running the program with a large number of test inputs. CPI is measured at intervals ranging from 100 to 1000 instruction depending on the program dynamic execution length.

On an average over all phases of all benchmarks, CPI samples are observed to be within 10% of the sample mean($\widehat{\mu}$) on all architectures considered in this study. Our main objective is to quantify the amount by which a future CPI sample can be away from $\widehat{\mu}$ for a given probability(p). Had we known the true probability distribution of the samples (ascertained only if true population set, S_i, built by exercising *all* paths within phase i is known), we could apply an appropriate probability density function to compute the confidence interval to contain a future CPI sample for probability p. Building S_i is computationally expensive. Hence we use Chebyshev inequality as it can be applied to any arbitrary distribution. Chebyshev inequality only requires the random variable(CPI) to have finite mean and variance. If variance is small, bounds obtained using Chebyshev inequality are tight.

Chebyshev's inequality: The inequality states that p, probability of a future sample, cpi_x, being greater or lesser than mean of S_i (μ) by a distance of k, is as follows,

$$P(|cpi_x - \mu| \geq k) \ \leq \ \frac{\sigma^2}{k^2} \tag{6.4}$$

Where, k is an arbitrary constant, μ is true mean of the distribution and σ^2 is true variance of the distribution. In the context of WCET analysis, the distance k is one of the factors in determining the accuracy of WCET estimate. If k is much greater than the mean CPI, the resulting WCET estimate will be much greater than the observed maximum cycles. Chebyshev inequality is also written in the following alternate form.

$$P(|cpi_x - \mu| \geq k \times \sigma) \leq \frac{1}{k^2} \tag{6.5}$$

We can use sample mean, $\widehat{\mu}$ and square root of sample variance, $\widehat{\sigma}$ in Eq(6.4) and Eq(6.5), provided variance of sample mean, Var($\widehat{\mu}$) is small.

$$Var(\widehat{\mu}) \ = \frac{\sigma^2}{n} \tag{6.6}$$

$\mathrm{Var}(\widehat{\mu})$ is inversely proportional to the number of samples, n. Hence with increasing n, $\mathrm{Var}(\widehat{\mu})$ decreases[68]. Since we have a large number of samples, we can confidently use $\widehat{\mu}$ and $\widehat{\sigma}$ in place of μ and σ to give the following equations.

$$P(|cpi_x - \widehat{\mu}| \geq k) \ \leq \ \frac{\widehat{\sigma}^2}{k^2} \tag{6.7}$$

$$P(|cpi_x - \widehat{\mu}| \geq k \times \widehat{\sigma}) \ \leq \ \frac{1}{k^2} \tag{6.8}$$

Applying Chebyshev inequality to $\widehat{S_i}$, we obtain an upper bound value, $\mathrm{CPI}_{i,u}$ which will be greater than any future CPI sample with probability p. We refer to $\mathrm{CPI}_{i,u}$ associated with a probability p as $PrCPI_{\mathsf{p}}$. Hence original timing equation, Eq(6.2) is modified to,

$$WCET_j \leq WIC_j \ \times \ PrCPI_{\mathsf{p}} \tag{6.9}$$

Example:

Inorder to compute bounded CPI associated with a particular probability, Eq(6.8) can be re-written as follows:

$$P(cpi_x \leq \widehat{\mu} + k \times \widehat{\sigma}) \leq 1 - \frac{1}{k^2} \tag{6.10}$$

If p is equal to 0.99, we set 1-$\frac{1}{k^2}$ to 0.99, which gives a value of 10 to k.

The program *bez* has two phases, P1 and P2. Assume it has the following values, $\widehat{\mu_1}$=1.15, $\widehat{\mu_2}$=1.27, $\widehat{\sigma_1}$=0.01, $\widehat{\sigma_2}$=0.001, WIC_1=112500, WIC_2=18176352, CPI bound of phase P1 is calculated as follows:

Applying Eq(6.10), bounded CPI at p=0.99 is $\widehat{\mu_1}$ + k × $\widehat{\sigma_1}$

which is 1.15 + 0.01 × 10 = 1.25

Similarly, CPI bound of phase P2 is calculated as follows:

Applying Eq(6.10), bounded CPI at p=0.99 is $\widehat{\mu_2}$ + k × $\widehat{\sigma_2}$

which is 1.27 + 0.001 × 10 = 1.28

According to Eq(6.9), we estimate WCET as

WCET ≤ WIC_1× 1.25 + WIC_2× 1.28

In the next section we shall see how the probabilistic WCET of the whole program can be obtained in terms of probabilistic WCET of the constituent phases. Since we use a theoretically

bounded value for WIC_j, which is a constant, the probabilistic guarantee associated with PrCPI_p applies to the resultant WCET of phase as well.

6.3 Estimating Probabilistic Program WCET

In the previous section, we treated phase CPI as a random variable. The CPI of a phase varies within a small percentage around the mean. We apply Chebyshev inequality to bound a future CPI sample with a given probability. This enables us to estimate the probabilistic WCET of a phase. If we treat WCET as our random variable, We can carry forward the same logic and estimate probabilistic WCET of the whole program. For ease of notation, let us term WCET as 'W', theoretical upper bound on IC (SWIC) as 'f' and CPI as 'C'. The WCET of a program with a number of phases can be formulated as in Eq.(6.9). Inorder to compute a bound on W, using Chebyshev inequality, we need to compute mean and variance of W.

$$W \;=\; \Sigma(f_i \times C_i) \tag{6.11}$$

$$Expectation(W)\ or\ E(W)\ or\ \mu_w \;= \Sigma(f_i \times E(C_i)) \tag{6.12}$$

As, f_i is a constant,

$$E(W) \;= \mu_w \;=\; \Sigma(f_i \times \mu_i) \tag{6.13}$$

$$Var(W) \;= \sigma_w^2 \;=\; \Sigma(f_i^2 \times Var(C_i)) \tag{6.14}$$

C_i values are not correlated with C_j values and hence assumed to be independent. The C_i and C_j values refer to measured CPI values within phase i and j. It has been observed through experimentation that one cannot guess CPI values of phase j with the help of phase i. Hence they are not correlated. Therefore we can write,

$$Var(W) \;= \sigma_w^2 \;=\; \Sigma(f_i^2 \times \sigma_i^2) \tag{6.15}$$

Applying Chebyshev inequality on W, on the lines of Eq(6.5),

$$P(W - E(W) \geq k \times \sigma_w) \leq \frac{1}{k^2} \tag{6.16}$$

$$P(W - E(W) \leq k \times \sigma_w) \leq 1 - \frac{1}{k^2} \tag{6.17}$$

$$P(W \leq E(W) + k \times \sigma_w) \leq 1 - \frac{1}{k^2} \tag{6.18}$$

Considering,

$$W \leq E(W) + k \times \sqrt{Var(W)} \tag{6.19}$$

$$W \leq \Sigma(f_i \times \mu_i) + k \times \sqrt{\Sigma(f_i^2 \times \sigma_i^2)} \tag{6.20}$$

We know that, $(\Sigma(X_i))^2$ is greater than $\Sigma(X_i^2)$. Hence we can say,

$$W \leq \Sigma(f_i \times \mu_i) + k \times \sqrt{(\Sigma(f_i \times \sigma_i))^2} \tag{6.21}$$

Which gives us,

$$W \leq \Sigma(f_i \times \mu_i) + k \times \Sigma(f_i \times \sigma_i) \tag{6.22}$$

Removing f_i outside,

$$W \leq \Sigma(f_i(\mu_i + k \times \sigma_i)) \tag{6.23}$$

Which essentially means, once we compute probabilistic bounds of phase CPI to estimate probabilistic phase WCET, the sum of probabilistic phase WCET gives the probabilistic WCET of the whole program as well. Similar to the case where we estimate probabilistic bounds on phase CPI, we use sample mean ($\widehat{\mu}$) and square root of sample variance ($\widehat{\sigma}$) in place of true mean and true variance in the above equations to obtain our bounds on execution time.

Example:

Continuing with the same example that we used earlier to compute bounds on phase CPI for two phases, we now show how to use Eq.(6.23) to estimate whole program probabilistic WCET. Treating W as a random variable and applying Chebyshev inequality on W itself, by the above derivation,

$$P(W \leq (f_1 \times (\widehat{\mu_1} + k \times \widehat{\sigma_1}) + f_2 \times (\widehat{\mu_2} + k \times \widehat{\sigma_2}))) \leq 1 - \frac{1}{k^2} \tag{6.24}$$

To get a bound on W, we proceed likewise as in the case of obtaining a probabilitic bound on phase CPI. Assume p=0.99, we first obtain k to be 10. W is calculated by substituting values for f_1 (WIC_1) and f_2 (WIC_2) and $\widehat{\mu_1}$, $\widehat{\mu_2}$, $\widehat{\sigma_1}$, $\widehat{\sigma_2}$, that were obtained in the previous example. If WIC_1=112500 and WIC_2=18176352, we estimate W as,

W $\leq$ 112500 $\times$ 1.15 + 18176352 $\times$ 1.27 + 10 $\times$ (112500 $\times$ 0.01 + 18176352*0.001)

The Chebyshev inequality described in the above equations provides both the lower and upper bounds for phase CPI. Since we are interested in estimating WCET, we need to obtain only the upper bound. Hence we use the one tailed version of Chebyshev inequality- *Chebyshev-Cantelli* inequality to obtain probabilistic upper bound on phase CPI in this work.

Chebyshev-Cantelli inequality: The inequality states that p, probability of a future sample, cpi_x, being greater than mean of S_i (μ) by a distance of k is as follows,

$$P(cpi_x - \mu \geq k) \ \leq \frac{\sigma^2}{\sigma^2 + k^2} \tag{6.25}$$

Stated in an alternate form,

$$P(cpi_x - \mu \geq k \times \sigma) \ \leq \frac{1}{1 + k^2} \tag{6.26}$$

The procedure to compute the CPI bound and the bound on execution time using Chebyshev inequality, described above can be similarly adapted to use Chebyshev-Cantelli inequality. For benchmarks that exhibit low variance in CPI, both Chebyshev and Chebyshev-Cantelli inequality tightly bounds phase CPI. However, applying the inequality to phases with high variance in CPI results in high $PrCPI_{\mathsf{P}}$ values and hence pessimistic WCET estimates. In the next section, we will see how such phases can be divided into smaller sub-phases to obtain tighter CPI bounds and hence tighter WCET estimates.

6.4 Phase Refinement

Code structure analysis [39] ensures that variation in instructions executed within a phase is much lesser than the corresponding variation across different phases. However, presence of *if-conditions* in loops or calls can cause high variance in instructions executed across loop iterations or call invocations. The *if-condition* of the inner-loop in *Bubble_sort* (Figure 6.2), when *true*, executes additional code compared to when the condition is *false*. The HCL graph for the routine created by profiling with a set of 5 different inputs, is shown in the same figure. Each loop is represented by a loop head node and a loop body node in the HCL graph. The edge associated with loop head node stores information about the number of times loop

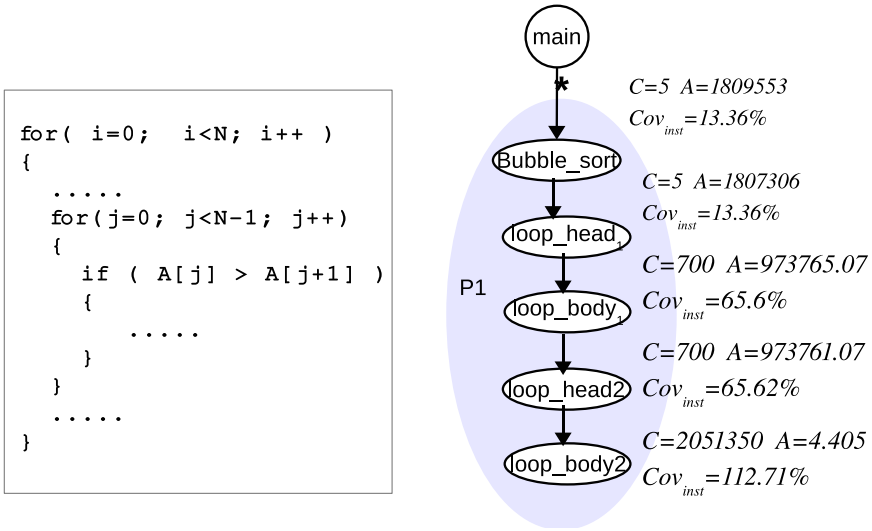


Figure 6.2: Code structure of *Bubble_sort* routine and the corresponding HCL graph.

head was executed(**C**), hierarchical average number of instructions(**A**) and CoV in instructions executed($\mathbf{CoV}_{inst}$) over different executions. Similarly the edge associated with loop body node stores these information pertaining to a loop iteration. It can be observed that loop edges have a very high CoV_{inst} hence do not qualify as software markers resulting in the whole routine being selected as a single phase [39]. Considering the entire loop as a single phase exhibits high variation in CPI(Figure 6.1).

In such cases, we could statically mark the code region associated with each iteration and hence each path per iteration as a different phase. But that would not work because,

a) The underlying architecture is based on a pipeline consisting of several stages. Multiple instructions are in flight at the same time. A phase should be lengthy enough to allow at least few instructions to completely execute to facilitate calculation of CPI of the phase.

b) Instrumenting every few instructions can hamper performance of the program that we are trying to measure.

Hence it appears that we need to find a mid point where phases are big enough, at the same time, small enough to obtain tight bounds on CPI.

An intuitive approach is to consider every x consecutive iterations of a loop, L, as a potential phase, which we term as a *window*, W. We emit dynamic execution information of every

window, W, in the form of a triple, defined as a *PC signature* (Figure 6.3), storing the following information.

PCbitmap: The bitmap is a vector of 4 integers (128 bits). The simulator hashes every instruction PC encountered and stores it into the bitmap. 127 bits are sufficient to map PC addresses in each phase of the benchmarks considered in this thesis.

CPI: CPI represents observed cycles per instruction while instructions belonging to W are executed.

IC: IC represents number of instructions executed that belong to W.

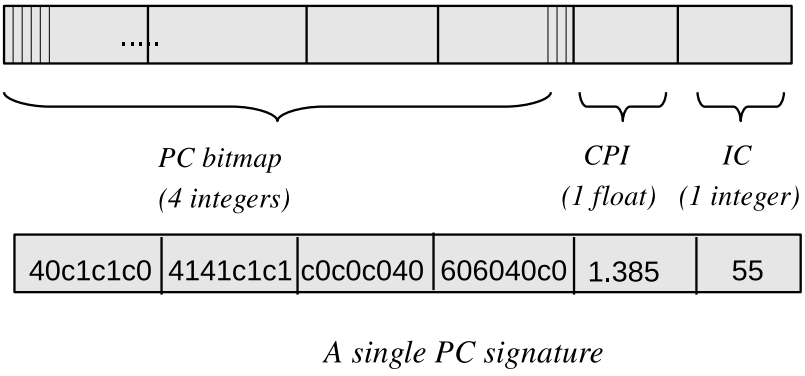


Figure 6.3: Format of a single PC signature.

6.4.1 Refinement Based on PC Signature

Refinement consists of three steps: trace generation, trace compression and classification of compressed trace into sub-phases.

Trace Generation

In order to generate a trace, we first identify the branch instruction that iterates loop L of the phase. If L is nested with several levels, we select the innermost loop. The simulator is modified to count x consecutive executions of the branch instruction of L. If $Min(|L_i|)$ denotes the minimum number of instructions executed in each loop iteration i of L, number of iterations

that make up a single window, x is defined as,

$$x = \left\lceil \frac{Phase\_length}{Min(|L_i|)} \right\rceil \tag{6.27}$$

where, *Phase_length* is the number of instructions that make up a phase. On a given architecture, *Phase_length* should be greater than the minimum number of instructions that have to be executed for at least one instruction commit. A *Phase_length* of *50* instructions suffices for all architectures considered in this thesis (Table 3.2). With x being a ceiling value, Eq.(6.20) might cause some windows to be composed of more than *50* instructions. x may not always be an exact multiple of $|L|$. Hence the execution time of the last few iterations will have to be added separately. If the phase has multiple loops, the same procedure is repeated for all loops within the phase.

A loop with small $|L_i|$ will have windows comprising of a large number of iterations. If $|L_i|$ is greater than minimum *Phase_length*, every iteration forms a window. If $|L_i|$ is well beyond minimum *Phase_length*, we can use code structure analysis to break it into smaller phases. The cycles taken by code preceding loop L of phase P, if any, is added separately.

Figure 6.4 describes necessary modifications made to *Simplesim-3.0* to generate PC signatures. At the end of a program run, we have a trace consisting of $\frac{|L|}{x}$ such signatures, where $|L|$ is the loop iteration count of L. The modified simulator code does not impact execution cycles of the program as PC values are read off the pipeline and processed in parallel. In the worst case, for every 50 instructions executed, a trace comprising of 6 words, is emitted out and the 4-word hash table is reset.

An integer vector that stores occurrences of each PC encountered would be more accurate to represent path information of every window, instead of the existing bitmap. But that would clearly not scale with x and would lead to huge traces. The bitmap is imprecise as we shall now see with an example.

Example: In Figure 6.2, assume the inner loop executes 18 instructions when the *if-condition* evaluates to *true* and 10 instructions when it evaluates to *false*. Let window size, x be 4 iterations. Consider two such windows W_1 and W_2. Assume in W_1: *if-condition* is *true* once and false three times. In W_2, *if-condition* was *true* three times and *false* once. The PCbitmap will be identical in both cases but IC(W_1) = $18 \times 1 + 10 \times 3 = 48$. IC($W_2$) = $18 \times 3 + 10$

```
/* Hash Table size */
#define TABLESIZE 4

/* PC bitmap in the form of Hash Table */
int PathVector[TABLESIZE];

/* Number of unique PC addresses tracked */
int HashValue = 127;

/* Path mask that isolates each of 128 bits of PC */
int PathMask[128] = { 0x1, 0x2, 0x4, ... };

......

void ResetHashTable()
{
 int i;
 for(i=0;i < TABLESIZE; i++)
  PathVector[i] = 0;
}

/* Actual Hash function */
int Hash(long PC)
{
  return (PC & 0xffff) % HashValue;
}

......

/* insert following code where instructions finish execution */

if(regs.regs_PC == <branchL>) /* branchL : iterating instruction */
{
  if(counter % x == 0)
  {
    /* Emit PathVector, IC, CPI of last x executions */

   ResetHashTable();
  }

  counter++;
}

/* Record PC that was executed into PC bitmap */
hashval=Hash(regs.regs_PC);
PathVector[hashval/WORDSIZE] |= PathMask[hashval];
```

Figure 6.4: Simulator code to implement trace collection for each window of loop L of phase P

$\times 1 = 64$. Hence IC serves to store extra information without bloating the trace.
Although seeming imprecise, the combination of IC and PCbitmap is observed to be sufficient to isolate high CPI variations in most cases.

Trace Compression

Lengthy program runs can produce megabytes of trace. But they are easily compressible owing to the repetitive nature of phases. A large number of consecutive windows have identical PC signatures which can be compressed(Figure 6.5). We look for consecutive triples that repeat to

Trace of a single run of Bub

PC Bitmap	CPI	IC
.....		
202020002020202020	1.250000	64
202020002020202020	1.250000	64
202020002020202020	1.250000	64
202020002020202020	1.250000	64
202020002020202020	1.250000	64
20202020202020203030202030303030	1.451610	93
2020200020202020202020202020	1.291140	79
202020002020202020	1.250000	64
202020002020202020	1.250000	64
202020002020202020	1.250000	64
202020002020202020	1.250000	64
.....		

Compressed trace of a single run of Bub

PC Bitmap	CPI	IC	#duplicates
.....			
202020002020202020	1.250000	64	147
20202020202020203030202030303030	1.451610	93	1
2020200020202020202020202020	1.291140	79	1
202020002020202020	1.250000	64	93
.....			

Figure 6.5: Signature trace of a single run of *Bubble sort* and its compressed version.

compress them. The time complexity of the compression algorithm is linear to the trace size.

Trace Classification

A one-to-one correspondence is observed between <PCbitmap, IC> and CPI in the trace for program *Bubble sort* (Figure 6.5), which is observed in other benchmarks as well. This happens because CPI is largely determined by the instructions that execute [39]. Based on this, we define a *sub-phase* as a unique pair of <PCbitmap, IC> values. All windows with the same <PCbitmap, IC> value belong to one sub-phase. For each such sub-phase, new confidence intervals are computed by applying Chebyshev-Cantelli inequality on CPI samples pertaining to that sub-phase.

The time taken by the classification algorithm is $O(m \times n)$, where m is the number of unique <PCbitmap, IC> pairs (sub-phases) and n is the number of entries in the compressed trace. On an average, number of sub-phases, m, detected for benchmarks used in this thesis is around 8 on all PISA architectures considered even if number of windows for some benchmarks go upto a few thousands. The size of compressed trace obtained across all inputs for a program, n, ranges from 2 MB to 2.7 GB. The average sub-phase size observed across all benchmarks is 79 resulting in an average instrumentation overhead of 1.26%.

6.4.2 Refinement Based on CPI Variance

In spite of refinement based on PC signature, certain sub-phases continue to exhibit high variance of CPI. Hence we add another level of refinement wherein the user can control the variance of CPI within the sub-phase. The classification will now be based on <PCbitmap,

IC, CPI-range>. The procedure repeatedly splits the sub-phase until the CPI values fall in the desired range giving rise to variance well within the specified limit(Figure 6.6)[2]. The time complexity of *Split* is O($n \times log(n)$), where n is the number of entries in sub-phase CPI file. The overall time complexity of the refinement procedure is O($m \times n \times log(n)$) where m is the number of original sub-phases.

```
/*****************   Refine sub-phase based on CPI **************/
/* Inputs:                                                       */
/* <CPI data for each sub-phase>                                 */
/* <Variance_threshold>                                          */
/* Output:                                                       */
/* <New Bounds on CPI for each new sub-phase>                    */
/*****************************************************************/

Procedure Split(sub-phase_CPI_vector)
 Compute variance of CPI;
 While (variance > Variance_threshold) do
   Split sub-phase_CPI_vector into two depending on range of values
 /* vector_1 range: [lower..mean] vector_2 range: [mean..upper] */
   Split(vector_1);
   Split(vector_2);
  end for
end Split
Procedure Bounds(sub-phase_CPI_file)
 Compute mean of CPI;
 Compute Variance of CPI;
 Compute Chebyshev bounds on CPI;
end Bounds
Procedure main
 for each sub-phase, i, do
  Split(i); // generates new sub-phases
  for each new-sub-phase, j, do
   Bounds(j);
  end for
end for
end main
```

Figure 6.6: Algorithm to refine sub-phase based on CPI variance.

6.4.3 WCET Estimation Using Sub-Phases

A phase represents a static code region. Whereas a sub-phase represents a group of consecutive loop iterations. Every single loop iteration is included in the analysis. Sub-phases do not overlap as each of them represent a different group of loop iterations. In order to estimate WCET in terms of sub-phases, Eq.(6.2) has to be suitably modified. Different phases can

[2]The term Chebyshev bounds used in Figure 6.6 refers to Chebyshev-Cantelli bounds

occur on execution with different inputs[11]. The same holds for sub-phases. The set of sub-phases that occur for a particular program run with input i forms a sub-phase sequence $\mathcal{S}_i$. Note that we are not interested in the exact order in which sub-phases occur. We only need the frequency of occurrence of each sub-phase. A sub-phase sequence ($\mathcal{S}_i$) obtained with input i, takes the form of an integer vector, $[s_{i.0},\ s_{i.1},\ \ldots\ ,\ s_{i.sp}]$ where sp is the total number of sub-phases appearing across all inputs. $s_{i.j}$ indicates the number of times sub-phase j occurs in the execution run of program with input i. Among two sequences, $\mathcal{S}_a$ and $\mathcal{S}_b$, obtained with inputs a and b, such that $s_{a.m} \geq s_{b.m}\ \forall$ m = {0,..,sp}, we include only $\mathcal{S}_a$. The number of unique sub-phase sequences that can occur range from 1 to upto a few hundred.

The CPI for each sub-phase possible, j, is bounded within limits for a particular probability value, p using Chebyshev-Cantelli inequality as discussed in the earlier sections to give $\mathrm{PrCPI}_{\mathsf{j.p}}$ for a probability p. The probabilistic WCET of each sub-phase , j, for a probability p, is bounded as

$$WCET_{j.p} \leq WIC_j \times\ PrCPI_{\mathsf{j.p}} \tag{6.28}$$

Where, WIC_j is the theoretical upper bound on IC of sub-phase, j. Since a sub-phase represents x iterations of a loop, L, WIC_j is computed as the maximum number of instructions executed per iteration of that loop L multiplied by x.

A phase is constituted of a number of sub-phases. The entire program is composed of a number of phases. However, depending on program structure, we can obtain multiple sub-phase sequences wherein each sub-phase sequence is associated with a corresponding probabilistic WCET. Hence for each unique sub-phase sequence, $\mathcal{S}_i$, we calculate the weight of each sub-phase, m, occurring in that sequence as,

$$w_m\ =\ \frac{s_{i,m}}{\Sigma_{j\in\{0,..,sp\}} s_{i,j}}$$

And consequently, probabilistic $WCET_i$ of the whole program for sequence $\mathcal{S}_i$, is bounded as described in section 6.3 as,

$$WCET_i \leq\ \Sigma_{j\in\{0,..,sp\}}\ (w_j\ \times\ SWW\ \times\ WIC_j \times\ (\mu_j + k \times \sigma_j)\ ) \tag{6.29}$$

Where, SWW is the theoretical upper bound on the number of windows or sub-phases possible for a given loop making up a program phase. μ_j and σ_j are the sample mean and square root of sample variance of CPI of sub-phase j occurring in sequence i. SWW is estimated as

$$SWW = \frac{|L|}{x}$$

Where, $|L|$ denotes the loop bound of L, x denotes the number of iterations per window considered as a sub-phase.
The probabilistic WCET over all possible sub-phase sequences, s, is estimated as,

$$WCET = max(WCET_1, ..., WCET_s) \tag{6.30}$$

6.4.4 Context Sensitivity

An analysis of a program fragment is said to be context sensitive if it takes into account the context in which the fragment appears. Context sensitive analysis has been observed to improve precision of WCET analysis significantly[73]. Context sensitive analysis is typically applied for procedures and loops. In this thesis, we treat a procedure appearing in two different contexts as two different procedures. Since we group iterations as sub-phases and encode path information taken in these iterations in the form of a PC signature, we can now distinguish loop iterations by considering them as sub-phases based on the PC bitmap and IC values. Also, it is observed that the first iteration of a loop takes more time to execute (greater CPI) than rest of the iterations[54, 73]. Hence we treat the first window (that includes the first iteration) of a loop as a separate sub-phase.

6.5 Evaluation

All our experiments are performed on benchmarks taken from *Mibench* and Mälardalen standard WCET project benchmark suite as mentioned in Chapter 3 for *Simplest*, *Inorder_complex* and *complex* architectures inorder to compare the estimates with that of *Chronos*, the open source static WCET analyzer. The benchmarks are compiled to MIPS PISA binaries with -O2 -static flags inorder to be compared with *Chronos*. *Simplescalar Version 3.0* is used to obtain CPI samples and generate traces of PC signatures with modifications described in section 6.4.1

for comparison on PISA architectures.Input selection is done with the criterion of adequate structural coverage of the program and with the widest possible range of data as described in section 3.2 of chapter 3. Each (benchmark, input) pair is executed with a minimum of 500 different inputs multiple number of times to model different initial states [9] to see that atleast one million CPI samples per phase are generated. Invalid inputs and inputs that terminate execution early are not considered for analysis. In the next section, we shall compare our technique with *RapiTime*, which is a commercial measurement based WCET analysis tool, which also gives probabilistic WCET estimates. More details about RapiTime and it's usage are described in the Appendix.

6.5.1 Impact of Coefficient of Variation of CPI on Probabilistic Upper Bound of CPI

Chebyshev-Cantelli inequality yields tight CPI bounds for phases that exhibit low CoV(CPI) as can be seen from Figures 6.7, 6.8 and 6.9 which plot $\mathrm{PrCPI}_{\mathsf{p}}$ at $\mathsf{p}=\{0.9, 0.95, 0.99\}$, normalized to mean CPI, for all program phases on *Simplest*, *Inorder_complex* and *complex* respectively. However, applying the inequality directly to phases like *Mat_P2*(*Inorder_complex and complex*) and *Nsch* with high CoV(CPI) results in pessimistic upper bounds of CPI, as can be seen from Figures 6.7, 6.8 and 6.9. Hence we need to refine such phases into smaller sub-phases. This will reduce CPI variance within a sub-phase and help yield tighter CPI bounds.

6.5.2 Impact of Refinement on Coefficient of Variation of CPI

We now compare sub-phases obtained using refinement based on unique <PCBitmap, IC> pairs with the corresponding unrefined phase based on their coefficient of variation of CPI. Figures 6.10, 6.11 and 6.12 group sub-phases into four categories as shown. *Lms* (*Simplest* architecture) and Janne_complex are not shown as they exhibit very low variance of CPI even without refinement. The proportion of sub-phases falling in each of the four categories for all architectures is summarized in Table 6.1. The results indicate that refinement based on PC signature successfully brings down the coefficient of variation of CPI for all three architectures. However, there are many sub-phases that continue to exhibit high CoV(CPI) post refinement as can be observed from the figures. Such sub-phases are further refined into smaller sub-phases based on CPI variance as outlined in Section 6.4.2.

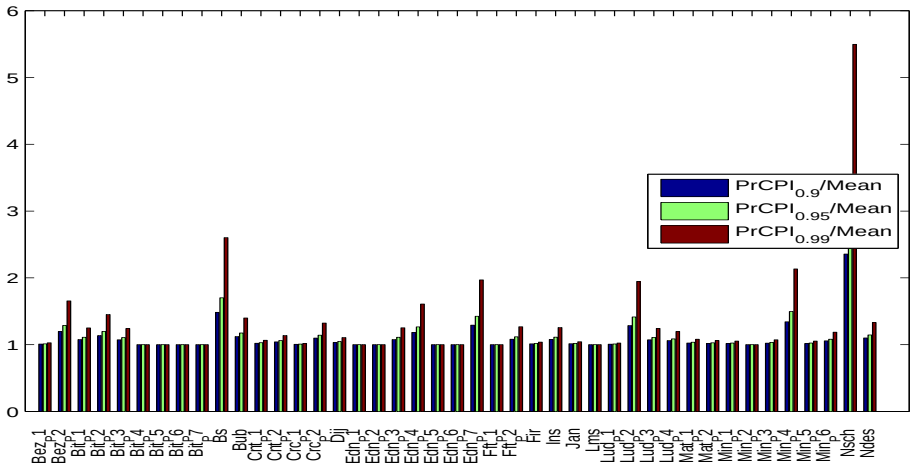


Figure 6.7: Ratio of probabilistic CPI upper bound to mean CPI at p={0.9, 0.95, 0.99} on *Simplest* architecture.

Table 6.1: Average proportion of sub-phases falling in all four categories on all PISA architectures.

Architecture	**0% < Var < 25%**	**25% < Var < 50%**	**50% < Var < 75%**	**Var > 75%**
Simplest	84.85	1.5	1.62	12.03
Inorder_complex	60.53	6.3	8.89	24.28
Complex	53.89	11.52	5.63	28.96

6.5.3 Accuracy of WCET

In this section, we evaluate the pessimism of WCET estimated using probabilistically bounded CPI at three different probability values p={0.9, 0.95, 0.99} on all PISA architectures. We also compare the pessimism of WCET at p=0.99 with the estimate made by *Chronos*. It is to be noted that the estimate given by *Chronos* is an *absolute* WCET estimate, whereas the estimate we compute in this chapter is probabilistic and depends on the probability at which we compute it. Figure 6.13 plots the ratio of estimated WCET to maximum observed cycles (*Pessimism in the WCET estimate*) observed when the proposed phases/sub-phases are used for p={0.9, 0.95, 0.99} for some of the programs. *Unrefined.architecture* and *Refined.architecture* bars represent the pessimism observed using unrefined phases and phases refined based on PC signature respectively for a particular architecture. The *50per-ref.architecture*, *10per-ref.architecture*, *5per-ref.architecture* and *1per-ref.architecture* bars indicate pessimism in WCET obtained using refined sub-phases with variance of CPI limited at {50%, 10%, 5% and 1%} of CPI variance of original sub-phase respectively on a particular architecture. *Simplest*, *Inorder_complex* and

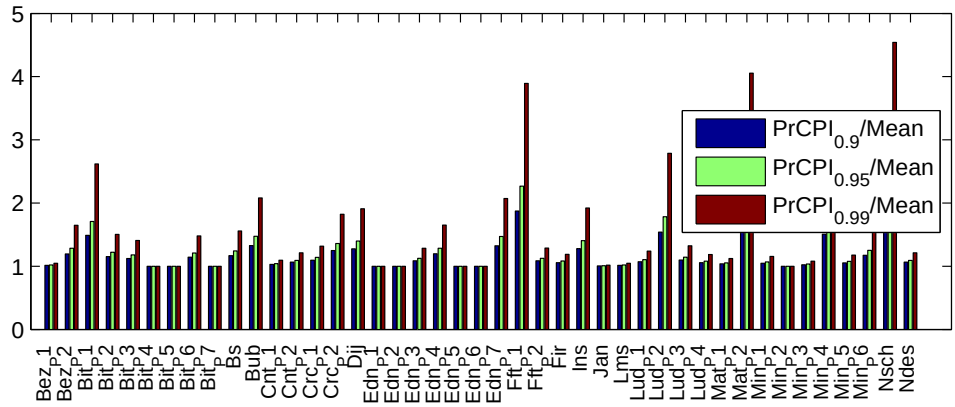


Figure 6.8: Ratio of probabilistic CPI upper bound to mean CPI at p={0.9, 0.95, 0.99} on *Inorder_complex* architecture.

complex architectures are abbreviated using *sim*, *inc* and *com* respectively in these figures.

It can be observed in figure 6.13 that, the pessimism in WCET estimates obtained using unrefined CPI values, at the three probability values are quite apart from each other at $\mathsf{p} = 0.99$. The large difference stems out of large variance in CPI and is undesirable as it results in pessimistic WCET estimates with increasing p. The difference however decreases as the phase is refined further and further. This trend applies for all other programs across all three PISA architectures. On *Simplest* architecture, which does not have a data cache, the difference quickly subsides either after refinement based on PC signature or the phase is refined to contain samples whose variance is limited to 50% of the variance of the original phase. On *Inorder_complex* and *Complex* which have both instruction and data caches and complex branch prediction schemes, additional levels of refinement are required for the difference in pessimism of WCET to decrease.

Figures 6.14, 6.15 and 6.16 compare the pessimism of probabilistic WCET estimated at p=0.99 with the estimate made by the static WCET analysis tool, *Chronos*, on *Simplest*, *Inorder_complex* and *Complex* respectively. The missing bars for *Chronos* for some programs on *Inorder_complex* and *Complex* architectures indicate that *Chronos* either gives a segmentation fault during analysis or goes out of memory. Benchmarks like *Lms*(*Simplest*), *Janne_complex* exhibit very low CPI variance as it is without any refinement. As a result, the WCET estimates are very precise (within 1% of maximum observed cycles) as shown in these figures. Benchmarks

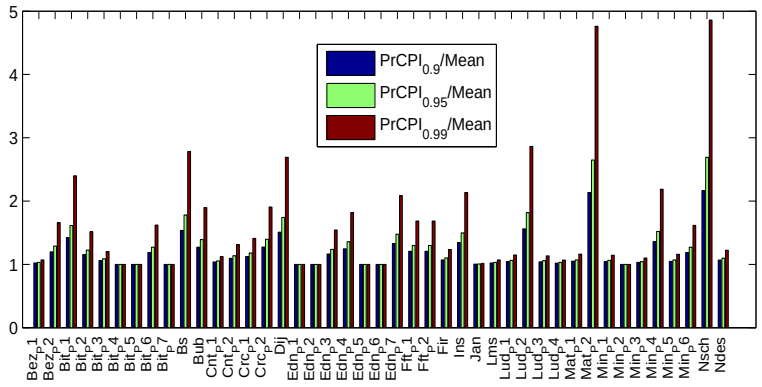


Figure 6.9: Ratio of probabilistic CPI upper bound to mean CPI at p={0.9, 0.95, 0.99} on *Complex* architecture.

like *Bezier* exhibit low variance in CPI post refinement based on signature. Other benchmarks like *Bitcount*, *Bubble sort* etc exhibit high variance in CPI when their phases are unrefined. However, as the phases are more and more refined, the pessimism of their estimates also improve.

Table 6.2 summarizes the reduction in pessimism due to various levels of refinement, the average pessimism of all refined estimates and the average improvement observed in the WCET estimate compared to that of *Chronos*. On all architectures, refinement based on PC signature results in an improvement in pessimism in the range of 22-26%. The improvement is larger with more levels of refinement. The average improvement in WCET estimate obtained by refinement based on PC signature at p=0.99 on *Simplest* architecture is 9.1%. It improves to 12.9% when the sub-phase is refined to contain samples that have half variance in CPI compared to the original sub-phase. Subsequent improvements on further refinement are marginal. The CPI values on *Simplest* architecture are more regular and hence variance quickly settles with 50-per refinement.

The average improvement in WCET estimate obtained by refinement based on PC signature at p=0.99 on *Inorder_complex* architecture is 23.1%. Subsequent levels of refinement improves the estimate further by 38.1%, 39.8%, 41.5% and 42.7%. A similar trend is observed in *Complex* architecture, where the average improvement in WCET estimate obtained by refinement based on PC signature at p=0.99 is 32.9%. And subsequent levels of refinement pushes the

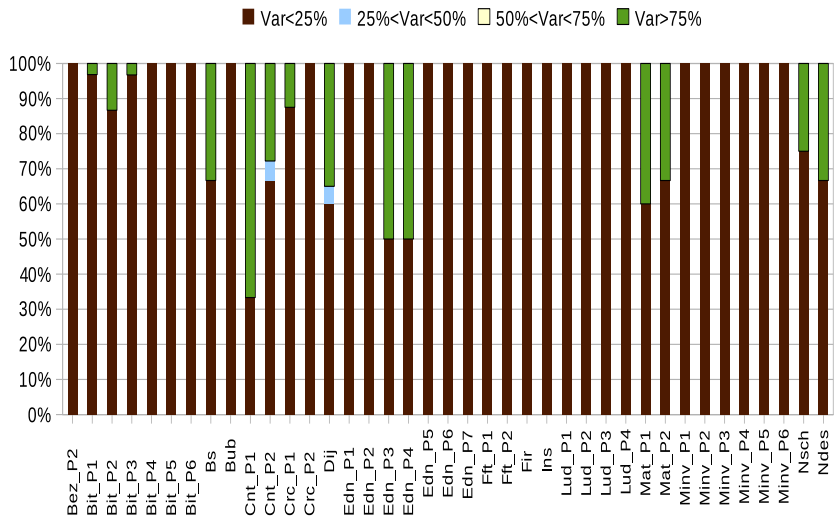


Figure 6.10: *Simplest:* Percentage breakup of sub-phases based on CoV(CPI).

improvement to 46.1%, 47%, 50.5% and 52.1%. In both architectures, CPI variation is more scattered compared to *Simplest* architecture and refinement based on CPI helps control the pessimism to a good extent.

Figure 6.17 plots the level of refinement needed to reach a point of zero CPI variance in every sub-phase of the benchmark. Zero variance implies a constant pessimism in WCET irrespective of the probability at which it is computed. The benchmarks falling under the black line have accurate WCET estimates either without refinement or when refined based on PC signatures alone and hence not considered for refinement based on CPI variance. *Bubble sort*, *Bitcount*, *Cnt*, *Dijkstra* and many other benchmarks continue to show variance in CPI even beyond a point when CPI variance is limited to 1% of CPI variance of the original sub-phase. With CPI variance of a sub-phase limited to 50% of original sub-phase CPI variance, 12 out of 18 benchmarks reach the point of maximum WCET accuracy on *Simplest* architecture. On *Inorder_complex* and *Complex* architecture, only 6 out of 18 benchmarks reach the point of maximum WCET accuracy when CPI variance of a sub-phase is limited to 50% of original sub-phase CPI variance.

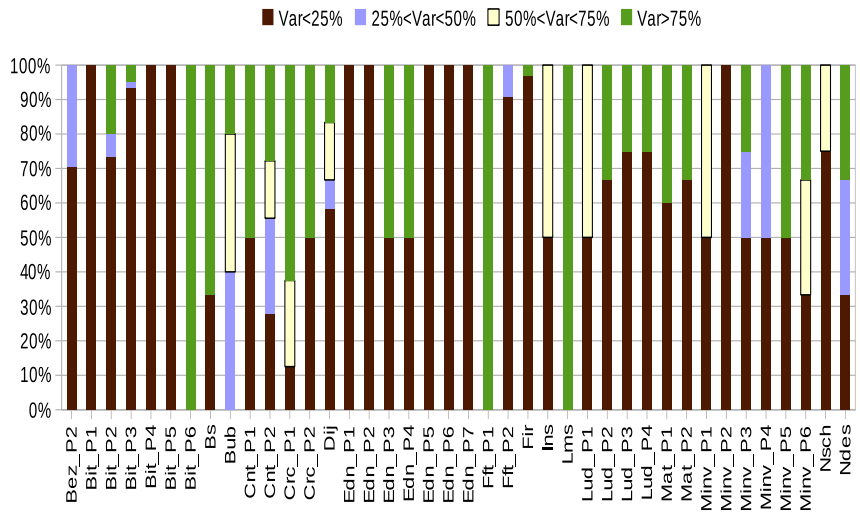


Figure 6.11: *Inorder_complex:* Percentage breakup of sub-phases based on CoV(CPI).

6.5.4 Impact of Refinement on Number of Sub-phases

Refinement splits a phase into smaller sub-phases based on PC signature. When a sub-phase is refined based on CPI variance, many more smaller sub-phases are generated. Figures 6.18, 6.19 and 6.20 plot the increase in number of sub-phases due to refinement based on PC signature(indicated by *Refined* and refinement based on CPI variance (*50-per*, *10-per*, *5-per*, *1-per*). Number of sub-phases reaches a saturation point for 95.3% of phases by the time CPI variance of a sub-phase is limited to half the CPI variance on *Simplest* architecture. On *Inorder_complex* architecture, 55.8% of phases reach saturation point with respect to number of sub-phases when CPI variance is limited to half the original CPI variance. On *Complex* architecture, the number of phases that reach saturation point decreases to 54.8%. The reason is the increase in complexity of the architecture from *Simplest* to *Complex* leading to more scattered CPI variation within a phase.

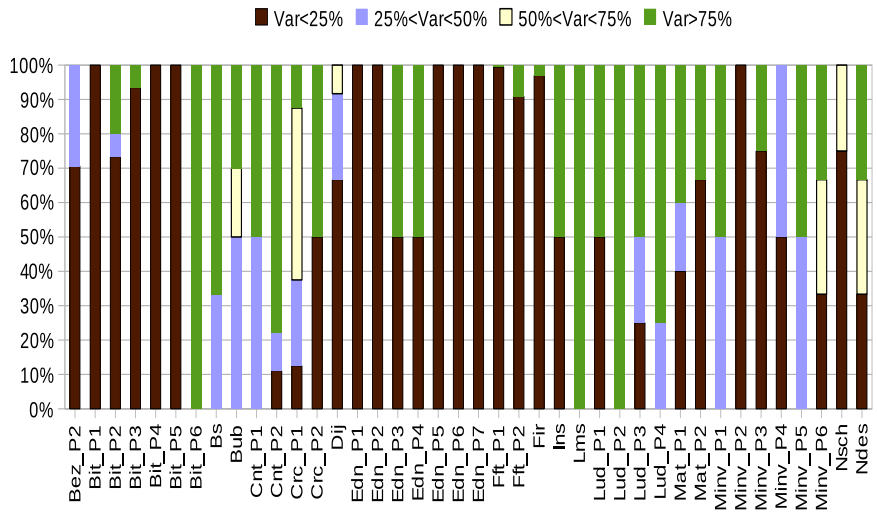


Figure 6.12: *Complex:* Percentage breakup of sub-phases based on CoV(CPI).

6.5.5 Compression

Table 6.3 compares the average sizes of trace obtained across all inputs before and after compression on both architectures. The highest compression factor observed is 24.8 on *Simplest* architecture as CPI patterns are more regular. The compression factor on *Inorder_complex* and *Complex* reduces to 14.6 and 14.9 respectively.

6.5.6 Impact of Refinement on Number of Samples

The process of refinement based on signature involves forming sub-phases by classifying the CPI samples into various categories depending on their PC signatures. Refinement based on CPI variance further forms smaller sub-phases out of the existing sub-phases by limiting the CPI samples to a particular variance value. Naturally, the sample size reduces with every level of refinement which is what is depicted in Figures 6.21, 6.22 and 6.23. Since variance of mean is inversely proportional to the number of samples, a large sample set is desirable. Hence in our experiments we select inputs which will result in atleast a million samples such that we are left with a few thousand samples at the maximum level of refinement. The maximum

Table 6.2: Impact of Refinement on pessimism of WCET and comparison with *Chronos*.

p	100-per	50-per	10-per	5-per	1-per
(Simplest)					
% Reduction in pessimism compared to unrefined WCET					
0.9	16.75	17.5	17.5	17.55	17.55
0.95	18.7	19.8	19.8	19.9	19.9
0.99	24.8	27.1	27.2	27.2	27.2
% Average Pessimism of all refined estimates					
0.9	26.5	24.3	24.3	24.3	24.3
0.95	28.7	24.8	25.8	25.8	25.8
0.99	38	26.8	26.8	26.8	26.8
% Improvement in accuracy compared to Chronos					
0.99	9.1	12.9	13.1	13.1	13.1
(Inorder_complex)					
% Reduction in pessimism compared to unrefined WCET					
0.9	16.9	22.9	23.4	24.2	24.7
0.95	18.3	26.6	27.2	28	28.5
0.99	22	35.9	36.9	38.1	38.7
% Average Pessimism of all refined estimates					
0.9	47.2	33.7	32.1	29.2	27.2
0.95	56.8	35.5	33.3	30	27.8
0.99	96.9	43.7	38.8	33.5	30.4
% Improvement in accuracy compared to Chronos					
0.99	23.1	38.1	39.8	41.5	42.7
(Complex)					
% Reduction in pessimism compared to unrefined WCET					
0.9	17.2	23.3	23.6	25.1	25.5
0.95	21.3	28.2	28.7	30.3	30.7
0.99	26.3	38.4	39.3	41.1	41.6
% Average Pessimism of all refined estimates					
0.9	52.3	36.7	36	29.2	27.3
0.99	63.6	38.9	37.8	30.3	28.1
0.99	210.7	48.9	46.2	35.2	31.9
% Improvement in accuracy compared to Chronos					
0.99	32.94	46.1	47	50.5	52.1

level of refinement is defined as that point when any further refinement does not improve the WCET accuracy. We saw about the maximum levels of refinement for all program phases in Figure 6.17. Table 6.4 describes the initial total number of samples per unrefined phase and the average number of samples per sub-phase of the maximum refined level.

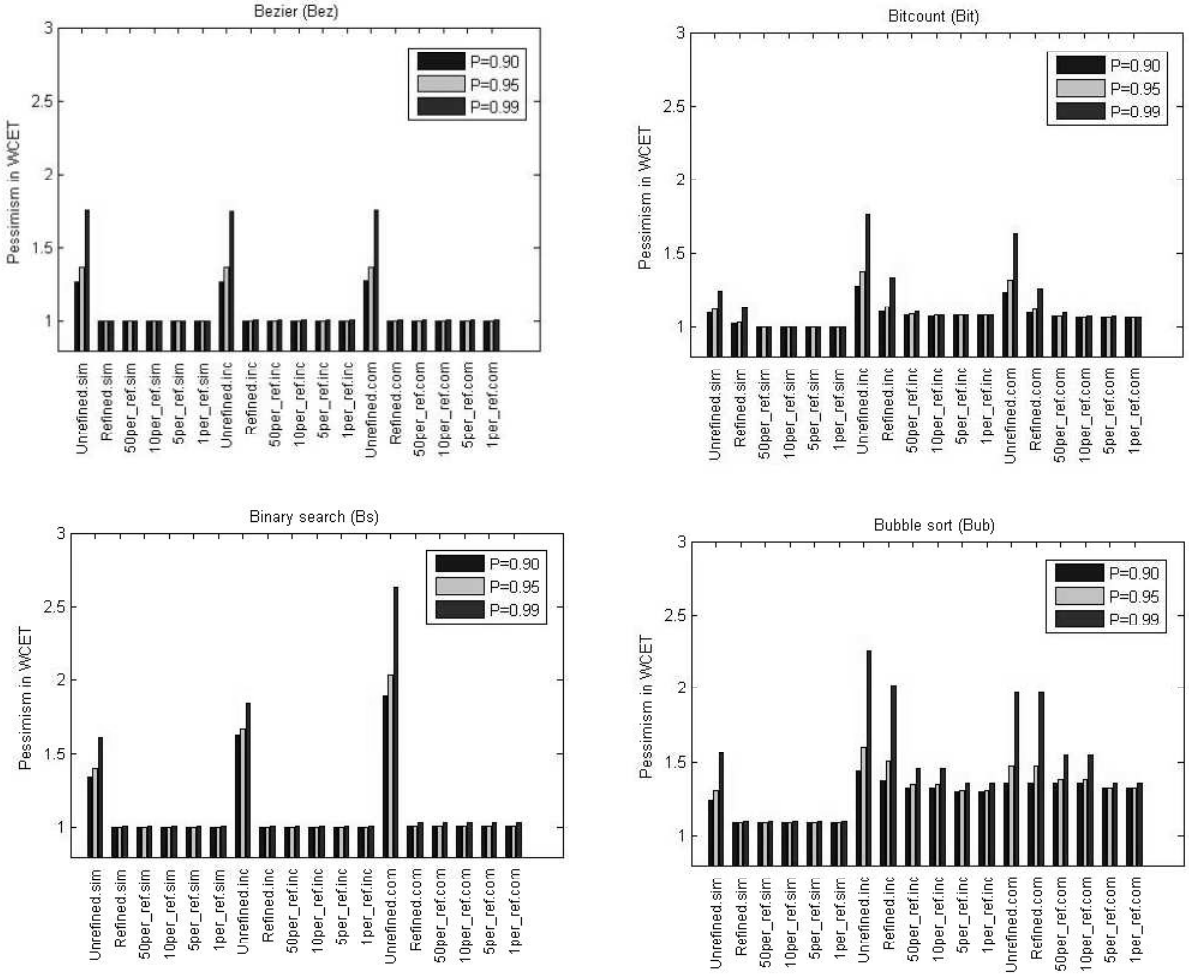


Figure 6.13: Pessimism in WCET estimate for all three probabilities on all PISA architectures for *Bezier*, *Bitcount*, *Bs* and *Bub*.

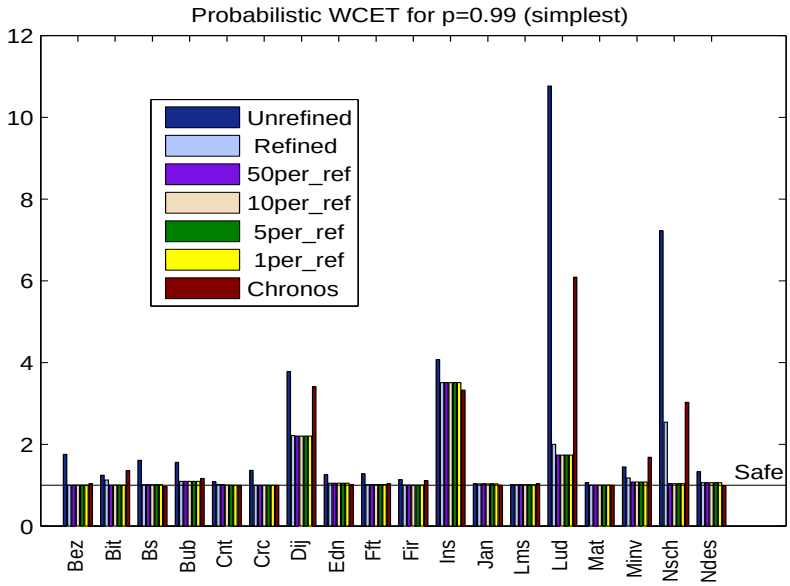


Figure 6.14: Comparison of Probabilistic WCET at p=0.99 with the corresponding estimate by *Chronos* on *Simplest* architecture.

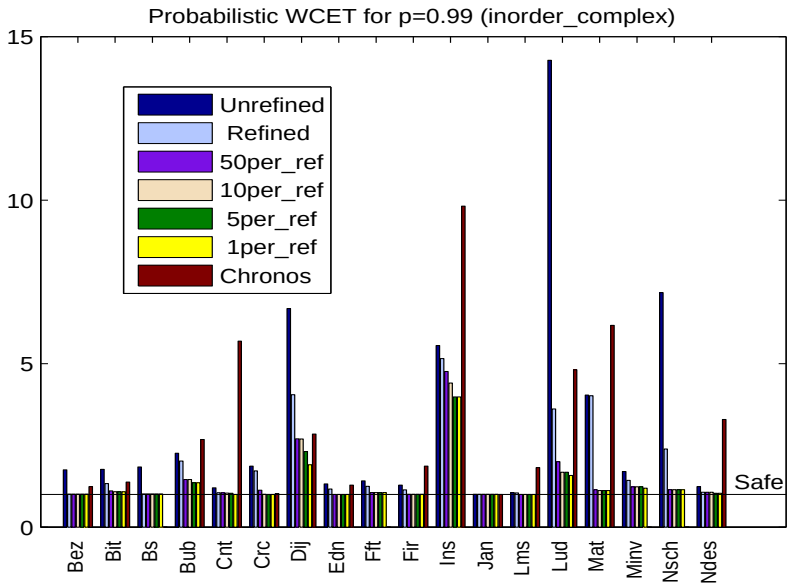


Figure 6.15: Comparison of Probabilistic WCET at p=0.99 with the corresponding estimate by *Chronos* on *Inorder_complex* architecture.

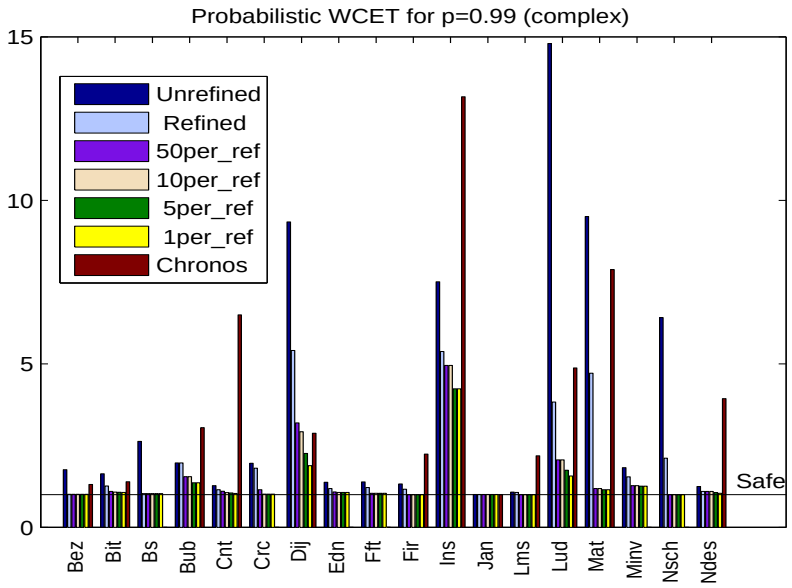


Figure 6.16: Comparison of Probabilistic WCET at p=0.99 with the corresponding estimate by *Chronos* on *Complex* architecture.

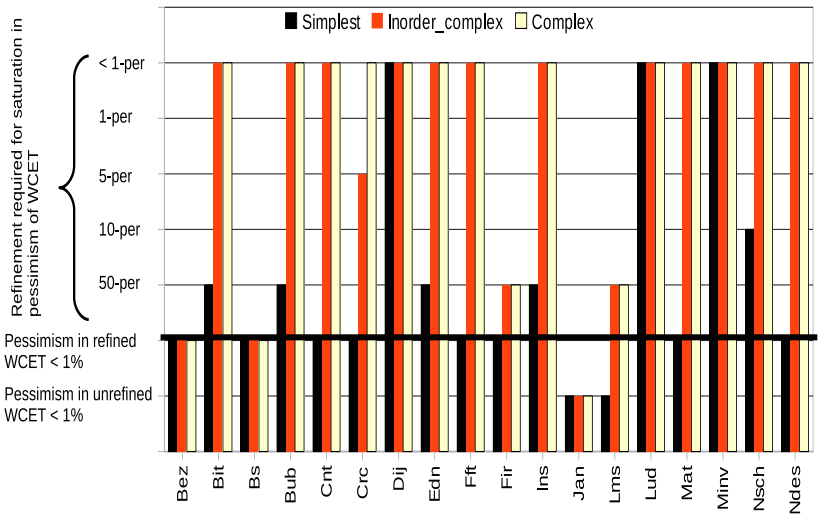


Figure 6.17: Amount of refinement required to reach zero variance in CPI within a sub-phase on all PISA architectures.

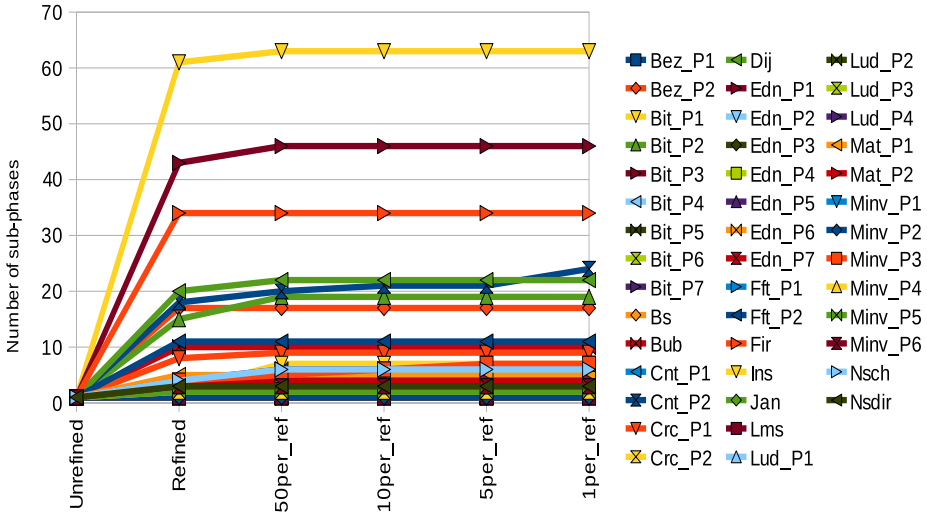


Figure 6.18: Impact of refinement on number of sub-phases on *Simplest* architecture.

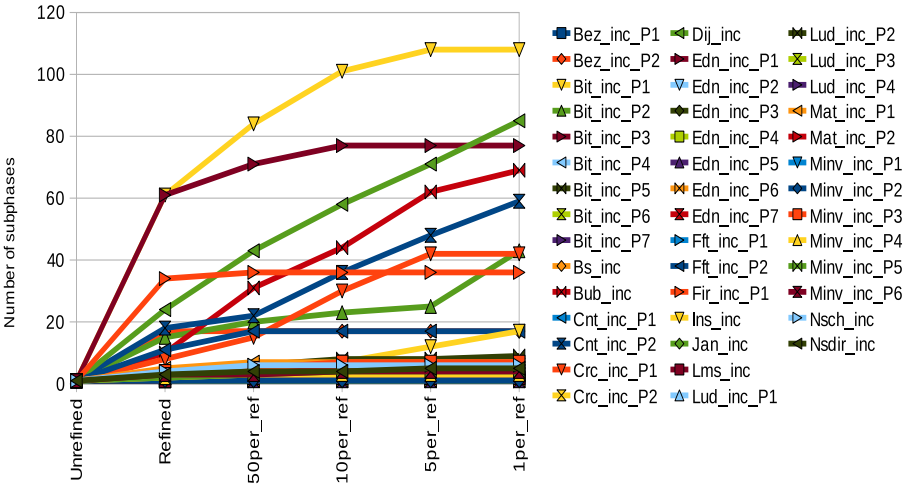


Figure 6.19: Impact of refinement on number of sub-phases on *Inorder_complex* architecture.

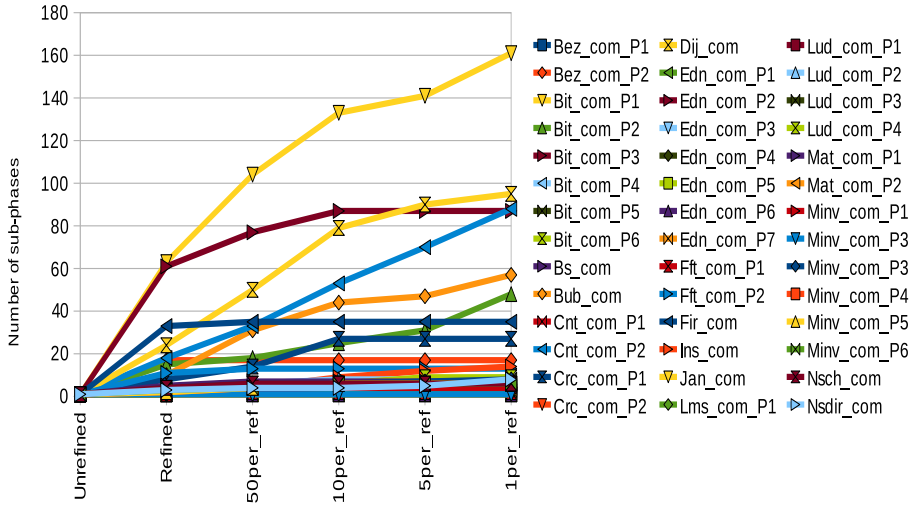


Figure 6.20: Impact of refinement on number of sub-phases on *Complex* architecture.

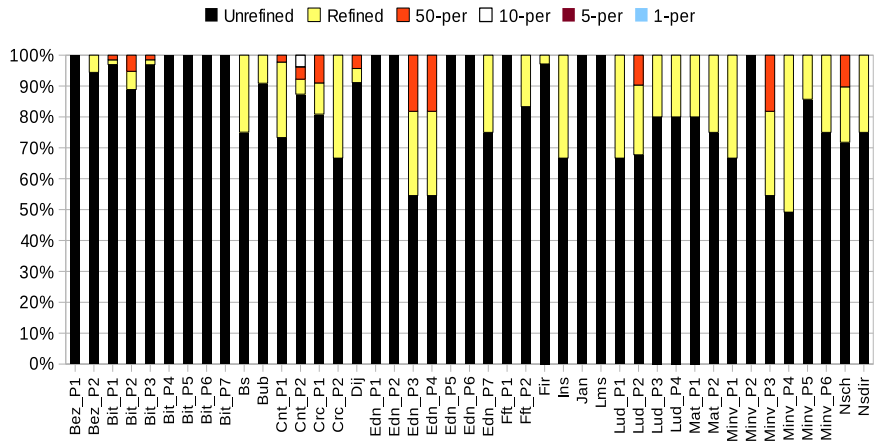


Figure 6.21: Impact of refinement on number of samples on *Simplest* architecture.

Table 6.3: Average trace size across inputs before and after compression.

Phase	**Trace size before compression**			**Trace size after compression**			**Compression Factor**		
	Sim	*Inc*	*Com*	*Sim*	*Inc*	*Com*	*Sim*	*Inc*	*Com*
Bez_P1	1.6M	1.6M	1.6M	68K	88K	88K	24:1	19:1	19:1
Bez_P2	52M	52M	52M	304K	471K	444K	175:1	116:1	120:1
Bit_P1	20K	20K	20K	16K	16K	16K	1.25:1	1.25:1	1.25:1
Bit_P2	24K	24K	24K	12K	16K	16K	2:1	1.5:1	1.5:1
Bit_P3	24K	24K	24K	16K	16K	16K	1.5:1	1.5:1	1.5:1
Bit_P4	24K	24K	24K	4K	4K	4K	6:1	6:1	6:1
Bit_P5	28K	28K	28K	4K	4K	4K	7:1	7:1	7:1
Bit_P6	24K	24K	24K	4K	4K	4K	6:1	6:1	6:1
Bit_P7	24K	24K	24K	4K	4K	4K	6:1	6:1	6:1
Bs	4K	4K	4K	4K	4K	4K	1:1	1:1	1:1
Bub	40M	42M	42M	8.8M	32M	32M	4.6:1	1.3:1	1.3:1
Cnt_P1	160K	160K	160K	20K	24K	24K	8:1	6.7:1	6.7:1
Cnt_P2	192K	192K	192K	20	28K	28K	9.6:1	6.8:1	6.8:1
Crc_P1	16K	16K	16K	12K	16K	16K	1.3:1	1:1	1:1
Crc_P2	364K	364K	364K	4K	364K	364K	91:1	1:1	1:1
Dij	2.4M	2.9M	1.4M	1.7M	2.6M	1.4M	1.4:1	1.14:1	1:1
Edn_P1	4K	4K	4K	4K	4K	4K	1:1	1:1	1:1
Edn_P2	4K	4K	4K	4K	4K	4K	1:1	1:1	1:1
Edn_P3	28K	28K	28K	8K	8K	8K	3.5:1	3.5:1	3.5:1
Edn_P4	20K	20K	20K	8K	8K	12K	2.5:1	2.5:1	1.6:1
Edn_P5	4K	4K	4K	4K	4K	4K	1:1	1:1	1:1
Edn_P6	4K	4K	4K	4K	4K	4K	1:1	1:1	1:1
Edn_P7	4K	4K	4K	4K	4K	4K	1:1	1:1	1:1
Fft_P1	788K	788K	788K	4K	788K	788K	197:1	1:1	1:1
Fft_P2	2.7M	3M	3M	428K	652K	548K	6.4:1	4.7:1	5.6:1
Fir	20K	20K	20K	4K	20K	20K	5:1	1:1	1:1
Ins	23M	22M	25M	284K	420K	416K	82.9:1	53.6:1	61.5:1
Jan	1.1M	1.1M	1.1M	4K	4K	4K	281.6:1	281.6:1	281.6:1
Lms	60K	56K	56K	4K	56K	56K	15:1	1:1	1:1
Lud_P1	172K	172K	172K	20K	20K	20K	8.6:1	8.6:1	8.6:1
Lud_P2	12K	12K	12K	12K	12K	12K	1:1	1:1	1:1
Lud_P3	212K	212K	212K	20K	24K	24K	10.6:1	8.3:1	8.3:1
Lud_P4	208K	208K	208K	20K	24K	24K	10.4:1	8.7:1	8.7:1
Mat_P1	696K	696K	696K	36K	48K	48K	19.3:1	14.5:1	14.5:1
Mat_P2	83M	83M	83M	3.6M	5.2M	5.2M	23.6:1	16.3:1	16.3:1
Minv_P1	180K	180K	180K	20K	24K	24K	9:1	7.5:1	7.5:1
Minv_P2	4K	4K	4K	4K	4K	4K	1:1	1:1	1:1
Minv_P3	12K	8K	12K	12K	8K	12K	1:1	1:1	1:1
Minv_P4	12K	12K	12K	12K	12K	12K	1:1	1:1	1:1
Minv_P5	180K	180K	180K	20K	28K	24K	9:1	6.4:1	7.5:1
Minv_P6	79M	79M	79M	3.5M	5M	5M	23.1:1	16.1:1	16.1:1
Nsch	4K	4K	4K	4K	4K	4K	1:1	1:1	1:1
Ndes	120K	120K	120K	76K	76K	76K	1.6:1	1.6:1	1.6:1

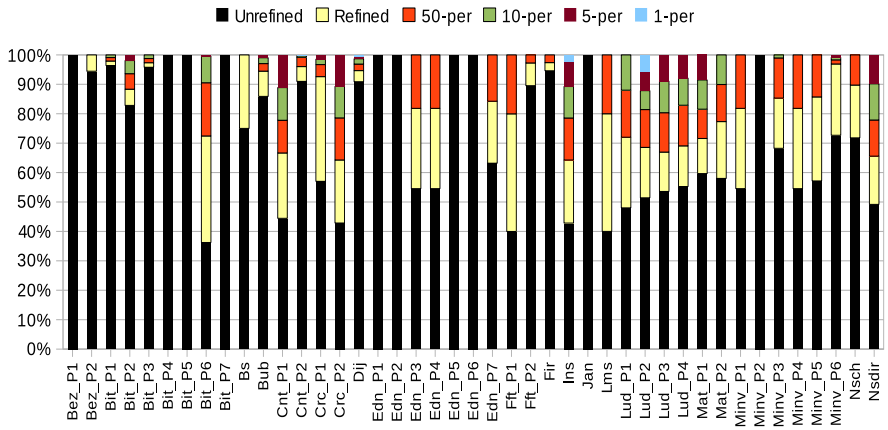


Figure 6.22: Impact of refinement on number of samples on *Inorder_complex* architecture.

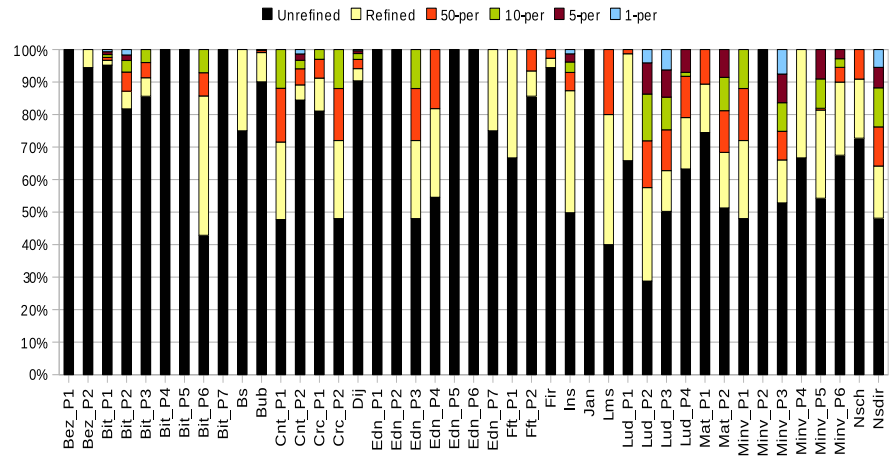


Figure 6.23: Impact of refinement on number of samples on *Complex* architecture.

Table 6.4: Initial number of samples and number of inputs prior to refinement.

Phase	**Number of Inputs**	**Number of samples prior to refinement**	**Number of samples post maximum refinement**		
			Simplest	*Inorder_complex*	*Complex*
Bez_P1	500	2.9e+7	2.9e+7	2.9e+7	2.9e+7
Bez_P2	500	5.01e+8	2.9e+7	2.9e+7	2.9e+7
Bit_P1	10500	5.2e+6	8.1e+4	4.8e+4	3.6e+4
Bit_P2	10500	5.2e+6	3.1e+5	1.2e+5	1.07e+5
Bit_P3	10500	5.2e+6	8.1e+4	6.8e+4	2.4e+5
Bit_P4	10500	5.2e+6	5.2e+6	5.2e+6	5.2e+6
Bit_P5	10500	5.2e+6	5.2e+6	5.2e+6	5.2e+6
Bit_P6	10500	5.2e+6	5.2e+6	6.2e+4	8.7e+5
Bit_P7	10500	5.2e+6	5.2e+6	5.2e+6	5.2e+6
Bs	10500	1.5e+5	5.01e+4	3.5e+4	3.5e+4
Bub	500	5.4e+8	5.4e+7	5.3e+6	1.2e+5
Cnt_P1	10500	5.4e+7	1.6e+6	1.09e+7	1.3e+7
Cnt_P2	10500	5.2e+7	2.2e+6	8.3e+4	8.4e+5
Crc_P1	10500	2.6e+6	2.9e+5	6.7e+4	9.8e+4
Crc_P2	10500	4.1e+6	2.04e+6	1.02e+6	1.02e+6
Dij	500	1.6e+7	6.9e+5	8.3e+4	7.3e+4
Edn_P1	10500	3.8e+5	3.8e+5	3.8e+5	3.8e+5
Edn_P2	10500	3.8e+5	3.8e+5	3.8e+5	3.8e+5
Edn_P3	10500	6.5e+5	2.1e+6	2.1e+6	1.6e+6
Edn_P4	10500	7.9e+5	2.6e+5	2.6e+5	2.6e+5
Edn_P5	10500	3.1e+5	3.1e+5	3.1e+5	3.1e+5
Edn_P6	10500	3.1e+5	3.1e+5	3.1e+5	3.1e+5
Edn_P7	10500	3.1e+5	1.03e+5	7.7e+4	1.01e+5
Fft_P1	500	8.1e+6	8.1e+6	4.08e+6	4.08e+6
Fft_P2	500	2.8e+7	5.7e+6	8.01e+5	2.2e+6
Fir	2000	7.9e+5	2.3e+4	2.3e+4	2.2e+4
Ins	500	4.1e+6	1.9e+6	2.12e+5	1.12e+5
Jan	500	1.2e+7	1.2e+7	1.2e+7	1.2e+7
Lms	600	4.9e+5	4.9e+5	2.5e+5	2.5e+5
Lud_P1	3000	2.5e+6	1.2e+6	6.3e+5	4.9e+4
Lud_P2	3000	6.09e+5	8.7e+4	6.7e+4	8.6e+4
Lud_P3	3000	1.1e+7	2.5e+6	1.6e+6	1.2e+6
Lud_P4	3000	1.8e+6	4.6e+5	2.6e+5	2.06e+5
Mat_P1	500	1.01e+7	2.5e+6	1.4e+6	1.4e+6
Mat_P2	500	7.4e+7	2.4e+7	1.3e+7	1.02e+7
Minv_P1	10500	6.2e+7	3.1e+7	2.1e+7	1.6e+6
Minv_P2	10500	2.8e+6	2.8e+6	2.8e+6	2.8e+6
Minv_P3	10500	2.4e+6	8.02e+5	3.7e+4	1.3e+4
Minv_P4	10500	1.01e+6	5.4e+5	4.4e+5	6.3e+5
Minv_P5	10500	7.2e+7	1.2e+7	1.8e+7	1.2e+7
Minv_P6	10500	9.6e+8	3.2e+8	5.5e+6	1.2e+6
Nsch	4302	1.3e+5	1.9e+4	1.8e+4	1.6e+4
Ndes	500	9.1e+5	3.02e+5	1.8e+5	1.03e+5

6.6 Comparison with RapiTime

In this section, we shall compare our phase based technique with the commercial measurements based WCET analyzer, *RapiTime*[102] Since both *RapiTime* and our techniques are measurement based, the comparisons are made with respect to various parameters such as accuracy of WCET estimate, number of instrumentation points and the time taken for estimating WCET. *RapiTime* is configurable to use an ARM simulator (Simit-ARM-2.1) underneath to generate time measurements. For our experiments we use the same simulator to generate CPI samples. We modify the simulator as described in section 6.4.1 to generate PC signatures during execution of the program. The architectural configuration used in case of the ARM simulator is described in Table 6.5. In the case of *RapiTime*, measurements may be taken by instrumenting the program at a very fine level such as the basic block level, referred to as FULL instrumentation. Alternatively measurements may also be taken at a much coarser level referred to as START_OF_SCOPES. *RapiTime* also supports taking measurements at the level of functions but these measurements are observed to result in inaccurate WCET estimates. Hence we use only FULL and START_OF_SCOPES instrumentation levels for comparison. More particulars regarding usage aspects of *RapiTime* can be found in the Appendix.

The FULL instrumentation level is finer than START_OF_SCOPES instrumentation level and is observed to result in more accurate WCET estimates as we shall see. Inorder to make a fair comparison with *RapiTime* on two instrumentation levels- fine(FULL) and coarse (START_OF_SCOPES), we likewise obtain measurements of CPI samples at two window sizes- $\mathsf{w1}$ and $\mathsf{w2}$, such that $\mathsf{w2} = 2 \times \mathsf{w1}$ and compare estimates obtained using $\mathsf{w1}$ with FULL and estimates obtained using $\mathsf{w2}$ with START_OF_SCOPES. Since *RapiTime* also generates probabilistic estimates, we generate estimates for *RapiTime* and our technique using the same probability values p={0.9, 0.95 and 0.99} and compare the estimates.

6.6.1 Accuracy of WCET

We begin with comparing the accuracies of WCET obtained by phase based technique and

SimIt-ARM-2.1	L1 16KB 32-way set associative Instruction cache, 8KB 32-way set associative data cache

Table 6.5: Architectural configurations used for experimentation.

RapiTime. The higher the pessimism in the WCET estimate (difference between WCET estimate and maximum observed cycles), the lower is the accuracy of the WCET estimate. Unsafe estimates that are lesser than observed maximum cycles are considered to be inaccurate. Figures 6.24 to 6.28 plot the pessimism of WCET using phase based technique at two window sizes w1, w2 and *RapiTime* at two instrumentation levels- START_OF_SCOPES and FULL.

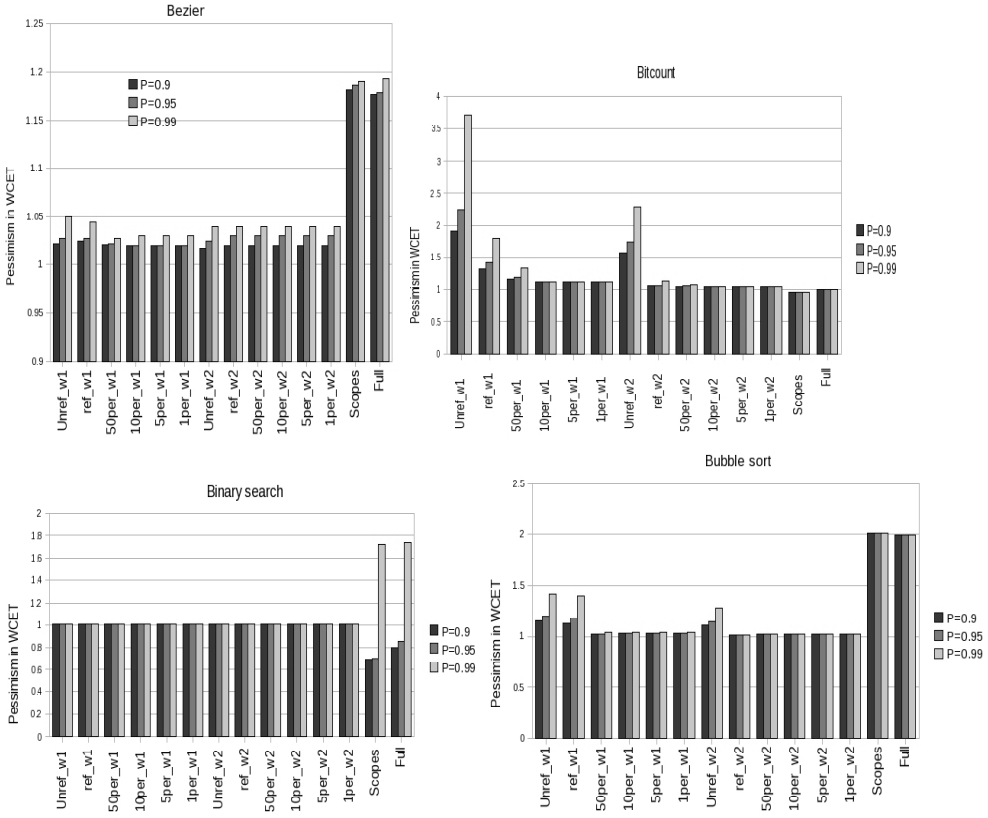


Figure 6.24: Comparison of pessimism in WCET estimate using *RapiTime* and phase based technique for *Bezier*,*Bitcount*, *Bs* and *Bubble sort*.

From Figure 6.24, it can be seen that *RapiTime* estimates WCET with greater accuracy for *Bitcount* compared to the phase based technique. The reason is that *Bitcount* is composed of various phases, some of which have very high variation in CPI. Hence unrefined estimates and refined estimates obtained on basis of signature are more pessimistic than the *RapiTime* counterparts. However, with additional refinement based on CPI values, they come very close to each other. Since *Bezier* exhibits a stable CPI variation, the phase based WCET analyzer

performs better than *RapiTime*. It can be observed from figure 6.6.1, that probabilistic WCET estimate obtained using *RapiTime*, at p={0.9, 0.95} is unsafe. At p=0.99, the pessimism jumps to 1.7. The corresponding probabilistic WCET estimates given out by the phase based technique is much more uniform as it is determined by CPI variation which has a lower variance compared to variance in execution count frequencies that is incorporated by *RapiTime* in its probabilistic model.

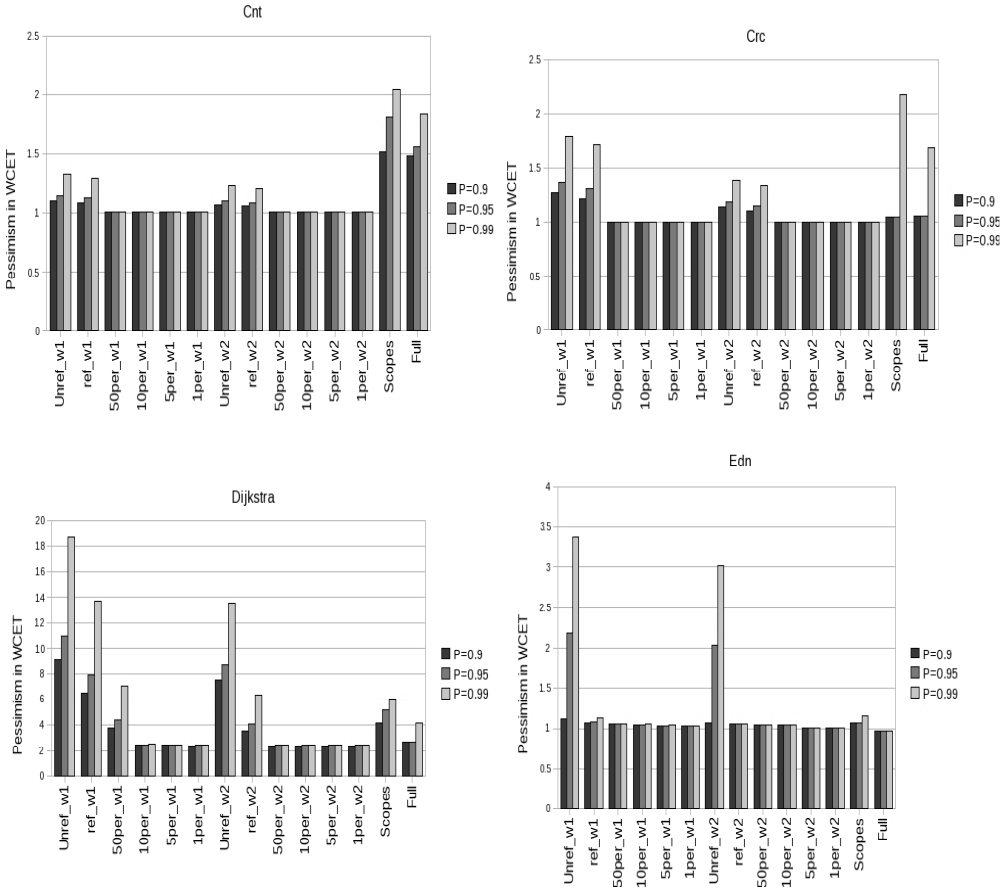


Figure 6.25: Comparison of pessimism in WCET estimate using *RapiTime* and phase based technique for *Cnt*,*Crc*, *Dij* and *Edn*.

For most programs, refinement based on limiting CPI variance to 50% of CPI variance of sub-phases obtained by refinement based on PC signatures is observed to be either close to or

more accurate than *RapiTime*. *Dijkstra* is an exception as can be observed from Figure 6.25, which requires higher levels of refinement to match the accuracy obtained using *RapiTime*. It can also be observed that estimates obtained using a higher window size (w2) saturate quickly compared to those obtained using w1. The reason is that as instrumentation granularity increases in the phase based technique, CPI variation becomes a lot more smoother[41]. In figure 6.26, we observe that the pessimism in WCET estimation by *RapiTime* for *Janne_complex* is highly pessimistic. *Janne_complex* has a two level nested loop and the entry to the inner loop is controlled by value of data variables that are modified within the program. By running the program with an exhaustive set of inputs it is observed that the inner loop is very sparsely visited. At present there is no way of annotating this information in *RapiTime*. A new feature in *RapiTime* which can convey this information as an annotation will greatly improve the pessimism in WCET. For this reason, we do not include Janne_complex in our comparison of average pessimism in WCET estimate over all benchmarks seen in *RapiTime* versus the phase based WCET analyzer.

Figure 6.29 depicts the average pessimism in WCET estimated over all benchmarks using *RapiTime* and phase based WCET analyzer using START_OF_SCOPES instrumentation and window size w2 respectively. At this instrumentation level, using even unrefined estimates show improvement over *RapiTime*. At p=0.99, WCET estimated using unrefined CPI is 7% better than *RapiTime*. Subsequent levels of refinement shows an improvement of 36.6%, 49.5%, 50.8%, 51.4% and 51.4% over *RapiTime*. The reason is START_OF_SCOPES instrumentation level yields very pessimistic estimates for certain benchmarks as it is a coarse level of instrumentation.

Figure 6.30 depicts the average pessimism in WCET estimated over all benchmarks using *RapiTime* and phase based WCET analyzer using FULL instrumentation and window size w1 respectively. From the figure it can be observed that, *RapiTime* estimates using FULL instrumentation is much more accurate than START_OF_SCOPES instrumentation. At p=0.9 and 0.95, estimates obtained by refinement based on signature are more accurate than *RapiTime*. At p=0.99, estimates obtained using refinement based on signature lag behind RapiTime by 10.6%. However further refinement based on CPI variance limiting COV of CPI to 50% and 10% of original CPI variance improves pessimism by 18% and 32% compared to *RapiTime*. Improvement on further refinement is marginal and is more pronounced at p=0.99 than at lower probability values. While there is a large disparity between the estimates made by *RapiTime*

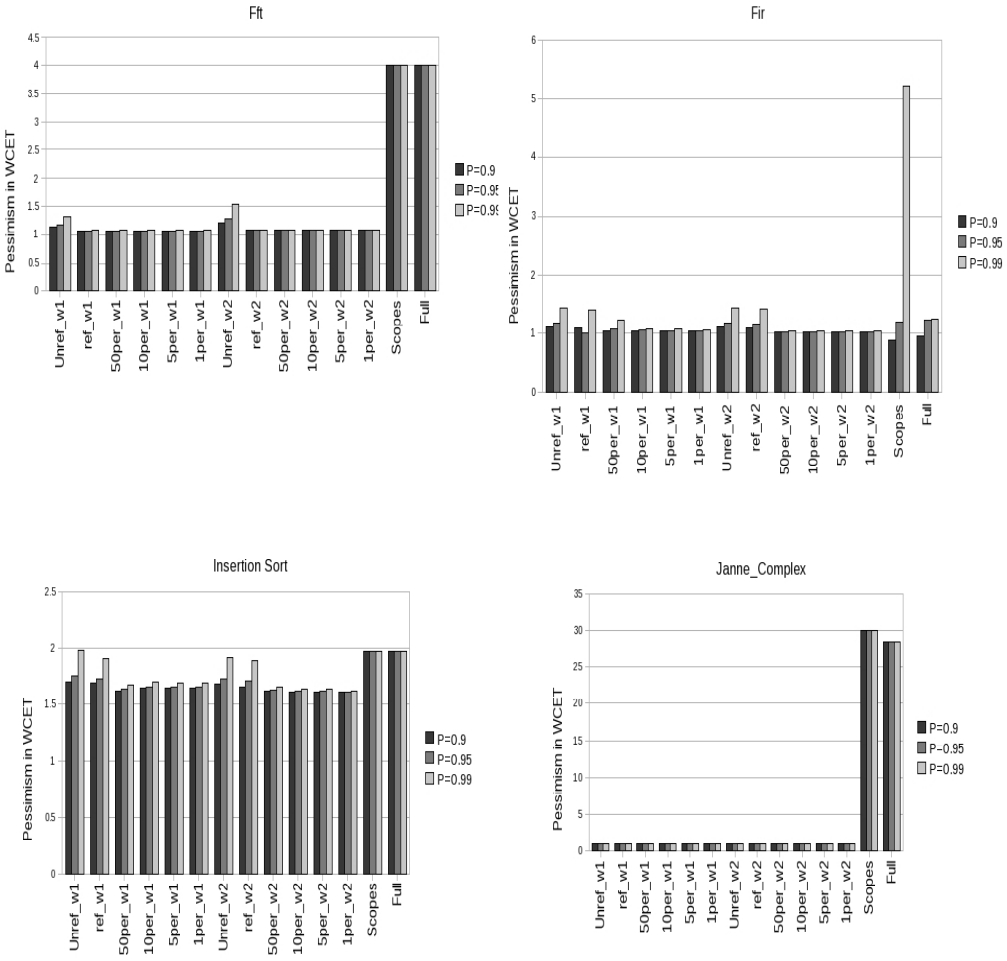


Figure 6.26: Comparison of pessimism in WCET estimate using *RapiTime* and phase based technique for *Fft*,*Fir*, *Insertion sort* and *Janne_complex*.

at two different instrumentation levels, the estimates are not largely different at the two window sizes used by the phase based WCET analyzer. The reason is that the repetition of CPI variation within a phase ensures the preservation of phase behavior at higher window sizes.

6.6.2 Number of Instrumentation Points

In this section, we shall compare the average number of instrumentation points used in *RapiTime* with FULL and START_OF_SCOPES instrumentation with w1 and w2 levels of instrumentation used in the phase based WCET analyzer taken across all programs. Figure 6.31 plots the

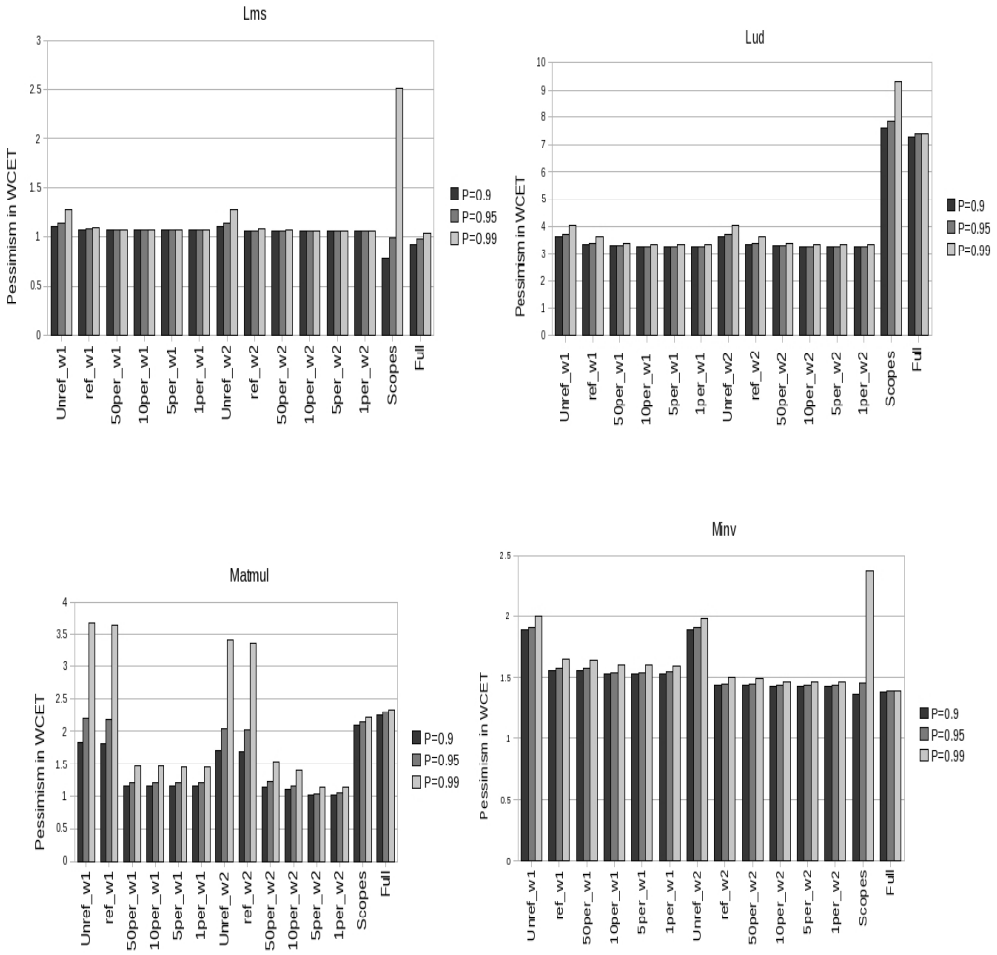


Figure 6.27: Comparison of pessimism in WCET estimate using *RapiTime* and phase based technique for *Lms*,*Lud*, *Matmul* and *Minv*.

number of instrumentation points used on an average for all configurations used as shown.

With a window size of w1, unrefined estimates require only 4.5% of the instrumentation points employed by RapiTime FULL instrumentation. With a window size of w1, refined estimates require 12% of the instrumentation points employed by Rapitime FULL instrumentation. With a window size of w2, unrefined estimates require only 5.22% of the instrumentation points employed by RapiTime SCOPES instrumentation. With a window size of w2, refined estimates require 10.3% of the instrumentation points employed by Rapitime START_OF_SCOPES instrumentation.

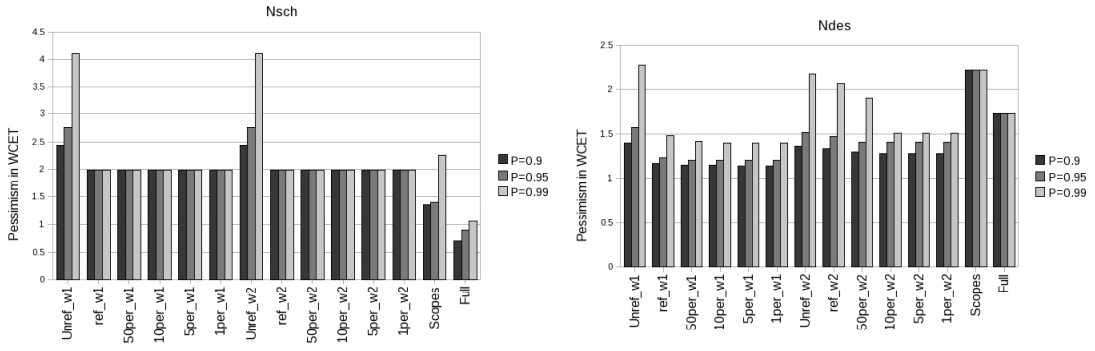


Figure 6.28: Comparison of pessimism in WCET estimate using *RapiTime* and phase based technique for *Nsch* and *Ndes*.

The phase based WCET analyzer uses the property of phase behavior and CPI homogeneity and can hence capture CPI information accurately by using only a small fraction of the instrumentation points otherwise required by conventional measurement based WCET analyzers like *RapiTime* and make accurate estimates that come very close to the accuracy achieved by *RapiTime* and in some programs that display stable CPI, even better. This way, program phase behavior helps build a non-intrusive WCET analyzer whose performance compares well with a commercial WCET analyzer such as *RapiTime*.

6.6.3 Time to Estimate WCET

Figure 6.32 compares the time taken by our phase based WCET analyzer at window size, w2 with the corresponding time taken by *RapiTime* at START_OF_SCOPES instrumentation granularity for all the programs. The time taken by the phase based technique is normalized with respect to the *RapiTime* counterpart as shown. Compared to START_OF_SCOPES, the time to analyze unrefined phases taken by the phase based WCET analyzer is on an average only 1/10 that of *RapiTime*. The amount of processing involved in the case of unrefined phases is minimal as only Chebyshev-cantelli bounds need to be computed. The analysis time using estimates obtained based on refinement with respect to signature is still about 3/4ths that of *RapiTime*. But additional levels of refinement based on CPI variance increases the analysis time further compared to *RapiTime*. Refinement to allow CPI variance of a sub-phase to 50% that of the original CPI variance leads to a 30% increase in analysis time compared to *RapiTime*.

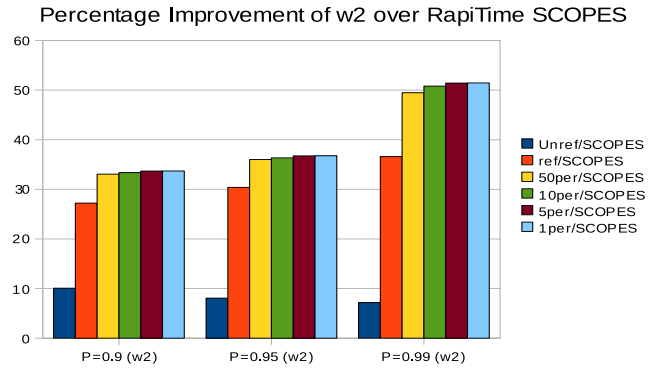


Figure 6.29: Average improvement in accuracy compared to RapiTime for both w2, w1 versus START_OF_SCOPES, FULL.

Further refinement beyond this point increases analysis time of the phase based method only marginally.

Figure 6.33 compares the time taken by our phase based WCET analyzer at window size, w1 with the corresponding time taken by *RapiTime* at FULL instrumentation granularity for all the programs. Similar to Figure 6.32, the time taken by the phase based technique is normalized with respect to the *RapiTime* counterpart. Compared to FULL, the analysis time using unrefined CPI taken by the phase based WCET analyzer is on an average about 1/7th that of *RapiTime*. The analysis time using estimates refined based on signature is still only about half of the time taken by *RapiTime*. Refinement to allow CPI variance of a sub-phase to 50% of that of the original CPI variance increases the analysis time, but which continues to be about 3/4ths of the time taken by *RapiTime*. Any further refinement beyond this point increases analysis time of phase based method only marginally. The reason of higher analysis time by *RapiTime* is due to large traces generated and the time spent in processing them.

6.6.4 Analysis Time versus Trace Size

Since both the phase based WCET analyzer and *RapiTime* are measurement based WCET analyzers, both produce traces when the program is instrumented and executed. Trace analysis time forms a major part of WCET analysis time. The most important factor that influences trace analysis time is the trace size. Figures 6.34 and 6.35 plot the growth of analysis time

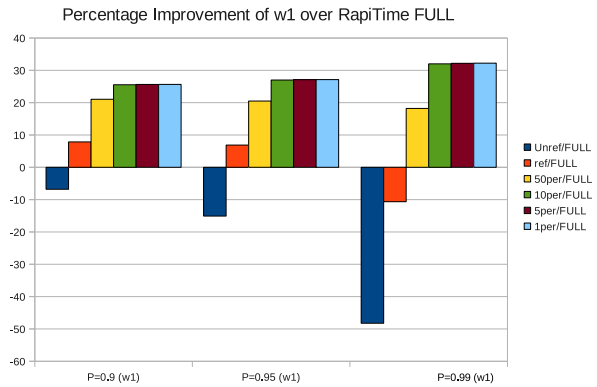


Figure 6.30: Average improvement in accuracy compared to RapiTime (w1 versus FULL.)

with respect to trace size for START_OF_SCOPES and FULL instrumentation respectively. The analysis time grows slowly in the case of START_OF_SCOPES compared to FULL instrumentation. The size of the trace generated by START_OF_SCOPES instrumentation is about half of the size of trace generated by FULL instrumentation. The analysis time is observed to be a direct function of uncompressed trace size in both cases.

Figures 6.36 and 6.37 plot the growth of analysis time using unrefined estimates with respect to trace size for window size w1 and w2 respectively. The left Y-axis plots both the number of instrumentation points and the trace size in bytes. The trace size generated using window size w2 is about half the size of the trace generated using w1. The analysis time in our case is not a direct function of the uncompressed trace size as it depends on several other factors such as computation of theoretical upper bound on IC and phase detection time. The growth of analysis time for both w1 and w2 are similar.

Figures 6.38 and 6.39 plot the growth of analysis time using estimates obtained using refinement based on PC signature and CPI variance for window sizes w1 and w2 respectively. The growth of analysis time is faster in case of refinement based on controlling CPI variance compared to refinement based on PC signature. Once again, the analysis time is not a direct function of trace size as analysis time is influenced by several other factors such as computation of theoretical upper bound on IC and phase detection time.

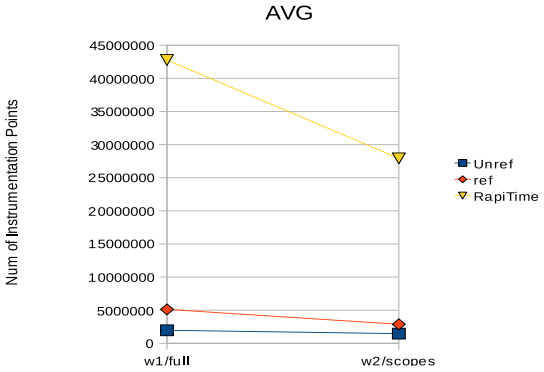


Figure 6.31: Average instrumentation points used in *RapiTime* and phase based WCET analyzer.

6.6.5 Scalability of Analysis Time

In the previous section, we looked at how analysis time grows with respect to trace size individually in the phase based WCET analyzer and *RapiTime*. In Figure 6.40, we plot the analysis times of both the tools together versus the corresponding trace sizes to see how the individual tools scale with increasing trace size. Since the analysis time is more dependent on the trace size than on the instrumentation mode, we include numbers from both the instrumentation granularities in this graph (FULL, START_OF_SCOPES, w1 and w2) in this graph.

Analysis time for unrefined estimates are too small to figure in this graph. Growth of analysis time taken of the phase based WCET analyzer are similar to *RapiTime* upto 420MB of trace. Beyond 420MB, analysis time by phase based WCET analyzer using refined estimates based on signature and CPI variance overtakes *RapiTime*. The highest analysis time is taken by *Dijkstra*- 40 minutes for refinement based on signature and 80 minutes for refinement based on controlling CPI variance (limiting COV of CPI of subphase to 50% of original value). In Chapter 8, we shall see that the analysis time can be reduced by parallelizing the process of WCET analysis itself. Since the result of analysis of each phase is independent of the other, they can be processed in parallel. This brings down the analysis time considerably.

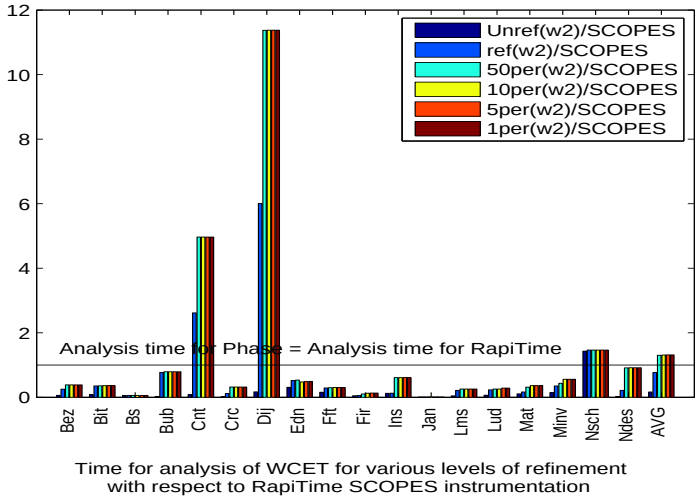


Figure 6.32: Comparison of WCET Analysis time using phase based technique(w2) versus *RapiTime*(START_OF_SCOPES).

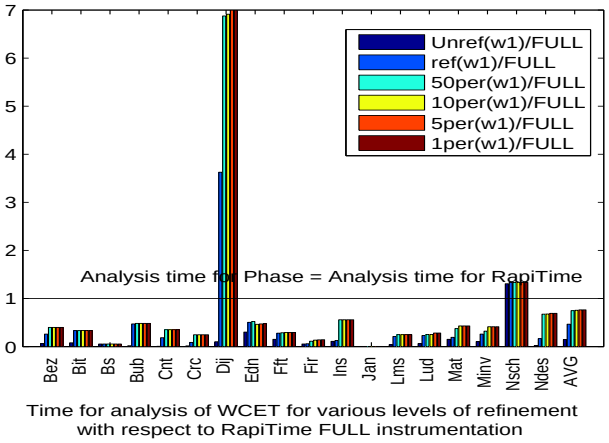


Figure 6.33: Comparison of WCET Analysis time using phase based technique(w1) versus *RapiTime*(FULL).

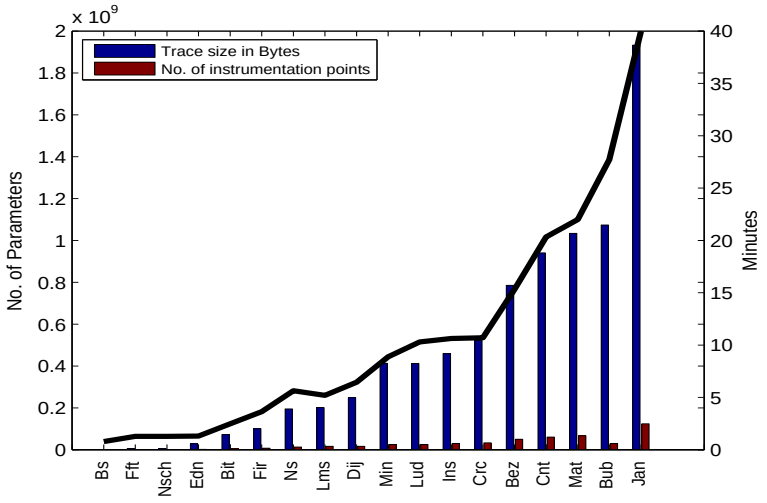


Figure 6.34: Growth of analysis time with trace size in case of *RapiTime*(START_OF_SCOPES).

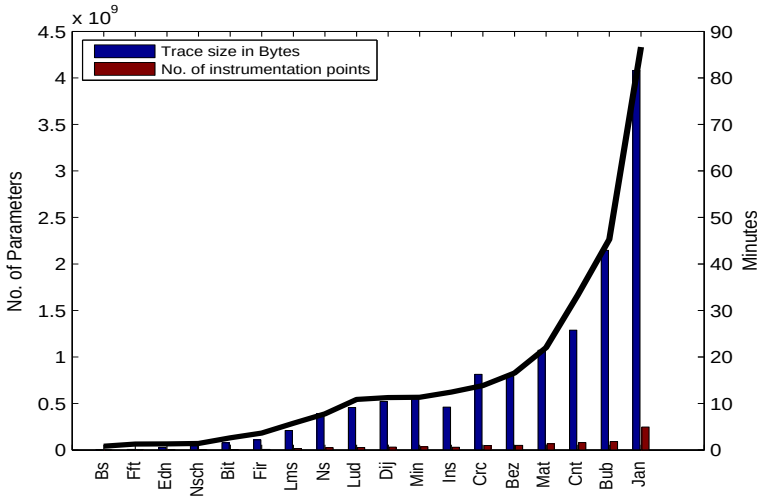


Figure 6.35: Growth of analysis time with trace size in case of *RapiTime*(FULL).

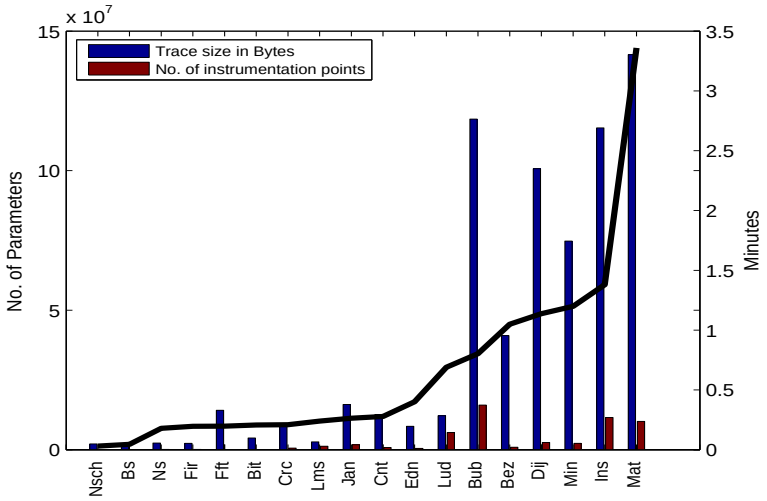


Figure 6.36: Growth of analysis time with trace size in case of phase based WCET analyzer using unrefined phase(w1).

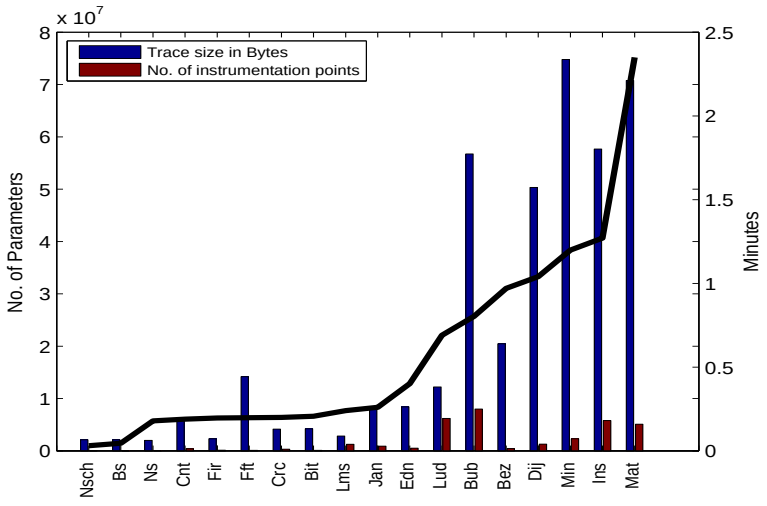


Figure 6.37: Growth of analysis time with trace size in case of phase based WCET analyzer using unrefined phase(w2).

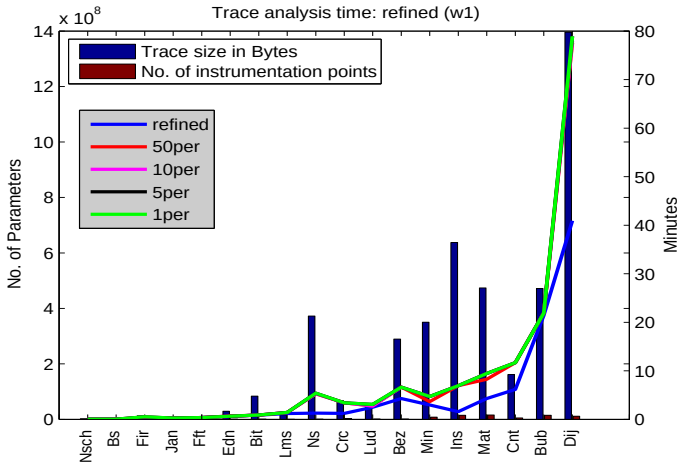


Figure 6.38: Growth of analysis time with trace size in case of phase based WCET analyzer using refined phase(w1).

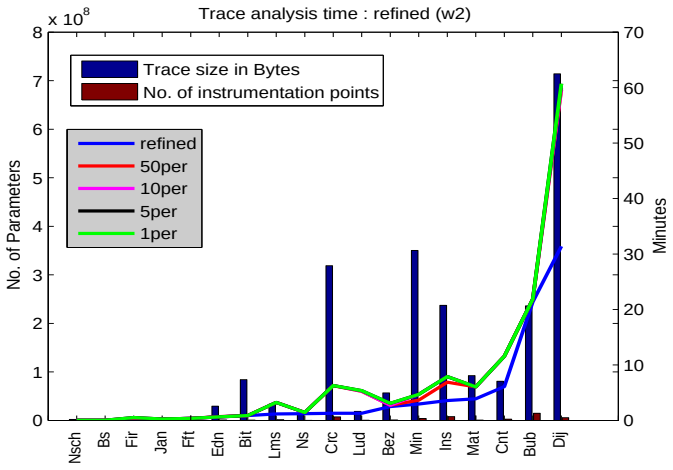


Figure 6.39: Growth of analysis time with trace size in case of phase based WCET analyzer using refined phase(w2).

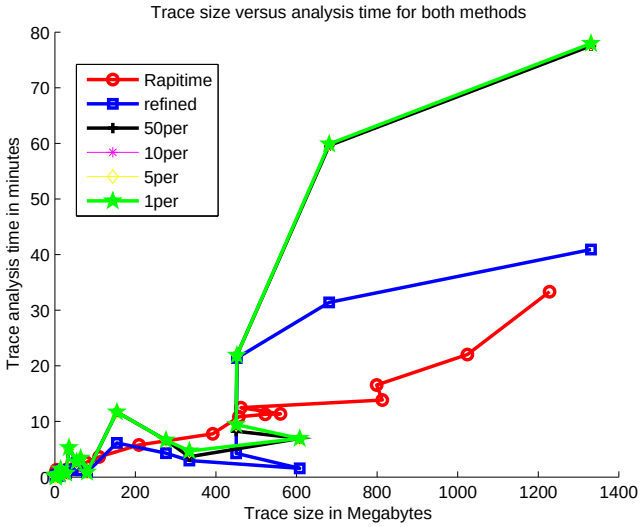


Figure 6.40: Scalability of analysis time with respect to trace size in case of phase based WCET analyzer and *RapiTime*.

6.7 WCET Analysis of DEBIE-1

All the results reported till now were based on WCET benchmarks created for purpose of evaluating WCET analyzers and embedded benchmark suites. In this section, we shall evaluate how our proposed WCET analysis technique performs with respect to a real-world benchmark, *DEBIE-1* (DEBris In orbit Evaluator)[106]. *DEBIE-1* is a standard instrument for monitoring space debris and meteoroids in near earth orbit. The onboard control software of DEBIE-1 was adapted to serve as an industry standard benchmark to evaluate several WCET analysis tools in the following way. DEBIE-1 was very hardware dependent in operation. It interacted with a specific multi-threaded real-time kernel and had specific peripherals and memory layouts. However, to let the program be tested on ordinary workstations during its development, the hardware and kernel dependencies were hidden by #defines and modules that formed a basic hardware abstraction layer (HAL) including also a kernel abstraction. The real DEBIE-1 benchmark was transformed into the *debie1* benchmark by removing all hardware and kernel specific components and adding a *harness* component. With much of the original code of DEBIE-1 being retained, the HAL was changed to connect to the harness instead of the real hardware and a real kernel. The software is written in C language. A copy of the software can be requested from Tidorum[107].

The control software of debie1 comprises of six tasks[103]. In the real system, these tasks are activated by interrupts and all of them have real-time deadlines. In debie1, the original behavior is simulated as debie1 is single threaded. In this thesis, we have applied our technique to estimate WCET for all of the six tasks. We have also conducted a similar experiment using *RapiTime* and the comparison of the two techniques are presented. These six tasks have featured in the WCET tool challenge[64], an event that happens once in every 3 years and brings together WCET tool developers and researchers all over the world[104]. Further information about the purpose of the tasks can be found in the WCET tool challenge wiki[104] and the DASIA'2000 paper on DEBIE-1[103].

6.7.1 Accuracy of WCET

debie1 Task 1

Each of the six tasks of debie1 are associated with a root function that is to be analyzed for

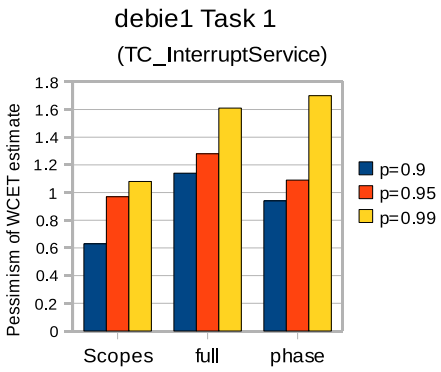


Figure 6.41: Comparison of Phase based technique and *RapiTime* for debie1 task 1.

WCET estimation. The root function of task 1 is *TC_InterruptService* which is the telecommand interrupt handler. Figure 6.41 describes the pessimism of WCET estimate obtained using *RapiTime* with START_OF_SCOPES and FULL instrumentation and the proposed phase based technique. *RapiTime* estimates with FULL instrumentation are more accurate and safer than those obtained with START_OF_SCOPES instrumentation. At p=0.9 and p=0.95, the pessimism reported by phase based technique is lesser by 8% and 19% than *RapiTime*. However, at p=0.99, the pessimism increases by 9% compared to *RapiTime*. The increased pessimism is due to the fact that there is high CPI variation while the code pertaining to task 1 is executed. Hence simple probabilistic inequalities like *Chebyshev* do not yield good results. The code base pertaining to task 1a is very small and does not exhibit phase behavior. Hence CPI variation cannot be refined by creating sub-phases out of phases as we saw earlier. An alternative way to reduce pessimism would be to gathering many more samples and use more complex probabilistic inequalities more in line with the true distribution of samples.

debie1 Task 2

The root function of task 2 is *TM_InterruptService* which is the telemetry interrupt handler. Figure 6.42 describes the pessimism of WCET estimate obtained using *RapiTime* with START_OF_SCOPES and FULL instrumentation and the proposed phase based technique.

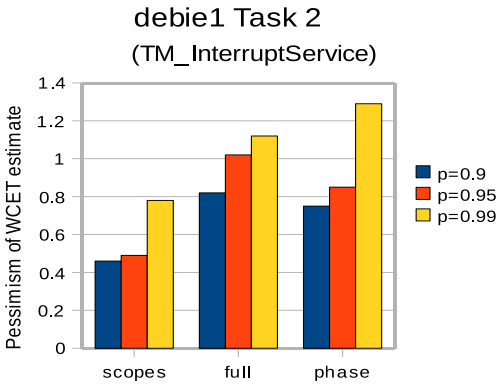


Figure 6.42: Comparison of Phase based technique and *RapiTime* for debie1 task 2.

RapiTime estimates with FULL instrumentation are more accurate and safer than those obtained with START_OF_SCOPES instrumentation. At p=0.99, the pessimism reported by the phase based technique is greater 17% compared to *RapiTime*. Task 1 and task 2 have very similar properties and hence are not good candidates for phase behavior.

debie1 Task 3

The root function of task 3 is *HandleHitTrigger* which refers to handling a situation when one of the four sensor units of the system registers a *hit* which is the impact of a particle of space debris or a natural speck of solid material on the sensitive foil on the exterior face of the sensor unit. Figure 6.43 describes the pessimism of WCET estimate obtained using *RapiTime* with START_OF_SCOPES and FULL instrumentation and the proposed phase based technique. It can be observed that the phase based technique reports very high pessimism compared to *RapiTime*. The increased pessimism arises out of two factors. The code base of task 3 is comprised of analog to digital conversion code, most of which comprise of polling loops that run until conversion is finished. Hence computation of theoretical upper bound on IC assumes that the loops iterate the maximum number of times, though in day to day operation this is not the case. The second reason for the pessimism is the high variation of CPI which can be refined as most of the code contains loops that exhibit phase behavior and we hence apply refinement based on signature and CPI variation. With maximum level of refinement, at

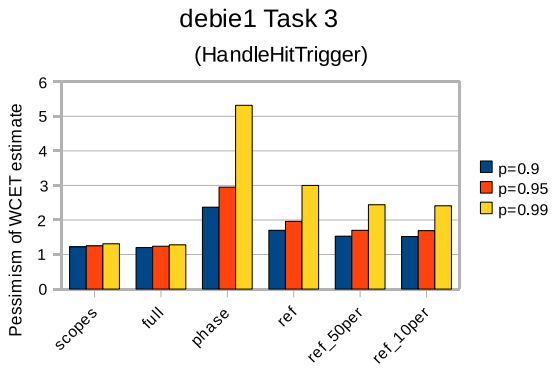


Figure 6.43: Comparison of Phase based technique and *RapiTime* for debie1 task 3.

p=0.99, the pessimism reported by the phase based technique is 89% higher than *RapiTime* with FULL instrumentation. Since *RapiTime* applies probabilistic functions on observed maximum frequency of basic blocks to compute the overall WCET estimate, it does not suffer from this added pessimism. By exercising the program with a test input set that gives 100% coverage, we could make our estimates better by using maximum observed instruction count instead of using theoretical upper bound on IC.

debie1 Task 4

The root function of task 4 is *HandleTelecommand* which refers to the execution of telecommands. Figure 6.44 describes the pessimism of WCET estimate obtained using *RapiTime* with START_OF_SCOPES and FULL instrumentation and the proposed phase based technique. At lower probability values, the results obtained using phase based technique with refined phases and *RapiTime* are similar. At p=0.99, the phase based technique reports 24% more pessimism than *RapiTime*. The root function for this task addresses execution of any telecommand in any state. As a result, computation of theoretical upper on IC considers all possible cases and ends up estimating a large value. Considering each type of telecommand separately can help in obtaining tighter bounds.

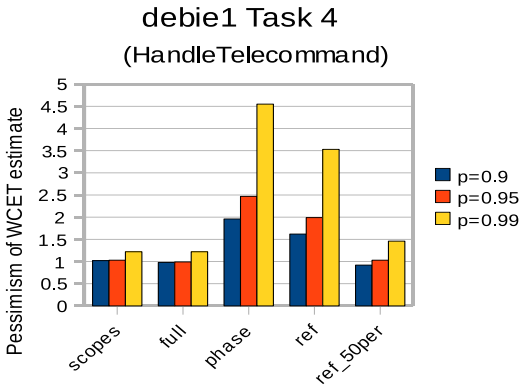


Figure 6.44: Comparison of Phase based technique and *RapiTime* for debie1 task 4.

debie1 Task 5

The root function of task 5 is *HandleAcquisition*. The acquisition task is responsible for reading the non-volatile (digital) sensor data from the sensor unit, evaluating the reality and quality of the hit event, and storing the event in the science data memory, an array of event records in data memory. The science data memory has a finite buffer space. Hence if it is full and cannot accomodate a new hit event, the acquisition task has to find the worst old event that can be evicted out to find place for the new event. This process is time consuming. Figure 6.45 describes the pessimism of WCET estimate obtained using *RapiTime* with START_OF_SCOPES and FULL instrumentation and the proposed phase based technique. Task 5 exhibits phase behavior that can be split into sub-phases based on PC signatures and CPI variation. Though the estimate obtained using phase based technique without any refinement shows more pessimism than *RapiTime*, estimate obtained by *RapiTime* with FULL instrumentation is twice as more pessimistic than the estimate obtained using phases with maximum level of refinement. Task 5 exhibits phase behavior that can be further refined for different paths by tracking their PC signatures. The CPI behavior is very regular and hence can be used to partition sub-phases further. As a result, the proposed technique works well for this task and is able to estimate WCET with high accuracy.

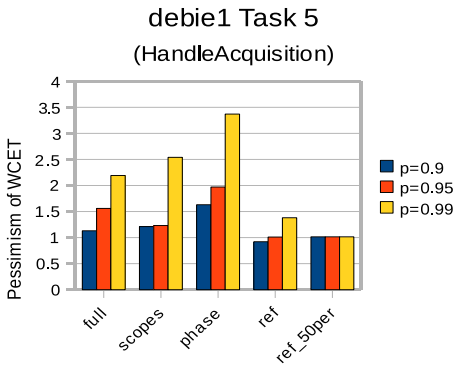


Figure 6.45: Comparison of Phase based technique and *RapiTime* for debie1 task 5.

debie1 Task 6

The root function of task 6 is *HandleHealthMonitoring* which is a house keeping task responsible for monitoring the health of the debie1 system. During normal operation, it is invoked periodically to monitor the state of various systems of debie1. The task follows a well defined sequence to conduct checks on the various components of debie1 as a result, many parts of the code base exhibit phase behavior. Figure 6.46 describes the pessimism of WCET estimate obtained using *RapiTime* with START_OF_SCOPES and FULL instrumentation and the proposed phase based technique. There is little difference between the estimate made by *RapiTime* at FULL instrumentation and the proposed technique using unrefined phases. However, using PC signatures, a significant portion of the code base can be split into sub-phases that show lesser CPI variation and thus brings down the pessimism by 37%.

6.7.2 Instrumentation Statistics

In this section, we shall compare the amount of instrumentation used by *RapiTime* and the proposed technique to estimate WCET. Figure 6.47 compares the relative proportion of instrumentation points used by *RapiTime* with FULL and START_OF_SCOPES instrumentation and the proposed phase based technique. The numbers to the right of the bars indicate the number of instrumentation points used in FULL instrumentation. On an average, phase behavior is observed to require only one third of the instrumentation points used in START_OF_SCOPES

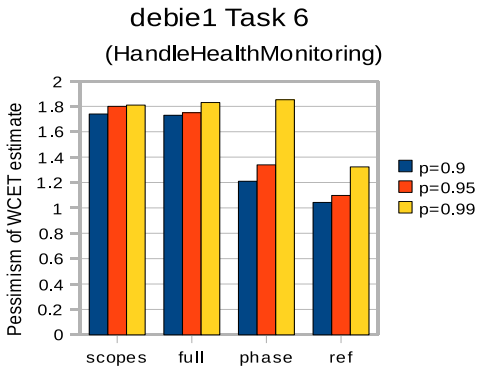


Figure 6.46: Comparison of Phase based technique and *RapiTime* for debie1 task 6.

instrumentation and about one fourth of the instrumentation points used in FULL instrumentation.

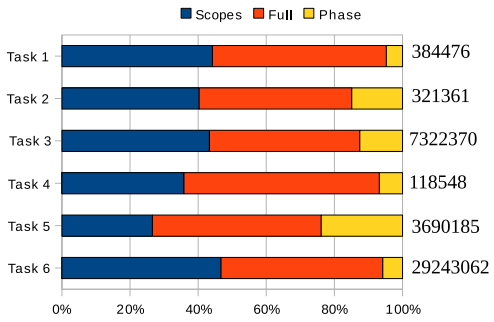


Figure 6.47: Comparison of number of instrumentation points for all tasks of debie1 used by *RapiTime* and the proposed technique.

Table 6.6 compares the uncompressed trace sizes generated by *RapiTime* with FULL and START_OF_SCOPES instrumentation and the phase based technique and also describes the number of inputs used to generate the trace. The trace generated by the proposed technique is much smaller. As we saw earlier, the uncompressed trace size is an important factor in determining the analysis time. As *debie1* comes with its own set of test harness that exercises most of the paths, we see each task is tested with a different number of inputs. It is important

to note that even with thousands of inputs, the trace generated by tasks 1, 2 and 4 indicate the small magnitude of its code base.

Table 6.6: Trace size in Megabytes and number of inputs.

Task number	Scopes	Full	Phase	Number of inputs
1	4.5	5.2	0.5	6580
2	4.5	5	1.6	2500
3	101	103	53	5380
4	1	1.6	0.3	3160
5	52	97	25	5262
6	427	434	152	36902

6.7.3 Analysis Time

Figure 6.48 compares the trace analysis time by *RapiTime* and the proposed technique. These times exclude the trace generation time as both take the same amount of time to generate the trace as it depends on the simulator speed. While we compared performance of *RapiTime* with our technique using other WCET benchmarks, we saw higher traces result in higher analysis time. The same is the case in *debie1*. In programs that run for a shorter amount of time, structural analysis takes a significant portion of analysis time of the proposed phase based technique. In tasks, where higher levels of refinement produce better accuracy, the analysis time overtakes *RapiTime* by a few percent. But overall, as the phase based technique produces lesser amount of trace, the analysis time is shorter than *RapiTime*.

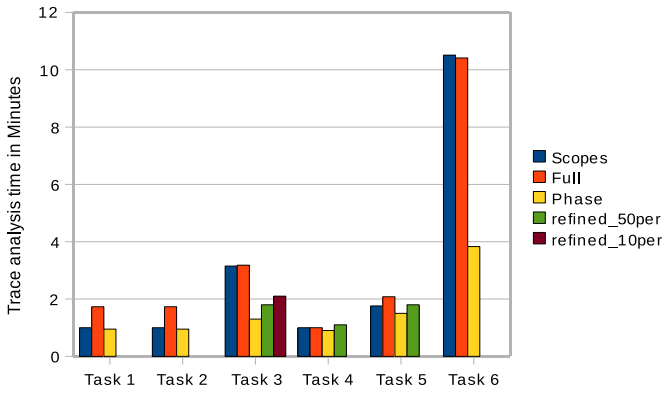


Figure 6.48: Comparison of analysis time of all tasks of debie1 by *RapiTime* and the proposed technique.

6.8 Related Work

6.8.1 Program Phase Behavior

Lau et al[41] observe that fixed length intervals can not find phase behavior especially if the period of the phase behavior is different from the fixed interval length and that there exists a hierarchy of phase behaviors and a phase can consist of smaller sub-phases. In order to identify variable length intervals, the code trace is analyzed for patterns. Traces of loops, calls, return traces are run with sequitur which finds variable length interval start and end points and also gives the hierarchy of phase behavior. We find phases by applying the code structure analysis algorithm[39] on a hieararchical call loop graph built with profiling. The edges of this graph store hieararchical information about the number of instructions executed across calls and loops and by looking at the variance in number of instructions executed, the algorithm selects edges that mark beginning of phases.

Annavaram et al[57] examine the use of code signatures obtained through periodic sampling to predict performance for data base applications and SPEC 2000. They use VTUNE[105] to sample hardware counters and PCs (sampled code signatures). The signature in [57] is referred as an extended instruction pointer (EIP) which is a bunch of unique PC addresses that are obtained by sampling and collecting performance data. EIP vectors are clustered and formed into phases. The objective of [57] is to predict accurate CPI using smaller number of samples.

In case of phase classification based on basic block vectors[28] every basic block executed is profiled whereas in [57], EIP profiling is done at a granularity of 100 million instructions. To quantify how accurately EIPvs can predict CPI, this work uses regression trees that can optimally subdivide EIPV space into groups such that CPIs of all EIPVs in that group have the theoretically smallest possible variance. Our work is different from [57] in many ways. Firstly our objective is timing analysis. Hence phases must be tied to code regions in our case. Hence we use code structure analysis[39], that builds a call loop graph annotated with profile information. In contrast to [57] that samples PCs every 100 million instructions, we record every PC into a bitmap for an interval consisting of a fixed group of loop iterations along with the number of instructions committed during that interval and the CPI during that interval. Phases are classified into sub-phases based on pairs of PC bitmap and IC values. The sub-phases are successively classified into smaller sub-phases with respect to CPI variation by partitioning samples into two categories with one category containing CPI samples that are lesser than the mean and the other category containing CPI samples that are greater than the mean.

Lau et al[40] show that the accuracy of EIP vectors identifying phases improves significantly if they are associated with loops and procedures and prove that there exists strong correlation between code and performance as CPI changes correspond correctly to code changes. Our work begins by identifying call loop boundaries and marking them as phases. The phases are then refined further based on code signatures and CPI variation.

6.8.2 Measurement Based WCET Analysis

Betts et al[3] observes that on the one hand, source-level instrumentation provides greater flexibility and is often the most convenient, but it is handicapped by the probe effect and on the other hand, less intrusive instrumentation normally demands some type of hardware support. Both problems have non trivial solutions. The first problem is addressed in [2] where instrumentation point graphs (IPGs) are constructed that model the information that flows between basic blocks. This is required in situations where there is availability of sparse instrumentation with only a few basic blocks containing instrumentation points. The trace is then parsed and using either an ITree which is based on tree based schema or IPET the final wcet is estimated. The second problem is addressed in [3] which describes how traces of only

branch instructions generated by typical hardware debug interfaces are analyzed to interpolate the missing instructions from the trace.

We stick to source code level instrumentation but use properties of program phase behavior to help us obtain accurate WCET estimates with sparse instrumentation. We describe a hybrid measurement based WCET analysis technique that is least intrusive and works with programs that exhibit phase behavior. The repetitive manner in which CPI varies in programs that exhibit phase behavior can be used to reduce instrumentation required in WCET analysis of such programs. We propose to instrument at the level of groups of loop iterations leading to a low instrumentation overhead of 1-2%. The number of iterations per group can be varied as per the requirement. Phases also help in compressing PC signatures considerably.

Bernat et al [29] measure execution time of basic blocks(execution time profiles or ETPs) and note their relative frequencies. The ETPs are convolved together to give probabilistic WCET estimates using three different scenarios- ETPs are mutually independent, ETPs are dependent, dependency is not known. The phase based timing model views execution time as a product of instruction count(IC) and CPI and estimates program WCET in terms of phases instead of blocks, instructions, segments or paths. We use probabilistic bounds on phase CPI to compute WCET of a phase.

Seshia et al[66] models the problem of estimating WCET as a game between the algorithm (player) and the environment of the program (adversary), where the player seeks to accurately predict the property of interest while the adversary sets environment states and parameters to thwart the player. Over several rounds, the player or algorithm learns enough about the environment to be able to accurately predict path lengths with high probability, where the probability increases with the number of rounds. To solve this problem, [66] employs a randomized strategy that repeatedly tests the program along a linear-sized set of program paths called basis paths, using the resulting measurements to infer a weighted-graph model of the environment, from which quantitative properties can be predicted. However the technique assumes that the timing of a program depends only on the control flow through that program. In general, the timing can also depend on characteristics of input data that do not influence control flow. Our technique works well in such situations as well, as the impact of all architectural components such as cache, branch predictors, pipelines et cetera is reflected in CPI. We build a phase based timing model that characterizes the timing of a phase by the average

phase CPI.

Edgar et al [71], Hansen et al [38] and Lu et al [97, 98] work with end to end program execution time samples and try to fit these samples into a Gumbel distribution using extreme value theory(EVT). Once the parameters of the distribution are computed, the estimate of WCET at various probabilities is available. Our work neither assumes any probability distribution of CPI samples nor tries to fit these samples into any distribution. We use Chebyshevs inequality that is applicable to any distribution, to compute bounds on CPI. The precision of our results will denitely improve if information regarding true probability distribution of CPI samples is available.

6.9 Conclusions

In the previous chapter, we saw that the homogeneous behavior of CPI within a phase can be used to reduce instrumentation in a measurement based WCET analyzer. The phase based timing model uses maximum of mean CPI observed across inputs to estimate WCET. However, WCET estimated thus is approximate and has no probabilistic guarantees. In this chapter, we introduce a way to obtain probabilistic bounds of phase CPI using the Chebyshev inequality. We assume no probability distribution of CPI samples and hence opt for Chebyshev inequality as it only requires samples to have finite mean and variance. The accuracy of CPI bound will certainly improve if the true probability distribution is known. Since we are interested in only the upper bound, we use a more optimal Chebyshev-cantelli inequality. If theoretical upper bound of instruction count (IC) is used in the timing equation, we obtain a probabilistic bound on program WCET using probabilistic bound of phase CPI. A probabilistic bound on WCET is more beneficial than absolute WCET as depending on the criticality of the application, WCET at the required probability can be derived.

Chebyshev-cantelli inequality works well with phases that exhibit low variance in CPI resulting in tight CPI bounds and accurate WCET estimates (Examples: Fir(Arch1), Jan(Arch2) and Lms(Arch1)). Some phases exhibit high variance in CPI. Applying Chebyshev inequality for such phases results in pessimistic WCET estimates (Mat(Arch2)). To isolate points of high variation in CPI, we refine such phases into smaller sub-phases based on PC signatures collected using profiling. We observe the following results for p=0.99. Refinement based on signatures

reduces average pessimism of WCET by 25%, 22% and 26% on *Simplest*, *Inorder_complex* and *complex* respectively. Refinement is designed to enable the user to control variance of CPI within a sub-phase, which is useful in programs like Bubble sort wherein CPI varies throughout program execution and points of high variation of CPI cannot be isolated based on PC signatures alone. We split a sub-phase into four levels (CPI variance within the subphase is limited to 50%, 10%, 5% and 1% of average CPI variance of the sub-phase obtained by refinement based on PC signature). Refining Bubble sort (Arch1) at these four levels reduces pessimism by 21%, 31%, 35% and 42% respectively.

The following improvements are with respect to Chronos at p=0.99. Average accuracy of WCET obtained by refinement based on signature improves by 9%, 23% and 33% on *simplest*, *Inorder_complex* and *complex* respectively. On *simplest*, average accuracy improves by refinement at the first level(50%) by 13%. The improvement is marginal beyond the first level. Average accuracy of WCET continues to improve following refinement at each of these four levels of CPI variance on *Inorder_complex* by 38%, 40%, 41.5% and 42.7%. Average accuracy of WCET continues to improve following refinement at each of these four levels of CPI variance on *complex* by 46%, 47%, 50.5% and 52.7%. The trace generated by the phase based WCET analyzer is highly compressible owing to the repetitive and homogeneous nature of phase behavior. The compression factor on *simplest*, where CPI variation is much more stable compared to other architectures, is 24.8%. On *Inorder_complex* and *complex*, where CPI variation is higher, the compression factor is 14.6% and 14.9% respectively.

The phase based WCET analyzer is also evaluated by comparing it with a commercial measurement based WCET analyzer, *RapiTime*. In the case of programs with high CPI variation, WCET estimated by using unrefined and refined estimates based on signature are more pessimistic than *RapiTime*. However further levels of refinement bring down the difference between the estimate made by the phase based WCET analyzer and *RapiTime*. In the case of programs with stable CPI, the estimates made by *RapiTime* are far more pessimistic than the phase based WCET analyzer. *RapiTime* gives an option to the user to instrument at two levels- START_OF_SCOPES and a much finer FULL level. Likewise, the phase based WCET analyzer is experimented with two window sizes w1 and w2 which is double the size of w1. At p=0.99, compared to estimates obtained by START_OF_SCOPES, the estimates made by the phase based WCET analyzer are more accurate by 7% on an average, even when using

unrefined CPI. Subsequent levels of refinement show an improvement of 36.6%, 49.5%, 50.8%, 51.4% and 51.4% over *RapiTime*.

RapiTime with FULL instrumentation gives much more accurate estimates than START_OF_SCOPES as it instruments the program at a higher number of points compared to START_OF_SCOPES. Estimates made by the phase based WCET analyzer using refinement based on signature are 10.6% more pessimistic on an average than *RapiTime*. However further levels of refinement make the estimates more accurate by 18.2%, 32%, 32.2% and 32.22% on an average over *RapiTime*. The phase based WCET analyzer makes use of the repetitive and homogeneous nature of CPI variation to reduce the instrumentation points compared to *RapiTime*. The average number of instrumentation points used by the phase based WCET analyzer when it has to estimate using unrefined phases is 4.5%(w1) of *RapiTime* (FULL) and 5.2%(w2) of *RapiTime* (START_OF_SCOPES). The average number of instrumentation points used by phase based WCET analyzer when it has to estimate using refined estimates is 12%(w1) of *RapiTime* (FULL) and 10.3%(w2) of *RapiTime* (START_OF_SCOPES). Unrefined estimates use CPI samples measured at 100-1000 instruction intervals. Refined estimates use CPI samples measured at a few loop iteration intervals and hence much higher in number.

With START_of_SCOPES/w2, the phase based WCET analyzer takes 1/10th, 3/4th of the time taken by *RapiTime* while using unrefined phases and phases refined based on signature. Further levels of refinement takes 30% more time than *RapiTime*. With FULL/w1, the time taken by the phase based WCET analyzer is always lesser than *RapiTime* (1/7th, half of, 3/4th that of *RapiTime* using unrefined, refined based on signature, refined based on CPI variance respectively). In case of *RapiTime*, the analysis time is a function of uncompressed trace size. In case of phase based WCET analyzer, there are several other factors that contribute to analysis time such as computation of theoretical upper bound of IC and phase detection. The analysis time of phase based WCET analyzer grows similarly compared to *RapiTime* for a trace of size less than 420MB. Beyond 420MB, the phase based WCET analyzer overtakes *RapiTime*. Due to the fact that the results of analysis of each phase are independent of the other, the analysis time can be further reduced by processing phases in parallel. While there is a large disparity between the estimates made by *RapiTime* at two different instrumentation levels, the estimates are not largely different at the two window sizes used by the phase based WCET analyzer. The reason is that the repetition of CPI variation within a phase ensures the

preservation of phase behavior at higher window sizes.

Our phase based technique is applied on the modified version of DEBIE-1 (debie1), which was specifically developed to evaluate WCET analyzers and the results are compared with *RapiTime*. debie1 is comprised of six tasks that are run routinely as part of system operations. The following comparisons are done at p=0.99 and compared with *RapiTime*-FULL instrumentation. Out of the six tasks, two tasks do not display phase behavior. As a result, *RapiTime* estimates WCET with greater accuracy than our proposed technique (Task1: 36% and Task2: 13%). Two other tasks exhibit phase behavior but since there is a large difference between theoretical upper bound on IC and maximum observed IC, the phase based technique exhibits greater pessimism in WCET estimate compared to *RapiTime* (Task3: 89% and Task4: 24%). Two other tasks exhibit phase behavior and WCET estimates are obtained with various levels of refinement and are more accurate than than *RapiTime* (Task5: 216% and Task6: 37%). The estimates can be further improved by using a more exhaustive test input set that guarantees sufficient coverage so that we could use maximum observed instruction count in place of theoretical upper bound on instruction count.

Chapter 7

Implementation of Phase Based Technique on a Native Platform

In the earlier chapters, we saw that WCET can be estimated with more accuracy if we consider phase-wise CPI than overall program CPI. We could apply simple probabilistic inequalities to obtain the probabilistic upper bound of CPI associated with a particular probability value. We also saw a much finer level of phase detection based on executed paths that isolates points of high variation of CPI. All these techniques were implemented using either a MIPS based simulator or an ARM based simulator. Simulators generally execute atleast 10 times slower than native execution. If a program runs for a very long time, simulation could take many hours or even days to finish. In such situations, a native execution is a much better alternative. In this chapter, we shall describe how the phase based technique can be implemented so that it can work directly on a native platform.

7.1 Performance API or PAPI

Cycles per instruction (CPI) is an important performance parameter that is used to commonly evaluate a processor. As a result, most processors are equipped with hardware performance counters that measure CPI with least intrusion. In this work, we use Performance API or PAPI[101] to measure CPI. PAPI is a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors.

Two kinds of counter interfaces exist- The high level API, which lets us to start, stop and

read counters for PRESET events on the cpu. The low level API manages hardware events in user defined groups called *event sets*. The low level API offers fine grained measurement and control of the PAPI interface. It provides access to both PAPI preset and native events. In this work, we use the low level API that allows us to focus on events that are of interest to us. The two events that are relevant to us are PAPI_TOT_INS and PAPI_TOT_CYC which help us measure total instructions and cycles respectively. The source code of the program is modified by adding calls to PAPI interfaces as shown in Figure 7.1.

The startup PAPI calls take up a few tens of thousands of cycles and occur only in the beginning of the program. Each of the PAPI_read calls that occur during execution of the program takes about 2000 processor cycles to execute. Hence we need to judiciously choose the instrumentation granularity such that the program execution time is not affected due to excessive PAPI calls. In the previous chapter, we saw that estimation of worst case phase CPI did not change significantly by having a larger instrumentation granularity. This happens because CPI varies repetitively within a phase. Hence we use a higher instrumentation granularity (one PAPI call per 500-2000 instructions) to accommodate PAPI calls and at the same time not affect the execution time of the program that we are trying to measure.

```
#include "papi_test.h"
.....
long long **values;
int EventSet=PAPI_NULL;

// Two events- PAPI_TOT_CYC and PAPI_TOT_INS
int num_events=2;
int papi_ctr=0, i_ctr=0, retval=0;

#define INSTR_GRAN 32

// maximum number of instrumentation points
long int num_tests=MAX_TESTS;

void function()
{

 ....

 for(....) {
  .....
 // record instantaneous CPI here
 papi_ctr++;
 if(papi_ctr%INSTR_GRAN == 0 && i_ctr<num_tests) {
  retval=PAPI_read(EventSet, values[i_ctr]);
  if(retval != PAPI_OK) {
   printf("error: PAPI_read!\n");
   exit(1);
  }
  i_ctr++;
 }
.....
}

int main(int argc, char** argv) {
// PAPI initialization code
 retval=PAPI_library_init(PAPI_VER_CURRENT);
 values=allocate_test_space(num_tests, num_events);
 if(retval != PAPI_VER_CURRENT) {
  printf("error: PAPI_library_init\n");
  exit(1);
 }

 if((retval=PAPI_query_event(PAPI_TOT_INS)) != PAPI_OK) {
  printf("error: PAPI_query_event(PAPI_TOT_INS)\n");
  exit(1);
 }
 if((retval=PAPI_query_event(PAPI_TOT_CYC)) != PAPI_OK) {
  printf("error: PAPI_query_event(PAPI_TOT_CYC)\n");
  exit(1);
 }
 PAPI_Start(EventSet);

// code starts here
 ......
// values[] contain set of CPI and IC measurements
}
```

Figure 7.1: Source code modifications to measure CPI using PAPI.

7.2 Partial Signatures

In the previous chapter, we saw that a phase could contain points of high CPI variation. Inorder to isolate such points of high variation, we used the notion of a *PC signature* to codify different paths in a compressed manner. The PC signature is composed of a bitmap that stores hashed PC addresses that have been executed in every x iterations of the loop that makes up the phase apart from the number of instructions (IC) executed in those x iterations and the CPI of the x iterations. The bitmap is easily implementable in a simulator as we have complete access of PC addresses that enter and commit in the pipeline. However, in case of native implementation, we cannot access the PC addresses of instructions in the pipeline using PAPI calls. We require specific hardware support inorder to be able to do that. As a result, we cannot form the bitmap which is a vital part of the PC signature. We can only store the IC and CPI of the x iterations. Hence instead of a complete PC signature which is a triple, now we have only a *partial PC signature* in the form of a tuple <IC, CPI>.

Inorder to evaluate the pessimism of resultant WCET estimates owing to the absence of the bitmap, we reuse the traces generated in the previous chapter but with the bitmaps removed. For this study, we choose three benchmarks out of our benchmark suites- *Bezier* which is observed to be very structured and is a well behaved program with little variance in IC and CPI, *Dijkstra* which exhibits high variation in IC and moderate variation in CPI and *Janne_complex* which has a complex structure but CPI is rather stable. Figures 7.2, 7.3 and 7.4 plot the pessimism of WCET estimate obtained using CPI samples of an unrefined phase (unref_p), phase refined based on complete signatures (ref_sig_p) and phase refined based on partial signature (ref_p). On an average, the use of CPI samples of phases obtained by refinement based on partial signatures yield WCET estimates that are pessimistic than those obtained using CPI samples of unrefined phases by {50%, 45% and 31%} , {48%, 44%, 33%} and {46%, 42%, 29%} on *Simplest*, *Inorder_complex* and *Complex* corresponding to probabilities p=0.9, 0.95 and 0.99.

The reason for increased pessimism in explained as follows. Using partial signatures we can refine a phase based on only IC. All tuples with the same IC fall into the same sub-phase. Assume we have a nested loop as shown in Figure 7.5. Assume that the number of instructions executed in the inner loop is 10 per iteration and bounds of both loops are 10 and that there are 10 other instructions executed in the outer loop apart from the inner loop. Assume 4 iterations

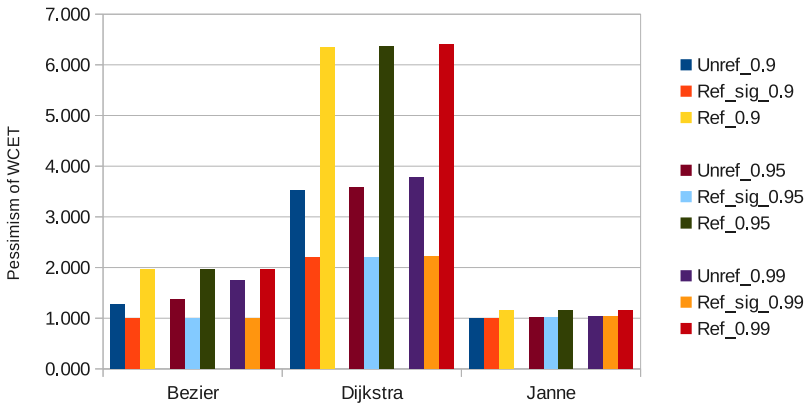


Figure 7.2: Impact of partial signatures on pessimism of WCET on Simplest architecture

are grouped into a window and instrumented. Since every window consists of 4 iterations and 4 does not divide 10. When we reach the inner loop boundary, the next window will contain additional instructions belonging to the boundary (remaining 2 iterations). This causes the number of instructions executed for every window to be 40, 40, 50, 40 and so on. According to our definition of sub-phases as described in Chapter 6, the maximum IC possible in every sub-phase represented by a bitmap signature is considered to compute WCET estimate. Hence we need a way to avoid including the extra instructions that get executed at the boundary for every loop iteration. Because if we include it, we end up computing a much higher instruction count which is even greater than the theoretical upper bound on IC.

7.2.1 Optimal Global Maximum

In this section, we describe the solution to reduce the pessimism in IC. Lets assume we have n sub-phases and that we have full signature information including the bitmap. Let the maximum IC for a bitmap B_i be LM_i (local maxima). According to the definition of sub-phase as described in the previous chapter, let f_i be the fraction of times, bitmap B_i occurs in the traces. Then we have,

$$SWW \times f_0 \times LM_0 \ + \ ... \ + SWW \times f_n \times LM_n \leq SWIC \tag{7.1}$$

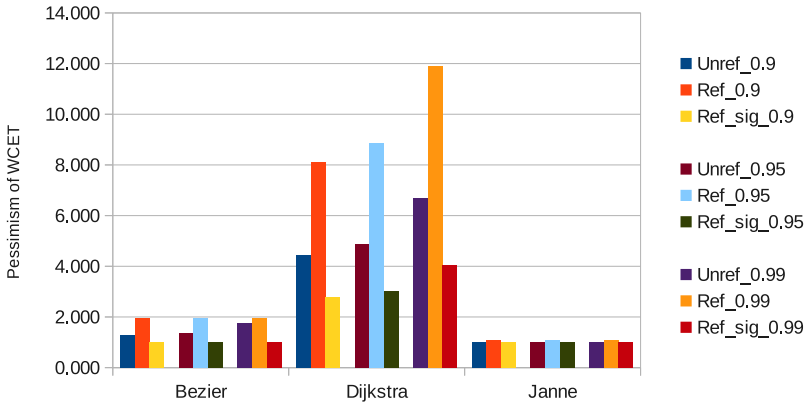


Figure 7.3: Impact of partial signatures on pessimism of WCET on Inorder_complex architecture

Where SWW is the theoretical upper bound on the number of windows possible and SWIC is the theoretical upper bound on IC.

However, we have only partial signatures with us. As a result, we do not know the unique local maxima IC values, LM_0, ... ,LM_n. We only have a global maxima value which is the maximum IC across all sub-phases pertaining to the loop. Let us term the global maxima value as GM. Hence we have the following equations.

$$GM \ = \ Max(LM_0, \ ... \ , LM_n) \tag{7.2}$$

$$SWW \times f_0 \times GM \ + \ ... \ + SWW \times f_n \times GM \leq SWIC \tag{7.3}$$

$$SWW \times GM \times (f_0 \ + ... \ + f_n) \leq SWIC \tag{7.4}$$

Since (f_0 + ... + f_n) equals 1, we have,

$$SWW \times GM \leq SWIC \tag{7.5}$$

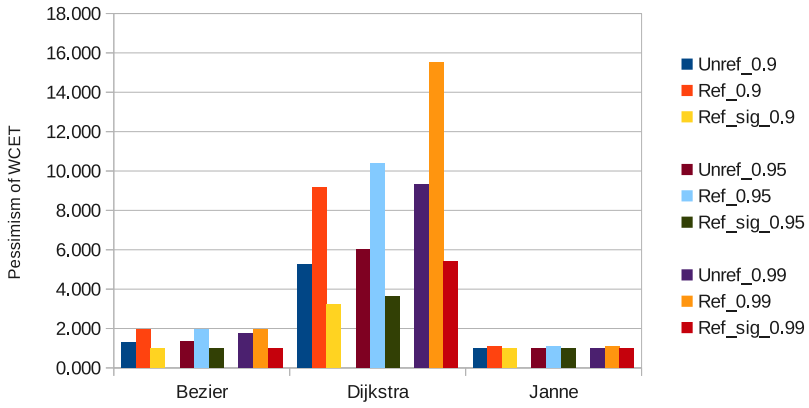


Figure 7.4: Impact of partial signatures on pessimism of WCET on Complex architecture

Hence the optimal value of GM should be,

$$Opt\_GM \leq \frac{SWIC}{SWW} \tag{7.6}$$

If x iterations comprise a window (sub-phase),

$$Opt\_GM \leq \frac{SWW \times x \times Max\_IC\_per\_iteration}{SWW} \tag{7.7}$$

$$Opt\_GM \leq x \times Max\_IC\_per\_iteration \tag{7.8}$$

Which essentially is what Opt_GM is supposed to be: x times the maximum possible IC per iteration.

Method

Using this information, we bump up the IC value of sub-phase i to Opt_GM if $IC_i <$ Opt_GM. If IC_i of sub-phase i is $>$ Opt_GM (likely boundary candidates), we bump it up to GM. This prevents all the iterations from being assigned a false high boundary value and hence brings down the pessimism. This solution brings down the pessimism of WCET estimate obtained

```
for(i=0;i<10;i++)
{
 ..... // 10 instructions in outer loop

  for(j=0;j<10;j++)
  {

    .......  // 10 instructions per iteration

    ctr++;
    if( ctr % 4 == 0 )  // window size = 4 instructions
      instrument;
  }
}
```

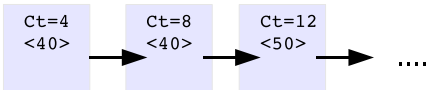


Figure 7.5: Cause of additional pessimism with partial signatures : an example

using partial signatures considerably as shown in Figures 7.6, 7.7 and 7.8. The estimates obtained by using Opt_GM is represented by ref_opt_p. The average pessimism of WCET estimates for all the three benchmarks considered are plotted in Figures 7.9, 7.10 and 7.11.

Table 7.1: Percentage Improvement by using Opt_GM instead of GM as maximum IC of a sub-phase.

	p=0.9	**p=0.95**	**p=0.99**
Simplest			
Ref_opt/Ref	40.95	40.89	40.66
Ref_opt/Ref_sig	-33.15	-33	-32.97
Ref_opt/Unref	3.6	5.89	13.97
Inorder_complex			
Ref_opt/Ref	41.4	41.69	42.47
Ref_opt/Ref_sig	-36.84	-38.04	-41.91
Ref_opt/Unref	2.72	4.33	8.86
Complex			
Ref_opt/Ref	46.16	47.02	49.35
Ref_opt/Ref_sig	-26.52	-26.4	-26.8
Ref_opt/Unref	12.29	15.16	22.3

The percentage improvement obtained by using Opt_GM compared to the estimate obtained using unrefined phases and refined phases with partial and full signature is described in Table 7.1. On *Simplest* architecture, refined estimates show more or less the same values as

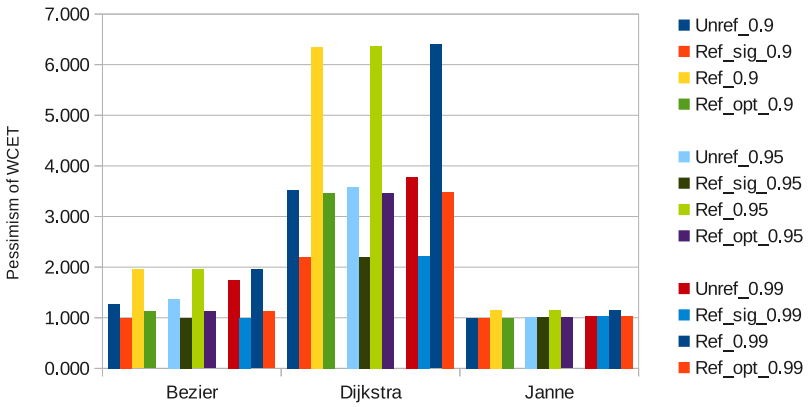


Figure 7.6: Pessimism of WCET estimates obtained using unrefined phase, refined phase with full and partial signatures on Simplest architecture

the architecture does not consist a data cache and predicts branches perfectly. Around 41% improvement is seen when Opt_GM is used with partial signatures compared to when GM is used. However the estimates are still 33% more pessimistic compared to the situation when full signatures are known. At p=0.99, the estimates obtained using Opt_GM are 14% more accurate than estimate obtained using unrefined phases.

On *Inorder_complex* architecture, refined estimates show slightly more pessimism with increasing probability as the architecture is more realistic and has a data cache and a realistic branch predictor. Around 41% improvement is seen when Opt_GM is used with partial signatures compared to when GM is used. However the estimates are still 38-40% more pessimistic compared to the situation when full signatures are known. At p=0.99, the estimates obtained using Opt_GM are 9% more accurate than estimate obtained using unrefined phases.

On *Complex* architecture, refined estimates show slightly more pessimism with increasing probability as the architecture is out-of-order and has a data cache and a realistic branch predictor. Around 48-49% improvement is seen when Opt_GM is used with partial signatures compared to when GM is used. However the estimates are still 26% more pessimistic compared to the situation when full signatures are known. The difference between estimates obtained

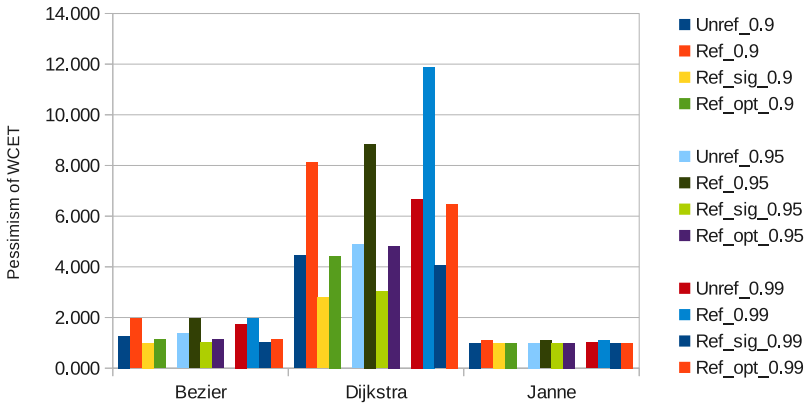


Figure 7.7: Pessimism of WCET estimates obtained using unrefined phase, refined phase with full and partial signatures on Inorder_complex architecture

using partial signatures with Opt_GM and those obtained using full signatures comes down in case of *Complex* architecture as the CPI variation in *Complex* is much more random than *Inorder_complex*. Refinement based on full signature works better on *Inorder_complex* than on *Complex*. At p=0.99, the estimates obtained using Opt_GM are 22% more accurate than estimate obtained using unrefined phases.

Using Opt_GM with partial signatures, we estimate WCET for these three benchmarks on the native platform. We run our experiments on AMD Athlon with a dual core processor each of speed 800 MHz, a cache of 512 KB and 2GB DDR memory. We use PAPI 5.0.1 to collect traces of IC and CPI. Figure 7.12 describes the pessimism of WCET estimate obtained by our technique on this native platform. The average pessimism of WCET estimated using unrefined phases on the native platform is 4, represented by the first bar in each probability. Using refinement based on partial signatures, the average pessimism comes down to 3.65(second bar). Subsequent refinement based on controlling CPI variance within a sub-phase to 50%, 10%, 5% and 1% of the original CPI variance brings down pessimism to 3.6, 2.78, 2.56 and 2.52 respectively(subsequent bars under each probability).

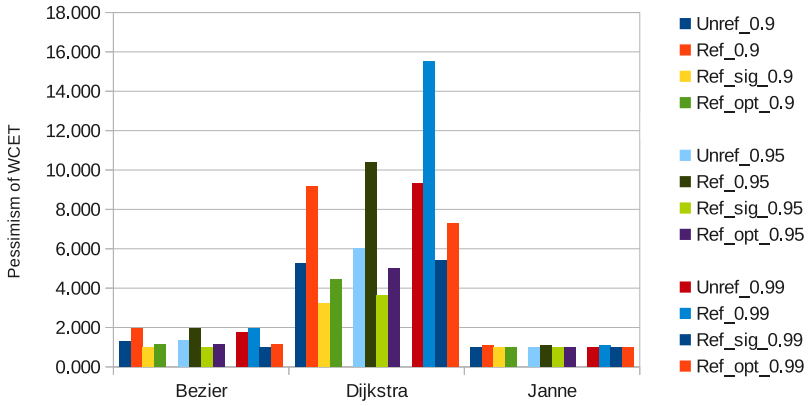


Figure 7.8: Pessimism of WCET estimates obtained using unrefined phase, refined phase with full and partial signatures on complex architecture

7.2.2 Instrumentation Overhead with PAPI

In this section we shall describe the instrumentation overhead due to PAPI calls. The instrumentation ratio for the three benchmarks considered is described in Figure 7.13. The average sub-phase sizes for *Bezier*, *Dijkstra* and *Janne_complex* on the native platform are 1841, 554 and 650 respectively. On an average PAPI is called 0.12% of the time the program is executed, to measure CPI. Since PAPI start up code takes a few thousands of cycles and some programs execute for only a few thousands of cycles, we execute such programs many times to overcome the effect of start up code. Figure 7.14 describes the time taken by the programs with and without PAPI code. The average overhead with respect to time due to PAPI across all benchmarks is 2.2%.

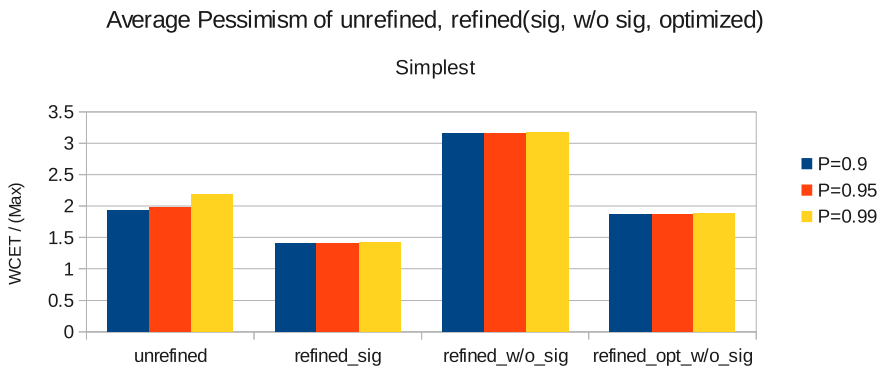


Figure 7.9: Average pessimism of WCET estimate using all kinds of phases on Simplest architecture

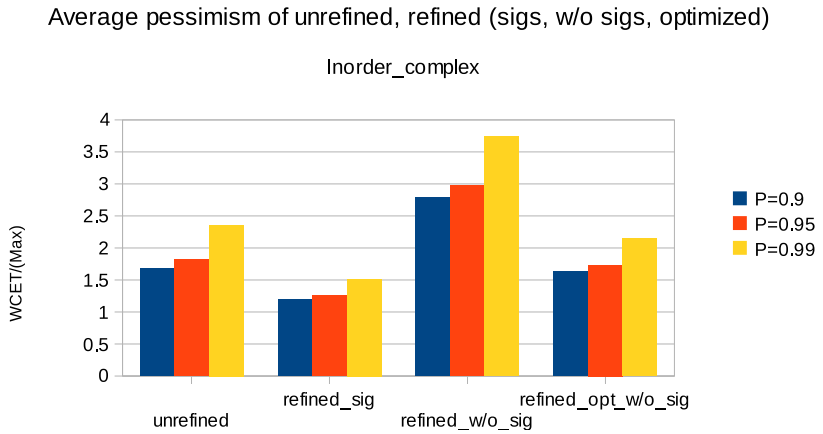


Figure 7.10: Average pessimism of WCET estimate using all kinds of phases on Inorder_complex architecture

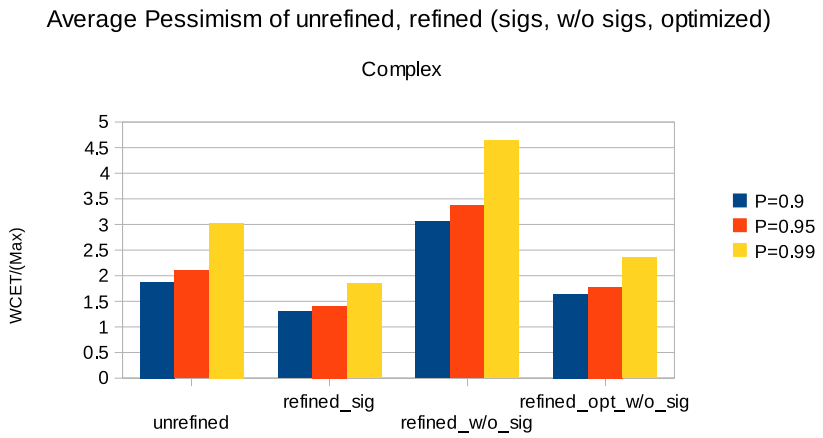


Figure 7.11: Average pessimism of WCET estimate using all kinds of phases on Complex architecture

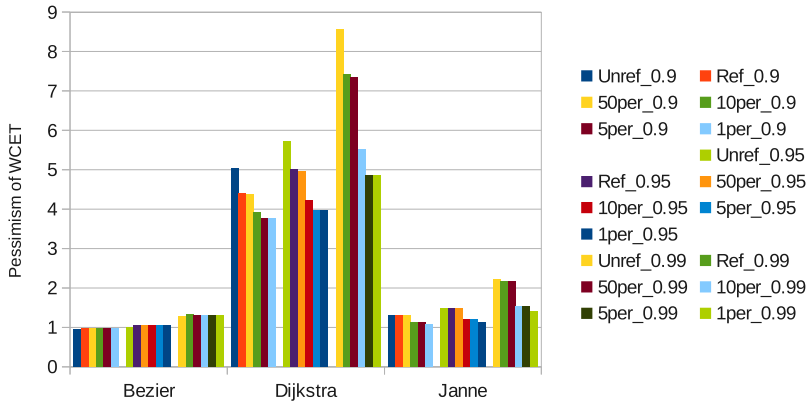


Figure 7.12: Pessimism of WCET estimate using unrefined and refined phases on native platform

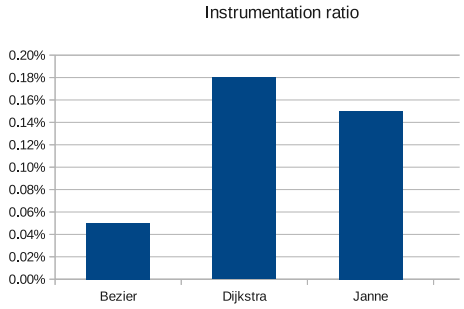


Figure 7.13: Percentage of time, PAPI is called during program execution

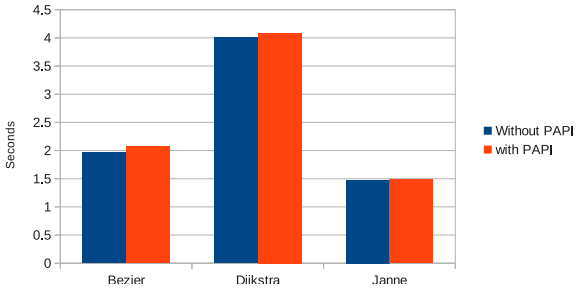


Figure 7.14: Time overhead of PAPI calls

7.3 Related Work

Corti et al[52] use performance counters in Motorola PowerPC 604e microprocessor to measure various parameters like cache misses, stalls in the pipelines and various other units. These values are used to calculate CPI that is multiplied by the basic block length to obtain cycles taken by basic blocks to execute on the target system. Our technique uses performance API (PAPI) to measure CPI at various instances within a phase. The CPI samples within a phase are used in either an absolute function or a probabilistic function to obtain absolute or probabilistic estimates worst case phase CPI which will be used to compute the overall program WCET. Different versions of PAPI are available on multiple architectures. Other measurement based tools either use hardware devices such as oscilloscopes, logic analyzers to obtain time traces, but in the case of these devices, it is difficult to find the correspondence between measurements and the program paths[34]. Hardware debug support comprising of trace buffers of limited memory have been used to collect timestamp traces of instructions in measurement based analyzers[3]. However the issue with these buffers is the rate at which these traces are produced and the finiteness of the trace buffer. If trace data is produced too fast, there could buffer overflows causing blackouts as a result of which, part of trace can be lost. Hence generally only branch instructions are timestamped and recorded in the trace buffer[3]. The phase based technique helps us to use sparse instrumentation to collect CPI at arbitrarily large granularities of instructions without significantly affecting the accuracy of phase CPI and hence worst case execution time. As a result, source code level instrumentation with PAPI can be used to obtain accurate measurements of CPI. Recently there have been efforts to support very fast access to hardware performance counters such as LiMiT (light weight microarchitectural tool kit)[45] that is 90x faster than PAPI-C requiring 12ns per access. We could benefit from such a tool that would reduce instrumentation overhead even further.

7.4 Conclusions

In this chapter, we describe how we can implement the phase based WCET analysis technique on a native platform. Simulations are atleast 10 times slower than native execution. For programs that take a long time to execute, collecting traces might take hours or even days. In such programs, collection of traces by native execution is very helpful. Implementation of

the technique on a native platform involves modifying the source code to make PAPI calls. Unless, we have special hardware support to record PC addresses of instructions that commit in the pipeline, we have to resort to using a partial PC signature instead of a full PC signature, <Bitmap, IC, CPI>. A partial signature is a tuple <IC, CPI> that represents the number of instructions executed in the sub-phase and the CPI of the sub-phase. In some programs that involve nested loops, there is an added pessimism in the estimation of worst case IC with partial signatures, as it considers additional instructions executed at the boundaries to compute maximum IC per sub-phase. On an average, the use of CPI samples of phases obtained by refinement based on partial signatures yield WCET estimates that are pessimistic than those obtained using CPI samples of unrefined phases by {50%, 45% and 31%} , {48%, 44%, 33%} and {46%, 42%, 29%} on *Simplest*, *Inorder_complex* and *Complex* corresponding to probabilities p=0.9, 0.95 and 0.99.

A simple correction in the timing equation that makes use of optimal global maximum IC per sub-phase alleviates the pessimism to a large extent (40-49% on *Simplest*, *Inorder_complex* and *Complex* architectures). If there is hardware support to associate instruction addresses with event counters, accuracy of WCET can improve further by (26-41%). PAPI calls takes around 2000 processor cycles to execute. Hence the program has to be instrumented with caution so as to not affect the execution time of the program which we are trying to analyze. Program phase behavior comes to the rescue here. Higher instrumentation granularities does not change the variation characteristics of CPI significantly. Hence we choose a much higher instrumentation granularity (one PAPI call per 500-2000 instructions) to measure CPI. The average instrumentation ratio (ratio of number of PAPI calls to the total number of instructions) is observed to be 0.12%. On an average, PAPI is observed to cause the execution time of programs to increase by 2.2%. In the next chapter, we shall see other advantages of phases in the context of program timing analysis.

Chapter 8

Other Advantages of Phases in Timing Analysis

In the previous chapters, we saw that program phase behavior helps build a very simple timing model that estimates WCET of a program in terms of its phases. The homogeneous variation of CPI within a phase enables us to use coarse instrumentation (ratio of instrumentation $<$ 5%). In this chapter, we shall see that phases offer additional advantages such as faster WCET analysis and estimation of WCET for a particular program run.

8.1 Parallelized WCET analysis

Existing WCET analysis methods can be classified into two categories- a *separated* approach where program structural analysis is decoupled from estimating the effect the underlying architecture or an *integrated* approach where structural analysis and estimating the effect of the underlying architecture are carried out side by side. Our approach clearly belongs to the separated approach as we estimate the theoretical upper bound on IC and worst case CPI separately. The estimation of both quantities are independent of each other. Hence estimation of theoretical upper bound on IC and worst case CPI can be done in parallel. However, if the program has multiple phases, program phase detection has to be carried out first before the theoretical upper bound on IC for each phase is determined.

Our approach is trace based and estimates worst case CPI by analysis of traces of CPI per phase or sub-phase. Some programs can be composed of a number of phases depending on its

structural complexity. Some programs with large running times can produce large amounts of trace. Hence trace analysis can be expensive in terms of time and space. In this regard, we are at an advantage of using phases to divide our program into smaller units of analysis. Since the analysis result of one phase is not dependent on the analysis of another phase, we can analyze the phase traces in parallel. This can be done because CPI is fairly an independent entity. The CPI of one phase is fairly independent of the other phases. No correlation is observed between CPI of two different phases of a program.

In an earlier chapter, we saw that the time taken to obtain WCET estimate based on phase refinement with respect to PC signature in the case of *Dijkstra* was about 40 minutes on ARM architecture, a major part of which is taken up by trace analysis. This is because, *Dijkstra* is comprised of a large number of phases based on PC signature which is further comprised of smaller number of sub-phases when refined based on CPI tolerance. The trace analysis time using our technique is observed to be about 4 times the time taken by *RapiTime*. We apply the following parallelization on *Dijkstra* to reduce trace analysis time for estimating WCET using phases refined based on signature. The parts of timing analysis that are carried out in parallel are prefixed by "[parallel]".

1. Run the profile based phase detection algorithm on the program with a few representative test inputs.

2. Note the binary instructions that mark phase boundaries.

3. Build CFG out of the program binary.

4. Demarcate the phases in the CFG.

5. [parallel]
   Distribute the work of computing the theoretical upper bound on instruction count for each phase i equally among n threads.

6. Run the program either on a simulator or on the native platform. The trace of CPI for each phase and sub-phases is generated in cpisig_i for each input i.

7. Compress cpisig_i for all i.

8. [parallel]
   Distribute the work of collecting unique PC bitmap/IC values from cpisig_i into temp_i equally among n threads.

9. Merge all the temp_i files into a single unique signature file.

10. [parallel]
    Distribute the work of collecting various IC for each unique signature i and the maximum IC equally among n threads.

11. Collate the various IC information and maximum IC into a file.

12. [parallel]
    Distribute the work of collecting CPI for each unique PC bitmap, IC pair (each sub-phase i) equally among n threads.
    Each thread in parallel calculates mean and variance of sub-phase CPI for each sub-phase i.
    Each thread in parallel calculates Chebyshev-Cantelli bounds of CPI for each sub-phase i and output this information into variance_sub_i.

13. [parallel]
    Distribute the work of collecting CPI for each unrefined phase i, equally among n threads. Each thread in parallel calculates mean and variance of phase CPI for each phase i. Each thread in parallel calculates Chebyshev-Cantelli bounds of CPI for each phase i and outputs this information into variance_i.

14. Using sub-phase signature information, create an awk file that records the occurrence of every sub-phase for a given input.

15. [parallel]
    Distribute the work of collecting sub-phase occurrence pattern for every input i equally among n threads. Each thread in parallel creates the frequency of occurrence information for every sub-phase for each input i.

16. Collate the frequency of occurrence of sub-phases for all inputs i into a single file.

17. Append CPI bound information for every sub-phase to this file.

18. Append information about the theoretical maximum number of windows possible (SWW) (section 6.4.3 in chapter 6) to this file.

19. Use the timing equation Eq.(6.23) to estimate WCET of unrefined phases and Eq.(6.29) to estimate WCET of refined phases and Eq(6.3) for programs that exhibit multiple phase patterns.

In addition to this, steps 1 and 2 is run in parallel with step 3. And step 13 is run in parallel with steps 9 - 12. Likewise, the trace analysis for estimating WCET using phases refined based on CPI variance is parallelized. We evaluate the speedup in analysis time obtained in the case of *Dijkstra* using multiple threads in Figure 8.1. The experiment was run on a multicore machine comprising of 8 Intel Xeon cores each at a frequency of 2.66GHz with a cache size of 6144KB. We define speedup with n threads as the ratio of the time required to execute steps 1 to 19 running n threads to the time required to execute steps 1 to 19 sequentially.

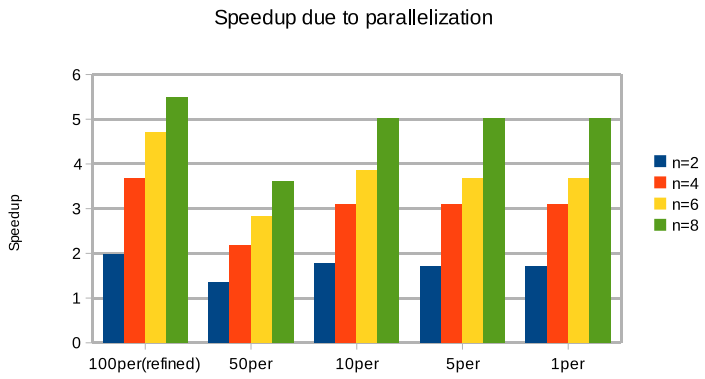


Figure 8.1: Speedup in trace analysis time with multiple threads.

8.2 Worst Case Remaining Execution Time (WCRET)

The primary focus of the thesis is to estimate worst case execution time of a program that holds across any permissible input. There also exists a dynamic counterpart of WCET of a

program that can be calculated at any instance during execution of a particular program-input pair, also known as the worst case remaining execution time (WCRET). Accurate estimation of the worst case remaining execution time at any point during a program's execution helps in improving scheduling tasks to maximize resource utilization. If WCRET is unknown, tasks might be unnecessarily over-provisioned with resources. If tasks finish in much less time than their anticipated worst case execution times, this would result in severe under utilization of resources[62]. The earlier this information is available to us, the better. It is highly desirable if the method allows us to learn from program behavior and helps us refine our existing WCRET estimates.

Program phase behavior allows us to estimate WCRET of a program fairly in advance and also refine the estimate in a very simple way. At the beginning of program execution, there is little information about the dynamic behavior of the program. Hence our initial estimate of WCET is the product of theoretical upper bound on IC and maximum of average CPI. As and when phases are entered into and information about actual phase CPI begins to emerge, we refine our estimate of WCRET. Our prime assumption is that variation of CPI within a phase is fairly homogeneous and repetitive and CPI will continue to vary in the same manner till the end of the phase. The program is intercepted at every ten thousand instructions and the WCRET is estimated at this point. Figure 8.2 describes our algorithm to estimate the WCRET of the program at each such point. If a program has multiple phase patterns, the algorithm can be modified to keep track of the currently occurring phase pattern and apply the corresponding timing equation. MIN_CPI in Figure 8.2 refers to the minimum number of intervals that should elapse before we start using current phase CPI to help predict future CPI. Max_CPI[i] refers to the worst case CPI of phase i estimated as described in chapter 5.

We apply this technique on some of the benchmarks that have been considered in this work on the *Inorder_complex* architecture which is of intermediate complexity. All programs are compiled by gcc with the -O2 and static flag inorder to be simulated by *Simplescalar*. The simulator code is suitably modified as described above in Figure 8.2. Each program is run with a specific input and the call to compute WCRET is inserted after the commit of every ten thousand instructions.

```
/***
 **   curr_phase: current phase
 **   cycles[i]: stores actual elapsed cycles for phase i
 **   instns[i]: stores actual number of instructions executed
 **   for phase I
 **   total_pred_cycles: Predicted worst case execution time
 **   for the current run
 **   SWIC[i]: theoretical upper bound on Ic for phase i
 **   sim_cycle: actual cycles elapsed till now
****/


Compute_WCRET () {

// Account for all phases that have occurred so far
for(i=0; i<curr_phase; i++) {
 total_pred_cycles += cycles[i];
}

// Account for current phase
if(count_cpi[curr_phase] > MIN_CPI) {
 total_pred_cycles += (SWIC[curr_phase] *
   (float) cycles[curr_phase]/instns[curr_phsae]);
}

// Account for all future phases
for(i=curr_phase+1; i<NUM_PHASES; i++) {
  total_pred_cycles += SWIC[curr_phase] * Max_CPI[curr_phase];
}

WCRET = total_pred_cycles-sim_cycle;

}
```

Figure 8.2: Algorithm to compute WCRET of a program at any instant of time during execution.

8.2.1 Evaluation

We plot the predicted WCRET at the commit of every ten thousand instructions. Inorder to evaluate the accuracy of predicted WCRET, we also plot the actual remaining cycles at each of these points for comparison purposes(computed by the difference of actual execution cycles and cycles elapsed till that point). Ideally, the two plots should coincide. For simple straight line programs with stable CPI behavior even across inputs, the predicted WCRET is accurate right at the beginning and remains so during execution of the program. *Matmul* is a classic example as shown in Figure 8.3.

In programs with a high maximum CPI, the initial predicted WCRET is pessimistic, but as we continue to learn about the dynamic behavior of CPI as execution progresses, we can refine our WCRET estimate accordingly. *Bitcount* is a good example with a high maximum CPI. But as we get to know more about dynamic behavior of each of its phases, we can get close to the actual remaining cycles as shown in Figure 8.4. The bends in the curve correspond to the phase transition points in *Bitcount*. In programs where the theoretical upper bound on IC

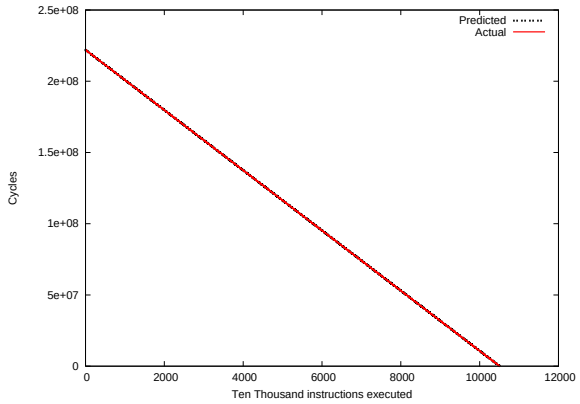


Figure 8.3: Predicted remaining cycles versus actual remaining cycles for *Matmul* (In-order_complex).

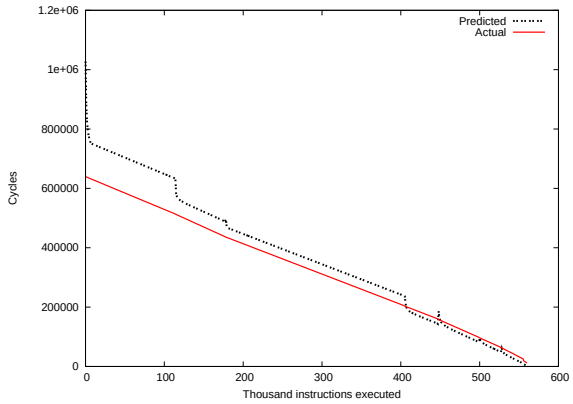


Figure 8.4: Predicted remaining cycles versus actual remaining cycles for *Bitcount* (In-order_complex).

is larger than the maximum observed instruction count, the corresponding WCRET estimate also is larger by that amount. *Bezier* and *Bubble sort* are classic examples of this as shown in Figures 8.5 and 8.6 respectively.

At present, we refine our WCRET estimate by making use of the current CPI information. If there is a large difference in the theoretical upper bound on IC and the number of instructions that have been executed, the estimated WCRET is always larger than the actual WCRET even at the end of program execution. If there is a way to learn and use the number of instructions that have been executed in the equation to estimate WCRET, our technique can estimate WCRET with high degree of accuracy. For programs with a simple structure, this can be

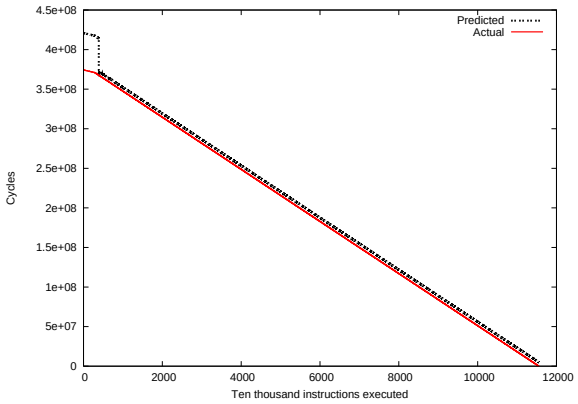


Figure 8.5: Predicted remaining cycles versus actual remaining cycles for *Bezier* (In-order_complex).

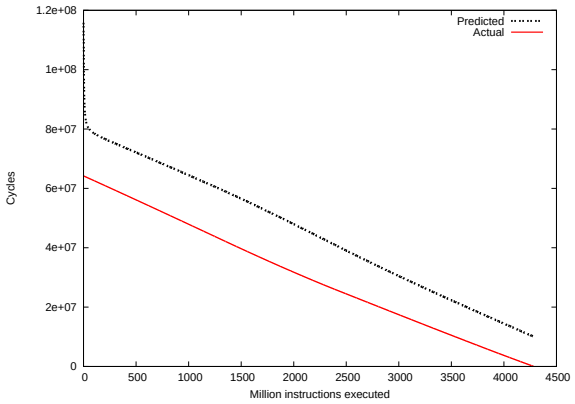


Figure 8.6: Predicted remaining cycles versus actual remaining cycles for *Bubble sort* (In-order_complex).

easily done. Lets consider *Bubble sort* which is composed of a 2 level loop nest. The outer loop iterates for N times where N represents the number of elements to be sorted. The inner loop iterations depend on the index of the outer loop. So essentially the inner loop body is executed $\frac{N\times(N+1)}{2}$ times. If we can keep count of the number of times the outer branch instruction is executed, we can use the number of instructions that have been executed so far and refine our WCRET estimate shown as follows. By keeping track of the number of instructions executed so far, we are able to predict the worst case remaining execution time with more accuracy as shown in Figure 8.7.

$$N = MAX\_ELEMENTS - bubcount;$$

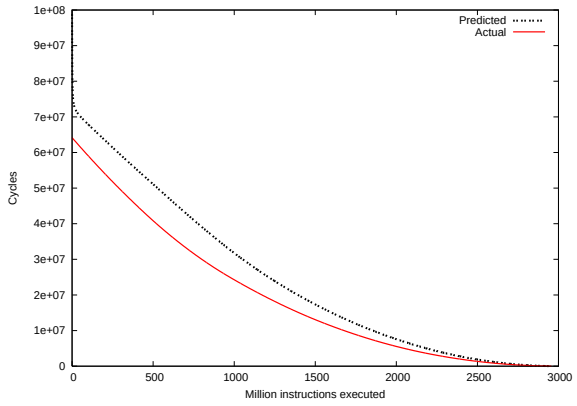


Figure 8.7: Predicted remaining cycles versus actual remaining cycles for *Bubble sort* (In-order_complex) tracking number of instructions executed along with CPI.

$$total\_predicted\_cycles \;=\; sim\_cycle + \frac{N \times (N+1)}{2} \times I \times \frac{total\_cpi[0]}{count\_cpi[0]};$$

Where, MAX_ELEMENTS is the size of the array that is being sorted. *bubcount* keeps count of the number of times the outer loop branch is executed. I is the number of instructions in the inner loop body.

8.3 Related Work

Kumar et al[83] describe a framework that finds dominant or recurring patterns of behavior from the profile of an application, and produces signatures (detection patterns) that can be used to identify the occurrence of these behaviors in future runs of the application, predict the occurrence of these behaviors sufficiently in advance, and make statistical assertions about the likelihood of the occurrence of these behaviors during the application's execution. The programmer can use these patterns to assess how well the application will satisfy various relaxed real-time deadlines. The method first profile-instruments a C application. It then runs the Statistical Analyzer tool described in [83] that detects patterns of behavior and generates prediction patterns and statistical guarantees for those. The patterns of behavior consist of segments of function call-chains, annotated with the statistics predicted for them. The call-chains are further refined into minimal distinguishing call-chain sequences that unambiguously detect the corresponding pattern of behavior when it starts to occur at runtime, and make

statistical predictions about the nature of the behavior. Kumar et al[83] expect that the statistical behavior of the function call at the top of the stack can be predicted using a short sequence of function calls occurring just below it in the call-stack, without going all the way to main in most applications. Apart from call sequence, behavioral statistics annotated by profile information are considered for a match. The matching sequence is picked to predict the future call sequence. The idea is to carry out an early prediction which can be done by seeing only a small prefix and predicting the entire suffix. A finite state machine sequence detector is used to predict the following pattern. Our technique works on the principle of phase behavior. Based on the observation that CPI varies homogeneously and repetitively within a phase, an early sequence of phase CPI is used to predict the subsequent sequence of phase CPI. Gupta et al[62] propose that correlation among different parts of the program can help refining the WCRET estimate. Only regions that matter in influencing execution time are considered. Correlation is computed at the statement level. For instance an assignment and a condition within the loop can be correlated. Post assignment, the remaining time estimate is likely to be refined better. Another possibility that can affect remaining execution time is a set of multiple assignments affecting a branch. The compiler must be modified to find out all such assignments that affect a branch. In contrast to [62] which is purely a static approach, our work is a dynamic approach and is centered around the CPI.

8.4 Conclusions

Apart from reducing instrumentation overhead, program phase behavior has other advantages too. The CPI of one phase is fairly independent of another phase. The result of trace analysis of one phase does not depend on the result of trace analysis of any other phase. As a result, trace analysis of multiple phases can happen in parallel thereby reducing the overall time to carry out WCET analysis. Further estimating theoretical upper bound on IC can be parallelized with estimating WCPI. In this chapter, we implemented such a parallel WCET analyzer that estimates WCET based on refinement with respect to PC signatures and observed the following speedup of {1.9, 3.7, 4.7, 5.5} with {2, 4, 6, 8} threads respectively. For estimation based on higher levels of refinement with respect to controlling CPI variance, the speedup is slightly reduced by a few percent. Program phases also help in estimating the worst case remaining

execution time of a program run with a particular input, at any point during its execution. Accurate knowledge of WCRET helps prevent over-provisioning of resources thereby increasing resource utilization. Estimation of WCRET is based on the homogeneous variation of CPI within a phase and involves predicting future CPI of a phase by looking at a prefix of occurring phase CPI. For certain programs with a simple structure, one can even predict the number of instructions that will further improve accuracy of WCRET.

Chapter 9

Conclusions and Future Work

The thesis explores several aspects of CPI-centric worst case execution time analysis and proposes techniques that estimate WCET with increasing levels of accuracy in each succeeding chapter. State of the art WCET analyzers estimate cost of executing a program in terms of cycles either by carrying out static analysis, or direct measurement or fitting statistical models to measured execution times. The thesis proposes to view execution time as a product of instruction count(IC) and cycles per instruction(CPI) and explores the advantages of this formulation in each chapter.

To begin with, a program is considered as a single unit and program WCET is estimated as a product of worst case IC and worst case CPI. Worst case IC is estimated as the theoretical upper bound on IC(SWIC). The advantage of using a bound on IC is that it can be used whenever building an exhaustive test input set that achieves complete coverage is difficult. Worst case CPI is estimated using various analytical and statistical functions of measured CPI samples. Using this basic timing model, a safe analytical combination that can be used for IC and CPI is determined to be SWIC, Max_Avg(CPI) which is the maximum of average CPI of the program observed across inputs. On *simplest* architecture, this combination gives a WCET estimate that is 9% pessimistic compared to *Chronos*. On *Inorder_complex* and *Complex* architectures, this combination improves accuracy in WCET estimate by 38% and 51.7% respectively compared to *Chronos*. A safe statistical combination that can be used for IC and CPI is determined to be SWIC, 99per(CPI) which is the 99th percentile CPI value observed across all inputs. On *simplest* architecture, this combination gives a WCET estimate that is 36.5% pessimistic compared to *Chronos*. On *Inorder_complex* and *Complex* architectures, this

combination improves accuracy in WCET estimate by 10.6% and 29.3% respectively compared to *Chronos*. The percentile CPI value is observed to be highly dependent on the distribution of CPI samples which inturn depends on the distribution of input data. Hence the analytical combination is preferred over the statistical one.

During our experiments, we observe that the overall program IC and CPI are correlated in many cases. This gives us an opportunity to optimize our WCET estimate. A scatter plot is constructed with a large vector of (IC, CPI) samples gathered over a large number of runs with different inputs. Five kinds of correlation are observed. In some programs, CPI and IC are negatively correlated with each other. With increasing IC, CPI decreases. In some programs, CPI and IC are positively correlated with each other. With increasing IC, CPI also increases. In some programs, irrespective of input, IC and CPI do not vary significantly. In some programs, with increasing IC, CPI saturates to a particular value. In some programs, the correlation is not clear. In programs where the correlation is clear, we can fit a curve to the points in the scatter plot and define CPI as a function of IC. Using this, we optimize WCET as a product of the theoretical bound on IC and f(IC). This estimate is observed to be much optimal than the product of worst case IC and worst case CPI in many cases. Using correlation information, on *Simplest* architecture, the WCET estimate is now only 4.7% more pessimistic than *Chronos*. On *Inorder_complex* and *Complex* architectures, the improvement in accuracy using correlation is 49% and 62.3% compared to *Chronos*. The correlation also helps us in benchmark classification. For instance, programs that exhibit the same IC, CPI values irrespective of input need not be tested exhaustively with respect to worst case execution time analysis.

The formulation of WCET as a product of a maximal function of IC and a maximal function of CPI works well for small programs wherein execution centers around a single loop and exhibits stable CPI throughout execution. However, this is seldom the case. Many a times, a program is composed of a well defined set of tasks. The dynamic behavior within each task might be different from the rest. This principle manifests itself as program phase behavior which refers to a phase like variation of certain architectural parameters like CPI during execution. Within each phase, the variation of CPI is homogeneous and centered around the mean. The coefficient of variation of CPI within a phase is much lesser than the coefficient of variation of CPI across phases. Using this observation, we propose a phase based timing model that

estimates WCET of a program in terms of its phases. On *Simplest* architecture, using phase information brings the WCET estimate very close to *Chronos* (1.73% higher than Chronos). On *Inorder_complex* and *complex* architectures, using phase information, the WCET estimates are 43% and 55.3% more accurate than *Chronos*. Phases have important implications on the instrumentation aspect of measurement based WCET analysis. Phases can be instrumented at arbitrarily large intervals without causing a significant impact on the accuracy of WCET. Phases are typically composed of hundreds or thousands of instructions. Hence instrumentation at the phase level is much more advantageous than instrumentation at the level of basic blocks and instructions.

The homogeneous and uniform nature of CPI within a phase helps us to obtain probabilistic bounds of phase CPI using simple probabilistic inequalities such as *Chebyshev* inequality which does not assume any particular distribution of data. Since the CPI samples that we have are not created by exercising *all* paths in the program, it helps to have a formalism by which we can bound a future unknown CPI sample to obtain a more robust WCET estimate. By using the theoretical upper bound on IC, which is a constant, we prove that the probabilistic bound on CPI can be extended to the whole program WCET as well. Based on these lines, we build a probabilistic WCET analyzer that can give us WCET estimates depending on the desired probability value.

Some phases can exhibit high variation of CPI. The primary reasons that could cause this could either be the presence of complex *if-conditions*, branch mispredictions, stalls in the pipeline or cache behavior. Applying probabilistic bounds as is, for such phases, yields a very pessimistic bound on CPI. For this reason, we isolate such points of high variation of CPI using a PC signature, which is a triple consisting of a PC bitmap, IC and CPI value, tracked for every x iterations of a loop. The bitmap encodes path information in a highly compressed manner and helps distinguishing loop iterations that execute different paths. The PC bitmap and IC values are observed to correspond to CPI values and are hence used to classify a phase further into smaller sub-phases. At p=0.99, using signatures, the average pessimism of WCET estimates across all benchmarks improves by 9%, 23% and 33% compared to estimates obtained by *Chronos* on *Simplest*, *Inorder_complex* and *Complex* respectively.

In some programs, CPI variation may not correspond to instruction execution patterns but be more influenced by the data it accesses and the order in which it accesses it. Hence we

make a provision to split a sub-phase into smaller sub-phases based on allowed coefficient of variation of CPI within a sub-phase. Further refinement based on controlling CPI variance within a subphase to 50%, 10%, 5% and 1% of its original value yields 12.9%, 13.1%, 13.1%, 13.1% improvement on *Simplest* architecture. On *Inorder_complex* and *Complex* architectures, the corresponding improvements are 38%, 40%, 41%, 43% and 46%, 47%, 50%, 52%.

The proposed probabilistic WCET analyzer is compared with a commercial probabilistic WCET analyzer, *RapiTime*. Comparisons are done at two levels of instrumentation supported by *RapiTime*. START_OF_SCOPES which is a coarse level and FULL which is a fine level of instrumentation. Compared to *RapiTime*, the average pessimism of WCET obtained by our technique based on PC signatures across all benchmarks at p=0.99 improves by 7% when programs are instrumented at START_OF_SCOPES granularity. Program phase behavior helps us to achieve this with only 10.3% of instrumentation points used by *RapiTime*. Further refinement based on controlling CPI variance within a subphase to 50%, 10% and 5% of its original value yields an improvement of 37%, 49% and 51% respectively. Any further refinement yields marginal improvement. WCET analysis based on signatures takes about 3/4ths of the time taken by *RapiTime* using START_OF_SCOPES. Further refinement based on controlling CPI variance takes 30% more time than RapiTime.

When *RapiTime* instruments at FULL granularity, the average pessimism obtained by our technique based on signatures is more pessimistic by 10.6%. However further refinement based on controlling CPI variance of a subphase to 50% and 10% of its original value yields 18% and 32% improvement. Any further refinement yields marginal improvement. Use of program phase behavior enables us to achieve this result with only 12% of the instrumentation points used by *RapiTime*. WCET analysis based on signatures takes half the time taken by RapiTime using FULL instrumentation. Further refinement based on controlling CPI variance takes about 3/4ths of the time taken by RapiTime.

The trace analysis time is found to be a major factor in deciding the overall WCET analysis time. In case of the phase based technique, the number of phases have an equal influence on the trace analysis time. However, since the results of trace analysis of one phase is not dependent on the other phase, traces of different phases can be analyzed in parallel thereby reducing the overall WCET analysis time. A parallel implementation based on these lines is observed to speedup the analysis time by factor of 1.98, 3.68, 4.71, 5.5 with 2 threads, 4 threads, 6 threads,

8 threads respectively.

The proposed technique is also implemented on a native platform using hardware performance counters accessed by performance API (PAPI). This feature is useful for programs that take a long time to simulate as simulation can be upto 10 times slower than native execution. Unless there is special hardware support to record PC addresses with events, we can have only partial PC signatures that store only IC, CPI information for groups of iterations. Use of partial signatures is observed to increase pessimism by 26-41% compared to estimates obtained by refinement based on full signatures but is more accurate than unrefined estimates by 8-22%.

Lastly, the homogeneity of CPI within a phase can be used in estimating the worst case remaining execution time of a program run with a specific input well before the program finishes execution. Predicting execution time early prevents holding onto resources for a longer time and leads to better resource utilization

The proposed technique is evaluated with respect to several desirable characteristics of a WCET analyzer which was touched upon in Chapter 2 in Tables 9.1 to 9.3 as follows. Most measurement based WCET analyzers are retargetable and hence this characteristic is not discussed in these tables. A comparison with some of the other measurement based tools is also presented.

Table 9.1: Comparison of proposed technique with other measurement-based tools with respect to desirable characteristics of WCET analyzers.

Proposed Technique	**RapiTime**	**Segments (TU Vienna)**	**IPG**	**GameTime**	**SWEET**
			Accuracy		
If CPI is stable, estimates are tight even at high probability. Multiple levels of refinement exist that can increase accuracy. Different granularity of instrumentation does not have significant impact on accuracy unlike RapiTime and Segments.	FULL estimates are more accurate than START-_OF_SCOPES estimates. With higher probability, the pessimism is generally observed to be high.	If segments are too small, the estimates can get pessimistic.	Pessimism at leaf node level gets propagated up to root. If a program has many procedure calls, estimates can get pessimistic.	For simple architectures with good timing predictability, perturbation is small and hence probability of accurate estimates is high.	Accurate for simple architectures with only L1 instruction cache and simple to moderate pipelines with bounded long timing effects and without any anomalies.
			Safety		
Usage of SWIC ensures higher degree of safety.	As basic block frequencies obtained using measurement is used to compute WCET, safety depends on coverage.	Can be unsafe due to insufficient state coverage.	Can be unsafe due to insufficient coverage by inputs.	As input coverage is based on paths, it has a higher chance of safety than other measurement based approaches that cover paths only within a segment.	The technique uses more of static analysis than measurements. Static analysis is used to predict values, branch prediction outcomes. Only pipeline analysis is performed by carefully controlled simulations using cycle accurate simulators.
			Non-intrusive Instrumentation		
Phase behavior allows us to use sparse instrumentation. We can afford to use source level instrumentation because of this.	START-_OF_SCOPES instruments the program at fewer points compared to FULL but is observed to give very pessimistic estimates. FULL provides accurate estimates but instruments every basic block.	Provides adaptable instrumentation. User can vary size of segments. However larger segments imply higher number of paths which will increase input test generation time.	This technique deals with sparse instrumentation. Given a few arbitrary ipoints, IPG models flow of information from one point to another.	Works with only measured end to end path execution times. Hence level of instrumentation required is low.	Not purely measurement-based. Only pipeline analysis is carried out using measurements at the basic block level.

Table 9.2: Comparison of proposed technique with other measurement-based tools with respect to desirable characteristics of WCET analyzers.

Proposed Technique	**RapiTime**	**Segments (TU Vienna)**	**IPG**	**GameTime**	**SWEET**
Time taken to estimate WCET					
Traces are small as phases span thousands of instructions. Hence analysis time is small. SWIC derivation and estimation of worst case CPI can be done in parallel. Further, trace analysis of each phase can be done in parallel.	Depending on trace size, analysis takes time to extract path information from time stamped traces. Time has been shown to be a direct function of non compressed trace size. Time stamps can overflow and hence need to be specially taken care of.	Input generation based on model checking takes larger time.	Trace parsing takes long time as time stamps record only sparse ipoints and the whole path has to be reconstructed.	Checking feasibility of basis paths and generation of basis paths take longest time. Number of iterations run by the algorithm to learn about the environment also determines the time.	Cannot be compared directly as it predominantly involves static analysis more than measurements.
Scalability					
Large number of phases observed to increase analysis time. There is inherent parallelism in the technique that can be exploited to make it scalable. For instance, parallel estimation of worst case IC (SWIC) and worst case CPI can be carried out and estimation of worst case CPI of several phases can be done in parallel. IPET is used to derive SWIC in this work. SWIC can also be derived using tree based schema or graph theoretical algorithms like longest path search.	START-_OF_SCOPES is more scalable compared to FULL. If trace size is large analysis can take a lot of time. In this technique path analysis and cost derivation is inseparable.	This technique also uses IPET. Larger segments make input test generation less scalable.	This technique also uses IPET and gives more accurate results compared to when ipoints are used with tree based schema. Using IPET, location of ipoints is insensitive to estimation accuracy. Alternatively, tree based schema, Itree can be used to obtain estimates but less accurate and highly sensitive to location of ipoints.	Number of tests to be run is polynomial in input size. Computing basis paths is time consuming as it involves feasibility checking using integer programming and SMT solving which is not easily scalable.	Path based technique is scalable but gives less accurate estimates. IPET gives more accurate estimates but is less scalable compared to the path based technique[6].
Computation of other related information apart from WCET					
Mapping WCET to path is difficult, however phases that are likely to lie on the worst case path can be obtained with little effort.	Path mapping is very clear, the tool is user friendly and has an excellent GUI and is used extensively in development and architectural exploration studies.	Since the technique uses IPET, the worst case path cannot be reconstructed unless the whole program is a segment (max_seg).	Paths can be reconstructed using ipoints hence worst case path can be identified.	Essentially path based hence worst case path can be obtained.	Using path based representation can help obtain worst case path.

Table 9.3: Comparison of proposed technique with other measurement-based tools with respect to desirable characteristics of WCET analyzers.

Proposed Technique	**RapiTime**	**Segments (TU Vienna)**	**IPG**	**GameTime**	**SWEET**
Other Remarks					
Since phase markers are tied to call loop boundaries, compiler optimizations are not a threat. The technique is simple, intuitive and highly transparent to the user and works seamlessly across all architectures. Estimates can get pessimistic for programs that are highly structurally complex (where SWIC >> MIC). In such cases, exhaustive inputs can be applied and MIC can be used instead of SWIC. High variation of CPI also causes pessimism which can be remedied by applying refinement to phases.	Unified contexts improve accuracy but take more time than separated contexts. Time stamps can overflow for lengthy executions. Source code level instrumentation especially at FULL level can impact program execution time. At such times, object level tracing has to be applied to reduce instrumentation overheads. FULL level can generate very large traces thereby increasing analysis time.	Infeasible paths cutting across segments need to be specially handled. The tool performs static analysis at the source code level hence needs assurance from the compiler that it will not significantly alter the structure of the program.	Ipoints to path reconstructibility is complex. Additional constraints need to be added to handle infeasible paths.	Loops have to be completely unrolled upto their safe bounds. Analysis assumes that the timing of the program depends only on control flow although timing can also depend on several characteristics of data that do not determine control flow.	To make use of automatic flow analysis, program has to be run with the SWEET research compiler. Pipeline analysis requires a highly controlled cpu model and works for pipelines with small to medium complexity.

9.1 Future Work

In this section we outline a few directions along which our work can be extended.

- The techniques proposed in this thesis apply to single core architectures. Phase behavior has been applied in multicore architectures too, though in different contexts such as core assignment[84], data placement and cache performance improvement[58]. A similar study can be carried out on such advanced architectures to estimate WCET. Since timing is to be estimated, the effects of cache synchronization and bus traffic have to be carefully accounted for. As several threads of a program can run in parallel on several cores, the timing model would be much more complicated than what was discussed in the thesis. Many cyber physical systems are distributed in nature. It would be interesting to see if we can apply phase behavior in such systems.

- The methods proposed to compute the theoretical upper bound described in this thesis use absolute loop bounds. However, we could assign loop variables to loops and derive bounds in the form of parametric timing equations which will help us obtain a parametric WCET estimate[70, 67]. Depending on the loop bounds, the WCET appropriately would change.

- It would be very useful if we could model the architecture aspects such as cache size, block size etc as part of the timing model, such that, we could predict WCET estimates for small variations in these parameters without having to redo the whole process of WCET analysis. Such an implementation could be very useful in architecture design exploration studies when different architectures are evaluated to pick the appropriate one for an application.

- Phases are seen in other parameters such as energy and power as well[25]. A similar model can be built to estimate worst case energy that would be very useful in embedded systems that have strict constraints with respect to power consumption and energy used.

In closing, this thesis proposes several techniques that can give accurate worst case execution time estimates associated with a particular probability for programs exhibiting phase behavior with minimal instrumentation. The thesis also describes a technique to predict the dynamic remaining worst case execution time of a program run. Phases have been used in various other

applications such as architecture simulation reduction, memory foot print reduction, power and energy optimizations and the like. This work describes how to apply phases in the context of timing analysis. The thesis also proposes a correlation based technique that exploits the inherent correlation between IC and CPI of a program that can give approximate WCET estimates in programs that do not exhibit any phase behavior.

Appendix A

Chronos: Specifics and Usage

Chronos[94] is a static WCET analysis tool developed by the embedded software research group at the National University of Singapore. It is released in the public domain and can be downloaded from [99]. Chronos has been widely used in the real time and embedded systems community for evaluation and comparison studies. We choose Chronos for comparison, as it models the same simplescalar MIPS architecture that we use for carrying out our experiments. Since Chronos is a static WCET analyzer, the tool provides an upper bound on the WCET estimate of a program which can never be exceeded by any of the program runs. The Chronos package consists of the main tool, linear programming solver based on LP[100], modified source code of Simplescalar Version 3.0 [113], modified version of GCC Version 2.7.3 along with a few binaries that comes along with Simplescalar- simpleutils-990811.tar.gz and simpletools-2v0.tgz. The details of installation can be found in [100]. Apart from these set of binaries, Chronos comes with a set of example programs and analysis outputs of these programs along with details on how the estimates were obtained. These examples help the user to quickly start using the tool with ease.

Chronos builds the program CFG out of the binary and carries out microarchitecture modeling which computes the cost of executing each basic block on the given architecture supplied in a config file. The format of specifying architecture components is similar to the format used by Simplescalar. The inputs to Chronos are hence the binary and the architectural specification given in the config file along with annotated loop bound information specified in a .cons file. In most cases, Chronos automatically derives infeasible path information by structural analysis[87]. The user can however populate the .cons file with additional flow constraints on

the CFG including infeasible paths. Chronos estimates WCET using the implicit path enumeration technique by solving an ILP problem. The individual estimates of cost of executing each basic blocks form the constants in the objective equation constructed for the purpose of estimating WCET. The variables of the objective equation indicate the execution frequencies of basic blocks which are to be maximized. The tool carries out structural analysis on the CFG and forms additional linear constraints based on flow equations of the CFG and creates an .lp file which consists of these equations in a format acceptable to the ILP solver. The user can then invoke the ILP solver which processes these equations and maximizes the objective equation subject to the constraints. The maximal value of the objective equation is the WCET estimate.

```
proc[0] cfg:
 0 : 004001f0 : [  1 ,    ]
 1 : 00400208 : [  2 ,  3  ]
 2 : 00400218 : [    ,  4  ]
......

proc[1] cfg:
 0 : 00400258 : [  1 ,  4  ]
 1 : 004002c8 : [  2 ,    ]
 2 : 004002f0 : [  3 ,    ]  P0
 3 : 00400310 : [  4 ,  2  ]
.....

proc[2] cfg:
 0 : 004005e8 : [  1 ,    ]  P1
 1 : 00400630 : [  2 ,    ]  P1
 2 : 00400680 : [    ,    ]
```

Figure A.1: Sample CFG output by Chronos.

Chronos tool can be used to dump program CFG. A sample CFG output is as shown in Figure A.1. Each procedure is addressed as proc[n]. Each entry in a procedure signifies a basic block. The binary address for the first instruction in the basic block is also provided as shown. At the end of each entry, there may be a set of numbers enclosed in square brackets which indicate the basic blocks connected to the current basic block by edges. A procedure call is indicated by a Pn at the end of an entry. For simple loops, Chronos computes loop bound information automatically. For complex while loops or loops whose bounds cannot be easily ascertained, in addition to the architectural specification in the config file, the user must

specify loop bound information in a .cons file in the following format.

```
C<pi>.<bj> = Bk
C<pi>.<bj> - Bj C<pk>.<bm> < 0
```

Where, pi indicates the procedure number, bj indicates the basic block number and Bk is the constant bound which is specified by the user. The first constraint specifies the bound on the execution frequency of `C<pi>.<bj>`. The second constraint specifies that `C<pi>.<bj>` executes atleast `Bj` times block `C<pk>.<bm>`.

The user can dump the program CFG using the following command.

```
commandline:> est -run CFG <benchmarkbinary>
```

Inorder to perform WCET analysis of a program, Chronos requires the program binary, the architectural specification in the .config file and the set of constraints specified in the .cons file. By running the following command

```
commandline:> est -config <benchmarkbinary.config> <benchmarkbinary>
```

an .lp file is generated which consists of the linear programming equation and the set of constraints derived from the CFG structure. By running the following command,

```
commandline:> lp_solve -rxli xli_CPLEX <benchmarkbinary.lp>
```

the final WCET estimate is obtained as output.

Appendix B

RapiTime: Specifics and Usage

RapiTime[102] is a commercial measurements based WCET analyzer developed by the research group at the University of York. RapiTime comes along with a user friendly guide that contains the installation instructions in detail. Apart from the actual tool, RapiTime also provides a tutorial package which consists of a set of examples which have been analyzed for WCET using RapiTime. RapiTime is a hybrid measurements based analyzer as it combines the static model of the software with detailed measurements of timing behavior obtained on the target system. The static model of the program is built by automatically parsing the source code to identify function boundaries, conditional structures and control flow. The static model is then populated with the times taken to execute each part of the software, in this case, a basic block. These times are derived from a *trace* of execution times of instrumented portions of the program on the target. The trace is a list of tuples consisting of the identifier of the basic block and its corresponding timestamp. The difference between the timestamp of the current basic block and the next indicates the time taken to execute the current basic block.

RapiTime is a measurements based tool and hence provides support to instrument program source code. RapiTime supports instrumentation at two levels- START_OF_SCOPES and FULL. RapiTime also provides instrumentation at the FUNCTION level but this option has not been observed to yield accurate WCET estimates. Hence we do not consider this option for our study. FULL is the finest level of instrumentation as it is added at the beginning and end of each block of code. This allows each individual sub-path through the code to be identified and so enables detailed code coverage information to be obtained. FULL instrumentation provides maximum accuracy in the computed worst case execution times and is the default

profile. START_OF_SCOPES is a coarser level of instrumentation compared to FULL. In case of START_OF_SCOPES, instrumentation points are added at the beginning of blocks of code but not at the end. START_OF_SCOPES instrumentation can result in some increased pessimism in the computed worst-case execution times.

The instrumentation code upon execution, writes the identifier of every instrumentation point to an output port and the rapitime trace box adds a timestamp for each of these instrumentation points and stores the trace data. The resulting trace is then combined with the static model of the program to generate reports. The trace data encodes paths that have been traversed during execution and hence can be used to obtain various coverage information along with average, best case and worst case execution times. RapiTime is based on probabilistic worst case execution time analysis[29] that computes an execution time profile for each block of code which contains distribution of execution times. Two types of distribution are stored. One is the set of measured execution times observed and the other is the set of computed distribution that shows the probability of the execution time exceeding certain values when the worst case path through the code is taken.

Inorder to estimate WCET using RapiTime, we need to take the following steps. The source code of the application that we want to analyze is coded in `my_application.c`. This file should contain a call to `my_main_function` which acts as a test driver and invokes the application with various test data. All declarations can be placed in `my_application.h`. If we do not want RapiTime to analyze a particular function, `procX`, we should include the following declaration within `my_application.h`

```
#pragma RPT instrument ("procX", FALSE);
```

We should also exclude the test driver, `my_main_function` from the analysis as the purpose of this function is only to test the application with various test data.

```
#pragma RPT instrument("my_main_function", FALSE);
```

Instrumentation options for functions that are to be analyzed should be specified in `my_rpt_annotations.h` as follows, depending on the level of instrumentation chosen by the user:

```
#pragma RPT instrument ("funcX", "TRUE", "FULL");
#pragma RPT instrument ("funcY", "TRUE", "START_OF_SCOPES");
....
```

The main function should contain the following declarations.

```
#include "rpt.h"
// include all other header files
#pragma RPT instrument ("main", FALSE);
int main(void);
{
RPT_init(void);
my_main_function();
RPT_Output_Trace();
return 0;
}
```

The root function of analysis must be specified by setting the value of ROOT in the makefile to the name of the function desired. PROG must be specified to be the name of the application, in this case, `my_application`. OBJS should be specified as the set of object files and ANNOT_FILE should be specified to be the annotation file, `my_rpt_annotations.h`. The usage can be easily understood by looking at the example code that comes as part of the tutorial package.

Once, `make` is executed, the program is processed in several stages. The source code is converted to an instrumented code by the binary utility tool, *cins* for C code which is then compiled and linked. *cins* also extracts structural information from each instrumented C source file, which is stored in the form of an .xsc file corresponding to each C source file. *xstutils*, another binary utility tool, analyzes the structural information contained in the .xsc file and generates a single RapiTime data base file with the extension, .rtd. *xstutils* is responsible for understanding annotations and thereby instruct RapiTime to behave in an appropriate way. *xstutils* is responsible for signalling any errors with respect to instrumentation such as there not being enough instrumentation points.

The instrumented executable can either be run using a simulator like SimIt-ARM-2.1[114] that is used by RapiTime by default. RapiTime can also be interfaced to run on a native platform to carry out measurements. Once the program is run, a trace file, *trace.txt* is generated as output. Since instrumentation points can cover every basic block, the trace file size can be

huge and is hence compressed into *trace.rpz* by RapiTime.

Once the trace is generated, *traceutils* pre-processes the trace data. For some large programs, the time stamps can exceed the size of timers used and can wrap around. This situation is handled by executing *traceutils* in the following manner.

```
commandline:> traceutils.exe trace.txt -o trace.txt -w <bitsize>
```

where <bitsize> should be a power of 2 and is large enough to ensure wraparound is fixed.

Finally the pre-processed trace is parsed by a *traceparser* which combines the computed time measurements of each basic blocks along with the measured and computed distribution information and program structure to give the final WCET estimates. RapiTime, being a probabilistic WCET analyzer generates estimates at various probability values. The end user can pick the estimate at the desired probability. Apart from WCET information, RapiTime also generates various reports that give valuable information about the program with respect to coverage by instrumentation points, call-tree which shows potential functions lying on the worst case path, execution time profiles that show variability in execution times due to hardware effects, untested code, code not on the worst case path, contribution of each basic block to the worst case execution time, observed distribution of execution times for the worst case path, links to source code, search and sort facilities on all data, variability in end to end execution times across different invocations.

References

[1] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg and F. Mueller. Virtual simple architecture (VISA):Exceeding the complexity limit in safe real-time systems. *In Proceedings of IEEE/ACM ISCA*, 2003, pages 350-361.

[2] Adam Betts and Guillem Bernat. Tree-Based WCET Analysis on Instrumentation Point Graphs. *In Proceedings of ISORC*, 2006.

[3] Adam Betts, Nicholas Merriam and Guillem Bernat. Hybrid measurement-based WCET analysis at the source level using object-level traces. *In Proceedings of WCET*, 2010, pages 54-63.

[4] A. C. Shaw. Reasoning about time in higher-level language software. *In IEEE Transactions on Software Engineering 1(2)*, 1989, pages 875-889.

[5] Amine Marref and Guillem Bernat. Towards Predicated WCET Analysis. *In Proceedings of WCET*, 2008.

[6] Andreas Ermedahl, Friedhelm Stappert and Jakob Engblom. Clustered Worst-Case Execution-Time Calculation. *In IEEE Trans. Computers 54(9).* 2005, pages 1104-1122.

[7] Andreas Ermedahl, Johan Fredriksson, Jan Gustafsson and Peter Altenbernd. Deriving the Worst-Case Execution Time Input Values. *In Proceedings of ECRTS*, 2009, pages 45-54

[8] Andreas Ermedahl. A Modular Tool Architecture for Worst-Case Execution Time Analysis. *Ph.D. thesis, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden*, ISBN 91-554-5671-5.

[9] Antoine Colin and Stefan M Petters. Experimental Evaluation of Code Properties for WCET Analysis. *In Proceedings of RTSS*, 2008, pages 190-199.

[10] Antoine Colin and Isabelle Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *In the Journal of Real-Time Systems, Volume 18*, 2000, pages 249-274.

[11] Archana Ravindar and Y. N. Srikant. Implications of Program Phase Behavior on Timing Analysis. *In Proceedings of INTERACT (HPCA)*, 2011, pages 71-79.

[12] A. Srivastava and A. Eustace. ATOM: A System for building customized program analysis tools. *In Proceedings of PLDI*, 1994.

[13] Bob Davies, Jean-Yves Bouguet, Marzia Polito and Murali M. Annavaram. iPART : An Automated Phase Analysis and Recognition Tool. *Technical Report.,IR-TR-2004-1.*

[14] Changpeng Fang, Steve Soner and Onder Zhenlin Wang. Instruction based memory distance analysis and its application to optimization. *In Proceedings of the 14 th International Conference on Parallel Architectures and Compilation*, 2005.

[15] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm, Reliable and Precise WCET Determination for a Real-life Processor. *In Proceedings of first Workshop on Embedded Software (EMSOFT), Vol. 2211 of LNCS*, 2001, pages 469485.

[16] Christian Ferdinand and Reinhard Wilhelm: Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *In Real-Time Systems 17(2-3)*, 1999, pages 131-181.

[17] Christian Ferdinand, Florian Martin, Reinhard Wilhelm and Martin Alt, Cache Behavior Prediction by Abstract Interpretation. *In Science of Computer Programming 35(2-3)*, 1999, pages 163-189.

[18] Florian Martin, Martin Alt, Reinhard Wilhelm and Christian Ferdinand. Analysis of Loops. *In Proceedings of 7th International Conference on Compiler Construction, LNCS 1383*, 1998, pages 80-94.

[19] Christoph Berg, Jakob Engblom, Reinhard Wilhelm: Requirements for and Design of a Processor with Predictable Timing. *In Dagstuhl Seminar Proceedings: Design of Systems with Predictable Behavior*, 2004.

[20] Christopher A. Healy and David B. Whalley. Automatic Detection and Exploitation of Branch Constraints for Timing Analysis. *In IEEE Transactions on Software Engineering*, Volume 28(8), 2002, pages 763-781.

[21] Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *In Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, 1977, pages 238-252.

[22] David Griffin and Alan Burns. Realism in Statistical Analysis of Worst Case Execution Times. *In Proceedings of WCET*, 2010, pages 44-53.

[23] E. Vivancos,C. Healy, F. Mueller and D. Whalley. Parametric Timing Analysis. *In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems.*, 2001, pages 88-93.

[24] Frank Mueller. Timing Analysis for Instruction Caches. *In the Journal of Real-Time Systems, Volume 18*, 2000, pages 217-247.

[25] Frederik Vandeputte, Lieven Eeckhout and Koen De Bosschere. Exploiting program phase behavior for energy reduction on multi-configuration processors. *Journal of Systems Architecture, Volume 53(8)*, pages 489-500.

[26] Frederik Vandeputte and Lieven Eeckhout. Phase Complexity Surfaces: Characterizing Time-Varying Program Behavior. *In High Performance Embedded Architectures and Compilers*, 2008, pages 320-334.

[27] F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real-time systems. *In Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005, pages 33-40.

[28] Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder and Timothy Sherwood. Using Machine Learning to Guide Architecture Simulation. *Journal of Machine Learning Research*, 2006, Volume 7, pages 343-378.

[29] Guillem Bernat, Antoine Colin and Stefan M Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. *In Proceedings of RTSS*, 2002, pages 279-288.

[30] Guillem Bernat, Antoine Colin and Stefan M Petters. pWCET: A Tool for Probabilistic Worst-case Execution Time Analysis of Real-time Systems. *Technical Report YCS-2003-353, University of York, England, UK.*

[31] Harry Frank and Steven. C. Althoen, Statistics: Concepts and Applications, *Cambridge University Press*, 1994.

[32] Hemedra S. Negi, Abhik Roychoudury, and Tulika Mitra. Simplifying WCET analysis by code transformations. *In Proceedings of WCET*, 2004.

[33] Henrik Theiling, Christian Ferdinand and Reinhard Wilhelm, Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *In Real-Time Systems 18(2/3)*, 2000, pages 157-179.

[34] Jan Gustafsson and Andreas Ermedahl. Experiences from Applying WCET Analysis in Industrial Settings. *Proceedings of the 10th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC)*, 2007.

[35] Jan Gustafsson. Usability Aspects of WCET Analysis. *In Proceedings of ISORC*, 2008, 346-352.

[36] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger and Bernd Becker, A Definition and Classification of Timing Anomalies. *In Proceedings of WCET*, 2006.

[37] Jakob Engblom. Processor Pipelines and Static Worst-Case Execution Time Analysis. *Ph.D. thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden.*

[38] Jeffery P. Hansen, Scott A. Hissam and Gabriel A. Moreno. Statistical-Based WCET Estimation and Validation. *In Proceedings of WCET*, 2009, pages 123-133.

[39] Jeremy Lau, Erez Perelman and Brad Calder. Selecting Software Phase Markers with Code Structure Analysis. *In Proceedings of CGO*, 2006, pages 135-146.

[40] Jeremy Lau, Jack Sampson, Erez Perelman, Greg Hamerly and Brad Calder. The strong correlation between code signatures and performance. *In IEEE International Symposium on Performance Analysis of Systems and Software*, 2005.

[41] Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood and Brad Calder. Motivation for Variable Length Intervals and Hierarchical Phase Behavior. *In Proceedings of ISPASS*, 2005, pages 135-146.

[42] Jinpyo Kim, Sreekumar V. Kodakara Wei-chung Hsu, David J. Lilja and Pen-chung Yew. Dynamic Code Region (DCR)-based Program Phase Tracking and Prediction for Dynamic Optimizations. *In Proceedings of International Conference on High Performance Embedded Architectures and Compilers*, 2005.

[43] J. Deverge and Isabelle Puaut. Safe measurement-based WCET estimation. *In Proceedings of WCET*, 2005.

[44] J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing and R. Wilhelm. Automatic Identification of Timing Anomalies for Cycle-accurate Worst-case Execution Time Analysis. *In Proceedings of the Ninth IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2006, pages 15-20.

[45] John Demme and Simha Sethumadhavan. Rapid Identification of Architectural Bottlenecks via Precise Event Counting. *In Proceedings of ISCA*, 2011, pages 353-364.

[46] John L. Henessy and David A. Patterson. Computer Architecture, Fifth Edition: A Quantitative Approach. *The Morgan Kaufmann Series in Computer Architecture and Design*, 2011.

[47] Kamran Ghani and John A. Clark. Automatic Test Data Generation for Multiple Condition and MCDC Coverage. *In Proceedings of ICSEA*, 2009, pages 152-157.

[48] K. Beyls and E. H. D Hollander. Reuse distance as a metric for cache behavior. *In Proceedings of IASTED*, 2001.

[49] Lamport, L. LaTeX: A Documentation System, Addison-Wesley Publishing Company, 1986.

[50] Laurent David and Isabelle Puaut. Static Determination of Probabilistic Execution Times. *In Proceedings of ECRTS*, 2004, pages 223-230.

[51] Liangliang Kong and Jianhui Jiang. A Worst-case execution time analysis approach based on independent paths for ARM programs. *In Wuhan University Journal of Natural Sciences, Volume 17, No. 5*, 2012, pages 391-399.

[52] Matteo Corti, Roberto Brega and Thomas Gross. Approximation of Worst-Case Execution Time for Preemptive Multitasking Systems, *In Proceedings of LCTES, ACM Workshop on Languages, Compilers and Tools for Embedded Systems, volume 1985 of Lecture Notes in Computer Science*, Springer-Verlag, 2000, pages 178-198.

[53] Michael J Hind, V T Rajan, Peter F Sweeney. Phase Shift Detection: A Problem Classification. *IBM Research Report RC-22887*, 2003.

[54] Michael Zolda, Sven Bünte and Raimund Kirner. Context-Sensitive Measurement-Based Worst-Case Execution Time Estimation. *In Proceedings of RTCSA*, 2011, pages 243-250.

[55] Michael Zolda, Sven Bünte and Raimund Kirner. Towards Adaptable Control Flow Segmentation for Measurement-Based Execution Time Analysis. *In Proceedings of RTNS*, 2009.

[56] Michael Zolda, Sven Bünte and Raimund Kirner. Context-Sensitivity in IPET for Measurement-Based Timing Analysis. *In Proceedings of ISoLA*, 2010, pages 487-490.

[57] Murali Annavaram, Ryan Rakvic, Marzia Polito, Jean-yves Bouguet, Richard Hankins and Bob Davies. The fuzzy correlation between code and performance predictability, *In Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, 2004, pages 93-104.

[58] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero and Alexander V. Veidenbaum. Improving Cache Management Policies Using Dynamic Reuse Distances. *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pages 389-400.

[59] Paul Keim, Amanda Noyes, Andrew Ferguson, Joshua Neal and Christopher Healy. Extending the Path Analysis Technique to Obtain a Soft WCET. *In Proceedings of WCET*, 2009.

[60] Raimund Kirner. The Programming Language wcetC. *Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria.*

[61] Raimund Kirner and Michael Zolda. Compiler support for measurement-based timing analysis. *In Proceedings of WCET*, 2011.

[62] Rajiv Gupta and Prabha Gopinath. Correlation Analysis Techniques for Refining Execution Time Estimates of Real-Time Applications. *In Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, 1994, pages 54-58.

[63] R. Arnold, F. Mueller, D. Whalley and M. Harmon, Bounding Worst-Case Instruction Cache Performance. *In Proceedings of RTSS*, 1994, pages 172-181.

[64] Reinhard von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Ploedereder, Armelle Bonenfant, Hugues Cassé, Sven Bünte, Wolfgang Fellger, Sebastian Gepperth, Jan Gustafsson, Benedikt Huber, Nazrul Mohammad Islam, Daniel Kästner, Raimund Kirner, Laura Kovacs, Felix Krause, Marianne de Michiel, Mads Christian Olesen, Adrian Prantl, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Simon Wegener, Michael Zolda and Jakob Zwirchmayr. The WCET Tool Challenge. *In Proceedings of 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2011, `http://www.mrtc.mdh.se/index.php?choice=publications&id=2620`

[65] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Muller, Isabelle Puaut, Peter Puschner, Jan Staschulat and Per Stenström. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *In ACM Transactions on Embedded Computing Systems (TECS), Volume 7(3)*, 2008, pages 1-53.

[66] Sanjit A. Seshia and Jonathan Kotker. GameTime: A Toolkit for Timing Analysis of Software. *In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011, pages 388-392.

[67] Sebastian Altmeyer, Christian Humbert, Björn Lisper and Reinhard Wilhelm. Parametric Timing Analysis for Complex Architectures. *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications*, 2008, pages 367-376.

[68] Sheldon M Ross. Introduction to Probability and Statistics for Engineers and Scientists. *Wiley*, 2009.

[69] Sheayun Lee, Jaejin Lee, Chang Yun Park and Sang Lyul Min. A flexible tradeoff between code size and WCET using a dual instruction set processor. *In Proceedings of the 8th InternationalWorkshop on Software and Compilers for Embedded Systems (SCOPES). Vol. 3199 of Lecture Notes in Computer Science*, 2004, pages 244-258.

[70] Sibin Mohan, Frank Mueller, William Hawkins, Michael Root, Christopher Healy and David Whalley. ParaScale: Exploiting Parametric Timing Analysis for Real-Time Schedulers and Dynamic Voltage Scaling. *Proceedings of the IEEE Real-Time Systems Symposium*, 2005, pages 233-242.

[71] S. Edgar and Alan Burns. Statistical Analysis of WCET for Scheduling. *In Proceedings of RTSS*, 2001, pages 215-224.

[72] Srikant, Y. N. and Shankar, Priti. The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition. *CRC Press, Inc.*, 2007, ISBN:142004382X 9781420043822

[73] Stefan Stattelmann and Florian Martin. On the Use of Context Information for Precise Measurement-Based Execution Time Estimation. *In Proceedings of WCET*, 2010, pages 64-76.

[74] Stefan Schaefer, Bernhard Scholz, Stefan M. Petters and Gernot Heiser. Static Analysis Support for Measurement-based WCET Analysis, *In IEEE Conference on Embedded and real-time computing systems and applications, Work-in-progress session*, 2006.

[75] Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudury, Timon Kelter, Peter Marwedel and Heiko Falk. A Unified WCET Analysis Framework for Multi-core Platforms. *In IEEE Real-Time and Embedded Technology and Applications Symposium*, 2012, pages 99-108.

[76] Sudipta Chattopadhyay and Abhik Roychoudury. Unified Cache Modeling for WCET Analysis and Layout Optimizations. *In Proceedings of RTSS*, 2009, pages 47-56.

[77] Sven Bünte. A Benchmarking Suite for Measurement Based WCET Analysis. *In ICSTW*, 2008, pages 353-356.

[78] Sven Bünte, Michael Zolda and Raimund Kirner. Let's get less optimistic in measurement-based timing analysis. *In Proceedings of SIES*, 2011, pages 204-212.

[79] Sven Bünte, Michael Zolda, Michael Tautschnig and Raimund Kirner. Improving the Confidence in Measurement-Based Timing Analysis. *In 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2011, pages 144-151.

[80] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *In Real-Time Systems Volume 17*, 1999, pages 183-207.

[81] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. *In Proceedings of the 20th IEEE RTSS*, 1999, pages 12-21.

[82] Thomas Lundqvist and Patrik Sandin. Towards a practical WCET analysis approach based on testing. *In Work-in-progress session of ECRTS*, 2008.

[83] Tushar Kumar, Jaswanth Sreeram, Romain Cledat and Santosh Pande. A profile-driven statistical analysis framework for the design optimization of soft real-time applications. *Proceedings of ESEC/SIGSOFT FSE*, 2007, pages 529-532.

[84] Tyler Sondag and Hridesh Rajan. Phase-based Tuning for Better Utilization of Performance-Asymmetric Multicore Processors. *In Proceedings of CGO*, 2011, pages 11-20.

[85] Usman Khan and Iain Bate. WCET Analysis of Modern Processors Using Multi-Criteria Optimisation. *In Proceedings of the 1st International Symposium on Search Based Software Engineering*, 2009, pages 103-112.

[86] Vivy Suhendra, Tulika Mitra, Abhik Roychoudury and Ting Chen. WCET centric data allocation to scratchpad memory. *In Proceedings of RTSS*, 2005, pages 223-232.

[87] Vivy Suhendra, Tulika Mitra, Abhik Roychoudury and Ting Chen. Efficient Detection and Exploitation of Infeasible Paths for Software Timing Analysis. *In Proceedings of DAC*, 2006.

[88] W. Liu and M. C. Huang. EXPERT: Expedited Simulation Exploiting Program Behavior Repetition. *In Proceedings of ICS*, 2004.

[89] W. Zhao, D. Whalley, C. Healy and F. Mueller. WCET code positioning. *In Proceedings of RTSS*, 2004, pages 81-91.

[90] Wegener, Joachim and Grochtmann, Matthias. Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing. *In the Journal of Real-Time Systems, Volume 15(3)*, 1998, pages 275-298.

[91] Wolf F. Behavioral Intervals in Embedded Software. *Kluwer Academic Publishers.* 2002.

[92] Wolf F., Ernst R. and Ye W. Path Clustering in Software Timing Analysis. *In IEEE Transactions on VLSI Systems 9(6)*, 2001, pages 773-782.

[93] W. Pugh. The Omega Test: A Fast ad Practical Integer Programming Algorithm for Dependence Analysis. *In ACM/IEEE Conference on Supercomputing*, 1991.

[94] Xianfeng Li, Yun Liang, Tulika Mitra and Abhik Roychoudury. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming, Vol. 69(1-3)*, 2007, pages 56-67, `http://www.comp.nus.edu.sg/~rpembed/chronos/download.html`

[95] Yan-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. *In IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, Vol. 16(12)*, 1997, pages 1477-1487.

[96] Yan-Tsun Steven Li. Cinderella 3.0 WCET analyzer. `http://www.princeton.edu/~yauli/cinderella-3.0/`

[97] Yue Lu, Thomas Nolte, Iain Bate and Liliana Cucu-Grosjean. A New Way about using Statistical Analysis of Worst-Case Execution Times. *In ACM SIGBED Review, 8(2)*, 2011.

[98] Yue Lu, Thomas Nolte, Iain Bate and Liliana Cucu-Grosjean. A Trace-Based Statistical Worst-Case Execution Time Analysis of Component-Based Real-Time Embedded Systems. *In Proceedings of ETFA*, 2011.

[99] `http://www.comp.nus.edu.sg/~rpembed/chronos/download.html`

[100] http://www.comp.nus.edu.sg/∼rpembed/chronos/chronos_manual.pdf

[101] http://icl.cs.utk.edu/papi

[102] http://www.rapitasystems.com

[103] http://www.mrtc.mdh.se/projects/WCC/2011/doku.php?id=bench:debie1

[104] http://www.mrtc.mdh.se/index.php?choice=publications&id=2620

[105] http://software.intel.com/en-us/intel-vtune-amplifier-xe

[106] http://www.spaceinfo.fi

[107] http://www.tidorum.fi/en/

[108] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[109] http://www.eecs.umich.edu/mibench/

[110] http://www.mathworks.in/products/matlab/

[111] http://www.absint.com

[112] http://www.bound-t.com

[113] http://www.simplescalar.com

[114] http://simit-arm.sourceforge.net/

Index