Sampling-based Program Execution Monitoring

by

Yanmeng Ba

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2010

Abstract

For its high overall cost during product development, program debugging is an important aspect of system development. Debugging is a hard and complex activity, especially in time-sensitive systems which have limited resources and demanding timing constraints.

System tracing is a frequently used technique for debugging embedded systems. A specific use of system tracing is to monitor and debug control-flow problems in programs. However, it is difficult to implement because of the potentially high overhead it might introduce to the system and the changes which can occur to the system behaviour due to tracing.

To solve the above problems, in this work, we present a sampling-based approach to program execution monitoring which specifically helps developers trace the program execution in time-sensitive systems such as real-time applications. We build the system model and propose three theorems which determine the sampling period or the optimal in different scenarios. We also design seven heuristics and an instrumentation framework to extend the sampling period which can reduce the monitoring overhead and achieve an optimal tradeoff between accuracy and overhead introduced by instrumentation. Using this monitoring framework, we can use the information extracted through sampling to reconstruct the system state and execution paths to locate the deviation. Based on the statistically significant data, we also model the trend of the sampling period with the instrumentation steps. Based on the modelling results, we devise a scheme for predicting the number of markers we need to reach a certain sampling period. Last, we build a tool chain to instrument and monitoring the software system and further prove the soundness of our approach.

Acknowledgements

The work presented in this thesis would not be possible if I did not have such a stimulating, creative and supportive supervisor, Professor Sebastian Fischmeister. Under his supervision, I feel happy and fulfilled by the growth in myself both academically and personally during the past two years. Majoring in Electrical Engineering before I came to UW, I appreciate Professor Fischmeister's patience and support which allowed me enough time to catch up and accomplish solid work. His acumen towards cutting-edge technologies and new methods along with his passion for pushing the envelope is encouraging. From his rigorous working attitude, I learned what defined good academia and integrity. I always believe that if a supervisor is willing to sit with his student to revise the student's paper sentence by sentence, that student is lucky. And I think I am the lucky one. I am also very grateful of the financial support provided by Professor Fischmeister, so that I can totally focus on my work and have a enjoyable life in Canada. The trip to Stockholm was fantastic and unforgettable. I would like to express my greatest thanks to Professor Fischmeister for giving me such an amazing learning experience in University of Waterloo.

I also would like to thank my dear supervisor Professor Yu Peng, in HIT back in China, for his constant understanding and spiritual support along my way, especially during my most difficult time in Canada. I know you are always there for me.

For me, my supervisors for years, my role models for life.

Special thanks to my dear friends in both China and Canada. I feel blessed to be always surrounded by you.

Finally, the greatest and most special thanks to my family. It does not matter where I am, every step I take, I know I am not alone. I love you all.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Driven by the evolution in the silicon and optical technologies, the field of computing has been developing rapidly for the past decades. The programs executing on computers have grown immensely in both functionality and size and are becoming more and more distributed. However, such programs and software systems pose a significant challenge for program testing, debugging and monitoring. In addition, software testing and debugging are expensive components of the software development and maintenance which take between 30 to more than 50 percent of the overall development budget [22, 31] and add considerable length to the development cycle. Thus, developing more efficient and more effective testing and debugging techniques becomes inevitable and imperative. Section 1.1 of this thesis motivates our research and gives a brief overview of the problem I am trying to solve. Section 1.2 presents the contributions of our work. Finally, Section 1.3 describes the organization of this thesis.

1.1 Motivation and Problem Overview

Traditional monitoring tasks involve instrumenting the code by inserting a full complement of probes into each instance of the software. For most such tasks, this approach requires probes at many points in the software, significantly increasing the program size and compromising its performance. Such overhead is in general unacceptable to users, and inapplicable in contexts in which resources are limited, such as embedded systems. Instead of collecting all the information from a program, recording information from only parts of program can decrease monitoring overhead and increase the efficiency of monitoring simple software. We can use probes to monitor and track the execution of program entities, such as variables and statements. Monitoring more sophisticated entities, such as execution paths, requires more sophisticated techniques.

Instrumentation is a key technique in many aspects of software development, such as software debugging and monitoring. The instrumented programs produce various information about the states of the program, such as data traces, that enable developers to locate the origins of bugs in the system under test. However, instrumentation and producing traces incur run-time overhead in the form of additional computation resources in terms of both space and execution time. This might impose critical issues for embedded systems whose computation resource is quite constrained. The overhead introduced by instrumentation may also interfere with system's timing and perturb its behaviour, which are unacceptable in context of hard real-time systems, such as heavily-loaded safety critical applications and time-sensitive applications, in which meeting the deadline and carrying out precise function at the specific time are entitled to have the top priority. In time-related *probe effects*, the overhead introduced by instrumentation prevents the developers from reproducing the misbehaviour which actually exists in the original defective program.

Targeting the above problems, we aim to develop a novel method with less and bounded overhead and provide a framework which assists more effective and efficient debugging and monitoring techniques of resource-constrained embedded systems. Our approach gathers run-time information using a sampling technique which records a subset of the program execution and events that occur, thus introducing much less overhead than regular monitoring yet providing a decent accuracy.

1.2 Thesis Contribution

The advantages of our approach are as follows: Compared with continuous monitoring, our sampling-based technique can greatly reduce the overhead introduced. Since the sampling rate is fixed, we can estimate and bound the overhead and impact on the system under test. Also, our approach can easily be combined with data-tracing methods. There are, however, several issues to be solved regarding sampling-based approaches:

First, we need to balance between overhead and correct reconstruction of the control flow. On one hand, we want to gather enough information to be able to reconstruct the execution path; on the other hand, monitoring should have only modest impact on the program.

Second, we need to address the problem of how many markers should be used. Each marker requires memory, so we want to use as few markers are possible.

Third, we need to devise an instrumentation algorithm that efficiently uses the available markers. Using markers well can reduce the number of required markers to achieve the target sampling period.

This paper makes several contributions to the issues mentioned above:

- We provide a formal framework that permits quantitative reasoning about many aspects involved in sampling-based mechanisms.
- We define optimality from both vertex and the whole control-flow graph perspectives.
- We provide theorems for termination conditions of instrumentation algorithms with an unlimited number of markers and with exactly one marker.
- We validate the general approach by proposing and comparing several algorithms for inserting markers into programs.
- We investigate interference among markers and propose tailored algorithms to compensate for this interference.

- We discuss a number of observations and insights obtained during the development of the algorithms.

- We devise a scheme for predicting the number of markers needed to reach a given sampling period and reason about the optimal tradeoff between markers and instrumentation steps.

- We build the tool chain to instrument and monitor the software system and further prove the soundness and effectiveness of our approach.

Besides debugging, sampling-based execution monitoring can also be used in performance profiling of software systems. Combined with tracing, it can give performance engineers a sufficiently detailed analysis of the system with low overhead [46], such as event relationships in time and reconstructing the dynamic behavior of a software system. In addition, sampling-based execution monitoring can be applied to code coverage testing [61] which finds the code exercised by a particular set of test input. Moreover, sampling-based execution monitoring can be a feasible and efficient technique for reducing the overhead while collecting profile information [42].

1.3 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 presents a survey of related work and gives an overview of the background material and past works. It first introduces testing and debugging techniques and defines some terminology. Then, it discusses monitoring, software instrumentation and program sampling.

- Chapter 3 first describes the problem targeted and then builds our system model (Section 3.1) which serves as the theoretical foundation for the thesis. Second, we propose two theorems which provide the termination conditions for instrumentations in different scenarios and give the corresponding examples to explain them (Section 3.2). We then propose a BFS-based algorithm which calculates the sampling period (Section 3.3). We analyze the interference between instrumentations in Section 3.4. Varying the interference and using the model, we experiment with different algorithms (Section 3.5) and interpret the results (Section 3.6).

- Chapter 4 first gives an overview of the tool chain developed to implement all aspects of the sampling based program execution monitoring. It then presents the details on how each element of the tool chain is implemented and the techniques involved in it.

- Chapter 5 provides a scheme for predicting the number of markers needed to achieve a certain sampling period and explains the scheme along with the modelling method when carrying out curve fitting. In addition, it presents the experiments from real bench marks and further proves the claims made in previous sections.

- Chapter 6 draws conclusions and discusses the possible future work.

Chapter 2

Background and Related Work

Software testing and debugging have great importance in software development since the quality of software system plays a more and more critical role in many aspects of society as well as people's everyday life. Intensive research has been done on software quality assurance. In this domain, numerous researchers have investigated software monitoring and instrumentation techniques which effectively assist software testing and debugging. However, with software becoming more and more complicated and the rapid evolution of operating environments, especially with the advent of embedded systems which have strict constraints on resources, continuous work is required to develop more efficient and cost-effective testing and debugging methods. This chapter presents previous works and papers that pertain to this thesis. Section 2.1 gives an overview of software testing and debugging including definitions and different strategies. Section 2.2 presents existing monitoring techniques found in the literature. Section 2.3 discusses different methods of software instrumentation used in both research and industry. Though the sampling-based method is quite new and not much work has been done on it, Section 2.4 gives a brief introduction to the work related to this topic.

2.1 Software Testing and Debugging

Testing and debugging are important procedures in embedded software development, as between 30 to 50 percent of the development cost is spent on testing and debugging [22, 31]. We define a software defect as "An incorrect step, process, data definition or result." [1]. Testing is a process which developers can use to assure the software quality by collecting information and it uncovers failures by checking the runtime behaviour of the system for potential violations of the possibly implicit specification. Debugging, on the other hand, focuses on revealing the errors that cause the failures, removing those errors, and verifying the correctness of the software. Thus, the steps most frequently seen in debugging are as follows [9]:

- Issue Recognition: identifying what is actually defined as incorrect in the system under test
- Intelligence Gathering: by inspecting the target, figure out how it works and how it produces the symptom
- Diagnosis: determining the fundamental bug
- Prescription: planning out how to remove or fix the bug
- Response: fixing or removing the bug
- Verification: checking the bug is removed from the target system and make sure the way it is fixed has not introduced other bugs
- Deployment: releasing the bug-free software system

There are various of causes of unreliability in software and computer systems. Figure 2.1 shows the cause-consequence diagram of fault, error, and failure. [8] along with [67] give the formal definitions of the terms in this figure:

- Defect: An imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced.

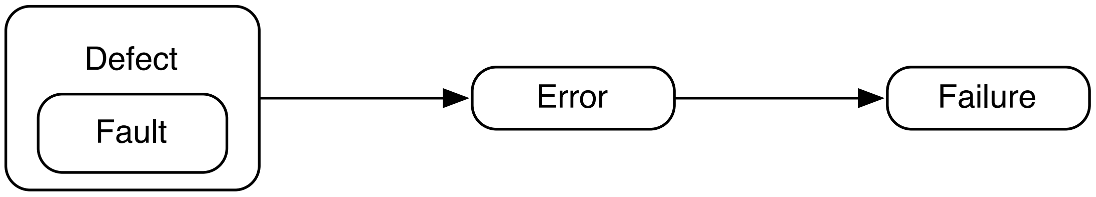


Figure 2.1: Cause-consequence Diagram of Fault, Error and Failure

- Fault: Abnormal condition that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function. Fault is the hypothesized cause of an error.
- Error: Discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition.
- Failure: Termination of the ability of a product to perform a required function or its inability to perform within previously specified limits.

In [53], B. Parhami generalizes the relationships between defect, fault, error, and failure as follows: At first, a software component may be defective. Some system states will show the defect. This exposure of a defect can lead to the development of faults. According to B. Parhami, if a fault is executed, it might contaminate the data which run through the system in such a way that it might cause some errors. However, incorrect information or states is not bound to make the subsystem malfunction. Different subsystems' designs have different error tolerances, which might handle some level of malfunction. Thus, B. Parhami suggests that a subsystem malfunction does not necessarily result in a disaster. Moreover, the degradation of service could lead to system failure which might cause catastrophe depending on the situation when the failure happens. To further clarify the above definition, here is an example [53]:

Example 1. *An aircraft begins to age as soon as it carries out its first flight. With passage of time, an aircraft with structural fatigue becomes defective. If the structure* *fatigue propagates a crack, the aircraft becomes faulty. A detection inspection of the aircraft during base maintenance can reveal the fault. However, the pilot does not notice this fault while flying the aircraft. When the internal load-bearing airframe structural components has less residual strength, the aircraft system becomes erroneous. When the structures no longer meet their damage tolerance requirements, they might unexpectedly produced cracks of a sufficient size and density in the structure to weaken it so much that it no longer has the intended residual strength, thus suffering sudden structural failure, which might result in explosive decompression in the worst case.*

To test or debug a software system, we must inspect the run time behaviour of the system and check how well this behavior is consistent with the specification. Equally important, the act of observation should not disturb or intrude on system behaviour. However, any form of observation is also an interaction — the act of testing can also affect what is being tested. In [30], Jason Gait defined the *probe effect* as "a characteristic behaviour of the execution trace of an incorrectly synchronized concurrent program when extraneous delays are introduced". In some situations, the presence of the debugger or the act of observation can affect some timing-related bugs in such a way that the bug disappears or the threads switch their execution order. The following is an example [67] of a probe effect.

Example 2. *In Figures 2.3 and 2.2, we make the assumption that two tasks A and B share a resource X and they both can carry out an operation on X. In addition, we assume that there is a semaphore S protecting the resource X and that the priority of task B is higher than that of task A. The execution time of each task varies according to different inputs, which leads to different accesses to the shared resource.*

1. *In Figure 2.2, task A terminates before task B is released, and thus performs an operation on X before B. The new value of X is $A(X)$. The entire operation will produce a value of X corresponding to $B(A(X))$.*

2. *As shown in Figure 2.3, task B locks the semaphore, and enters the critical region before task A. Task B then preempts A and performs an operation on*

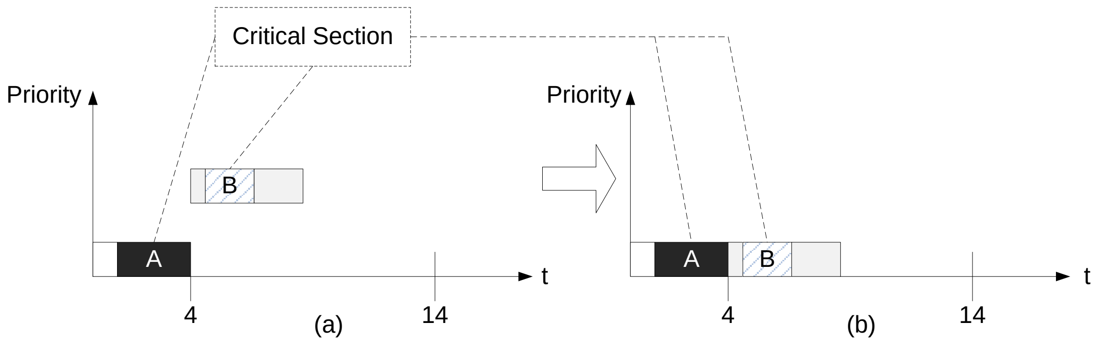


Figure 2.2: Task A terminates before task B

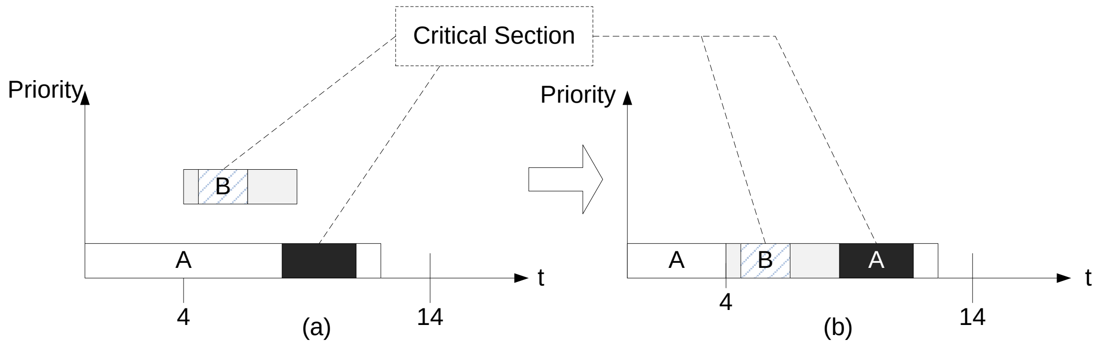


Figure 2.3: Task B preempts task A

X. The new value of X is B(X). The entire operation will produce a value of X corresponding to A(B(X)).

To test task A's behavior, we attached a probe to it and extend its execution time in such a way that only scenario (2) runs. As a result, scenario (1) will never show up during run time. If B(A(X)) is faulty, due to an error in task A, this error will never be revealed. After testing which shows no errors, we remove the probe in task A. However, scenario (1) will occur again and the erroneous calculation B(A(X)) may be executed, causing a failure.

2.2 Monitoring

In [72], Wong defines system monitoring as "the observation of specific activities or events that occur in an information system as specified by a predefined set of rules or polices". Previous work on system monitoring proposes the following categories [72]:

- monitoring frameworks
- monitoring to facilitate program understanding
- monitoring through code instrumentation
- monitoring for quality assurance
- dynamic monitoring

2.2.1 Monitoring Framework

For monitoring applications, it is more cost efficient to selectively record information. That is, instead of storing the entire temporal history of data, the monitoring framework records the history of the data object from the moment when a certain event happens. In [19], Bertino et al develop an event-based temporal object data model which can selectively store the past values of object attributes. In addition, they devise an event language which allows combining database operations, conditions on the database options, temporal and periodic events, using several operators. With this model, by keeping track of selective values, they develop a more efficient support for monitoring frameworks and provide the monitoring application with a more meaningful and effective set of data objects.

Detecting subsequence pattern has great importance in a monitoring system for suspicious activities. In [33], Robert Gwadera et al argues that setting thresholds for alarm is critical in observing long sequences of events, since it makes the monitoring system avoid false alarms. He suggests that setting the threshold too low results in too many false alarms while setting the threshold too high will fail to detect the real intrusions. To address the above issues, the authors in the paper carried out a quantitative analysis and devise a monitoring framework to set up the threshold that avoids false alarms and reduces the probability of missing the real intrusion.

In [63], Janusz Sosnowski et al proposes a on-line monitoring framework that combines hardware and software monitoring. This framework involves three techniques: event monitoring, performance monitoring at high and low architectural levels. These three techniques cover different scopes and relate to various system resources.

2.2.2 Monitoring to Facilitate Program Understanding

In [32], Neil M. Goldman argues that applications obtains a great portion of functionalities from their binary codes which is less amenable to static analysis than is the source code. In addition, he argues that compilers destroy a large amount of the structure so that the binary code cannot give answers to questions on observation of program's runtime behavior. To address the above issues, in this paper, he develops a tool named *Smiley* to help an analyst observe and record a program's runtime behavior and achieve an understanding of the program's implementation.

Mohlalefi Sefika et al, in [60], point out that the implementation of a software system tends to deviate from its original or intended design. He also considers this deviation undesirable since it makes the system hard to understand and modify. Thus, to avoid such undesirable deviation, he suggests that developers should make use of codified design principles supplemented by checks to ensure that the actual implementation adheres to its design guidelines. In the paper, they presents a hybrid computer-aided approach by integrating logic-based static analysis and dynamic visualization which provides multiple perspective of the code. Their approach provides a close monitoring of the implementation's faithfulness to its intended design through all stages of system's development.

2.2.3 Monitoring through Code Instrumentation

In [38], Amir Kishon et al introduce monitoring semantics, an extension of a language's standard semantics which captures monitoring activity. Monitoring semantics can provide a realistic basis for constructing effective monitors. Specifically, the they believe that by instrumenting the semantics into the program their approach has a decent degree of generality, safety and modularity which ease the process of reasoning about monitors. Furthermore, they indicate that their approach is very practical: it enables practical implementations of monitors.

A. W. Moore et al in [49] observed that kernel instrumentation tended to give an accurate record of what had happened in the kernel of a system; it is common to use kernel instrumentation when high precision is needed. However, through intensive study, he speculates that kernel instrumentation has several disadvantages, such as the difficulty to debug the code in the kernel, the availability of kernel source-code and the system crashes caused by the errors in kernel code. Due to these drawbacks of kernel instrumentation, he suggests that passive network monitoring can be an alternative to kernel instrumentation with several advantages such as no modifications to the operation of the monitored systems and the collection of data does not impact the machines being monitored. By comparing the pros and cons of these two methods, he presents methods by which the discrepancies between the results of the two techniques can be minimized.

William N. Robinson pointed out in [55] that analyzing software requirements is difficult and deviation software from requirements (if there is any) tends to cause errors. Targeting this challenge, he presents a framework to monitor the requirements of software while it is being executed by instrumentation and provides assurances about the state of a software's execution. According to him, this framework allows for automated support and uses a combination of assertion and model checking to inform the monitor. In another publication [56], He presents another requirement monitoring framework, called ReqMon. According to him, this monitoring framework can increase the visibility of requirements compliance provided by the system under test and employs monitoring tools to visualize the extent to which information systems comply with the stated requirements.

2.2.4 Monitoring for Quality Assurance

In structural testing, monitoring the software's execution determines which program entities have been executed. However, as pointed out by Raul Santelices et al in [58], this monitoring method introduces large overhead to the program execution. With covering all statements being a basic testing strategy, he suggests that cover all definition-use associations (DUAs) is more effective. Based on branch monitoring, in the paper, he presents a technique which can efficiently monitor DUAs. By speculating the technique, he presents two approaches for monitoring DUAs, that is, branch monitoring based and direct DUA-monitoring. Moreover, he indicates that there are efficiency and precision trade-offs between these two approaches and discusses the scenarios under which branch-monitoring can be used. He also present a tool, called DUA-FORENSICS which implements all aspects of their techniques.

In [23], Jim Bowring et al present a technique, called *software tomography*, for monitoring the deployed software products. According to them, software tomography is a low-impact and minimally-intrusive monitoring technique. As they describes in the paper, the technique divides the task into subtasks each of which introduces little instrumentation, and assigns these subtasks to single software instances for monitoring. To yield the original monitoring information, the technique at last synthesizes the information from each instance.

As Chenglian Peng et al point out in [54], due to the large scale and complicated architecture in distributed software systems, traditional static analysis fails to solve the emerging problems efficiently. Targeting the parallel and distributed computers, he introduces a method based on on-line monitoring. He presents a monitor system, called MS-1, which is a distributed event-driven hybrid monitor system with a synchronous clock system. In the paper, he suggests combining the control of the monitor system, collection of event trace and analysis tool an OM fulfills on-line monitor. In addition, with the internal states and the dynamic behaviour, he believes that the OM can help debug and test the computer systems effectively and efficiently.

2.2.5 Dynamic Monitoring

Monitoring depends on software instrumentation to collect run-time data from the programs. During monitoring, the instrumented running programs usually generate a huge amount of instrumentation data. Processing and storing the data introduces overhead and may even perturb execution. In [47], Barton P. Miller et al develop a performance measurement tool, called Paradyn, which monitors long running parallel applications using dynamic instrumentation and an adjustable sampling rate to reduce the collected performance data amount. According to him, by using a hypothesis set of performance problems, the time driven control can change the sampling rate accordingly. Further more, he points out that, in Paradyn, if incoming data show a problematic pattern then the sampling rate can be increased to get more data and thus achieve a better understanding of the problem.

In performance monitoring of parallel system, Jerry C. Yan considers the facility that can capture and display the program execution as a worthy feature. To accomplish this facility, in [73], he develops AIMS which is parallel monitoring system that employs event driven data collection mechanisms for MPI applications. According to Jerry C. Yan, interested parts of an application are instrumented before execution. He also indicates that when instrumented application executes, it generates performance data as a side effect. He also states that, with the performance data displayed on workstations, the user will have a good observation of program's behavior and have a way to trace the operation sequence.

2.3 Software Instrumentation

Software instrumentation tools have great importance in program analysis, profiling, performance evaluation, and bug detection. Instrumentation is "a technique for inserting extra code into an application to observe its behavior" [45]. Though introducing minor side effects such as increase in execution time, instrumentation should maintain the program structure and functionality. To achieve this purpose, as pointed out by Marina Biberstein et al in [20], instrumentations should not remove program elements; variables defined by the original program might be read out but not written. According to them, instrumentation may add its own variables, and those variables may be read or written. He further suggests that, instrumentation can insert new code into the original program, and invoke other methods from this instrumented code. However, he indicates that in such invocation original variables can not be modified.

There are a variety of applications for program instrumentation. As summarized in [39], instrumentation is usually used to collect program profile and run-time information for various testing, debugging and analysis applications, such as detecting program deviation, dynamic slicing and alias analysis. It can also be used to monitor and track the program behavior [20]. However, software instrumentation is intrusive because it introduces considerable overhead to the execution that might perturb the behaviour of the original programs. Researchers have proposed several methods to reduce the cost of instrumentation overhead [39, 16, 48]. Software instrumentation collects information of program execution by inserting instrumentation statements which might print out program location or variable values [69]. Instrumenting *printf* statements is a naive approach which is tedious and inflexible. It might also result in "probe effect".

Developer can carry out instrumentation at different stages: in the source code, at compile time, post link time, or at run time, as shown in Figure 2.4 [52]:

Generally, software instrumentation can be categorized as follows:

- Where to instrument:
  1. Source code instrumentation: instrument the source program
  2. Binary instrumentation: instrument the executable directly
- When to instrument:
  1. Static instrumentation: code is inserted to programs before runtime
  2. Dynamic instrumentation: code is inserted and removed from programs during execution

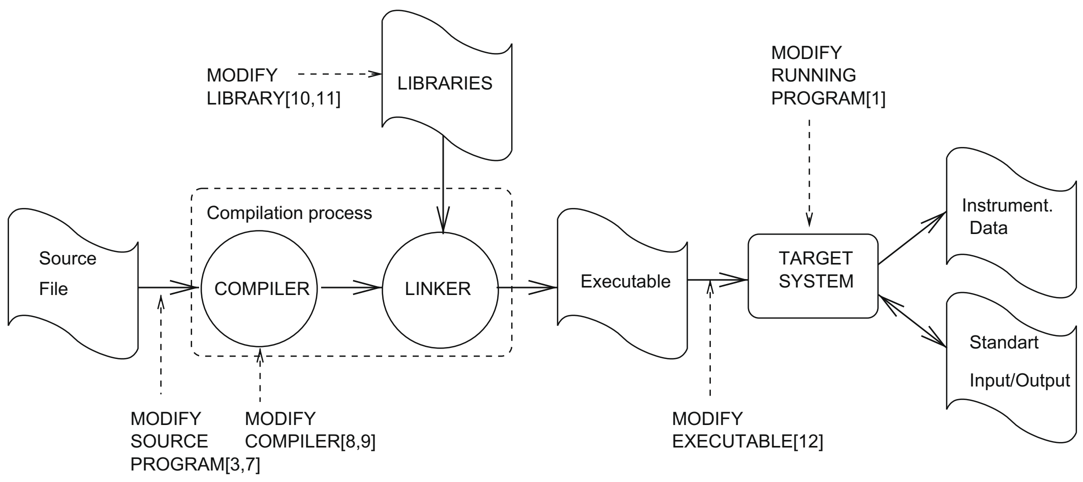


Figure 2.4: Software Instrumentation on Different Stages of Compilation

In [70], Zhonglei Wang summarizes that "the idea behind source code instrumentation is to insert timing information or analysis code into original source code to estimate execution time or to profile other software behaviour of interest". He also gives some reasoning about the advantages of the source code instrumentation as follows: source code instrumentation has good portability meaning the technique can be ported to different platform, since the output of a source code instrumentation is also source code just with additional sentences which provide or generate timing or analysis information; another advantage is that the instrumented software can be combined with hardware simulation.

Source code instrumentation shows promising results on software performance estimation. SciSim [70] framework proposes an infrastructure to carry out static instruction scheduling for superscalar architectures during instrumentation. According to Zhonglei Wang, the framework models runtime interactions between software and microarchitecture by combining instrumented code and microarchitecture simulators. Moreover, it instruments source code according to debugging information. A similar work in [36] uses code instrumentation methods in software profiling to make an estimation on the task latencies and memory access. The source code instrumentation engine in the framework insert extra function calls inside software tasks.

In [21], Aimen Bouchhima suggests that software instrumentation can also be used in embedded software annotation to enable performance modelling in high level hardware/software co-simulation environment. [21] proposes a "cross annotation" technique that allows instrumentation of embedded software on a basic block level. Consistent with Zhonglei Wang in [70], Aimen Bouchhima indicates that the advantage about source level instrumentation strategy is that it tries to get rid of host processor dependency by instrumenting the original source code. However, he argues that finding the basic block boundaries in the source code and inserting the instrumentation call is a very difficult task. The reason for this difficulty is, as he points out, the complex and rich syntax of the source code along with the effect of compiler optimizations. In [20], as Marina Biberstein points out, another problem about instrumenting annotated program is that instrumentation might perturb the integrity of annotation, making them invalid, and generating unpredictable results during the execution of the programs, since the tools used for instrumentation are unaware of the semantics of information which is passed by the annotation mechanism. In this paper, Marina Biberstein presents a solution to address this interaction problem.

Comparing with source code instrumentation, binary instrumentation tools offer the following advantages: since they instrument the binary or the executables, they are independent of the compiler and the source language [64]; in addition, they do not require recompilation and can take advantage of the processor characteristics [64]; Morever, in [51], the author suggests that "It also gives 100% instrumentation coverage of user-mode code". In dynamic binary instrumentation, analysis code is added to and removed from the original code of program at run-time.

One use for software instrumentation is to monitor control-flow. From the perspective of when to instrument the program, there are usually two types of software instrumentations: the static instrumentation insert the instrumentation code to the program before it executes, while the dynamic instrumentation instruments the program when it is running. The most commonly used static binary instrumentation tool is ATOM (Analysis Tools with OM) [64], is implemented by extending OM and provides a framework for building customized program analysis tools, such as basic block counting tool and cache modelling tool. It also provides selective instrumentation. Thus, user can specify the points to be instrumented, the procedure calls to be made, and the arguments to be passed on his own. For dynamic binary instrumentation, Pin provided by Intel is a valid option. It is a software instrumentation system that carries out binary instrumentation on Linux applications during their run time. Following the model of ATOM, PIN [45] makes the tool writer analyze an application at the instruction level. Unlike ATOM, it does not instrument an executable statically, but rather adds the code dynamically while the executable is running. Pin carries out the instrumentation using a just-in-time compiler. Both ATOM and PIN work on object modules. There are also other instrumentation frameworks available, such as Valgrind and DynamoRIO. Compared with PIN, these tools are not fully automated. For example, Valgrind [51] depends on the tool writer to add special operations to their intermediate representation to perform inlining. However, by instrumenting the executable with extra code, these software instrumentation methods might change the timing of the execution of the program unexpectedly and unpredictably. Thus, they are not soundly applicable to the real-time systems where timing has the top priority. Related work investigated software-support perspectives [17] and hardware-based approach [74]. Meanwhile, monitoring control-flow is especially expensive, and there is little work done so far to characterize or bound its cost.

One disadvantage of binary instrumentation is that a tool that performs binary instrumentation is usually limited to a specific instruction set architecture, since an object file cannot be ported to other architectures. The binary instrumentation tools ATOM is designed for Alpha AXP architecture [64]. Pin is originally designed for Intel Itanium architecture, and PIN 2 extends its support to four architectures: IA32, EM64T, Itanium and ARM [45].

From the modern language point of view, RAIL [24] (the Runtime Assembly Instrumentation Library) is a general purpose code instrumentation library for .NET platform. According to Bruno Cabral, the developer of this framework, RAIL allows manipulating assemblies in an object-oriented way and provides high level instrumentation patterns for assisting the program in code instrumentation. Moreover, with the help of RAIL, assemblies can be instrumented at runtime.

In [25], Anil Chawla et al suggest that, there are two common approaches of collecting run time information for Java program:

- Through a library which can rewrite the bytecode, adding instrumentation to the code
- Using an aspect-oriented language

However, as they argue in the paper, both of these two approaches have their own limitations: the first approach is expensive and hard to reuse and modify, while the second approach cannot provide information at the basic-block level. Targeting the above problem, they provide an extensive, configurable and generic framework that can gather information from an executing program. According to them, the framework lets the user define instrumentation tasks thus provides an easy method to instrument entities located in different parts of the code and collect distinct information from these entities.

Another application of program instrumentation is software dynamic translator (SDT) for self-managing systems which dynamically modify and control the execution of a program for code transformations at run-time, detection and repair of program faults. In [40], Naveen Kumar et al present a scalable and flexible framework, called FIST, which is also a dynamic instrumentation system. The instrumentation primitives in the framework are portable across different SDT infrastructures and machine architectures with both variable length and fixed-length instruction sets.

Software instrumentation can also aid program execution monitoring. In [65], Kevin S. Templer et al present an automatic software instrumentation tool, called CCI, which instruments C programs for program monitoring and visualization. In their method, they reduce the runtime overhead by instrumenting selective events. The tool also provides ways of extracting high level events.

In static instrumentation, the instrumentation is inserted into the code before execution and remains in the code afterwards. Although simple, static configuration is not generally efficient in terms of the information gathered versus overhead introduced, especially for long running programs. Thus, static instrumentation introduces immense overhead to both time and space. The instrumentation results in the growth of code and the instrumentation stays in the code leading to unnecessary time overhead. The large volume of data collected by static software instrumentation is a problem since storing and processing the data consume system resources such as memory, disk space, and CPU time.

Instrumentation also provides aids in structural testing which checks that a given coverage criterion is satisfied. In [48], Jonathan Misurda et al propose a demand-driven approach for instrumenting a program to perform different types of structural testings based on the execution path. According to them, the framework employs dynamic instrumentation to generate structural software testing tools.

Although the data trace produced by the instrumented programs make it possible for developers to locate bugs in the system, producing these data traces introduce large runtime overhead and might perturb the system's timing and behaviour. To solve the above problem, the authors in [28] develop an instrumentation technique for applications with temporal constraints and give reasoning about space and time for software instrumentations.

2.4 Program Sampling

Several works apply the concept of sampling to program debugging: using random sampling in statistical debugging to isolate bugs [44]; debugging programs given sampled data from thousands of user runs [75]; a sampling infrastructure for gathering information from a large number of executions [43]. These works focused on using the sampling concept to gather run-time information from program executions in workstation software. Paradyn [14], which monitors long running parallel applications, uses dynamic instrumentation and adjustable sampling rate to reduce the performance data amount collected. The time driven control changes the sampling rate using a hypothesis set of performance problems. If incoming data show a problematic pattern then sampling rate is increased to get more data and thus a better vision about the problem. Jonathan Misurda et al also build a tool called Jazz [48] which inserts and removes the instrumentation dynamically according to the demand.

Chapter 3

Sampling-based Program Execution Monitoring

3.1 System Model and Terminology

This work concentrates on multi-process single-threaded applications like the ones found in background/foreground systems. This structure dominates the embedded software domain due to its maintainable structure and efficient resource utilization [41, 29]. Note that about 85 percent of all embedded systems use 8-bit or smaller architectures [66].

We also assume that the system supports interrupts and has at least one high-precision timer as commonly found in microcontrollers. For example, the ATmega128 microcontroller has four timers.

3.1.1 Model Definition and Terminology

To analyze and reconstruct the execution path of an application, we convert a source program to a directed graph, representing the program's control flow. We define the control-flow graph as $G = \langle V, E \rangle$.

In G, each vertex ($v \in V$) represents a basic code block in a program. The entry vertex v_{en} is the start of the program. The exit vertex v_{ex} is the termination of the program. Edge $e := \langle v_s, v_d \rangle$ represents the specific transition from a source vertex v_s to a destination vertex v_d. It assumes that G is an *unweighted graph* with $e := \langle v_s, v_d \rangle = 0$, which means that there is no delay in the transition between two vertices.

We define the function $c : V \to \mathbb{N}$, which specifies the required execution time for a vertex v. For example, $c(v_0) = 10$ means that the basic code block at vertex v_0 requires 10 time units for its execution.

We define a path p as a sequence of adjacent vertices $v_i \to v_{i+1} \to \ldots \to v_k$. The execution time of a path p is the sum of the execution times of all vertices and is defined as $c_p(p) = \sum c(v_i)$ for all $v_i \in p$. An execution path r is the actual path executing from the entry vertex v_{en} to the exit vertex v_{ex}.

Our approach periodically takes samples from the execution information and program state. In this context, we define a sample as a triple $s := \langle state, v, t \rangle$ where v represents the vertex sampled, t represents the time stamp when we take the sample and *state* represents the program state (e.g. the values of some variables) at that time stamp. We define the sampling period T as the constant time interval Δt between two adjacent samples, that is, $T = \Delta t = t_{i+1} - t_i$ for two adjacent samples $s_i := \langle state_i, v_i, t_i \rangle$ and $s_{i+1} := \langle state_{i+1}, v_{i+1}, t_{i+1} \rangle$.

Furthermore, to evaluate the quality of the sampling period, we define the function $\mathrm{pathfind}_\mathrm{t}(v_i, v_j, \Delta t)$ with $\Delta t = t_j - t_i$ returning all possible paths between two vertices while Δt represents the execution time interval between v_i and v_j. We define the sampling period as *too long*, if multiple paths exist between two vertices of two samples, which is indicated by $|\mathrm{pathfind}(v_i, v_j, \Delta t)| > 1$,where $v_i, v_j \in V$. We define a sampling period as *sufficient*, if only one path exists between two vertices.

We form the concept of *optimality* for the sampling period with respect to both a vertex and a complete control-flow graph. If a sampling period T is sufficient and a sampling period $T + \epsilon$ is too long, T is the optimal sampling period for the starting vertex in the given control-flow graph. In other words, sampling after T permits only one path between the two samples and $T + \epsilon$ permits multiple paths. ϵ is the

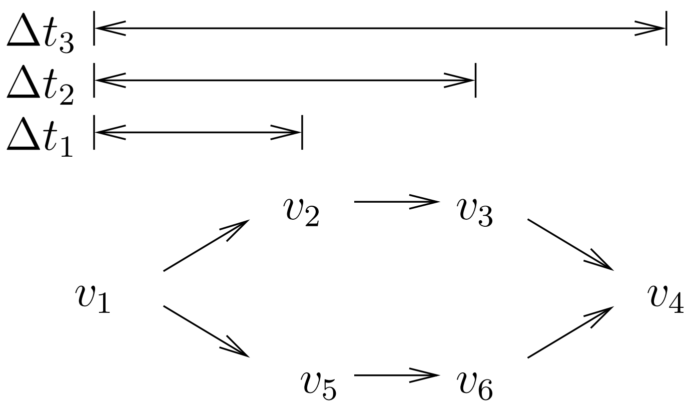


Figure 3.1: Different sampling periods for one control flow

smallest integral positive quantity of execution times in a control flow graph. Stating this formally, starting from a specific node v_i, we say the sampling period T is *optimal* for the node v_i, if $|\mathrm{pathfind_t}(v_i, v_{next}, T)| = 1$ while $\left|\mathrm{pathfind_t}(v_i, v'_{next}, T + \epsilon)\right| > 1$.

For a control-flow graph $G = (V, E)$, the *optimal* sampling period is the minimum of the *optimal* sampling periods of vertices in that control-flow graph. Thus, we define the *optimal* sampling period T_{opt} as $T_{opt} = min(T_1, \ldots, T_k)$ where T_i is the *optimal* sampling period for $v_i \in V$ with $V = (v_1, \ldots, v_k)$.

Example 3. *Figure 3.1 shows an example of a control-flow graph, a starting vertex v_1, and several sampling periods $\Delta t_1 = 1$, $\Delta t_2 = 2$, and $\Delta t_3 = 3$. All basic blocks have the same execution time $c(v_i) = 1$. From our definitions, for vertex v_1 the sampling period Δt_1 is sufficient, since $|\mathrm{pathfind_t}(v_1, v_2, \Delta t_1)| = |\mathrm{pathfind_t}(v_1, v_5, \Delta t_1)| = 1$;$\Delta t_2$ is optimal, since $|\mathrm{pathfind_t}(v_1, v_3, \Delta t_2)| = 1$ while $|\mathrm{pathfind_t}(v_1, v_4, \Delta t_2 + 1)| = 2$; Δt_3 is too long, since $|\mathrm{pathfind_t}(v_1, v_4, \Delta t_3)| = 2$.*

3.1.2 Markers

To increase the sampling period and reduce monitoring overhead, we introduce the concept of markers and extend a sample with state information. A marker can be a system element such as the program counter, because a vertex in the control-flow graph is a basic block in the source code. Alternatively, a marker can also be a newly introduced variable solely used for monitoring the software. For the remainder of this paper, we will only use extended samples and thus $s := \langle state, v, t\rangle$ where *state* is also a tuple defined as $state := \langle m_1, \ldots, m_k\rangle$ with m_i representing a marker of a system state such as memory, processor word, I/O registers, or our introduced variables which we can carry out arithmetic operations on or assign values to.

We thus refine the pathfind function as $\text{pathfind}_\text{s}(v_i, v_j, state_i, state_j, \Delta t)$ where $state_i$ and $state_j$ are the state elements of the corresponding samples. Using the function pathfind_s, a sampling period T is *optimal*, if $|\text{pathfind}_\text{s}(v_i, v_{next}, state_i, state_{next}, T_i)| = 1$ while $\left|\text{pathfind}_\text{s}(v_i, v'_{next}, state_i, state'_{next}, T_i + \epsilon)\right| > 1$.

As stated above, markers are special variables that can be used for extending the optimal sampling period. We introduce such new markers and increment their values at well-placed locations. We give the following example to show how the markers work.

Example 4. *Figure 3.2 shows a program control flow. All basic blocks have the same execution time* $c(v_i) = 1$.

Without introducing a monitoring variable a*, we use function* $\text{pathfind}_\text{t}(v_i, v_j, \Delta t)$ *to find the* optimal *sampling period. Starting from vertex* v_1*, there are three possible paths afterwards. If we take the sample after time 1, then* $|\text{pathfind}_\text{t}(v_1, v_i, 1)| = 1$ *with* $i = 2, 3, 4$*. However, if we take the sample after time 2, then* $|\text{pathfind}_\text{t}(v_1, v_5, 2)| = 2$*. Figure 3.3 shows the mechanism of this function. Thus, the* optimal *sampling period for node* v_1 *is* $T_1 = 1$*. Applying the same mechanism, for every other vertex* v_i *with* $i = 2, 3, 4, 5$ *in the control-flow graph, the* optimal *sampling period* T_i *with* $i = 2, 3, 4, 5$ *is* $4, 3, 3, 2$ *respectively. Thus, for the whole control-flow graph* $G = \langle V, E\rangle$*, the* optimal *sampling period* T_{opt} *is* 1.

Using the monitoring variable a*, Figure 3.4 shows the resulting optimal sampling period. We will use function* $\text{pathfind}_\text{s}(v_i, v_j, state_i, state_j, \Delta t)$ *to select the* optimal *sampling period. Starting from vertex* v_1*, the mechanism is shown in Figure 3.4.*

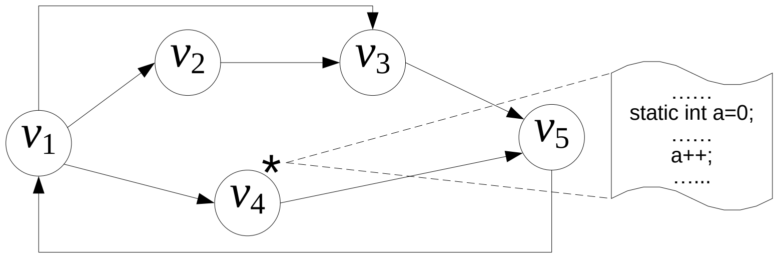


Figure 3.2: Control-flow graph with marker instrumented

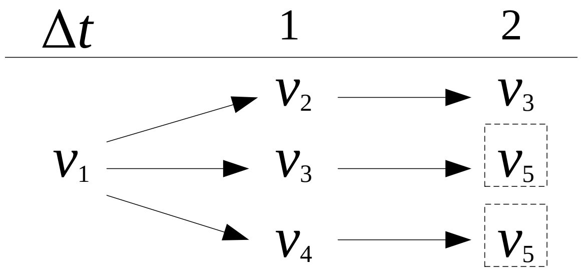


Figure 3.3: $\mathrm{pathfind_t}(v_i, v_j, \Delta t)$

While $|\mathrm{pathfind_s}(v_1, v_i, state_1, state_i, 4)| = 1$ *with* $i = 2, 3, 4, 5$ *and state* $:= \langle a \rangle$, $|\mathrm{pathfind_s}(v_1, v_5, state_1, state_5, 5)| = 2$. *Thus, for vertex* v_1*, the* optimal *sampling period is* 4*. Applying the same mechanism, for every other vertex* v_i *with* $i = 2, 3, 4, 5$ *in the control-flow graph, the* optimal *sampling period* T_i *with* $i = 2, 3, 4, 5$ *is* $7, 6, 6, 5$ *respectively. Thus, for the whole control-flow graph* $G = \langle V, E \rangle$*, the* optimal *sampling period* T_{opt} *is* 4*. Compared with the previous example, introducing marker a increases the* optimal *sampling period* T_{opt} *by a factor of 4.*

3.2 Theoretical Optimum

By using markers, we can increase the sampling period without losing any essential information about the execution paths. We call inserting such markers into vertices instrumentation.

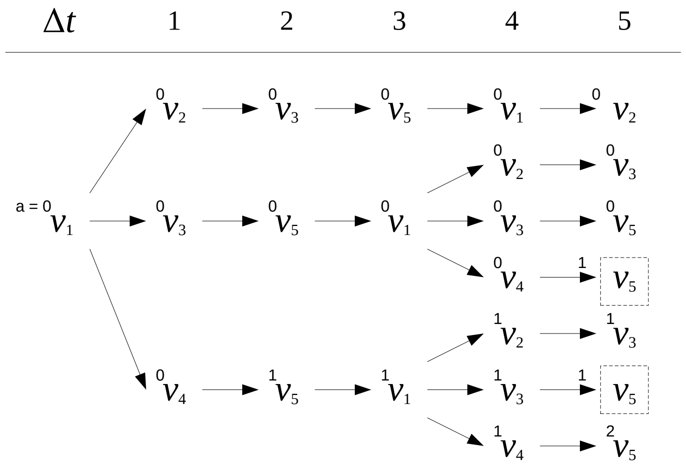


Figure 3.4: $\text{pathfind}_{\text{s}}(v_i, v_j, state_i, state_j, \Delta t)$

An important problem is to understand the limitations of such marker-based instrumentation. We therefore provide theorems to find the theoretic sampling period and decide on the termination conditions.

We require two additional definitions for the theorems. A *path pair pp* can be defined as two paths which have the same entrance vertex and the same exit vertex with their exit vertices overlapping in time, but no other vertices between the entrance vertex and the exit vertex overlap in time. That is, $|\text{pathfind}_{\text{s}}(v_{en}, v_{ex}, state_{en}, state_{ex}, \Delta t_{ex})| = 2$ while $|\text{pathfind}_{\text{s}}(v_{en}, v_{next}, state_{en}, state_{next}, \Delta t_{next})| = 1$, where $0 < \Delta t_{next} < \Delta t_{ex}$. A *path set* is defined as a set of paths of which any two paths constitute a path pair as defined above.

Lemma 1. *In a path pair pp, a path which starts from the entrance vertex v_{en} and ends at any other vertex except the exit vertex v_{ex} is unique. Formally,* $|\text{pathfind}_{\text{s}}(v_{en}, v_{next}, state_{en}, state_{next}, \Delta t)| = 1$*, where $v_{next} \in V_{pp}$ and $v_{next} \neq v_{ex}$.*

From the definition of a path pair, we can draw the following conclusion:

Lemma 2. *(Optimal Vertex Sampling Period) In a control-flow graph, all path pairs starting at vertex v_i constitute a vector $PP_{v_i}\langle pp_1, pp_2, \ldots, pp_k, \ldots, pp_m\rangle$, with each* *path pair starting at time t_{ien} and ends at time t_{kex}, with $k = 1, 2, \ldots, m$. The optimal sampling period of this v_i is defined as $T_{opt_i} = \min |(t_{kex} - t_{ien})| - \epsilon$, with $k = 1, 2, \ldots, m$.*

Therefore, to calculate the theoretic optimal sampling period, it is essential to find path pairs for every vertex in the control-flow graph. We propose the following approach to find path pairs starting from v_i: we construct an array of vertex states ordered by time; starting from v_i, we search v_i's child vertices and their corresponding states; then, we compare the state of child vertex v_j with that of v_k in the state array. If they meet the path pair conditions, we will say that the path which starts from v_i and ends at v_j and the path which starts from v_i and ends at v_k constitute a path pair; if the conditions are not met, we will treat that child vertex as a father vertex, add it to the state array after sorting and continue to search for path pair.

Definition 1 (Optimal Sampling Period)**.** *For a control-flow graph with N vertices, the optimal sampling period of the whole graph is the minimum sampling period of all sampling periods for all vertices. Formally, $T_{opt} = min(T_{opt_1}, \ldots, T_{opt_N})$.*

By choosing a proper strategy to find the vertices which are instrumented with markers and thus making the states of overlapping exit vertices different, we can extend the path pair and therefore increase the sampling period. However, the number of markers to instrument with is limited. With the the number of markers increasing, the following situation would occur. After the number of markers reaches a certain value, no matter how we instrument the vertices with markers, we cannot extend the path pair any further. In other words, we cannot increase the sampling period any more. In this situation, regardless how many markers are available, we can no longer distinguish the two paths in the path pair. Obviously, we should terminate the instrumentation process at this point. We propose the following theorem to draw this termination condition:

Theorem 1 (path pair Termination)**.** *For two paths p_1 and p_2 in a path pair, if they meet the following conditions:*

- *they have the same vertices with the same number of appearances but possibly a different order in time*

- *the states of the corresponding vertices are the same*

we can no longer instrument vertices with markers to differentiate these two paths, thus reach the theoretical optimum sampling period for this path pair.

Proof. We use proof by contradiction to prove our theorem. Suppose that we reach a path pair with its two paths (p_1 and p_2) violating the above conditions in Theorem 1. For example, the two paths have different vertices between them or the two paths have exactly the same vertices but the numbers of their appearances in these two paths differ. We assume that the sampling period $T_{falseopt}$ we get here is the theoretic optimum sampling period. However, if we instrument the distinct vertices of the two paths or the identical vertices which have different numbers of appearances in two paths with markers, we can still extend this path pair and form a new path pair whose sampling period is larger than $T_{falseopt}$. This contradicts the assumption that $T_{falseopt}$ is the theoretic optimum sampling period, as we can still distinguish these two paths. In this way, we prove that our theorem is correct. □

When a path pair satisfies the conditions in Theorem 1, we can terminate the instrumentation process since we can no longer distinguish the two paths through instrumenting vertices with markers.

Theorem 2 (Single Marker Termination)**.** *If the optimal sampling period can be extended by instrumenting a path pair or a path set with only one marker is an SAT problem.*

Proof. For a path pair pp_k, all the vertices except the entrance and exit vertices constitute an internal vertex set Ω_k. In Ω_k, all vertices that can be used to instrument with markers constitute the set Φ_k, with all the other vertices which can not be instrumented constituting the set Ψ_k. Then, $\Omega_k = \Phi_k \cup \Psi_k$. In Φ_k, the vertices, all of which cancel out the instrumentation when they are instrumented with markers at the same time, constitute Υ_k. Thus, $\Upsilon_k \subseteq \Phi_k$. When $cond_k = (v_{k1} \vee v_{k2} \vee \ldots \vee v_{kj}) \wedge (\overline{v_{c1} \wedge v_{c2} \wedge \ldots \wedge v_{cm}}) \wedge (\overline{v_{kj+1}} \vee \overline{v_{kj+2}} \vee \ldots \vee \overline{v_{kj+m}})$,with $\Phi_k = \{v_{k1} \ldots v_{kj}\}$, $\Upsilon_k = \{v_{c1} \ldots v_{cm}\}$ and $\Psi_k = \{v_{kj+1} \ldots v_{kj+m}\}$, is satisfiable, we can distinguish the two paths in a path pair k.

$$\begin{tikzcd}
 & v_b \arrow[r] & v_c \arrow[dr] & \\
v_a \arrow[ur] \arrow[r] \arrow[dr] & v_c \arrow[r] & v_e \arrow[r] & v_d \\
 & v_e \arrow[r] & v_b \arrow[ur] &
\end{tikzcd}$$

Figure 3.5: Scenario when theoretical optimum is reached using only one marker

The theoretical optimum for a graph using only one marker is reached, when we get to a path set, where $\Omega = cond_1 \wedge cond_2 \wedge \ldots \wedge cond_N$ can never be satisfied. $\square$

Example 5. *As shown in Figure 3.5 we get to a path set* $S = \{p_1, p_2, p_3\}$ *with* $p_1 = v_a \rightarrow v_b \rightarrow v_c \rightarrow v_d, p_2 = v_a \rightarrow v_c \rightarrow v_e \rightarrow v_d$ *and* $p_3 = v_a \rightarrow v_e \rightarrow v_b \rightarrow v_d$*. For path pair* $pp_{12} = \{p_1, p_2\}$*, the two paths* p_1 *and* p_2 *can be distinguished using only one marker, as* $cond_{12} = (v_b \vee v_e) \wedge (\overline{v_b \wedge v_e}) \wedge \overline{v_c}$ *is satisfiable. Similarly, the two pairs of paths in path pairs* $pp_{13} = \{p_1, p_3\}$ *and* $pp_{23} = \{p_2, p_3\}$ *can be distinguished respectively, with* $cond_{13} = (v_c \vee v_e) \wedge (\overline{v_c \wedge v_e}) \wedge \overline{v_b}$ *and* $cond_{23} = (v_b \vee v_c) \wedge (\overline{v_b \wedge v_c}) \wedge \overline{v_e}$ *satisfiable. However,* $cond_{12}$*,* $cond_{13}$ *and* $cond_{23}$ *can not be satisfied at the same time using only one marker. In other words,* $\Omega = cond_{12} \wedge cond_{13} \wedge cond_{23}$ *can never be satisfied. At this point, we reach the theoretic optimum using only one marker.*

3.3 Calculating the Sampling Period

However, in practice, we encounter path pairs much more often than path set as the likelihood is quite small for three or more paths to have the same entry and exit vertices with the same time span. Thus, it is both practical and important to develop an algorithm that has a polynomial runtime complexity to calculate the sampling period for path pairs.

Given a control-flow graph $G = \langle V, E \rangle$, to calculate the optimal sampling period for a vertex in the path pair, we propose the following algorithm based on the breath-first-search (BFS) [26] to implement $pathfind_s$.

This algorithm is based on breadth-first search (BFS). Firstly, we pick a starting vertex *vertex* by setting its *state* as OPEN. We also build a set V_{open} which contains all vertices that are adjacent to vertex *vertex* and set it to OPEN as well. In set V_{open}, we choose the vertex which has the least execution time t_{min} as the next starting vertex v_{next} to move to. At the same time, we update the sampling period by increasing it by t_{min} and the execution time of all vertices in set V_{open} by decrementing them by t_{min}. We build another set V_{toopen} which contains all the vertices that are both adjacent to and reachable from v_{next}. At last, we check the set V_{toopen}. If it contains a vertex whose *state* is OPEN and execution time is greater than zero, we say the optimum sampling period for that vertex is reached and return the current sampling period as *optimum*. If not, we repeat the above procedure until we reach the *optimum* conditions stated above.

Since the algorithm uses BFS, the runtime complexity for our algorithm is $O(|V| + |E|)$.

3.4 Instrumenting Control Flows

As stated above, to increase the sampling period, we introduce markers to the control-flow graph. In this section, we present our instrumentation approaches, analyze the related issues caused by the instrumentation and give our strategies to resolve these issues.

3.4.1 Increment VS Assignment

Instrumentation algorithms can use markers in different ways. One method is to increment the value of the marker each time the marker is hit. The other assigns a fixed number to the marker. We provide the following two examples to prove that neither of these two options is better than the other.

```
Vertex $v := \langle state, time \rangle$,
Edge $e := \langle v_{src}, v_{dst}, cond, updates \rangle$
**for** all $v \in V$ **do**
    $v.state \Leftarrow$ CLOSED
**end for**
$v_{en}.state \Leftarrow$ OPEN
$tResult \Leftarrow 0$
$V_{toopen} \Leftarrow \{\}$
$V_{open} \Leftarrow \{\}$
**loop**
    **if** $V_{open}$ is empty **then**
        **for** all $v \in V_{toopen}$ **do**
            $v.state \Leftarrow$ OPEN
        **end for**
        $V_{open} \bigcup \{v | v \in V and v.state = \text{OPEN}\}$
        $V_{toopen} \Leftarrow \{\}$
    **end if**
    $t_{min} \Leftarrow min(v.time)$ of all $v \in V_{open}$
    $v_{next} \Leftarrow v$ where $v \in V_{open}$ with $v.time = t_{min}$
    $tResult \Leftarrow tResult + t_{min}$
    **for** all $v$ in $V_{open}$ **do**
        $v.time \Leftarrow v.time - t_{min}$
    **end for**
    **for** all $e \in E$ with $e.v_{src} = v_{next}$ and $eval(e.cond) = T$ **do**
        **if** $e.v_{dst}.state =$ OPEN and $e.v_{dst}.time > 0$ **then**
            break from *loop*
        **else**
            create state for $e.v_{dst}$ and execute updates (e.updates) on this state
            $V_{toopen} \Leftarrow V_{toopen} \bigcup e.v_{dst}$
        **end if**
    **end for**
    $v_{next}.state \Leftarrow$ CLOSED
**end loop**
**return** $tResult - 1$
```

Algorithm 1: Find optimal sampling period for a vertex

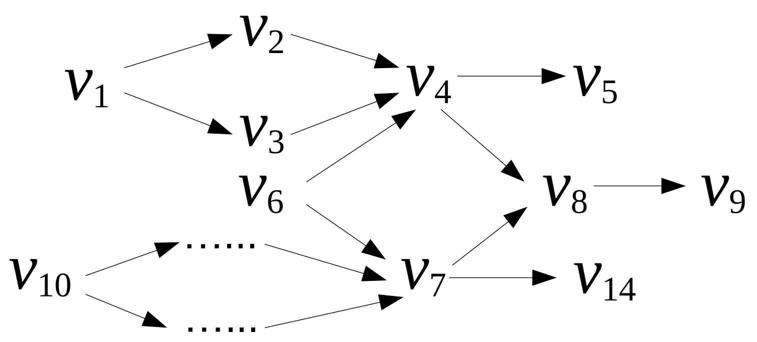

$$\begin{tikzcd}
 & v_2 \arrow[dr] & & & \\
v_1 \arrow[ur] \arrow[dr] & & v_4 \arrow[r] \arrow[dr] & v_5 & \\
 & v_3 \arrow[ur] & & v_8 \arrow[r] & v_9 \\
 & v_6 \arrow[uur] \arrow[dr] & & & \\
 & \ldots\ldots \arrow[r] & v_7 \arrow[uur] \arrow[r] & v_{14} & \\
v_{10} \arrow[ur] \arrow[r] & \ldots\ldots \arrow[ur] & & &
\end{tikzcd}$$

Figure 3.6: Scenario where increment works but not assignment

Figure 3.5 shows a path set. By adding the marker with a different assignment to the last vertex before the exit vertex (e.g., $v_c \leftarrow a = 1$, $v_e \leftarrow a = 2$ and $v_b \leftarrow a = 3$) in each path, we can distinguish these three paths in a sample taken at v_d. However, according to Theorem 2, we cannot distinguish these three paths by increment-based marker methods. Thus, the assignment-based marker method can instrument cases that the increment-based one cannot.

Figure 3.6 shows another case. We can instrument v_4 or v_7 with increment-based markers and distinguish the two paths in path pair $pp_{(6,8)}$. However, since v_4 and v_7 are also the exit vertices of two other path pairs $pp_{(1,4)}$ and $pp_{(10,7)}$, instrumenting these two vertices by assignments renders any previous instrumentations invalid, as several paths will share the same marker value and can invalidate another instrumentation at a later point. As shown, if we use increment instead, we will be able to solve this problem and distinguish the paths. Thus, the increment-based method can instrument cases that the assignment-based method cannot.

Since each method can instrument at least one case that the other cannot instrument, both methods have their justification as they can instrument different sets of control-flow problems. Instrumentation with assignment can be used in the scenario in which there is no previous increment marker introduced in the path set or path pair. Instrumentation with increment can be used as long as the elements in path set or path pair are not isomorphic.

3.4.2 Interference

Greedy instrumentation suffers from the problem that the instrumentation at a subsequent step may influence the instrumentations of previous steps. This can happen in either a direct or an indirect way. In *direct interference*, the subsequent instrumentation adds a marker to a vertex which is already part of a previous path pair or path set and this breaks the original instrumentation for that particular path pair, which relies on one of the paths being free of instrumentation. In *indirect interference*, the subsequent instrumentation adds a marker to a vertex which reveals a new path pair with a shorter time span. The following two examples show these effects.

Example 6 (Direct interference)**.** *We assume a control-flow graph, a greedy instrumentation strategy with only one available marker, and that each vertex has an execution time of one time step. Figure 3.7(a) shows the initial state of a path pair $pp_{(1,4)}$. Any greedy strategy will pick either v_2 or v_5 to instrument with a marker. Here we assume that the strategy picks v_2 as shown in Figure 3.7(b).*

v_5 is also part of another path pair $pp_{(6,11)}$ with a longer time span as shown in Figure 3.7(c). In this example, there will be a problem if the greedy strategy picks v_5 in the subsequent instrumentation step instead of v_{10}, v_7, or v_8. Instrumenting v_5 breaks the original instrumentation for path pair $pp_{(1,4)}$, since both paths in path pair $pp_{(1,4)}$ are then instrumented and can not be distinguished from each other.

Example 7 (Indirect interference)**.** *We make the same assumptions as that in Example 6. As shown in Figure 3.8, the greedy algorithm first discovers the path pair $pp_{(1,4)}$ and instruments v_5 to distinguish the two paths. By instrumenting v_5, the greedy algorithm also distinguishes the path pair $pp_{(6,11)}$, so the algorithm will not notice it—we now call it hidden—and instead see the path pair $pp_{(12,14)}$ as the next path pair with the shortest time span. If the algorithm now instruments v_8, it will reveal the hidden path pair which will cause a decrease in the sampling period.*

While a greedy algorithm can eliminate direct interference—see our SAT-based algorithms—eliminating indirect interference is hard, as it requires the algorithm to search for hidden path pairs with all possible marker configurations.

$$\begin{tikzcd}
 & v_2 \arrow[dr] & \\
v_1 \arrow[ur] \arrow[dr] & & v_4 \\
 & v_5 \arrow[ur] &
\end{tikzcd}$$

(a) Initial state

$$\begin{tikzcd}
 & v_2^* \arrow[dr] & \\
v_1 \arrow[ur] \arrow[dr] & & v_4 \\
 & v_5 \arrow[ur] &
\end{tikzcd}$$

(b) Greedy pick

$$\begin{tikzcd}
 & v_2^* \arrow[dr] & & \\
v_1 \arrow[ur] \arrow[dr] & & v_4 \arrow[r] & v_9 \\
 & v_5 \arrow[ur] \arrow[r] & v_{10} \arrow[dr] & \\
v_6 \arrow[ur] \arrow[dr] & & & v_{11} \\
 & v_7 \arrow[r] & v_8 \arrow[ur] &
\end{tikzcd}$$

(c) Problem if next greedy pick is v_5

Figure 3.7: The problem of direct interference during greedy instrumentation

$$\begin{tikzcd}
 & v_2 \arrow[dr] & & & \\
v_1 \arrow[ur] \arrow[dr] & & v_4 \arrow[r] & v_9 & \\
 & v_5^* \arrow[ur] \arrow[r] & v_{10} \arrow[dr] & & \\
v_6 \arrow[ur] \arrow[dr] & & & v_{11} & \\
 & v_7 \arrow[r] & v_8 \arrow[ur] \arrow[dr] & & \\
 & \cdots \arrow[ur] & & \cdots \arrow[dr] & \\
v_{12} \arrow[ur] \arrow[dr] & & & & v_{14} \\
 & \cdots \arrow[r] & \cdots \arrow[r] & \cdots \arrow[ur] &
\end{tikzcd}$$

Figure 3.8: The indirect interference caused by a greedy pick

3.4.3 Algorithms

We design seven different greedy algorithms (strategies) to find a suitable instrumentation. In our algorithms, the potential candidate vertex to instrument with markers to distinguish the two paths in a path pair is the one that is distinct in either of the two paths or is contained in both paths but has different numbers of appearances.

The seven strategies can be divided into three categories:

- I Degree-Based: The algorithms make decisions based upon the largest or smallest sum of in-degree and/or out-degree of the vertices in the context of the whole directed graph.
- II Frequency-Based: The algorithms make decisions based upon the occurrence frequency in the path pair, such as the most or least frequently occurring vertex.
- III SAT-Based: The algorithms transform the instrumentation into a SAT problem and compute a solution to find the instrumentation. The weighted SAT algorithm tries to combine the frequency-based method with the SAT-based ideas.

The following gives a detailed description of how each strategy works:

1. InOutDegreeMaxStrategy : It firstly finds potential candidate vertices in a path pair. Then among these vertices, it finds the one that has the largest sum of in degree and out degree in the context of the whole directed graph as the to-be instrumented vertex.

2. OutDegreeMaxStrategy: Among the potential candidate vertices, it finds the one that has the largest out degree in the context of the whole directed graph as the to-be instrumented vertex.

3. OutDegreeMinStrategy: Just the opposite of OutDegreeMaxStrategy, it finds the one that has the smallest out degree in the context of the whole directed graph as the to-be instrumented vertex.

4. SimpleGreedyMaxStrategy: Firstly it builds a hash table $ht = \langle key, keyvalue \rangle$ with key being Vertex's ID and key value being vertex's sum of difference of appearance in each path pair across the whole path set. Then among the potential candidate vertices, it finds the one that has the largest key value in the hash table as the to-be instrumented vertex.

5. SimpleGreeyMinStrategy: Just the opposite of SimpleGreedyMaxStrategy, it finds the one that has the smallest key value in the hash table as the to-be instrumented vertex.

6. GreedySATStrategy: We use this strategy to decrease the interference between different path pairs. Firstly, it separates the potential candidate markers between $T1$ and $T2$ and constructs a truth table. Secondly, it lists all the possible combinations which make the SAT expression true in the true table and return the combination as the to-be instrumented vertices. In the second run, while it constructs the true table, it also considers the outcome of the first run, that is, uses conjunction with the first. Instead of instrumenting one vertex each run, this strategy instruments several vertices in one run. Using SAT, it efficiently decreases the interference between different runs to the minimum.

7. MultipleGreedySAT: It is an extension of GreedySAT and focuses on instrumenting different markers. It firstly use SAT in GreedySAT to get the list of vertices. Then, it constructs a true table for the header of the list and gets all the possible scenarios to make the SAT expression true. In the later runs, it also considers the outcome of the previous runs when constructing the true table. In each run, instead of instrumenting using only marker, it instruments with different markers.

3.5 Simulation Method

To validate the theorems and the concepts of this work, we build an instrumentation engine that instruments control flow graphs. The engine provides the framework to test different heuristics but also computes the theoretic optimum following Theorem 1. The outputs are the instrumentation vertices, the required execution time, and the resulting sampling period.

Our inputs are realistic and statistically significant. The input data consist of 5 000 control flow graphs which model typical C program flows [68]. We generate these control flows with a customized version of Task Graphs For Free [27]. Regarding the SAT-based heuristic, we implement it using a SAT solver called SAT4J [2] in our instrumentation engine. One experiment run works as follows: we select a control flow graph, a heuristic (or the optimum algorithm), and the number of available markers and pass these values to the instrumentation engine. The engine computes the input control-flow graph and returns the sampling period, vertex to instrument, and the required execution time. Since the computational work is quite intensive, we perform our simulation through the Canadian super computing cluster called SHARCNET [3] collecting about 3.2 million instrumentation data points from up to 50 instrumentation steps, seven strategies, and several multi-marker configurations.

Our data successfully pass these integrity checks: (1) the execution time of the heuristic increases with the number of instrumentation steps, (2) no sampling period found by a heuristic is greater than the optimal sampling period, and (3) on average, the sampling period increases with the increase in the number of instrumentation steps.

The data distribution differs from a normal distribution (Shapiro-Wilk normality test for the data series varies around $p = 10^{-15}$). Thus, we rely on median values and testing procedures free of the normality assumption.

3.5.1 Instrumentation Performance Metric

To compare the performances of the algorithms, we take the maximum sampling period achieved in each run per algorithm and sum them; see Eq. (3.1). This metric is robust against direct and indirect interference outlined in Section 3.4.2.

$$P = \sum max(T_i) \tag{3.1}$$

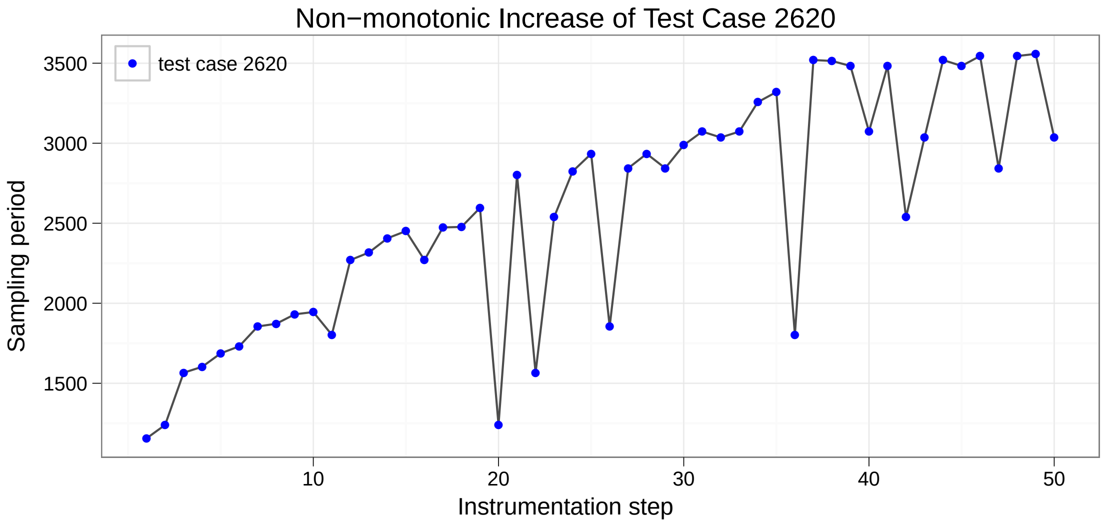


Figure 3.9: Example of interference of greedy instrumentation

3.5.2 Monotonicity Metric

Interference in the greedy instrumentation algorithms has an unpleasant effect on the monotonicity of the algorithms: a subsequent instrumentation may decrease the sampling period. This is contrary to what the user expects. Since each instrumentation introduces overhead, it is generally expected that the sampling period increases accordingly. Figure 3.9 shows an example of this behaviour. X-coordinate represents each instrumentation step with the same marker (the same variable in this case). Y-coordinate is the sampling period resulting from each instrumentation.

We use the following monotonicity metric to evaluate the algorithms:

$$M = \frac{N}{\sum d_i} \tag{3.2}$$

with

$$d_i = \begin{cases} 0 & \text{if } T_{run_i} - T_{run_{i+1}} \leq 0, \\ T_{run_i} - T_{run_{i+1}} & \text{otherwise} \end{cases}$$

In the metric, we use d_i to denote the decrement of the sampling periods between two instrumentation steps run_i and run_{i+1}, if the sampling period of one instrumentation step run_i is greater than that of its subsequent instrumentation step run_{i+1}. $\sum d_i$ denotes the sum of decrements in the entire instrumentation steps for a test case. N denotes the number of the total instrumentation steps. This monotonicity describes the reciprocal of the average decrement across the entire process of instrumentation steps for a test case using a specific strategy and gives a general assessment of that strategy, since the decrement represents the interference introduced by instrumenting vertices with markers.

3.5.3 Execution time

We measure the execution time by comparing the time stamp when the execution of heuristic starts with the time stamp after the instrumentation step is completed. The sum of all these times for one run provides the total execution time for that run. While we will not be able to compare quantitative results, because of the heterogeneity of SHARCNET, we will draw conclusions based on the similarity of the algorithms.

3.6 Results and Discussion

Following the experimental methods presented we give our experiment results which are sound and show statistical integrity. We also discuss the results and present the corresponding interpretation.

3.6.1 Instrumentation Performance

We follow the recommended guidelines for multiple testing [18]. We check that all input data for calculating the performance metric have roughly the same shape (single bell-like shape with a cut-off left tail) for all algorithms. The instrumentation performance differs significantly among the algorithms (Kruskal-Wallis Rank Sum

Test returns $p = 2^{-6}$). Using a Bonferroni correction for multiple testing among our algorithms, we test an individual algorithm with a $p \leq \frac{0.05}{91}$ to be accepted.

Figure 3.10 shows the result of the performance measurements for up to 50 instrumentation steps and compares it with the theoretic maximum achievable following Theorem 2. X-coordinate represents different heuristics with either multiple markers or a single marker. Y coordinate is the normalized experiment results of sampling period. The higher the performance value, the better. For the single-marker algorithms—the right part of the figure—the degree-based algorithms outperform the others except the 'max impact' algorithm. We use the Wilcoxon rank sum test with continuity correction and it shows that the differences among the degree-based algorithms are insignificant while it shows a difference between all degree-based algorithms and the 'min impact' as well as the SAT-based algorithms. An interesting point is that the SAT-based algorithms perform significantly worse than any of the other algorithms. Part of this is, because bad early decisions in the SAT algorithm cannot be undone by a later instrumentation. While, for example, the degree-based algorithms may break a previous instrumentation and cause direct interference, the SAT-based ones cannot do this, because they preserve all previous instrumentations.

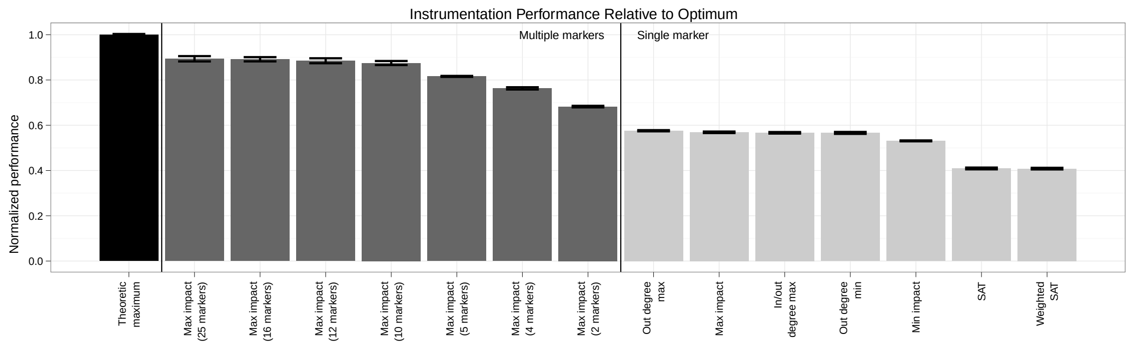


Figure 3.10: Instrumentation performance of all algorithms.

Using multiple markers improves the performance and asymptotically approaches the optimum. The middle part of Figure 3.10 shows the 'max impact' algorithm with different markers. Since 'max impact' performs similarly to other degree-based algorithms, if we were using a different algorithm, it would result in the same data. The gains achieved with small marker increases are significant. However, once the number of markers grows beyond ten, the results no longer differ using the Wilcoxon rank sum test with the adjusted significance level.

3.6.2 Monotonicity

Besides instrumentation performance, we also investigate monotonicity by the monotonicity metric defined in Section 3.5.2. Figure 3.11 shows the monotonicity of all heuristics normalized to SAT. X-coordinate represents different heuristics with either multiple markers or a single marker. Y coordinate is the normalized experiment results of sampling period. The higher the monotonicity value, the better. The left part of the figure shows the results of using only one marker. We use the same statistical test procedures as mentioned above to establish statistical significance. The SAT-based algorithms clearly outperform all the other algorithms. The reason is that the SAT-based heuristics always carry forward the previous path pairs and thus guarantee that a subsequent instrumentation avoids interfering with a previous one. The remaining monotonicity only originates from indirect interference. We can also conclude that in general approximately 20% of the interference in the instrumentation is indirect interference while 80% is direct interference.

Using multiple markers, we try to: (1) increase the achieved sampling period, and (2) improve monotonicity. We try to increase the sampling period, because if one marker is no longer sufficient (Theorem 2), we can use another marker until we hit the optimum for path pairs (Theorem 1). Figure 3.10 shows that we have achieved this. We also hope to improve monotonicity by reducing the interference between subsequent instrumentation steps. Whenever we switch to a new marker, we avoid interfering with a previous instrumentation. The results are quite surprising.

The right part of Figure 3.11 shows monotonicity with multiple markers. While using multiple markers improves the monotonicity of the heuristics, the improvements are still rather limited, and at some point become insignificant in general. However, individual cases can benefit significantly, as Figure 3.12 shows. X-coordinate rep-

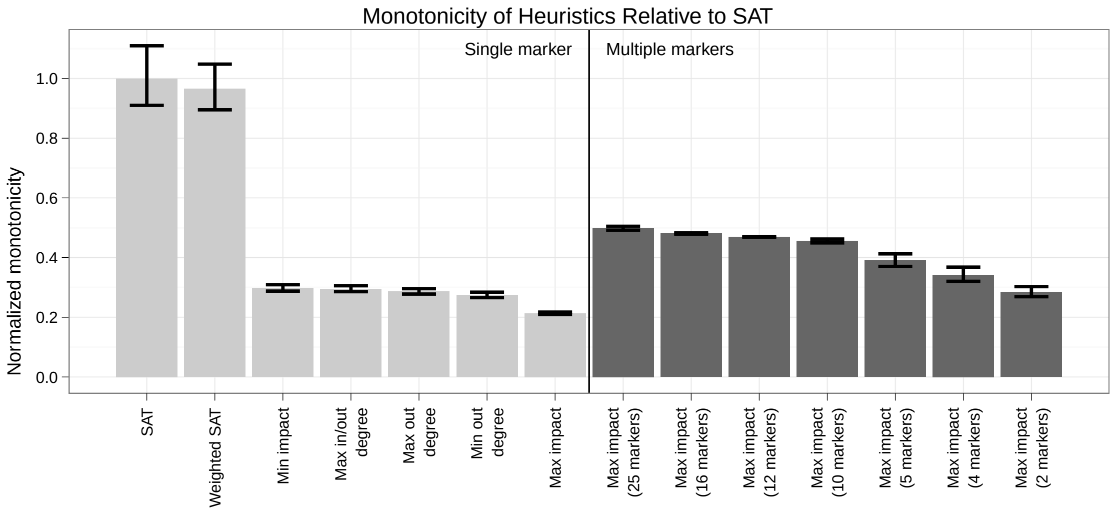


Figure 3.11: Monotonicity of heuristics

resents each instrumentation step with either multiple markers or a single marker. Y-coordinate represents the sampling period resulting from each instrumentation steps. The SAT-based heuristics still shows a better overall monotonicity than the 'max impact' heuristic with 25 markers.

3.6.3 Execution Time

We use SHARCNET to compute the results and collected the execution time of each instrumentation step. Based on the differences in the available computation time and platforms on SHARCNET, the results are purely informal and allow us to draw only conclusions when we can justify them algorithmically. For example, the weighted SAT algorithm builds on the SAT algorithm and uses a timeout to bound the execution time. Its execution time is about three orders of magnitudes greater than the SAT-based test. However, the complexity of the weighted SAT algorithm yields no improvement as seen in the performance and monotonicity analysis before.

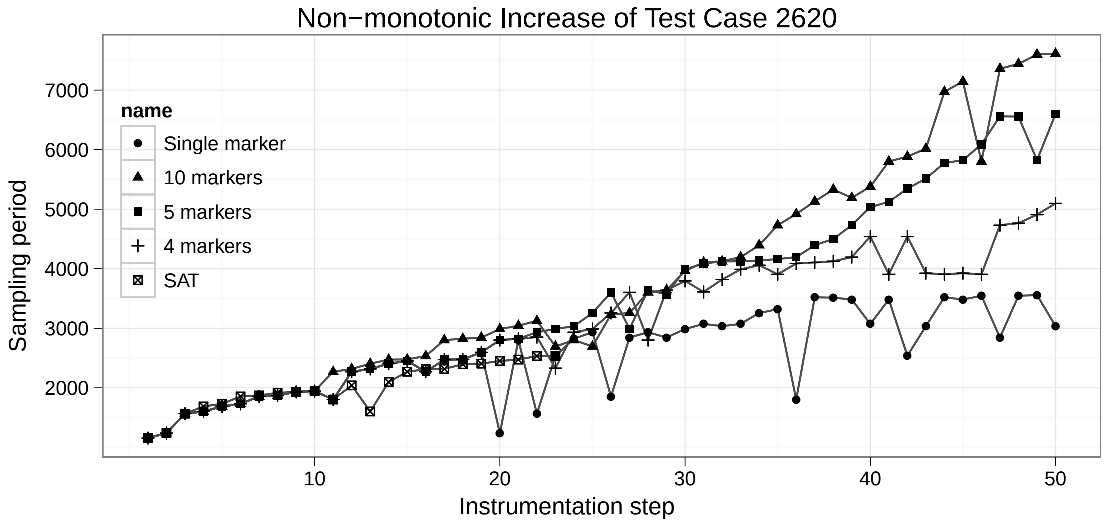


Figure 3.12: Improving monotonicity with multiple markers

3.7 Summary

In this Chapter, we proposed a framework for sampling-based monitoring to determine the execution path of the program and analyzed different algorithms for instrumenting a control-flow graph. We defined the system model and proposed two theorems based on it to determine when to stop instrumentation. While all heuristics worked to increase the sampling period, the degree-based heuristics outperformed the SAT-based ones, but the SAT-based ones achieved a higher monotonicity. Through normalized comparison, the SAT-based heuristic proved to be superior to others in terms of monotonicity. We further concluded that only 20% of interference was from indirect interference. We showed how to increase the sampling period by using multiple markers. However, this method had a limitation in that overusing markers did not pay off as much as we expected in the long run.

Chapter 4

SBMTC: A Sampling-based Monitoring Tool Chain

To investigate and evaluate the effectiveness and efficiency of the sampling-based program execution monitoring, we implemented our method and built a tool chain with it. The tool chain, called SBMTC, consists of a CFG generation, an execution time measurement, a sampling-marker search engine, a marker insertion engine, and a sampling-monitoring engine.

4.1 Tool Chain Overview

Based on the methodology proposed in Chapter 3, I built a tool chain to implement the sampling-based program execution monitoring method. Figure 4.1 shows the work flow of the tool chain.

In this section, I briefly describe the functionality of each module in the tool chain as follows and I will give more details in the subsequent sections.

1. CFG Generation: Using CIL, it generates the segmentation and CFGs of the C programs.

2. Execution Time Measurement: It instruments programs with timestamps to measure the execution times of basic blocks and annotates vertices in CFG.

3. File Conversion: It converts the .dot files into the input files readable by the Sampling-Marker Search Framework.

4. Sampling-Marker Search Framework: It uses heuristics to find the vertices to instrument and then calculates the optimal sampling period based on those vertices. I have given extensive discussion on the heuristics and the methods employed in this framework in Chapter 3.

5. Marker Info Extraction: It extracts the information about marker instrumentations from the output files of Sampling-Marker Search Framework and generates the input file which is also part of the input files of Marker Insertion Engine.

6. Marker Insertion Engine: It instruments the source code with the markers using the information provided by the previous steps.

7. Sampling-Monitoring Engine: It uses either "shared memory" or "GDB automated by Python" to sample the running program and collect the run time information.

4.2 Control Flow Graph Generation

This section explains how to use CIL to generate control flow graph from a C program along with some limitations using this method and other potential options.

The methodology of sampling-based program execution monitoring works with control flow graphs. To implement the tool chain based on the methodology proposed in the previous chapter, first of all, we have to extract the control flow graph from a given program. However, due to the complexity imposed by various programs and programming styles, extracting or generating the control flow graph from a program

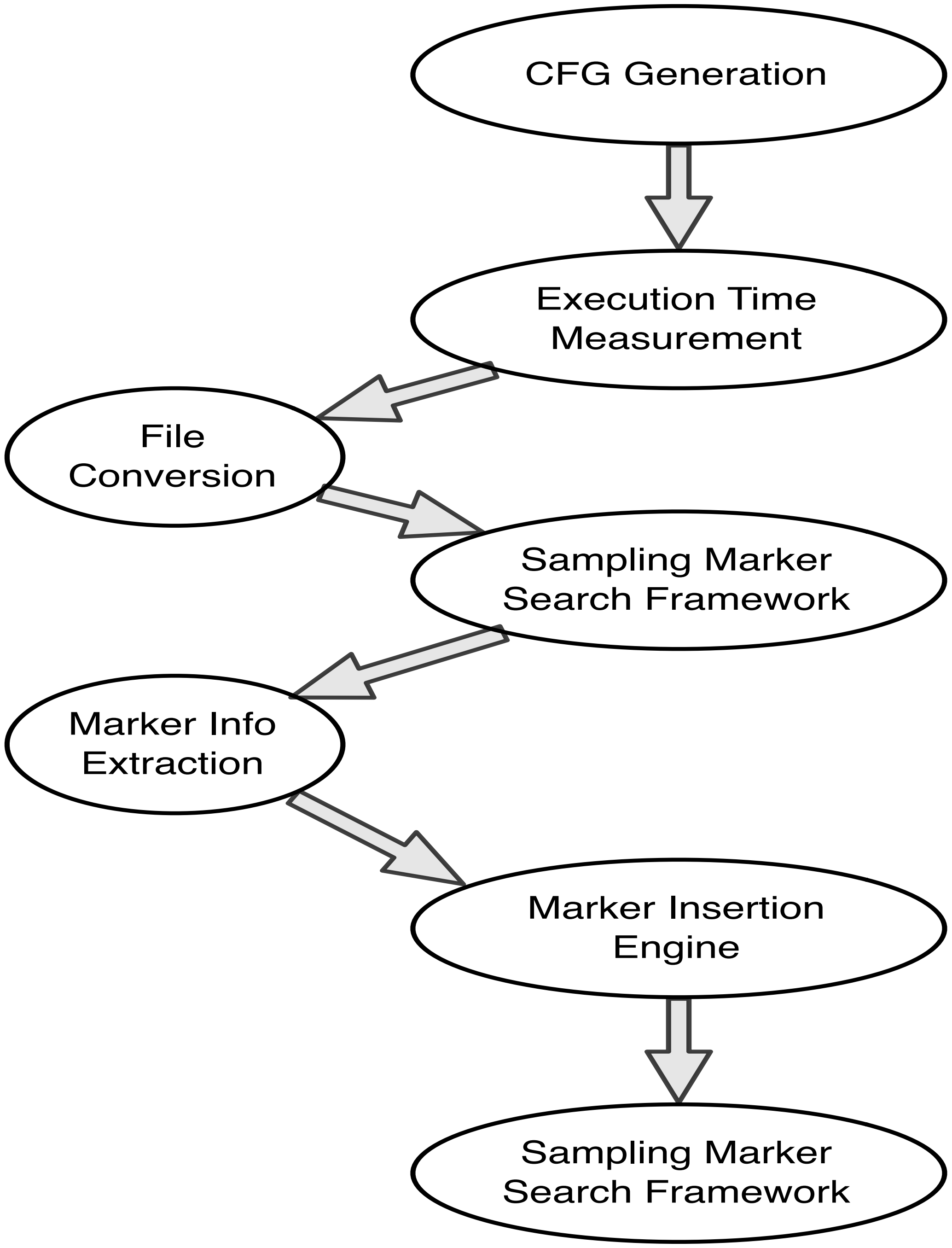

CFG Generation
Execution Time Measurement
File Conversion
Sampling Marker Search Framework
Marker Info Extraction
Marker Insertion Engine
Sampling Marker Search Framework

Figure 4.1: Work Flow of the Tool Chain

is also a standalone problem and is beyond the scope of my current research. Thus, the method employed in this thesis is just one of the many methods that tackle the CFG generating problem.

A control flow graph (CFG) is a directed graph with a single entry. A CFG is “comprised of vertices representing basic blocks with directed edges representing the flow of control in a program” [57]. A basic block represents a stream of successive statements where the control flow comes in at the start and exits at the end, without any branching except at the end of the basic block [15]. Though flexible in dealing with low-level constructs, the C program language is difficult to analyze and instrument by either humans or automated tools [50].

Before choosing to use CIL, I examined a number of existing tools that might be useful for generating control flow graph. None of the those tools met all my requirements on a CFG or served other components well enough to finish the tasks in the tool chain. Some (e.g. [71, 37]) are designed for specific compilers and are thus too low level for me to carry out the implementation. In [50], George C. Necula et al argue that the low-level representations generally lose structural information on high-level constructs, such as types and loops. Thus, he suggests that the printed out low-level representation is hardly faithful to the original C code. Some (e.g. [5]) are so high-level that detailed analyses are not supported, though they can visualize control flow graph in a fancy way. An intermediate language for generating the CFG which can be used to instrument the source code should be simple to analyze, close to the C source code so that conclusions from the tool can be mapped back to the statements in the source code, and able to handle real-world code [50]. CIL, which is a highly-structured subset of C, meets all these requirements.

CIL (C Intermediate Language) is a high-level representation. It allows an analysis and a source-to-source transformation of C programs [4]. Thus, it provides us with a way to analyze the C source code and generate the corresponding control flow graph from a program. CIL has served us very well for this purpose in the tool chain.

In the tool chain, I use the Library of CIL Modules which is very helpful for program analysis and transformation. Specially, the Control-Flow Graphs (CFG) Module is what I use in the tool chain and is one of the best options to generate the

CFG.

The API of this module provides many possible actions. By invoking the module on the source file, one can configure the tool in various ways, such as generating CFG for one function at a time and printing the CFG in *.dot* form.

I chose the style which generates the CFG of every function. I modified some source code of the CFG module, because the original version cannot get the location information of the function. There is some additional code to produce the line number of every block, since I have to insert marker in these blocks in the following steps.

The CIL tool is quite sensitive about the format of a given program and requires the C source file be well formatted. I had the following observations about CIL when using it to generate the CFG of a C source file:

1. One C statement on one line: To get the CFG of a C program, CIL has to get all the *instr* blocks of that program and requests that one line contain one C statement. If there are multiple statements in one line, CIL won't get the right starting or ending line number of that *instr* block which is critical for inserting either "timestamps" or "markers" into the source code in the following steps.

2. The calculation process (such as "$i++$") can not be in the control structure of a program, such as "if", "while" and "for" statements. If the calculation process is in the control statement, CIL will consider that control sentence as one *instr* block, which is not really the case. Furthermore, considering the control sentence as one *instr* block, the framework (which will be explained in the following section) inserts timestamps right before and after the control statement and will of course get syntax errors in later runs.

3. CIL can not handle some situations such as "precompiler". For example, there are some programs starting with "#ifdef" or "#elseif" which is so complex that CIL cannot get the ideal results.

4. For CIL to process, the curly braces "{" and "}" must be put in one line

respectively, otherwise there will be some syntax errors during the processing by CIL.

To prepare a well-formatted C program for CIL to process, I use GNU *Indent*, which changes the appearance of a C program by inserting or deleting whitespace. Besides understanding a great amount about the syntax of C, *Indent* also tries to handle the incomplete and misformed syntax [7]. I just used a small part of *Indent* program to tackle the problems mentioned in the above observations. The usage of the *Indent* involved in the tool chain is as follows:

```
indent -bl -nhnl input.c > output.c
```

with

- `-bl` option meaning "put braces on line after if". It formats the program like this:

```
if ( x < 0 )
 {
   x - - ;
 }
```

- `-nhnl` option meaning "Do not prefer to break long lines at the position of newlines in the input"

Then, by running a script, I delete all the comments in the program. Following the above steps, I got the program well formatted for CIL to process.

CIL takes the well-formatted C program and generates the corresponding control flow graph which can be visualized as that in Figure 4.2. In this Figure, the format of each vertex is as follows:

vertex ID: vertex type: the line numbers the vertex covers in source code

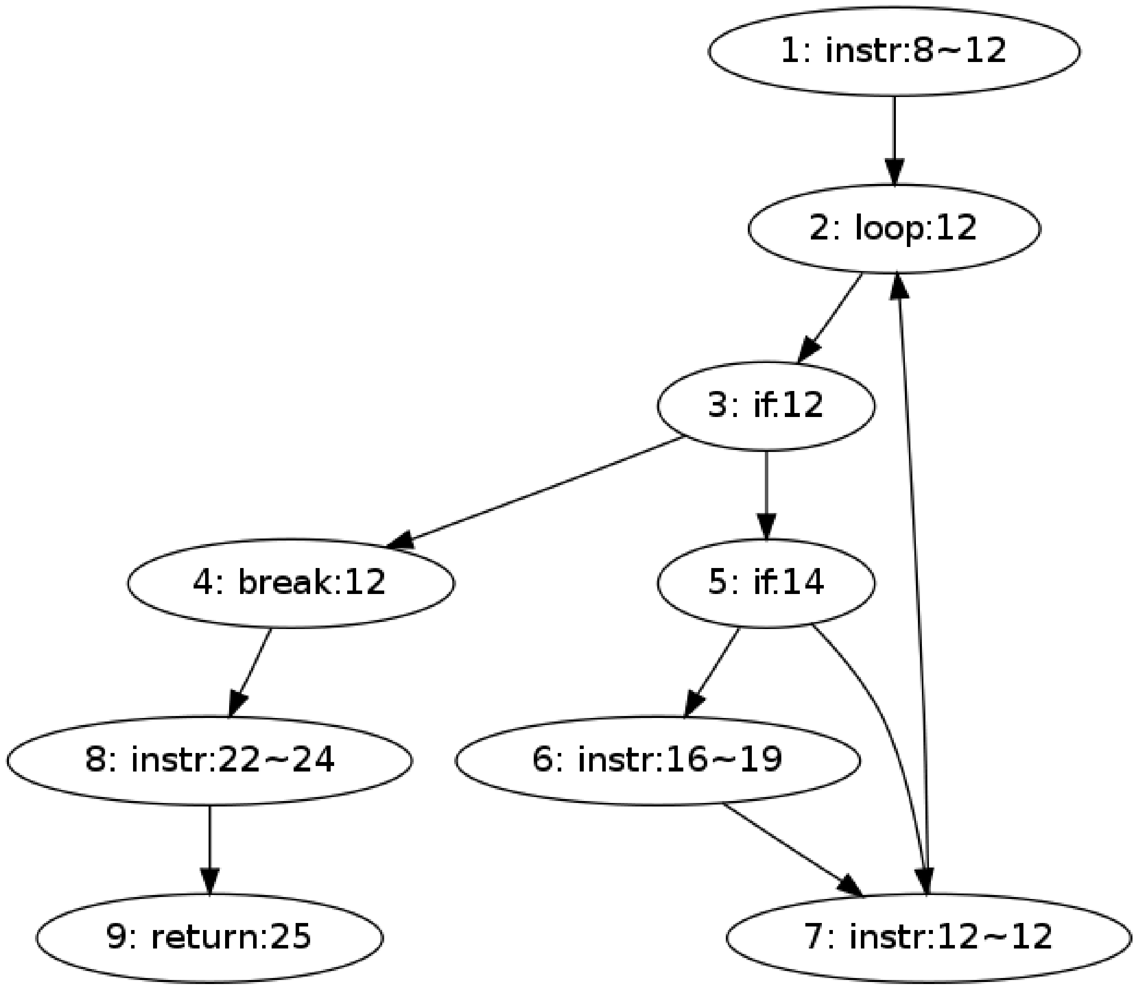


Figure 4.2: Control Flow graph generated by CIL

For “vertex type”, *instr* represents the regular basic block in the source code while others represent control statements or termination statement, such as `if` and `break`.

At this point in the *.dot* file generated by CIL, we have a program’s CFG, basic block label and line number map available. Once the CFG and the line number map are generated, we can instrument the program with either timestamps or markers according.

4.3 Execution Time Measurement

4.3.1 General Working Flow

The methodology of sampling-based program execution monitoring works with control flow graph which is a directed weighted graph with the value of a vertex being the execution time of that basic block. However, since the execution time measurement can be a standalone problem and is not the focus of my current research, I addressed this problem to a reasonable degree that is enough for carrying forward the tool chain.

There are three challenges for performing the execution time measurement on the instrumented source code:

1. Determine the granularity of the measurement
2. The resolution of the measurement has to be high enough to catch the extremely short execution time of the basic block.
3. Due to the high resolution of the time measurement, the counter has to be large enough in order to not overflow when measuring large basic blocks with long execution time.

After segmenting the program into basic blocks, as discussed in Section 4.2, the tool chain places instrumentation points before and after each basic block to measure the corresponding execution time. The general work flow of this execution time measurement is shown as Figure 4.3. The original input files of this working flow are C source file and the corresponding CFG file generated by CIL framework. Then, according to the program partition provided by the CFG, the tool inserts time stamps before and after each basic block of the C source code. Then, the tool runs the instrumented program and collects the information about execution time for each basic block. At last, with information provided by the original CFG file and *tes.c.touinfo* file which is the configuration file, the tool generates a new CFG with execution time for each basic block. The subsequent CFG file with execution times of basic blocks can be visualized as Figure 4.4.

4.3.2 Using Time Stamp Counter

Since the execution times of some basic blocks are extremely short, we need a counter with very high resolution to catch the execution times of those basic blocks. To achieve this purpose, I chose to use *Time Stamp Counter*. *Time Stamp Counter* (TSC) is a high-resolution way to get CPU timing information. It is a 64-bit MSR (model specific register) that increments every clock cycle since reset. We can use the `RDTSC` (read time-stamp counter) instruction to access this counter [34].

The following gives some details on how I measure the execution time of each basic block in practice. In my code, the instruction `__asm("RDTSC")` reads the time stamp counter and is wrapped in the function `getClock()`. The function `getStartClock()` is placed right before the beginning of a basic block to mark the starting time of that basic block. Function `getEndClock()` is placed right after the end of the same basic block to mark the ending time of that basic block. The difference between the timing values provided by function `getStartClock()` and function `getEndClock()` is the execution time of that basic block.

On a fast processor, we might encounter the situation when the time stamp counter overflows. However, this is not a problem to use the Time Stamp Counter in my measurement. A simple calculation can prove this point. The time stamp counter measures “cycles” and not “time”. For example, four hundred million cycles on a 400 MHz processor is equivalent to one second of real time, while the same

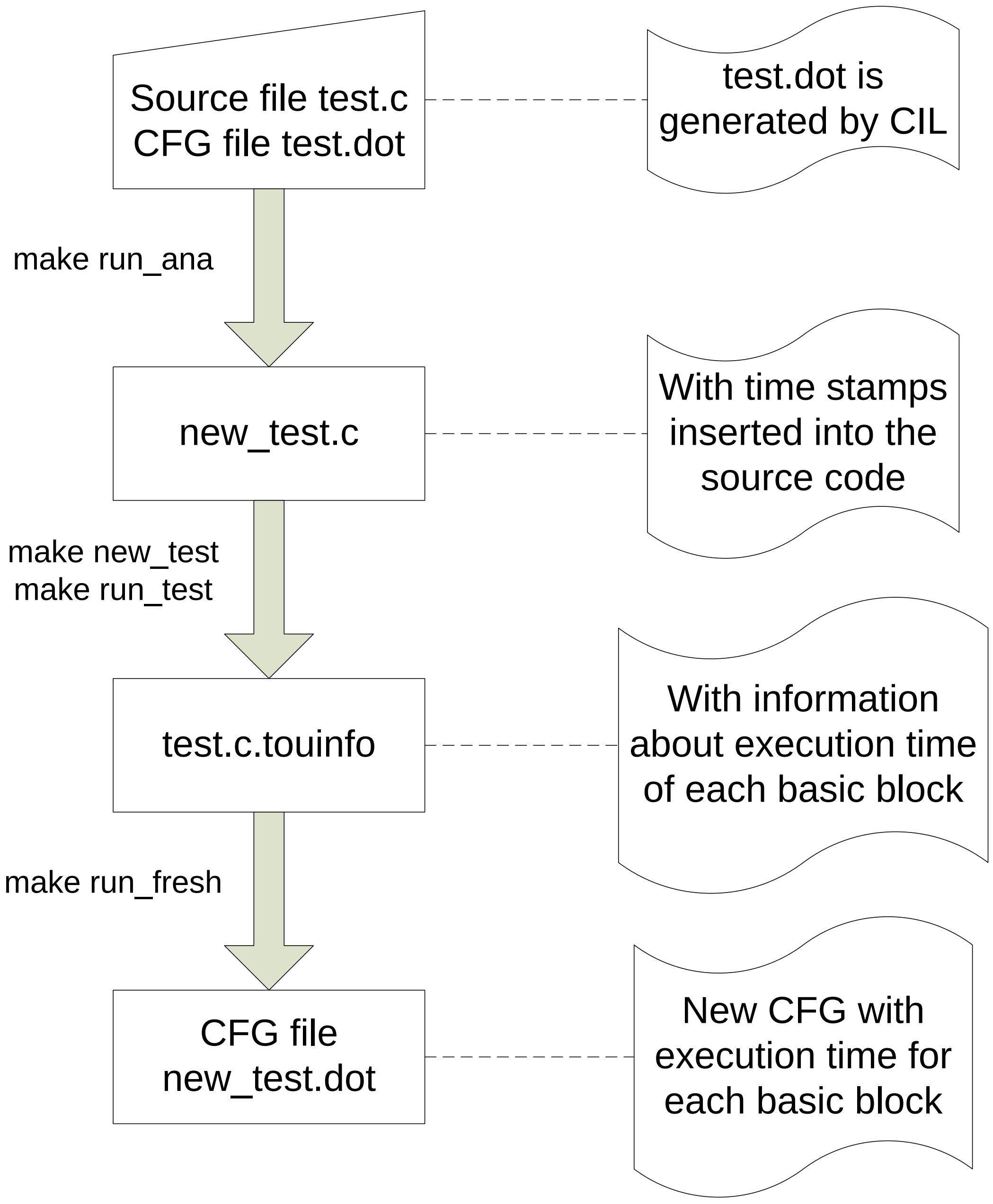

Source file test.c
CFG file test.dot
test.dot is generated by CIL
make run_ana
new_test.c
With time stamps inserted into the source code
make new_test
make run_test
test.c.touinfo
With information about execution time of each basic block
make run_fresh
CFG file new_test.dot
New CFG with execution time for each basic block

Figure 4.3: Working Flow of Measuring Execution Times of Basic Blocks

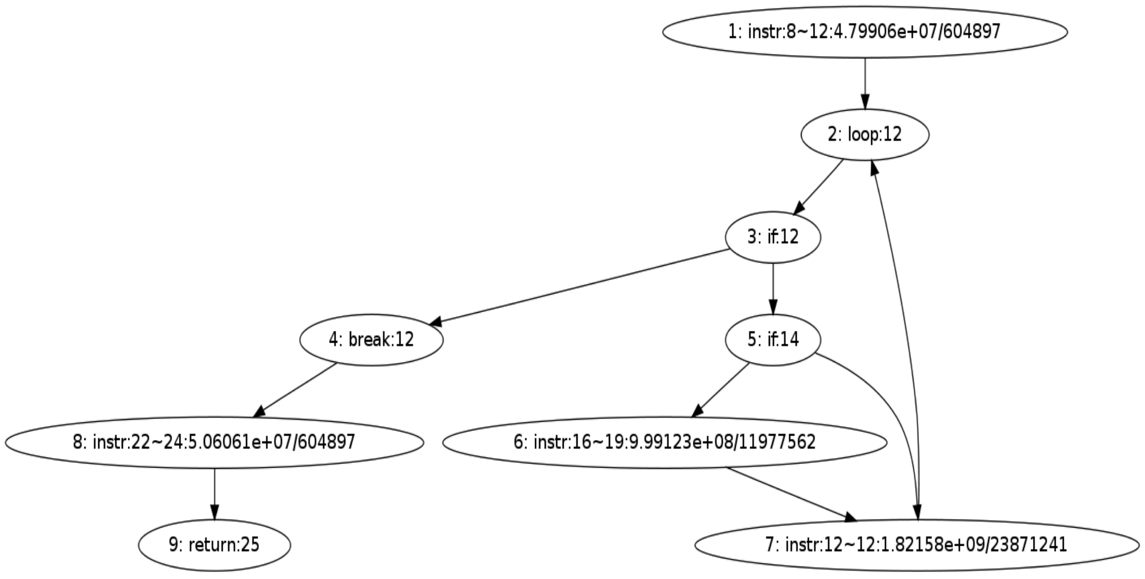

1: instr:8~12:4.79906e+07/604897
2: loop:12
3: if:12
4: break:12
5: if:14
8: instr:22~24:5.06061e+07/604897
6: instr:16~19:9.99123e+08/11977562
9: return:25
7: instr:12~12:1.82158e+09/23871241

Figure 4.4: Visualized CFG with execution times of basic blocks

number of cycles on a 800 MHz processor is only one-half second of real time. To get the "real" time, we should convert the cycle counts into time units, where:

$$\#seconds = \frac{\#cycles}{frequency} \tag{4.1}$$

Note: frequency is given in Hz, where: 1,000,000 Hz = 1MHz

On a 1596MHz processor (the "klagenfurt" machine I used), the time for TSC to overflow (starting from reset) is:

$$2^{64} cycles \times \frac{1 second}{1596 * 10^6 cycles} = 366.5 years$$

which is long enough for running any of the test cases I used for the experiments.

4.3.3 Issues Affecting the Accuracy of TSC

Though the timer stamp counter provides a high-resolution and low-overhead way to get CPU timing information, according to [34], the following issues might prevent us from getting reliable and accurate results from TSC:

1. Out-of-Order-Execution
2. Data Cache and Instruction Cache
3. Register Overwriting
4. Counter Overflow

My implementation does not take the issues 1–3 into account and we've already proved that issue 4 will not be a problem. Thus, we here assume that the timing information provided by TSC is accurate. However, there is code coverage problem existing in the current execution time measurement. Though the instrumentation of basic blocks is complete, there are still basic blocks with the execution time of zero because they are not executed during program's run. Here, we consider the basic blocks with zero execution to have low coverage expectation and assign a small value to them as their execution time assuming that the small value won't affect the calculation of the optimal sampling period.

4.4 Sampling Methods

The last module of the tool chain is the sampling engine, which periodically samples the instrumented running program and collects run time information. During the implementation, the following issues arise:

1. The execution times of basic blocks and the optimal sampling periods obtained from the previous steps are in the unit of "cycle" instead of "real" time, thus, they have to be converted into "real" time when the sampling is carried out. This is platform dependent, since different processors have different main frequencies.
2. Since the optimal sampling period is quite short in terms of "real" time, the clock used here has to have high enough resolution to carry out this sampling.

3. The method employed should introduce as little overhead as possible to the running program.

There are two options of implementing this sampling engine:

1. GDB based sampling automated by Python extension
2. Shared memory based sampling

Besides providing more details on these two methods in the following sub-sections, I also analyze the pros and cons of each method and the corresponding suitable working situation.

4.4.1 GDB based Sampling

GDB is a general-purpose software debugger developed by GNU project team. GDB is portable and usually works on Unix-like systems. It can debug many programming languages, such as Ada, C, and C++. It also supports a large number of processors for embedded system such as i386, ARM, and MIPS [35]. GDB provides the programmers with adequate functionalities to trace and alter the execution of programs. It offers methods to monitor and modify the values of programs' internal variables, and even make it possible to call functions independently.

Though GDB is an excellent debugging tool by itself, using it with some other language or integrating it with other environments is not that straight forward. Python is an object-oriented scripting language known for its ability to support various programming paradigms. One can write procedural, functional, and object-oriented code in Python [59]. Integrating Python scripting into gdb gives a full control over gdb from Python and automates the debugging process. This means that we can extend gdb with our own commands or create functions to operate with data structures. The many features of python-gdb include the following [11]:

1. Writing new commands

2. Convenience functions

3. Pretty-printing

4. Auto-loading of Python code

5. gdb from Python

6. Bringing up a GUI

Since the highest timing resolution that can be achieved by solely using Python is one second, which is too long to implement the required nanosecond sampling period, I have to embed a C timing library whose function can be called from within the Python script. Fortunately, there is a library, called `ctypes`, in Python to help me realize this. For Python, `ctypes` is a foreign function library which provides C compatible data types and offers a method to call functions in DLLs or shared libraries. Thus, we can wrap these libraries in pure Python [6]. `ctypes` exports the `cdll` which loads libraries by accessing them as attributes of these objects [6]. In order to carry out the "sleep" function whose "sleeping" time is actually the optimal sampling period calculated and is also quite short, I construct a dynamic shared object which contains a subroutine to solve the timing issue involved here. This dynamic shared object is loaded by `cdll` at the beginning of the Python script. Inside the subroutine of the dynamic shared object, the `nanosleep()` function provides a timing resolution which is high enough to implement of optimal sampling period of a program. Table 4.1 shows the input files of this sampling engine.

By reading these input files, the Python script will get the PID of the instrumented running program, the markers whose values are to be exported later and calculate the optimal sampling period in "real" time which can be converted from "cycles" given the frequency of the processor which is provided in the config file. The script first attaches GDB to the instrumented running program and, between two sleeps with sleeping time equal to the optimal sampling period, prints the values of the markers to an output file.

Table 4.1: Input Files of GDB based Sampling Engine

Input Files	**File Content**
PID file	pid of the application under test
config file	marker number and marker name, sampling period in “cycles” and the frequency of the processor in “GHz”
instrumentation file	the line numbers which contains the inserted markers
instrumented source file	the source file which has been instrumented with markers

4.4.2 Shared Memory based Sampling

As mentioned before, another option to implement the sampling engine is to use *Shared Memory*. *Shared Memory* is an efficient way to pass data between programs or processes. The idea behind the shared memory is that one process creates a memory portion which other processes can access [10]. After the memory is mapped to the address space of the processes which share the memory region, no kernel involvement will happen when two processes pass data between each other [12]. Thus, it decreases the time for system calls and increases the efficiency. However, the shared memory itself does not provide any synchronization function. That is, before the first process finishes writing to the shared memory, there is no automatic method to prevent a second process from writing to the shared memory. Thus, some form of synchronization between the processes using the shared memory is necessary. There are various forms of synchronization available: mutex, condition variables, and semaphores [12].

Following the general server-client scheme of using shared memory in [13], we can consider the whole shared memory based sampling as the server-client framework. The server is the SHMSample project (the sampling engine), while the client is the

C test case which is instrumented with markers. Thus, the scheme of the shared memory based sampling is as follows:

- The sampling engine should be started before any C test case. The sampling engine should perform the following tasks:
    1. Apply for a shared memory which has a memory key and keep the returned shared memory ID
    2. Attach this shared memory to the sampling engine's address space
    3. Initialize the shared memory
    4. Wait for all C test case' completion
    5. Detach the shared memory
    6. Remove the shared memory
- For C test case part, the procedure is similar:
    1. Apply for a shared memory with the same memory key and keep the returned shared memory ID
    2. Attach this shared memory to the C test case's address space
    3. Use the memory
    4. Detach all shared memory segments
    5. Exit

To be more specific: the C test case just writes the marker value into the shared memory, while the SHMSample project is running and waiting for the data. There is one timer in the SHMSample project, which controls the sampling period and makes the sample engine periodically read the marker values from the shared memory. The working scheme for the shared memory based sampling method is shown in Figure 4.5.

I use the following functions to implement the shared memory between the instrumented running program and the sampling engine. The `shm_open()` function is

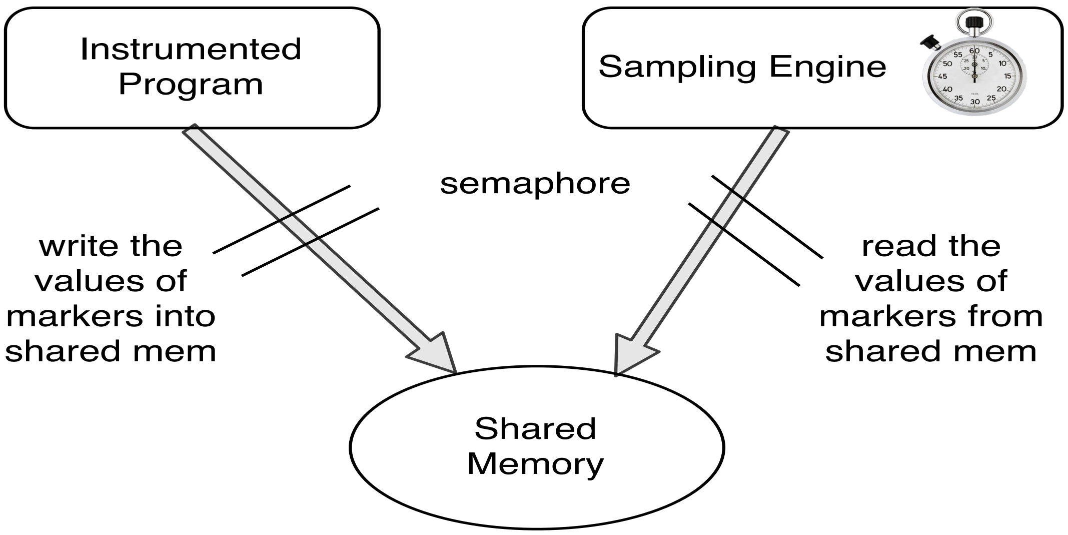

Instrumented Program
Sampling Engine
semaphore
write the values of markers into shared mem
read the values of markers from shared mem
Shared Memory

Figure 4.5: The Working Scheme of Shared Memory Based Sampling Method

used to connect a shared memory with a file descriptor. Functions can use the file descriptors `shm_open()` creates to refer to the shared memory object. Then, I create and initialize a semaphore to protect the shared variable (the global `_marker_0`, for example) using `sem_open()`. At last, `mmap()` establishes a mapping of a shard memory object into a process address space. By using `sem_wait()`, the instrumented program writes the values of markers into the shared memory and prevents other processes from writing to the shared memory at the same time. After the instrumented program is done with updating the marker values in the shared memory, it uses `sem_post()` to inform other processes (sampling engine, for example) that it is safe to write to the shared memory now. The usage of `sem_wait()` and `sem_post()` is the same for the sampling engine when it periodically reads marker values from the shared memory.

The second option which employs shared memory introduces much less overhead to the running program than the first option, since the sampling engine does not pause the running program as opposed to the one using GDB. However, the first option is also useful in the scenario where multiprogramming is not supported.

4.5 Summary

This chapter presents a tool chain which is developed according to the methodology of sampling-based program monitoring discussed in the previous chapter and gives the details on the working scheme of each element in the tool chain. The tool chain generally works well with the C files or C projects which contain multiple files and implements the sampling-based program monitoring method.

Chapter 5

Predicting the Number of Markers

Based on our observations, it becomes inefficient to use only one marker throughout the instrumentation in the long run. When using one marker, the sampling period on average increases with the number of instrumentation steps until it reaches either the optimum or the termination criteria, assuming the trend of the curves in Figure 3.9 and Figure 3.12. However, from Figure 3.9, the increment in the sampling period between one instrumentation step and its predecessor decreases. Still worse, interference invalidates the previous instrumentation thus decreasing the sampling period between instrumentation steps sharply.

5.1 Scheme of Adding a New Marker

Figure 3.12 shows that using multiple markers can improve monotonicity by reducing the interference between subsequent instrumentation steps and achieve higher sampling period than that achieved by using only one marker given a certain number of instrumentation steps. Inspired by the above promising results, we feel that using multiple markers has its benefits in resource-constrained embedded systems. Given the expected sampling period and the threshold, we can predict how many markers we need to achieve a certain sampling period with fewer instrumentation steps than those by using only one marker. Thus, we can enhance the efficiency of the instrumentation. Note that it is the system's resource allocated for the debugging that determines the expected sampling period. In the modelling of the changing of markers, we define the following terms:

1. *effective gain*: the actual increment of the sampling period between an instrumentation step and its predecessor and the overhead introduced by adding a marker (either new or old), denoted as G_E. Formally,

$$G_E = \frac{T_i - T_{i-1}}{C} \tag{5.1}$$

   while T_{i-1} is the sampling period of the current instrumentation step, T_i is the sampling period of the next instrumentation step and C is the cost of adding a marker (either new or old).

   Thus, we can define the effective gain from adding a new marker as follows: $G_{EN} = \frac{T_{iN}-T_{i-1}}{C_N}$, while T_{iN} is the potential sampling period that can be achieved by adding a new marker and C_N is the overhead (or cost) introduced by adding that new marker. We can also define the effective gain from using the same old marker like this: $G_{EO} = \frac{T_{iO}-T_{i-1}}{C_O}$, with T_{iO} being the sampling period achieved with the same old marker as the one inserted by the previous step and C_O being the overhead introduced by the marker.

2. *absolute gain*: it makes an evaluation of the effective gain by using the same marker against the effective gain by adding a new marker, formally:

$$G_A = \frac{G_{EO}}{G_{EN}} = \frac{T_{iO} - T_{i-1}}{T_{iN} - T_{i-1}} \times \frac{C_N}{C_O}. \tag{5.2}$$

   Both C_N and C_O are constants determined by the system under test, thus $\frac{C_N}{C_O}$ is also a constant C_s. Thus, we can re-write the *absolute gain* as $G_A = \frac{T_{iO}-T_{i-1}}{T_{iN}-T_{i-1}} \times C_s$.

With the above definitions, in the modelling, we need to consider the following factors on the changing of markers:

1. The resources that the embedded system allocates to the debugging, sampling-based monitoring to be specific: this factor determines the minimum sampling period (T_{min}) the sampling-based monitoring method should have. The expected sampling period that the instrumentation tends to achieve must be equal to or greater than this value.

2. The threshold (H) which determines when to add a new marker to the instrumentation: this value is pre-determined by the system under test or the user. However, we need to compare it with the *effective gain* or the *absolute gain* of the next instrumentation step to determine whether to add a new marker or not.

3. Finally, we compare the *absolute gain* G_A with the given threshold H and determine whether to add a new marker or not. If $G_A < H$, we will add a new marker into the instrumentation; otherwise, we continue to use the same old marker.

Using the above model, we can determine when to add a new marker to the instrumentation to achieve a certain sampling period with a certain amount of resource provided by the system under test.

5.2 Curve Fitting of the Sampling Period Trend

Since the data (in terms of instrumentation steps and the corresponding sampling periods) obtained from different heuristics using only one marker is statistically significant and reliable, we can safely use the data to generalize the trend which the sampling period of the instrumentation follows. The discrete data shown in each heuristic using only one marker besides the ones in Figure 3.9 and Figure 3.12 suggests that the sampling period increasing with the number of instrument steps might follow the *Logarithm* or *Power Law* curve.

Here, we decided to use least-square fitting to model the trend. The reason for using this method is as follows. In the least-square fitting, the sum of squared

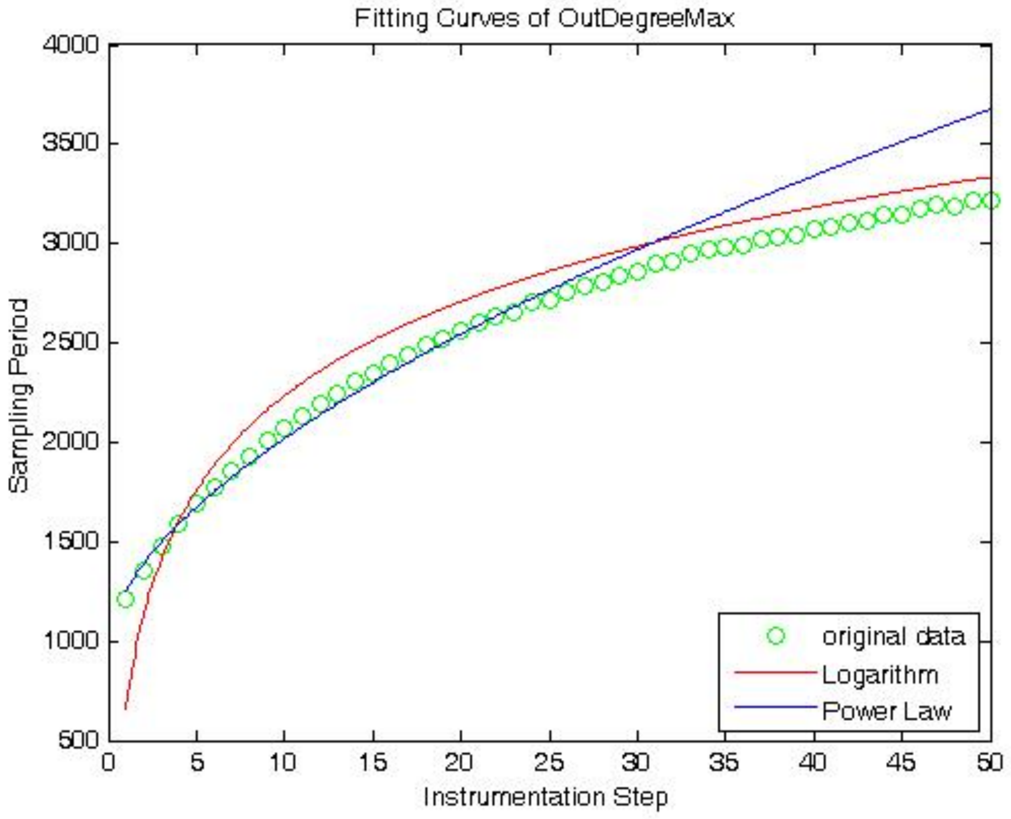


Figure 5.1: Power Law Trend

residuals provided by the model has its least value, with residuals being the difference between the experimental value and the value calculated by the model. Instead of the absolute residual, we use the sum of the squared residual to get a better continuous differentiable quality. Figure 5.1 shows the curve fitting results using both *Logarithm* and *Power Law*.

From Figure 5.1, we have the following indications: Even though the modelled *Logarithm* curve may have a better fitting in the long run, it gives a bad prediction on the trend of early instrumentation steps since the difference between the points on the curve and the actual experimental data is quite large. However, the modelled *Power Law* curve has a much better fitting with the experimental data during the early instrumentation steps. In our modelling, the early instrumentation steps carries more importance than the latter instrumentation steps, since we will use the early points of the curve to determine whether to add a new marker or not. Based on the above indications, we naturally choose to regard the *Power-law* curve as the trend of the sampling period increasing with the instrumentation steps and use it to make the marker prediction in the following section.

5.3 Optimal Threshold

Before continuing our discussion, we need to define another two terms:

1. *instrumentation pattern*: the number of markers and the corresponding total number of instrumentation steps to achieve a certain sampling period.
2. *optimal threshold*: the threshold used to achieve a certain sampling period with the best balance between the number of markers and the number of instrumentation steps.

From the discussion of Section 5.1 and Section 5.2, given the cost of inserting a same marker and adding a new marker and the threshold provided, we can predict the total number of markers we need to achieve a certain sampling period. Since the costs of markers are constants which are provided by the system under test, we can get different instrumentation patterns by tuning the threshold H. We get different instrumentation patterns as shown in Figure 5.2(a), with X-coordinate representing independent threshold, Y-coordinate on the left-hand side representing the number of instrumentation steps and Y-coordinate on the right-hand side representing the number of markers. To compare the results in Figure 5.2(a), we normalized the data and showed the normalized results in Figure 5.2(b). From these two figures, we can get the following conclusions:

1. Adding new markers reaches the expected sampling period with fewer instrumentation steps than using only one marker.

2. There is a tradeoff between the number of markers and the number of instrumentation steps.

3. By increasing the value of threshold, we increase the number of markers needed while decreasing the total number of instrumentation steps. Given the costs of adding a new marker and inserting with a same marker, by choosing the optimal threshold, we can achieve the best balance between the number of markers and the total number of instrumentation steps, thus optimize the overhead introduced by the instrumentation.

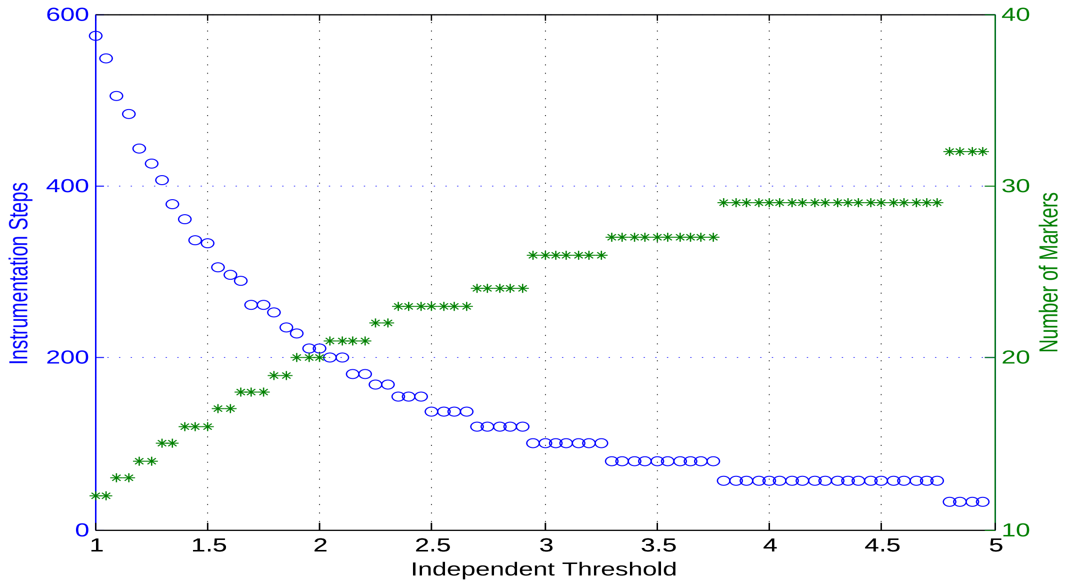
5.4 Experiments

Using an embedded benchmark, called MiBench, we performed experiments on the SBMTC to prove the soundness of the sampling-based program execution monitoring. MiBench offers a set of 35 embedded applications which are divided into six suites with each suite targeting a specific area of the embedded market. All the programs in MiBench are available as standard C source code, which makes MiBench a perfect testing candidate for SBMTC. MiBench also provides small and large data sets. The small data set represents a light-weight embedded application of the benchmark, while the large data set provides a more stressful, real-world application. To collect more run-time information, we only use the large data set when running the test cases from MiBench. We ran the experiments on an unloaded 2.4GHz Intel Core 2 Quad Q6600 processor with 3GB of memory and Ubuntu Linux 9.10.

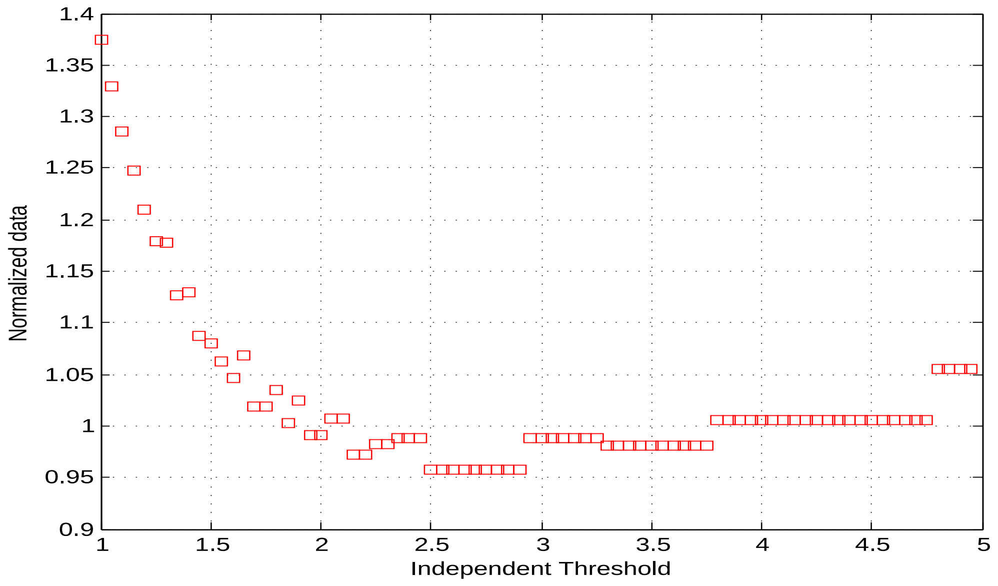
Figure 5.3 shows the experiment result from test case *sha* in the *Security* category of MiBench. From the figure, we have the following observations which reinforce the conclusions we get from the simulation data:

1. In the general trend, the sampling period increases with the increase of instrumentation steps.

2. Interference between instrumentation steps does exist.



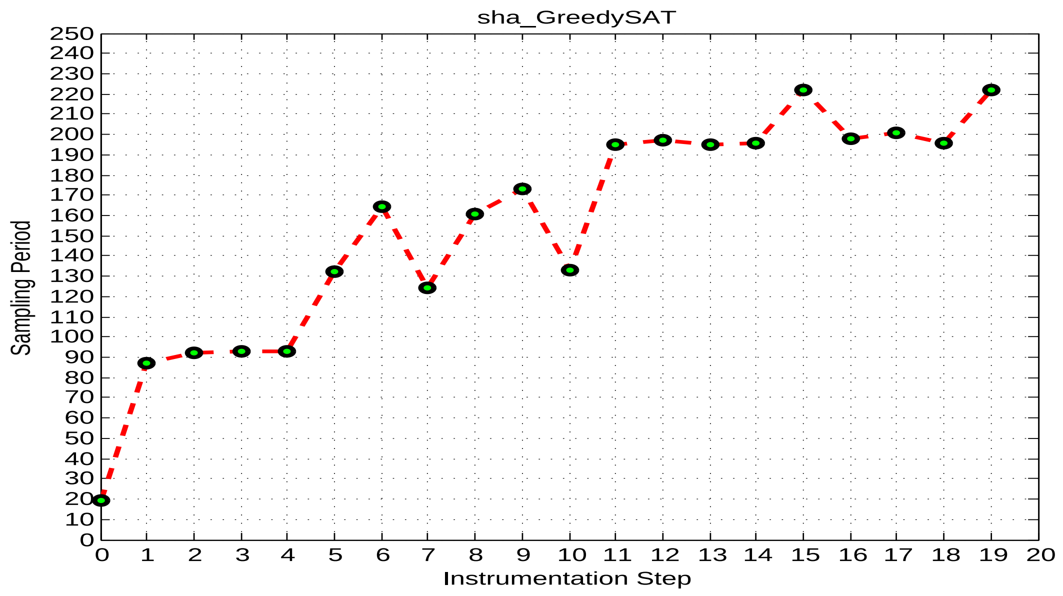
(a) Instrumentation Steps VS the Number of Markers



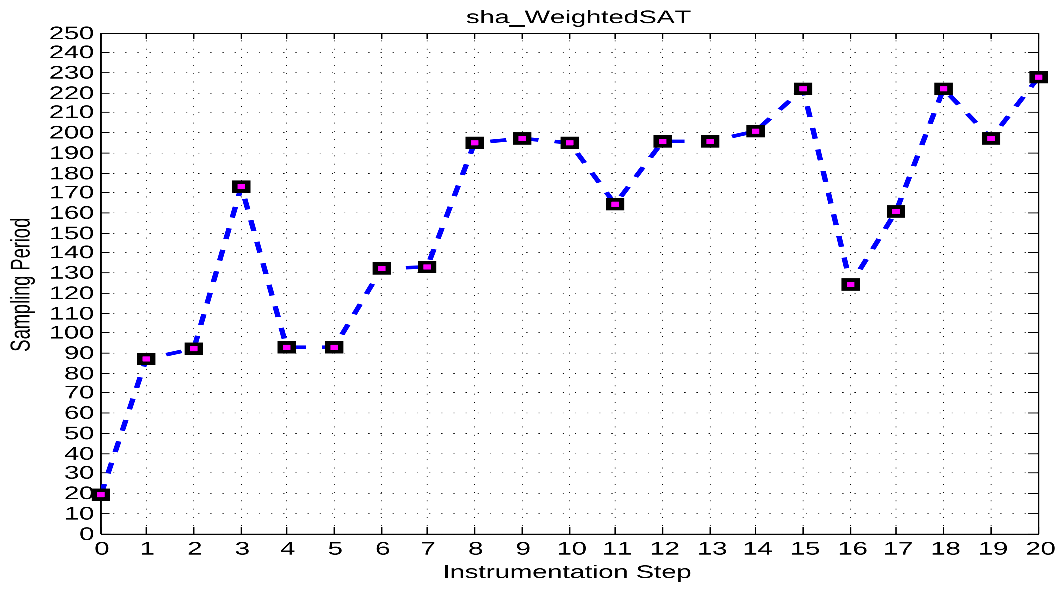
(b) Normalized Result

Figure 5.2: Tradeoff between Instrumentation Steps and the Number of Markers

3. The SAT heuristics cannot avoid indirect interference between instrumentation steps, though it eliminates the direct interference.



(a) sha with GreedySAT



(b) sha with WeightedSAT

Figure 5.3: Interference Found in Benchmark Program

Chapter 6

Conclusions and Future Work

Determining the execution path of a program helps locate bugs in a program. Software monitoring and instrumentation is one of the most important methods to debug and analyze the software system. However, for real-time embedded systems, the developer needs to bound the instrumentation overhead. This thesis addresses the above issue by providing a sampling-based program execution monitoring framework with the bounded overhead.

6.1 Conclusions

In our paper, we proposed a framework for sampling-based monitoring to determine the execution path of a program and permitted quantitative reasoning of many aspects involved in the mechanism. We also proposed and analyzed different algorithms for instrumenting a control-flow graph and propose the notion of optimality from both the vertex and the whole CFG perspective. We defined the system model and proposed two theorems based on which to determine when to stop instrumentation with an unlimited number of markers and with exactly one marker. We validate the general approach by proposing and comparing several algorithms for inserting markers into programs. Moreover, we investigated interference among markers from different instrumentation steps and proposed tailored algorithms to compensate for this interference. We discussed a number of observations and insights from the development of the algorithms. While all heuristics worked to increase the sampling period, the degree-based heuristics outperformed the SAT-based ones, but the SAT-based ones achieved a higher monotonicity. Through normalized comparison, SAT-based heuristic proved to be superior to others in terms of monotonicity based on which we further concluded that only 20% of interference was from indirect interference. We showed how to increase the sampling period by using multiple markers. However, this method had a limitation: over provisioning markers did not pay off as much as we expected in the long run. Based on our statistically significant and reliable data, we generalized the trend which the sampling period follows. Furthermore, we devised a scheme to predict the number of markers to reach a certain sampling period with a tradeoff with the number of instrumentation steps. At last, we built a tool chain to implement the sampling-based program execution monitoring methods and proved the soundness of the methods through experiments on benchmarks.

6.2 Future Work

The presented work fills the first pieces in a holistic framework for sampling-based execution monitoring. There are several ways to extend and improve the current work. The following are a number of possible paths to carry out the future work on this subject.

- **Optimization:** There is room for optimization by improving the algorithms to achieve both longer sampling periods and better monotonicity. However, we also need to investigate decision criteria when to switch markers before moving on to industrial case studies.
- **More Empirical Studies:** The lack of proper data and the limitation of the current benchmark prevented us from conducting a systematic evaluation of the current tool chain and comparing its effectiveness with the previous approaches. More case studies would help us better understand the pros and cons of both the framework and the tool chain. It would be good to test our tool chain on

a large number of software systems from different domains. Particularly, large software system and programs are good candidates for this purpose.

- **Randomized and Dynamic Sampling Period:** We use the fixed sampling period in the current framework and tool chain. By using this method, we might omit certain sections of program that periodically run between two samples. Randomized sampling periods might be used to avoid this miss. As suggested in [62], randomized sampling can discover sections of code that fixed sampling does not discover, thus providing more accurate snapshots of the software systems. Another extension is to devise a dynamic sampling scheme. According to different system conditions and criteria, the framework can dynamically tune the sampling period to reveal more information about the software system and decrease overhead.

- **Consider Code Coverage Constraints:** In our framework, we assumed that we could get full coverage of the program. However, in some test cases, some parts of the program never get executed which automatically invalid these test case. Future studies can take the code coverage into account and handle the test cases which do not have the desired code coverage.

- **If Statement**: It would be also interesting to investigate on the possibility of instrumenting *if* statements.

References

[1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages –, Dec 1990. 8

[2] SAT4J. web page, Oct 2009. `www.sat4j.org`. 41

[3] SHARCNET: Shared Hierarchical Academic Research Computing Network. web page, October 2009. `www.sharcnet.ca`. 41

[4] CIL - Infrastructure for C Program Analysis and Transformation (v. 1.3.7). web page, May 2010. `www.cs.berkeley.edu/~necula/cil/`. 52

[5] Control Flow Graph Factory. web page, May 2010. `www.eclipseplugincentral.com/Web_Links-index-req-viewlink-cid-1219.html`. 52

[6] ctypes A foreign function library for Python. web page, June 2010. `http://docs.python.org/library/ctypes.html`. 62

[7] GNU Indent - beautify C code. web page, May 2010. `www.gnu.org/software/indent/`. 54

[8] Ieee standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages C1 –15, jan. 2010. 8

[9] Introduction to Debugging . web page, June 2010. `http://www.gamedev.net/reference/articles/article2322.asp`. 8

[10] IPC: Shared Memory. web page, June 2010. `http://www.cs.cf.ac.uk/Dave/C/node27.html`. 63

[11] PythonGDB. web page, June 2010. `http://sourceware.org/gdb/wiki/PythonGdb`. 61

[12] Shared Memory Introduction. web page, June 2010. `http://www.kohala.com/start/unpv22e/unpv22e.chap12.pdf`. 63

[13] What is Shared Memory? web page, June 2010. `http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/shm/what-is-shm.html`. 63

[14] Vikram S. Adve, John Mellor-Crummey, Mark Anderson, Jhy-Chun Wang, Daniel A. Reed, and Ken Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 50, New York, NY, USA, 1995. ACM. 22

[15] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. 52

[16] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, New York, NY, USA, 2001. ACM. 17

[17] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994. 20

[18] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995. 43

[19] E. Bertino, E. Ferrari, and G. Guerrini. An approach to model and query event-based temporal data. In *Temporal Representation and Reasoning, 1998. Proceedings. Fifth International Workshop on*, pages 122 –131, 16-17 1998. 12

[20] Marina Biberstein, Vugranam C. Sreedhar, Bilha Mendelson, Daniel Citron, and Alberto Giammaria. Instrumenting annotated programs. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 164–174, New York, NY, USA, 2005. ACM. 16, 17, 19

[21] Aimen Bouchhima, Patrice Gerin, and Frédéric Pétrot. Automatic instrumentation of embedded software for high level hardware/software co-simulation. In *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 546–551, Piscataway, NJ, USA, 2009. IEEE Press. 19

[22] B. Bouyssounouse and J.Sifakis, editors. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, volume 3436 of *LNCS*. Springer, first edition, May 2005. 1, 8

[23] Jim Bowring, Alessandro Orso, and Mary Jean Harrold. Monitoring deployed software using software tomography. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–9, New York, NY, USA, 2002. ACM. 15

[24] Bruno Cabral, Paulo Marques, and Luís Silva. Rail: code instrumentation for .net. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1282–1287, New York, NY, USA, 2005. ACM. 20

[25] Anil Chawla and Alessandro Orso. A generic instrumentation framework for collecting dynamic information. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4, 2004. 21

[26] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001. 34

[27] R.P. Dick, D.L. Rhodes, and W. Wolf. Tgff: task graphs for free. In *Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on*, pages 97–101, Mar 1998. 41

[28] S. Fischmeister and P. Lam. On time-aware instrumentation of programs. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 305 –314, 13-16 2009. 22

[29] S. Fischmeister and I. Lee. *Handbook on Real-Time Systems*, chapter Temporal Control in Real-Time Systems: Languages and Systems, pages 10–1 to 10–18. Information Science Series. CRC Press, 2007. 25

[30] Jason Gait. A probe effect in concurrent programs. *Softw. Pract. Exper.*, 16(3):225–233, 1986. 10

[31] M.P. Gallaher and B.M. Kropp. The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards & Technologg Planning Report 02–03, May 2002. 1, 8

[32] N.M. Goldman. Smiley - an interactive tool for monitoring inter-module function calls. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 109 –118, 2000. 13

[33] R. Gwadera, M. Atallah, and W. Szpankowski. Reliable detection of episodes in event sequences. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 67 – 74, 19-22 2003. 12

[34] Intel Corporation. *Using the RDTSC Instruction for Performance Monitoring*, 1997. 57, 59

[35] Jeong-Hoon Ji, Gyun Woo, Hyung-Bae Park, and Ju-Sung Park. Design and implementation of retargetable software debugger based on gdb. In *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on*, volume 1, pages 737 –740, 11-13 2008. 61

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

3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. 18

[37] Holger Kienle and Urs Holzle. Introduction to the suif 2.0 compiler system. Technical report, Santa Barbara, CA, USA, 1997. 52

[38] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: a formal framework for specifying, implementing, and reasoning about execution monitors. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 338–352, New York, NY, USA, 1991. ACM. 14

[39] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. Low overhead program monitoring and profiling. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 28–34, New York, NY, USA, 2005. ACM. 17

[40] Naveen Kumar, Jonathan Misurda, Bruce R. Childers, and Mary Lou Soffa. Instrumentation in software dynamic translators for self-managed systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 90–94, New York, NY, USA, 2004. ACM. 21

[41] Jean J. Labrosse. *MicroC OS II: The Real Time Kernel*. CMP Books, 2002. 25

[42] Edward Lee and Craig Zilles. Branch-on-random. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 84–93, New York, NY, USA, 2008. ACM. 4

[43] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154, New York, NY, USA, 2003. ACM. 22

[44] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM*

SIGPLAN conference on Programming language design and implementation, pages 15–26, New York, NY, USA, 2005. ACM. 22

[45] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM. 16, 20

[46] Edu Metz, Raimondas Lencevicius, and Teofilo F. Gonzalez. Performance data collection using a hybrid approach. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 126–135, New York, NY, USA, 2005. ACM. 4

[47] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce, Irvin Karen, L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28:37–46, 1995. 16

[48] Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa. Demand-driven structural testing with dynamic instrumentation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 156–165, New York, NY, USA, 2005. ACM. 17, 22, 23

[49] A. W. Moore, A. J. McGregor, and J. W. Breen. A comparison of system monitoring methods, passive network monitoring and kernel instrumentation. *SIGOPS Oper. Syst. Rev.*, 30(1):16–38, 1996. 14

[50] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag. 52

[51] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM. 19, 20

[52] Ahmet Özmen. An entropy-based algorithm for data elimination in time-driven software instrumentation. *J. Syst. Softw.*, 82(5):907–913, 2009. 17

[53] B. Parhami. Defect, fault, error,..., or failure? *Reliability, IEEE Transactions on*, 46(4):450 –451, dec 1997. 9

[54] Chenglian Peng, Baifeng Wu, and Xiaoguang Sun. Test by distributed monitoring. In *Test Symposium, 1999. (ATS '99) Proceedings. Eighth Asian*, pages 218 –223, 1999. 15

[55] W.N. Robinson. Monitoring software requirements using instrumented code. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pages 3967 – 3976, 7-10 2002. 14

[56] W.N. Robinson. Implementing rule-based monitors within a framework for continuous requirements monitoring. In *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on*, pages 188a – 188a, 03-06 2005. 14

[57] Tetsuo Saitou, Mitsugu Suzuki, and Tan Watanabe. Dominance analysis of irreducible cfgs by reduction. *SIGPLAN Not.*, 40(4):10–19, 2005. 52

[58] Raul Santelices and Mary Jean Harrold. Efficiently monitoring data-flow test coverage. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 343–352, New York, NY, USA, 2007. ACM. 15

[59] M. Schatten. Reasonable python or how to integrate f-logic into an object-oriented scripting language. In *Intelligent Engineering Systems, 2007. INES 2007. 11th International Conference on*, pages 297 –300, june 2007. 61

[60] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 387–396, Washington, DC, USA, 1996. IEEE Computer Society. 13

[61] Alex Shye, Matthew Iyer, Vijay Janapa Reddi, and Daniel A. Connors. Code coverage testing using hardware performance monitoring support. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 159–163, New York, NY, USA, 2005. ACM. 4

[62] Alex Shye, Matthew Iyer, Vijay Janapa Reddi, and Daniel A. Connors. Code coverage testing using hardware performance monitoring support. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 159–163, New York, NY, USA, 2005. ACM. 79

[63] J. Sosnowski and M. Poleszak. On-line monitoring of computer systems. In *Electronic Design, Test and Applications, 2006. DELTA 2006. Third IEEE International Workshop on*, page 5 pp., 17-19 2006. 13

[64] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004. 19, 20

[65] K.S. Templer and C.L. Jeffery. A configurable automatic instrumentation tool for ansi c. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 249 –258, 13-16 1998. 21

[66] David Tennenhouse. Proactive computing. *Commun. ACM*, 43(5):43–50, 2000. 25

[67] Henrik Thane. Monitoring, testing and debugging of distributed real-time systems, 2000. 8, 10

[68] Mikkel Thorup. All structured programs have small tree width and good register allocation. *Inf. Comput.*, 142(2):159–181, 1998. 41

[69] Ben L. Titzer and Jens Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 59–68, New York, NY, USA, 2005. ACM. 17

[70] Zhonglei Wang, Antonio Sanchez, and Andreas Herkersdorf. Scisim: a software performance estimation framework using source code instrumentation. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 33–42, New York, NY, USA, 2008. ACM. 18, 19

[71] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The suif compiler system: a parallelizing and optimizing research compiler. Technical report, Stanford, CA, USA, 1994. 52

[72] Yat Fai Alfred Wong. Policy driven software monitoring, 2007. 12

[73] J.C. Yan. Performance tuning with aims-an automated instrumentation and monitoring system for multicomputers. In *System Sciences, 1994. Vol.II: Software Technology, Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 2, pages 625 –633, 4-7 1994. 16

[74] Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. Anomalous path detection with hardware support. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 43–54, New York, NY, USA, 2005. ACM. 20

[75] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112, New York, NY, USA, 2006. ACM. 22