Efficient Online Path Profiling

A Thesis submitted for the degree of

Doctor of Philosophy

in the Faculty of Engineering

by

Kapil Vaswani

Department of Computer Science & Automation

Indian Institute of Science

Bangalore, India

October, 2007

Dedicated to

my beloved parents

Efficient Online Path Profiling

Submitted for the degree of Doctor of Philosophy
September 2007

Abstract

Most dynamic program analysis techniques such as profile-driven compiler optimizations, software testing and runtime property checking infer program properties by *profiling* one or more executions of a program. Unfortunately, program profiling does not come for free. For example, even the most efficient techniques for profiling *acyclic, intra-procedural paths* can slow down program execution by a factor of 2. In this thesis, we propose techniques that significantly lower the overheads of profiling paths, enabling the use of path-based dynamic analyzes in cost-sensitive environments.

Preferential path profiling (PPP) is a novel software-only path profiling scheme that efficiently profiles given subsets of paths, which we refer to as *interesting* paths. The algorithm is based on the observation that most consumers of path profiles are only interested in profiling a small set of paths known a priori. Our algorithm can be viewed as a generalization of the Ball-Larus path profiling algorithm. Whereas the Ball-Larus algorithm assigns weights to the edges of a given CFG such that the sum of the weights of the edges along each path through the CFG is unique, our algorithm assigns weights to the edges such that the sum of the weights along the edges of interesting paths is unique. Furthermore, our algorithm attempts to achieve a minimal and compact encoding of the interesting paths; such an encoding significantly reduces the overheads of path profiling by eliminating expensive hash operations during profiling. Interestingly, we find that both the Ball-Larus algorithm and PPP are essentially a form of arithmetic coding. We use this connection to prove that the numbering produced by PPP is optimal.

We also propose a programmable, non-intrusive *hardware path profiler* (HPP). The hardware profiler consists of a path detector that detects paths by monitoring the stream of retiring branch instructions emanating from the processor pipeline. The path detector can be programmed to detect various types of paths and track architectural events that occur along paths. The second component of the hardware profiling infrastructure is a Hot Path Table (HPT), that collects accurate hot path profiles.

Our experimental evaluation shows that PPP reduces the overheads of profiling paths to 15% on average (with a maximum of 26%). The algorithm can be easily extended to profile inter-procedural paths at minimal additional overheads (average of 26%). We modeled HPP using a cycle-accurate superscalar processor simulator and find that HPP generates accurate path profiles at extremely low overheads (0.6% on average) with a moderate hardware budget. We also evaluated the use of PPP and HPP in a realistic profiling scenarios. We find that the profiles generated by HPP can effectively replace expensive profiles used in profile-driven optimizations. We also find that even well-tested programs tend to exercise a large number of untested paths in the field, emphasizing the need for efficient profiling schemes that can be deployed in production environments.

Declaration

Acknowledgments

It gives me great pleasure to acknowledge faculty, colleagues and friends who have contributed immensely to this thesis and to four eventful years at the Indian Institute of Science. First and foremost, I would like to acknowledge my advisors, Prof. Y. N. Srikant and Prof. Matthew Jacob, for their extraordinary guidance, inspiration and steadfast support through these years. Under their guidance, I found the freedom to express myself without inhibitions and explore my interests. The depth and foresight in their thought processes was exemplary. Working with them has been an absolute honour.

I would also like to acknowledge P. J. Joseph, my colleague at the Department of Computer Science and Automation, who was instrumental in shaping my thoughts and my research over these years. I am grateful to him for volumes of technical advise that he was ever so forthcoming with during our discussions and for his inspirational and calming presence. I would like to acknowledge my co-researcher Aditya Thakur for his critical contributions to the thesis. I thoroughly enjoyed my interactions with him in reading group meetings and discussions over a cup of coffee. Working and collaborating with him was a truly enriching experience.

I am extremely grateful to Aditya Nori, Trishul Chilimbi and Sriram Rajamani, researchers at Microsoft Research. They brought a fresh perspective to the problems we collaborated on and were never short of support and encouragement. I learnt a great deal from the way they approached problems, their persistence, and their attention to detail. I also thank Michael Bond for his help with the Scale compiler and Microsoft's Phoenix compiler team for their assistance with the path profiler implementation. I also thank Prof. Chiranjib Bhattacharya, Prof. Sirish Shevade, Prof. Priti Shankar and Prof. Govindarajan for their technical inputs.

I owe a great deal to my colleagues Anand Vardhan, Sujit and Bharath Kumar for creating a pleasant and enriching environment at the Compiler lab. Their presence, cooperation and advise went a long way in ensuring an enjoyable stay at IISc. I am also thankful to Mr. Pushparaj, scientific officer at the Department of CSA for his help in setting up some of the computational facilities, without which much of our research would not have been possible. I sincerely thank the staff at the Department of CSA, including Mr. George Fernandes and Mrs. Lalitha for handling the plethora of administrative matters ever so efficiently.

I would also like to acknowledge my friends Neerav Abani and Joseph Viathara for their company through the years. Finally, I owe a great deal to my parents and my brother Nikhil for being extremely patient and supportive in difficult times and for having trusted and encouraged me through the years. They have and will continue to be role models for me.

Contents

List of Figures

List of Tables

"To measure is to know"

Lord Kelvin

Chapter 1

Introduction

We are in the midst of a computing revolution. Our lives are increasingly dependent on computing systems that range from tiny mobile devices to large data centers that drive the internet. Ideally, we would like to ensure that the systems we use and rely so heavily on are provably correct, reliable, cost-effective and deliver high performance. However, this is rarely the case in practice. Systems deployed in the real world break down frequently despite rigorous design, development and testing procedures. And performance remains a real concern despite significant advances in processor technology.

This apparent fragility of modern-day systems can be attributed to the ever-increasing size and complexity of both hardware and software components. Not long ago, most applications were designed to solve relatively simpler problems, were developed from scratch by small teams of expert programmers and ran on relatively simple hardware. Consequently, it was possible for programmers to reason about both the correctness and performance of most parts of the system. This is rarely the case with systems we build today. Software applications routinely span several million lines of code. These applications are typically composed from a large number of distributed components written in different languages and developed simultaneously by geographically distributed teams of programmers. Very often, programmers are expected to maintain and extend large and unfamiliar code bases. The hardware that runs these applications is equally complex with multiple cores, complex memory, network and I/O subsystems being the norm.

One of the primary reasons why programmers can build software systems at such a large scale is the presence of several layers of abstraction in the programming stack. For instance, programming models such as object-oriented programming and aspect oriented programming allow programmers to write code at a higher level of abstraction. Such programming models also simplify the use of libraries and off-the-shelf components, enabling large-scale reuse of software. Virtual-machine based execution adds several other layers of abstraction, enabling features such as architecture-independence, runtime optimizations, automatic memory management and memory safety via type checking and process

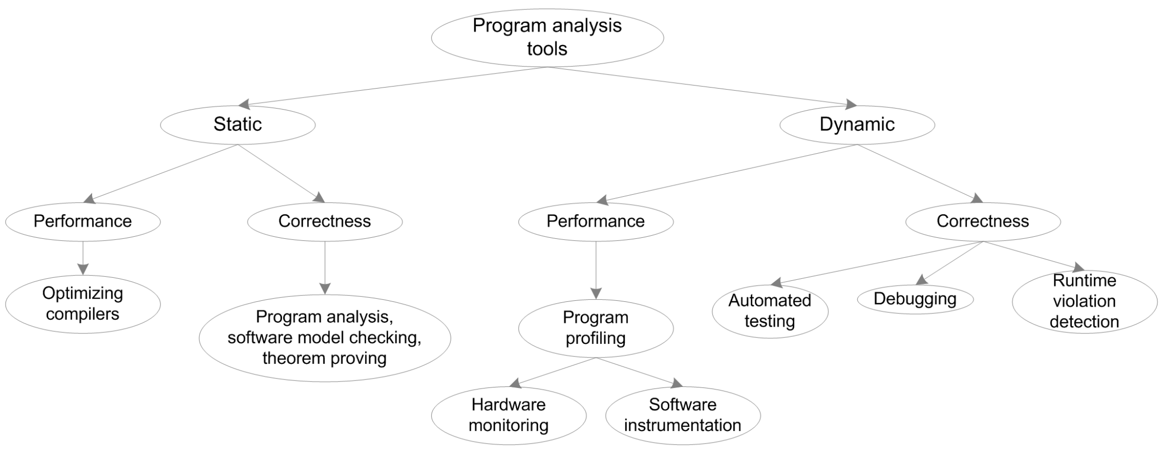


Figure 1.1: Overview of automated tools and techniques for analyzing performance and correctness of programs.

isolation. Microprocessor vendors have used the Instruction Set Architecture (ISA) as another layer of abstraction to improve processor performance while maintaining backward compatibility. The ISA, which defines aspects such as the instructions supported by the processor, the native data types, number of architectural registers etc, serves as the external interface of a processor family. Processor architects are free to change the processor micro-architecture (i.e. the implementation of the ISA) as long as the micro-architecture adheres to the ISA. This layer of abstraction has enabled several micro-architectural optimizations such as caching and speculative execution and has even allowed architects to change the internal model of execution via techniques such as out-of-order execution.

While these layers of abstraction are critical to the process of designing large scale software systems, they have an unwarranted side-effect. These layers introduce a disconnect between the programmer's view of the system and the code that eventually runs on the hardware. This disconnect makes it virtually impossible for programmers to manually reason about a system's performance, and to a certain extent, about its correctness. Consider for instance the use of off-the-shelf libraries. A library is typically associated with an external interface and implicit rules that govern the use of the library. Unless these rules are explicitly stated and interpreted correctly, a programmer could unintentionally introduce a bug by not conforming to the rules. The use of libraries also makes it harder for programmers to identify and alleviate performance bottlenecks since the bottlenecks may lie deep within the library code. Similarly, the use of OO features, JIT compilation, garbage collection and micro-architectural optimizations such as multi-level caches, speculation and out-of-order execution also add to the complexity of reasoning about an application's performance.

1.1 Automated program analysis

One way of bridging the divide between the programmer's and the machine's view of an application is through automated program analysis tools. Ideally, such tools should automatically (a) identify parts of an application that perform sub-optimally, infer the reasons behind the sub-optimal performance and suggest possible solutions to alleviate performance anomalies, and (b) detect whether an application meets a given correctness criteria, identify the *root cause* of any violations and if possible, suggest a suitable fix. It is now widely recognized that such tools are critical to the development of large scale computing systems and their design continues to be an area of active research (see Figure 1.1 for a classification of some of the commonly used program analysis tools).

There exist two fundamentally different approaches to the design of automatic program analysis tools. **Static program analysis** methods infer program properties simply by analyzing the program's source code. For example, compilers can statically detect and eliminate redundant expressions using data flow analysis. Similarly, a software model checker can detect whether a program violates a given safety property by building an abstract model of the program from its source code. On the other hand, **dynamic program analysis** techniques infer program properties by *monitoring* the program during one or more of its executions. A program's execution may be monitored by injecting probes into the program (instrumentation), via dedicated hardware or by observing the program's response to a set of inputs.

Because of this difference in approach to program analysis, static and dynamic techniques exhibit different characteristics. Static analysis techniques can infer *stronger* program properties i.e. properties valid across all inputs to the program. For instance, a compiler flags an expression as redundant only if the expression is redundant along all paths from program entry to the statement containing the expression. Similarly, a static verification tool certifies a program to be correct only if the correctness criteria is met for all inputs. Furthermore, static analysis techniques can detect defects early in the software design cycle. On the other hand, a dynamic analysis can only prove properties based on observations made during a finite number of runs of the program. A dynamic race detection algorithm, for instance, may be able to detect the presence of a race condition in a given run of a multi-threaded program, but cannot prove that the program is race-free. For these reasons, static analysis tools are generally desirable.

Unfortunately, despite significant recent progress in the design of scalable static analysis tools, the use of automated static analysis tools has been restricted to proving simple properties of large programs or complex properties of simple programs. For example, modern compilers for languages like Java can statically detect simple forms of memory safety violations such as stack overflows even in large programs. On the other hand, static tools that analyze complex properties such as the shape of heap-allocated data structures,

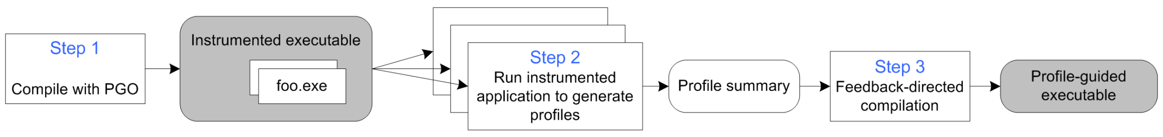

Step 1
Compile with PGO
Instrumented executable
foo.exe
Step 2
Run instrumented application to generate profiles
Profile summary
Step 3
Feedback-directed compilation
Profile-guided executable

Figure 1.2: Overview of compile-profile-recompile cycle that generates binaries tuned to a specific program behavior.

presence of races or determine the worst case execution time of a program do not scale to large programs because of the prohibitively large state space involved in the analysis. Many static analysis tools address the state space explosion problem by loosing precision. For instance, most compilers prefer using imprecise and inexpensive pointer analysis algorithms, trading off compilation time with optimization opportunities. Dynamic techniques do not suffer from these deficiencies because the analysis is restricted to a small number states that occur during one or more executions of a program. Furthermore, most dynamic analysis techniques are extremely precise because they can precisely track the values of program variables at each program point, witness all state transitions and have access to information that is available only at runtime (such as the state of the heap, I/O and network requests and hardware performance metrics). It is therefore not surprising that several dynamic tools have proven successful in analyzing complex properties for large programs. We now discuss three scenarios where dynamic techniques have been employed with great success.

Feedback-directed compiler optimizations. Traditionally, optimizing compilers were designed as static tools that identify, and where ever possible, eliminate inefficient operations and coding errors in programs. In the static compilation model, compiler optimizations statically analyze a program's source code and eliminate inefficient operations via one or more semantics-preserving program transformations. It turns out that in most optimizations, identifying inefficient code is not always easy because both the cost of the operation and the benefit of the optimization depend on several external factors such as program input, interactions between optimizations, and the underlying micro-architecture. For instance, given a choice, it may be beneficial to optimize a piece of code that is more likely to be executed frequently rather than code that is rarely executed. Since many of these factors are inherently dynamic, static compilers are unable to resolve these cost-benefit tradeoffs, which, in turn, reduces the effectiveness of optimizations.

Feedback-directed optimizations [45] overcome these limitations by using a dynamic approach to optimizations. To employ these optimizations, programmers follow a *compile-profile-recompile* cycle (illustrated in Figure 1.2). During the first compilation step, the program is instrumented to collect information about the its runtime behavior. Next, the instrumented binary is executed using a set of *representative* inputs provided by the programmer, generating a set of *profiles* in the process, where each profile characterizes an aspect of the program's dynamic behavior. Finally, the profiles are fed back to the compiler, which uses them to resolve cost-benefit trade-offs that occur during optimizations. In essence, the use of feedback in the form of execution profiles *biases* the optimizations towards the common case behavior of a program, resulting in highly tuned binaries. Feedback-directed optimizations are now supported by most industry-strength compilers and experience suggests that these optimizations can result in substantial performance gains.

Software testing. Software testing is a dynamic analysis technique that attempts to find errors by subjecting a given program to a number of inputs and checking the outputs against the expected response. Unlike static analysis, testing does not exhaustively verify correctness for all possible program inputs and hence, does not guarantee program correctness. However, it has been found that the reliability of software components is correlated with the degree of testing i.e. components tested more rigorously are less likely to fail when deployed. Testing is also preferred because it does not require a property or a specification of software artifact, both of which are hard to obtain (although the availability of either can improve testing efficiency). Furthermore, testing methods are not constrained by the size or complexity of programs being tested. Both manual and automated testing techniques have been employed for checking large programs, ranging from compilers to enterprize applications.

Runtime invariant detection and checking. The criteria for correctness of programs are often expressed as *invariants*. An invariant is a program property that does not change during the course of a program's execution. For example, programmers often use assert statements to specify intended invariants; expressions in the assert statements define the set of values program variables are allowed to take. Checking whether user specified invariants are not violated along any path in the program is central to program verification. Unlike static analyzes, which try and prove invariants statically, runtime invariant checking is concerned with checking for invariant violations during program execution. This is typically achieved by placing probes at specific locations and tracking the values of program variables. For instance, the Java runtime checks for violations of the (implicit) invariant that array indices do not exceed the declared bounds. Similarly, a dynamic race detector [44] checks for violations of the invariant that all accesses to a shared memory location are ordered by the happens-before relation.

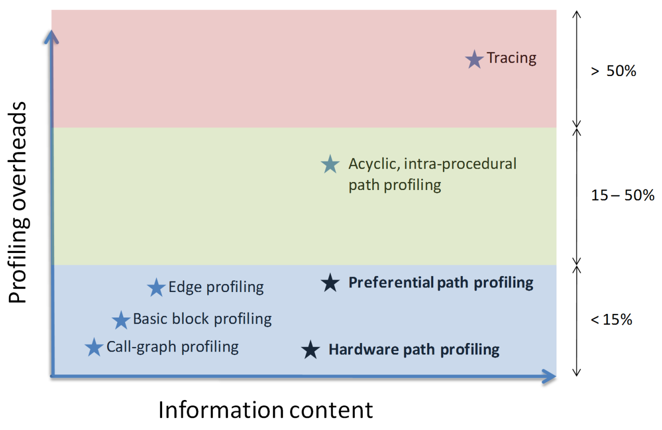


Figure 1.3: Trade-off between information content and profiling overheads for various control flow profiling techniques.

1.2 Efficiency of dynamic program analysis

Virtually all dynamic analysis techniques have a common characteristic - the analyzes are based on **profiling/monitoring** one or more executions of the program. Unfortunately, although significantly more efficient than static analysis, program profiling does not come for free. Depending on the nature of profile data being collected, profiling may slow down a program's execution considerably. High profiling overheads adversely effect the efficiency of dynamic analysis techniques and often preclude their use in cost-sensitive settings.

The close connection between profiling overheads and the efficiency of dynamic analysis is best illustrated by considering the class of *control flow profiling* techniques. As the name suggests, a control flow profiling technique attempts to characterize the flow of control in a program's execution by identifying parts of the program exercised in a given run, and optionally, the temporal order and frequency with which they were exercised. Figure 1.3 illustrates some of the most commonly used control-flow profiling techniques and compares their overheads. *Program tracing*, a profiling technique that generates an exhaustive trace of events that occur during a program's execution, lies on one end of the spectrum. While a trace records information in sufficient detail for any dynamic analysis technique, tracing programs is extremely expensive. Not surprisingly, the use of full program tracing has been mostly restricted to research environments. On the other end of the spectrum are a number of *point profiling* techniques, which count the frequency with which specific points in the program are exercised in a given execution. For instance, a basic block profiler tracks the number of times each basic block in the program was executed. Similarly, a call-graph profiler counts the frequency with which a function was called from each of its callers.

Point profiling techniques are simple and extremely inexpensive because collecting these profiles requires only a few additional instructions per program point. However,

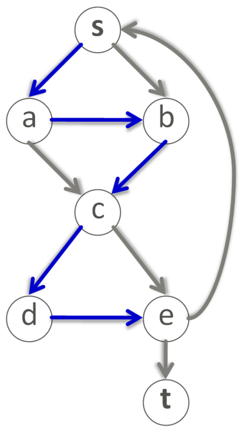


Figure 1.4: Acyclic, intra-procedural paths through a control flow graph.

this increased profiling efficiency comes at the cost of a loss in precision. Point profiles are coarse abstractions of a programs control flow behavior. These profiles are *lossy* because a programs actual control flow behavior cannot be accurately recovered from a point profile. In other words, a given point profile may correspond to several executions of a program, some of which may not even be feasible. Recent research has shown that this loss in precision may reduce the efficacy of dynamic analysis techniques that use these profiles. For instance, it has been shown that profile-driven optimizations such as superblock scheduling are less effective if driven by point profiles instead of more accurate profiles. Similarly, software testing is less effective if code coverage is measured using point profiles. This trade-off is fundamental to any profiling technique and the way this trade-off is resolved has a large bearing on the efficiency of dynamic analysis techniques.

1.3 Path profiling

In the case of analyses that use control flow profiles, this trade-off between precision and profiling overheads can be resolved using a profiling technique known as **path profiling** (proposed by Ball and Larus [6]). A path profiler monitors the control flow of a program at the granularity of **acyclic, intra-procedural paths**. An acyclic, intra-procedural path (simply referred to as path for the rest of this article) is defined as any sequence of instructions that originates at the beginning of a procedure or a loop and terminates at the end of a procedure or at a loop back-edge. Consider the control flow graph shown in 1.4. The node s represents the entry node to the function and t represent the exit/return node. The sequence of blocks sbcdet constitutes a path that begins and ends at a procedure boundary. Similarly, blocks sabcde constitute a path that terminate at a loop back-edge.

Much like a point profiler, a path profiler tracks the frequency with which each path was exercised in a given execution. Path profiles are a succinct and pragmatic abstraction of a programs dynamic control-flow behavior. Control-flow information captured by a path profile is much more precise than basic block or edge profiles. Also, path profiles are considerably smaller than complete instruction traces. Recording program paths has proved valuable in a wide variety of areas such as computer architecture, compilers, debugging, program testing, and software maintenance [5]. Several compiler optimizations perform better when trade-offs are driven by accurate path profiles [2, 20, 55]. Program paths are also a more credible way of measuring coverage of a given test suite. In addition, abstractions of paths can help automatic test generation tools generate more robust test cases. Finally, program path histories often serve as a valuable debugging aid by revealing the instruction sequence executed in the lead up to interesting program points.

While the benefits of using path profiles over point profiles are obvious, let us consider how path profilers fare on the overheads front. Our measurements of an implementation of a state-of-the-art path profiler [6] indicate an average execution time overhead of 50% with as much as a 132% overhead in the worst case and other studies report similarly high overheads [9]. Although this represents a significant reduction over whole program tracing, the overheads are already an order of magnitude more than point profiles. The overheads of profiling increase manifold if the scope of profiling is extended beyond acyclic, intra-procedural paths [30, 48]. This has limited the use of path profiles in favor of point profiles in tasks such as measuring test coverage and driving profile-guided optimizations such as code placement, inlining, unrolling, and superblock scheduling.

Apart from these traditional usage scenarios, we envisage the use of path profiling in several cost-sensitive environments. For instance, in **residual path profiling**, we are interested in determining the set of paths that a deployed program executed in the field but were not exercised during testing. This information could be used to improve and augment test suites, and if included with bug reports resulting from field failures, could help pinpoint the root cause of errors. Another scenario involves ascertaining whether paths that were identified as hot paths during testing and used to optimize the program continue to remain hot during field usage. In addition, we might want to gather detailed information about these paths, such as cache misses, page faults, and variations in execution time, without resorting to sampling techniques [39]. Finally, we might be interested in efficiently tracking a subset of paths in deployed software that meet a certain criteria, for example, paths that access safety or security critical resources, or that exercise an error prone code region. *These tasks can be performed only if paths can be profiled at low cost and high accuracy.*

1.4 Contributions of the thesis

In this thesis, we propose two techniques low overhead path profiling techniques namely **preferential path profiling** (PPP) [49] and **hardware path profiling** (HPP) [51] that enable the use of path profiles in cost-sensitive environments.

PPP is a novel software-only path profiling scheme that efficiently profiles arbitrary path subsets, which we refer to as **interesting** paths. PPP is based on the observation that several consumers of path profiles rely on profiling a small subset of paths that are known *a priori*, which we refer to as *interesting paths*. Our algorithm can be viewed as a generalization of the Ball-Larus algorithm [6]. The Ball-Larus algorithm assigns weights to the edges of a given CFG such that the sum of the weights of the edges along each path through the CFG is unique. Our algorithm generalizes this notion to a subset of paths; it assigns weights to the edges such that the sum of the weights along the edges of the interesting paths is unique. Furthermore, our algorithm attempts to achieve a minimal and compact encoding of the interesting paths; such an encoding significantly reduces the overheads of path profiling by eliminating expensive hash operations during profiling. In addition, our profiling scheme separates interesting paths from other paths and is able to classify paths during program execution. The ability to classify paths is important for many scenarios such as residual path profiling. We also propose an inter-procedural extension of PPP called IPPP, which profiles paths that span across procedures [13]. Interestingly, we find that both the Ball-Larus algorithm and PPP are essentially a form of arithmetic coding [52], a technique commonly used for data compression. We use this connection to prove an optimality result for the compactness of path numbering produced by PPP.

Unlike PPP, HPP is a programmable, non-intrusive scheme that profiles paths in hardware. HPP detects paths by monitoring the stream of retiring branch instructions emanating from the processor pipeline during program execution. HPP's path detection mechanism can be programmed to detect several types of paths and track various architectural events that occur along paths, a task hard to emulate using software path profilers. The detector generates a stream of path descriptors, each describing a path that has just executed. The second component of HPP is a *Hot Path Table* (HPT), a compact hardware structure that processes the stream of path descriptors generated by the path detector. The HPT is designed to collect accurate hot path profiles, irrespective of the type of path being profiled, the duration of profiling and the architectural metric associated with paths.

We have implemented and evaluated both path profiling schemes over a number of standard benchmark programs. Our evaluation shows that the overheads of our profiling schemes are comparable or even lower than point profiles (Figure 1.3). The two profiling schemes make for an interesting comparison. While PPP generates accurate profiles, it incurs slightly higher overheads (15% on average). HPP on the other hand requires hardware support and generates profiles with slightly lower accuracy but imposes significantly lower overheads (0.6% on average). Consequently, PPP is naturally suited for applications in software testing and invariant checking where the accuracy of profiling is critical. On the other hand, HPP is geared towards use in online, profile-driven and micro-architectural optimizations, where the overheads of profiling are the primary concern.

1.5 Organization of the thesis

This rest of thesis is organized into five chapters. Chapter 2 contains a background to various hardware and software profiling schemes and illustrates their use in several dynamic analysis techniques. Chapter 3 describes the preferential path profiling algorithm, the inter-procedural version of the algorithm and its application to residual path profiling. Chapter 4 discusses the connection between path profiling and arithmetic coding and provides the proof of optimality of preferential path profiling based on this connection. Chapter 5 presents the hardware path profiling scheme. We conclude with pointers for future work in Chapter 6.

Chapter 2

Literature Survey

The notion of profiling paths through a program execution has always generated significant academic interest. However, the utility of path profiles in understanding and improving program behavior has been recognized only recently. This had lead to significant interest in developing efficient and scalable schemes for profiling paths. In this chapter, we discuss some of the software and hardware path profiling schemes and describe a few areas where path profiling has added significant value.

2.1 Software path profiling

Ball and Larus [6] first demonstrated the feasibility of obtaining path profiles in software automatically and efficiently. The proposed instrumentation-based profiling scheme splits the dynamic instruction stream into acyclic, intra-procedural sequences, also referred to as Ball-Larus (BL) paths and tracks the frequency of occurrence of each path. A detailed description of this scheme can be found in Chapter 3.

Young and Smith [56] define a path as the last k branches of the execution trace leading to an edge. They propose a scheme for profiling k-paths by first constructing a *path CFG*, where each node corresponds to a k-path in the original graph and each edge represents the last edge of the path. Although k-paths have found utility in specific compiler optimizations like superblock scheduling [55], profiling these paths is significantly more expensive than profiling Ball-Larus paths (for reasonable values of k).

Subsequent research in path profiling has focused on alleviating two drawbacks of Ball-Larus profiling. First, path profiling does not provide information about a program's control flow across procedure and loop boundaries, limiting its utility in inter-procedural and aggressive loop optimizations. Efforts to overcome this limitation include the Whole Program Path (WPP) [30] and extended path profiling [48]. A WPP is a complete sequence of the acyclic, intra-procedural paths traversed by a program. The sequence is compressed online by generating an equivalent context free grammar, which is compactly represented as a DAG. Despite high compression ratios, the WPP's size and profiling overheads hinder its widespread use. Inter-procedural path profiling [33] extends Ball-Larus profiling beyond intra-procedural paths. As a compromise between Ball-Larus paths and the WPP, Tallam et al [48] propose the notion of *extended* paths − paths that extend beyond loop or procedure boundaries. They also propose a profiling scheme that reduces overheads by approximating the frequencies of extended paths from a profile of overlapping path fragments. Both these techniques have considerably higher overhead than the Ball-Larus technique (average execution time overheads for profiling extended paths are reported to be nearly 4 times the overheads of Ball-Larus path profiling).

Several researchers have proposed a variety of techniques to reduce the overhead of Ball-Larus style path profiling [3, 4, 8, 9, 25, 53], a critical factor in cost-sensitive dynamic optimization systems. Selective path profiling [3] uses a variation of Ball-Larus numbering where edges are visited in a specific order to ensure that interesting paths are assigned a unique number that is higher than the non-unique numbers assigned to other paths, while minimizing the number of counter updates needed to compute the path number. The proposed numbering scheme is a variation of the Ball-Larus numbering scheme where edges that occur along interesting paths are numbered later than other edges. However, they find that once the number of interesting paths increases beyond a small threshold, their edges covered most of the DAG and their technique offered little advantage over Ball-Larus numbering. In addition, they make no attempt to ensure that the interesting paths are compactly numbered. Instead of minimizing the number of counter updates needed to compute a path number, we optimize the compactness of numbers assigned to interesting paths. This reduces overhead by enabling the use of a path array in place of a hash table. In comparison, our compact numbering scheme is effective even when the number of interesting paths is large.

Both *targeted path profiling* [25] and *practical path profiling* [9] attempt to efficiently profile hot program paths starting from an edge profile by eliminating unneeded instrumentation. Targeted path profiling eliminates profiling cold paths by excluding cold edges and not instrumenting paths that the edge profile predicts well. It uses Ball-Larus numbering for labeling the remaining paths. Practical path profiling attempts to improve over targeted path profiling using a variety of techniques to eliminate a larger number of paths. It also performs intelligent instrumentation placement to further reduce overhead. To minimize overhead, practical path profiling may need to classify warm edges as cold and consequently could compromise the quality of the path profile. It also uses Ball-Larus numbers to uniquely identify the remaining paths.

Structural path profiling [53] is a lossy, online path profiling scheme that achieves lower path profiling overheads by partitioning methods into a hierarchy of nested graphs based on their loop structure. Each resulting graph has a much smaller number of paths and can be profiled independently. For instance, in a function with a nested loop, the outer loop is profiled first. Once path profiles for the outer levels have been collected, inner level loops are profiled until the bottom level is reached. The profiling algorithm constructs a global path profile by adjusting the local profiles of the structure graphs. *Continuous path profiling* is another lossy variation of the Ball-Larus profiling scheme directed at online environments. The scheme uses a combination of full Ball-Larus instrumentation and sampling to mitigate the overheads of maintaining hash tables while path profiling. While methods are instrumented with counter update instructions as per the Ball-Larus scheme, the hash tables are updated only occasionally.

Although these path profiling schemes reduce profiling overheads without much loss in accuracy, they cause an increase in the time required to obtain representative profiles, which in turn delays the optimization process and results in fewer optimization opportunities being exploited. Moreover, it is not clear whether these schemes can be extended for profiling other varieties of paths or for profiling program binaries, as is required in binary translation/optimization systems. Our software profiling technique is orthogonal to both as it proposes a new dense numbering scheme for interesting paths that minimizes the overhead of profiling these paths. It is also more general as it can be applied to scenarios such as residual path profiling (detecting paths not exercised by a test suite), where the techniques that targeted/continuous/structred path profiling use to reduce instrumentation overheads do not apply.

2.2 Hardware profiling

The importance of efficient program profiling techniques in improving performance has also been recognized by computer architects. Most modern processors provide architectural support for performance monitoring, typically in the form of event counters [46, 47]. To meet the requirements of profile-guided optimizations, hardware profilers that construct approximate point profiles using information from the processor have also been proposed [15, 17, 34].

Conte et al [15] propose an approach that uses the branch handling hardware present in most commercial processors for profiling purposes. The profiler samples the branch prediction hardware using kernel mode instructions at every kernel entry point to construct weights of a Weighted Control Flow Graph. Merten et al [34] describe a mechanism that monitors instructions at the retirement stage of the processor pipeline to identify program hotspots. ProfileMe [17] is a hardware framework that profiles instructions by tracking their progress through the pipeline and collecting information about associated events. The Relational Profiling Architecture [22] provides mechanisms that allow the software to issue queries about information regarding program behavior These queries are serviced by profiling hardware that monitors the pipeline. Zilles and Sohi [57] suggest a solution that advocates dedicated hardware for profiling in the form of a programmable profiling co-processor that receives profile samples from the main processor, processes it and feeds it back to the main processor. Our hardware profiler subsumes existing hardware profiling schemes and has comparatively wider applicability because of its ability to collect various types of path profiles and the added ability to associate architectural metrics with program paths.

2.3 Applications of path profiling

Path profiles have found applications in a wide variety of dynamic program analysis techniques. While many dynamic analysis techniques have benefitted by simply using richer path profiles instead of basic block or edge profiles, several other dynamic analyzes have been formulated using paths exclusively.

2.3.1 Profile-guided Optimizations

Profile-guided optimizations are a special class of compiler optimizations based on the philosophy of making the common case faster. For programs, the common case is defined by behavior that a program exhibits most frequently. For example, in a banking application, depositing and withdrawing money are likely to be the most common tasks whereas opening and closing of accounts not exercised as frequently. Traditional compiler optimizations do not distinguish between the common and uncommon case and consider all parts of the program as equally important during optimizations. On the other hand, profile-guided optimizations (PGO) use program profiles generated using a set of *representative inputs* to identify the common case and optimize the common case more aggressively. Consequently, these optimizations generate highly tuned program binaries that perform significantly better for the common case.

Traditionally, compilers have relied on simple point profiles to identify the common case and drive PGO, primarily because these profiles are simple and inexpensive to collect. More recently, several researchers have shown that the effectiveness of these optimizations can increased if path profiles are used instead. We now discuss a few optimizations that are known to benefit from richer profile information.

Path profile-guided superblock scheduling

Instruction scheduling is a compiler optimization that reschedules instructions (typically within a basic block) such that the resulting schedule utilizes hardware resources more efficiently, hides the latency of expensive operations and has a lower latency of execution. Superblock scheduling [23] is a variant of instruction scheduling that schedules code at the

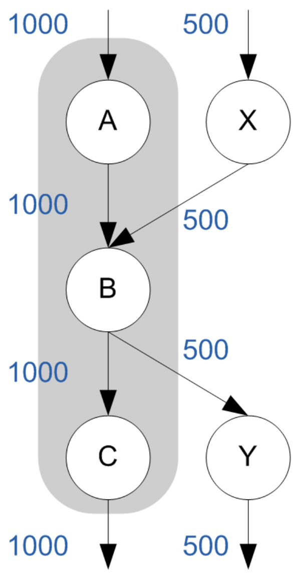


Figure 2.1: A scenario where an edge profile may not accurately reflect control flow behavior. If blocks ABC are candidates for superblock scheduling, the edge BY is a side exit. From the edge profile, it is not clear how often the blocks ABC are likely to complete. The frequency with which the side exit is taken lies between 0 and 500.

granularity of *superblocks* instead of basic blocks. A superblock is a single-entry, multiple-exit sequence of instructions with the additional characteristic that the last exit(s) (exits at the end of the superblock) are more likely to be taken than the others. In other words, once a superblock is entered, all the instructions in the superblock should be executed with high probability. In superblock scheduling, superblocks are formed using a control flow graph restructuring transformation known as *tail duplication*. Once superblocks, are formed, the the compiler schedules instructions in the superblock while assuming that the instructions belong to one basic block. Since the size of a superblock is typically much larger than the size of individual basic blocks, the scheduler has greater flexibility in reordering instructions, which results in better schedules. However, to to compensate for the effects of reordering instructions within the superblock and ensure correctness, the scheduler inserts compensation code along edges leading to the side exits.

Traditionally, superblock formation has relied on basic block or edge profiles to identify sequences of hot basic blocks that can be transformed into superblocks. Young et al [55] show that in certain scenarios, an edge profile may not accurately reflect the control flow behavior of a program. Figure 2.1 (adapted from [55]) illustrates one such scenario. In such scenarios, the superblock scheduler will create superblocks in which control flow escapes from one of the side exits and does not reach the last exit(s) as often as expected, which reduces the benefits of scheduling. In contrast, superblocks formed using path profiles are more likely to execute to completion, making best use of the optimized schedule.

Path-profile guided PRE

Occasionally, programmers and compiler introduce expressions whose value has already been evaluated in the program and is available for use at the statement containing the expression. Such expressions are said to be *redundant* and optimizations that detect and eliminate such expressions are an integral part of all optimizing compilers. Partial Redundancy Elimination (PRE) is an optimization that eliminates partially redundant expressions. An expression is said to be *partially redundant* if there exists a program point such that the expression is evaluated along all paths from the point to the exit (the expression is said to be *anticipable* at the program point) but the value of the expression is available only along a subset of paths from the entry to that point (the expression is *partially available* at the program point). PRE uses code hoisting to introduce instructions that evaluate the expression along paths where the expression is not redundant. This code transformation makes subsequent evaluations of the expression completely redundant and can be safely eliminated without introducing extra evaluations of the expression along any path in the program.

The traditional formulation of PRE is not applicable for expressions that are partially available and partially anticipable. This is because removing these expressions using PRE transformations as described above would introduce extra evaluations of the expression along some paths in the program. If these paths turn out to be frequently executed, the extra evaluations could slow down the program instead of improving performance. Gupta et al [20] propose a path profile-directed formulation of PRE that eliminates such expressions. The proposed approach uses path frequencies to determine the benefits of eliminating partially redundant expressions and the cost of extra evaluations introduced due to code hoisting. The optimization is performed only if the benefits exceeds the cost i.e. the cumulative frequencies of paths along with the expression is removed is higher than the frequency of paths along with an extra evaluation is introduced. A similar approach can be used to eliminate partial dead code [19].

Precise dataflow analysis using path profiles

Dataflow analysis is a program analysis technique used to detect facts about program variables that hold at various points in the program. A dataflow analysis is said to be conservative if the facts computed by the analysis at a program point hold along all paths from program entry to that point. The conservative nature of dataflow analysis ensures that compiler optimizations based on the analysis are always safe. However, the safety of compiler optimizations comes at a cost. A conservative dataflow analysis is forced to loose information at join points in the program. Consider the join point in Figure 2.2. if a dataflow fact $X = 2$ holds at the exit of node A but does not hold at the exit of node B, the analysis must conservatively assume that the the value of X is unknown at the entry

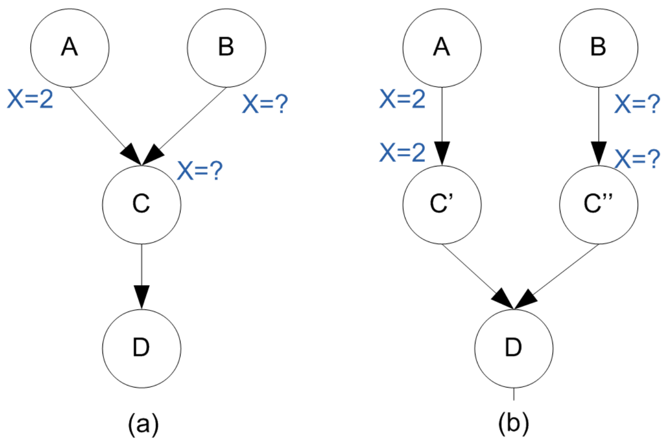


Figure 2.2: A control flow graph illustrating the loss in precision due to the conservative nature of data flow analysis. In (a), the dataflow fact X=2 holds at the exit of node A but not at the exit of B. Due to the merge, the analysis conservatively assumes that the dataflow fact does not hold at the entry of C. Figure (b) shows the version of the graph after splitting the merge. Here, the fact X=2 holds at the entry of node C'.

of node C. This loss in precision due to join points reduces the number of optimization opportunities and adversely effects the benefits obtained from compiler optimizations.

Ammons and Larus [2] propose an approach that improves the precision of data flow analysis by restructuring the control flow graph to eliminate join points along frequently executed paths identified using a path profile. Eliminating join points along hot paths ensures that the cost of restructuring the control flow, which involves some amount of code duplication, is more than offset by the increase in the number of optimization opportunities along these paths.

2.3.2 Software debugging and testing

Bugs in programs are associated with a *cause* and a *symptom* [7]. When bugs are found, either by verification tools, via in-house testing or in a deployed program, their symptoms are reported back to the programmer, typically in the form of a stack trace and a memory dump. Subsequently, much of the debugging effort is spent in finding the root cause of bugs from these symptoms, a process that involves tracing (possibly backwards) the sequence of events that lead to the symptom and identifying the sub-sequence that caused the program to transition into an invalid state. Paths are naturally suited for reasoning of this kind since paths efficiently capture a majority of event sequences that occur in a given program execution. Although existing software debuggers do not support paths explicitly, efforts have been make to make debuggers more path aware. Bruegge et al [11] propose the use of path expressions as a generic mechanism for specifying program behavior that a user wants to monitor in a debugging session. For example, the path expression `Open(Read`

`+ Write)*Close` encodes the correct usage of the file API. A path-aware debugger can translate this expression into a finite-state automata, monitor the automata in parallel with the program's execution, and invoke a user-specified action if the automata reaches an error or final state. Path profiles can be used for localizing faults directly by comparing profiles obtained from correct and faulty runs and identifying paths that whose execution is highly correlated with faulty runs [41].

Paths also provide a more credible way of measuring the effectiveness of a test suite. The number of distinct paths exercised by a test suite more accurately represents the range of behaviors tested than statement or basic block coverage [14]. For this reason, paths are ideal targets for automatic test case generation tools. These tools can identify paths that are likely to cause failures and determine inputs that drive the program along these paths.

2.3.3 Computer architecture

Paths exhibit several interesting characteristics that can be exploited to improve processor performance. Paths exhibit *locality* - programs usually execute a small number of paths repeatedly. Furthermore, the manner is which a program exercises paths is very predictable. The locality and predictability of paths is critical to micro-architectural optimizations such as branch prediction and trace caches.

A branch predictor allows a pipelined processor to eliminate control dependencies by predicting the direction and the target address of a branch instruction well before the branch is resolved and speculatively executing instructions on the predicted path. Early branch prediction schemes such as the bimodal predictor and local branch predictors relied completely on the local history of a branch to drive predictions. On the other hand, global branch prediction schemes such as gShare and path-based branch predictors [36] exploit the observation that a branch's behavior depends on the path taken to arrive at the branch. These predictors track the branch instruction's history separately for each path leading to the branch instead of aggregating the history for all paths. High branch prediction accuracy in high performance processors such as Intel's Xeon processors, AMD's K8 and the Alpha EV6 can be attributed to the use of global branch predictions schemes.

Trace caches [43] exploit locality in paths too, albeit in a different way. A trace cache stores frequently executed sequences of instructions known as traces. A trace contains instructions that span across several basic blocks from non-consecutive addresses in memory. A trace-based processor can predict [24] and fetch a whole trace at at time, reducing the average fetch latency. Because of locality in paths, a reasonably sized trace cache can be expected to hold a majority of the program's working set. Since traces fetched from a trace cache are executed atomically, processors can aggressively optimize the traces before placing them in the cache [37].

Chapter 3

Preferential Path Profiling

3.1 Introduction

Let us first examine why existing path profiling schemes incur relatively high overhead. The efficient path profiling scheme proposed by Ball and Larus, which forms the basis of all path profilers, assigns weights to edges of a control flow graph (CFG) such that all paths are allocated unique identifiers (i.e., the sum of the weights of the edges along every path is unique) [6]. During program execution, the profiler accumulates weights along the edges and updates an array entry that corresponds to this path identifier. Unfortunately, for functions with a large number of paths, allocating an array entry for all program paths is prohibitively expensive, if not infeasible. Consequently, path profiler implementations are forced to use a hash table to record path information for such functions. Although using a hash table is space efficient as programs typically execute only a small subset of all possible paths, it incurs significantly higher execution time overhead as compared to updating an array entry. Previous work has shown that hash tables account for a significant fraction of the overhead attributable to path profiling [25].

To address this problem, we propose *preferential path profiling* (PPP), a novel path profiling scheme that efficiently profiles arbitrary subsets of paths, which we refer to as interesting paths. Our algorithm can be viewed as a generalization of the Ball-Larus algorithm, which forms the core of all existing path profiler implementations. As mentioned earlier, the Ball-Larus algorithm assigns weights to the edges of a given CFG such that the sum of the weights of the edges along each path through the CFG is unique. Our algorithm generalizes this notion to a subset of paths; it assigns weights to the edges such that the sum of the weights along the edges of the interesting paths is unique. Furthermore, our algorithm attempts to achieve a minimal and compact encoding of the interesting paths; such an encoding significantly reduces the overheads of path profiling by eliminating expensive hash operations during profiling. In addition, our profiling scheme separates interesting paths from other paths and is able to classify paths during program execution.

The ability to classify paths is important for many applications such as residual path profiling. Interestingly, we find that both the Ball-Larus algorithm and PPP are essentially a form of arithmetic coding [42,52], a technique commonly used for universal data compression. We make use of this connection to prove an optimality result for the compactness of path numbering produced by PPP.

This chapter makes the following contributions. First, we describe a new algorithm, called preferential path profiling (PPP), for compactly numbering arbitrary path subsets that improves upon Ball-Larus numbering (Section 3.3). Next, we propose a low-overhead interprocedural path profiling scheme based on PPP in Section 3.4. We present an experimental evaluation of our PPP implementation that demonstrates it incurs significantly lower overhead as compared to Ball-Larus profiling (Section 3.6). In Section 3.5, we show that the PPP algorithm can be naturally adapted to perform intra-procedural and inter-procedural residual path profiling. Our experiments demonstrate that residual path profiling provides valuable quantitative information on the effectiveness of software testing.

3.2 Preliminaries

In this section, we briefly describe the Ball-Larus algorithm for profiling acyclic, intra-procedural paths through a control flow graph (CFG) of a program and motivate our problem using a simple example.

3.2.1 Definitions

For the purposes of profiling acyclic, intra-procedural paths (henceforth referred to as *paths*), each procedure in the program is represented as a *directed acyclic graph* (DAG). Each DAG is a graph $G = (V, E, s, t)$, where V represents nodes or basic blocks in the procedure, E is the set of edges between nodes and the nodes s and t are the start and exit nodes representing function entry and exit. The maps $src(e)$ and $dest(e)$ denote the source and destination nodes respectively of an edge e. For every node $v \in V$, $out(v)$ denote the set of edges emanating from v in G, and $succ(v)$ represents all the immediate successor nodes of v. An acyclic, intra-procedural path p is a sequence of nodes from s to t. The function $paths(G)$ refers to all acyclic, intra-procedural paths in G. The function $paths_G(e) : E \to 2^{paths(G)}$ represents all paths in G that traverse an edge e. Conversely, the function $edges : P \to 2^E$ maps every path p in G to the set of edges that belong to p.

An assignment of weights to the edges of G is represented as a map $W : E \to \mathbb{Z}$. The relation $pathid : P \to \mathbb{Z}$ maps each path to a *path identifier*, and is defined as follows.

$$pathid(p) \overset{\text{def}}{=} \sum_{e \in edges(p)} W(e)$$

```
procedure $computeBLIncrements$ $(G)$
Assume:
(a) $G = (V, E, s, t)$ is a DAG.
(b) $W : E \rightarrow \mathbb{Z}$ is an empty map.
Returns: The map $W$ defined for all edges such that every path in $G$ is assigned a unique weight.
1: $N_t := 1;$
2: for all nodes $v \in V$ in reverse topological order do
3:     $N_v := 0;$
4:     for all edges $e \in out(v)$ do
5:         $W(e) := N_v;$
6:         $N_v := N_v + N_{\mathsf{dest}(e)};$
7:     end for
8: end for
```

Figure 3.1: The Ball-Larus Algorithm.

3.2.2 Ball-Larus Profiling

The Ball-Larus profiling scheme works over the control flow graph representation of each procedure in the program. The analysis involved in the profiling scheme is modular because control flow across procedure boundaries is ignored. Hence, each procedure in the program is analyzed independently. Furthermore, since the goal is to profile acyclic paths only, the profiling scheme converts the control flow graph into a DAG, ignoring all control flow across loop boundaries. The conversion from the control flow graph to a DAG involves the following simple steps.

1. Identify loop back-edges in the control flow graph.
2. For each loop back-edge from node v to v', add an edge from s to v and an edge from v' to t.
3. Remove all loop back-edges from the control flow graph.

Note that the resulting DAG retains all acyclic, intra-procedural paths in the original control-flow graph and all these paths begin at the start node s and end at the exit node t. Once the DAG (say G) has been constructed, the Ball-Larus algorithm assigns weights to the edges of G such that for every path $p \in paths(G)$, $pathid(p)$ is unique, and is equal to a number between 0 and $N - 1$, where $N = |paths(G)|$. The algorithm *computeBLIncrements*, shown in Figure 3.1, performs one bottom-up pass through G and processes its nodes in reverse topological order. With each node v, the algorithm associates

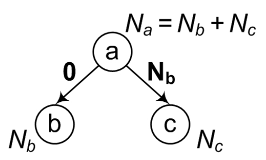


Figure 3.2: Assignment of weights to edges using the Ball-Larus algorithm

```
procedure computePathIdentifier (G, W, p)
Assume:
(a) $G = (V, E, s, t)$ is a DAG.
(b) The map $W : E \rightarrow \mathbb{Z}$.
(c) A path $p$ is a sequence of nodes through $G$.
Returns: The path identifier for the path $p$
1: return $\sum_{e \in edges(p)} W(e)$;
```

Figure 3.3: Computing the Ball-Larus identifier of a path from an edge assignment.

a count N_v that indicates the number of paths from v to the exit node t (Line 5). At each node, the *computeBLIncrements* traverses the list of successor nodes and assigns weights to the corresponding outgoing edges. This algorithm is based on a simple idea stated in the following lemma [6].

Lemma 3.2.1 Let $G = (V, E, s, t)$ be a DAG. The number of paths from any node v in G to the exit node t is equal to the sum of the number of paths from each of v's successor nodes to t.

Figure 3.2 illustrates how *computeBLIncrements* uses this invariant to compute an edge assignment. Assume that the algorithm is processing node a with two successors b and c that have N_b and N_c paths to the exit node t. Also assume that these paths have already been assigned identifiers from 0 to $N_b - 1$ and $N_c - 1$ respectively. The algorithm assigns a weight 0 to the edge (a, b), and a weight N_b to the edge (a, c). This ensure that the paths from a to t are assigned identifers from 0 to $N_b + N_c$ - 1, which is also equal to $N_a - 1$ (from Lemma 3.2.1). In general, the weight assigned to an edge is equal to the sum of the number of paths from all previously processed successor nodes to t (Line 6).

The Ball-Larus path profiler instruments the edges of the CFG with instructions to increment a counter by the weight assigned to the edge. When the instrumented program executes, it simulates the procedure *computePathIdentifier* (Figure 3.3). When a path terminates, the value in the counter represents the path that just executed and can be used for book-keeping.

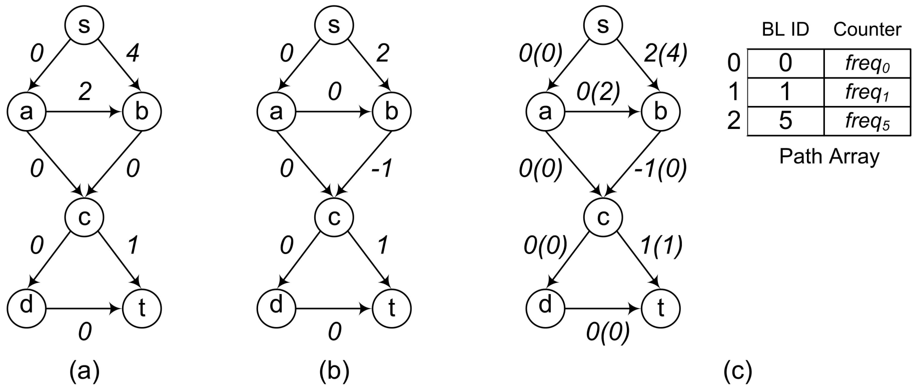


Figure 3.4: Motivating example for PPP. *(a) A* DAG *G with 6 paths with edges numbered using the Ball-Larus algorithm. (b) G with edges having only* PPP *assigned numbers for three interesting paths $I = \{sacdt, sact, sbct\}$. (c) G with edges assigned two numbers, a* PPP *number and a Ball-Larus number (in parenthesis). The path array is accessed using the* PPP *counter.*

3.2.3 An illustrative example

We start with an example that illustrates a drawback of the Ball-Larus profiling scheme, and also shows how our profiler works on this example. Consider the function in Figure 3.4(a). The DAG G in Figure 3.4(a) is obtained from the CFG of the function. This figure also shows the weights assigned by the Ball-Larus algorithm to edges of G. Note that the sum of the weights of edges along every path from the start node s to the final node t is unique, and all paths are allocated identifiers from 0 to $N - 1$, where N is the total number of paths from s to t. If N is reasonably small (less than some threshold value T), the profiler can allocate an array of counters of size N, and track path frequencies by indexing into the array using the path identifier and incrementing the corresponding counter. However, the number of potential paths in a procedure can be arbitrarily large (exponential in the number of nodes in the graph) and allocating a counter for each path can be prohibitively expensive, even infeasible in many cases. Path profilers overcome this problem by using a hash table of counters instead of an array, relying on the observation that only a small number of paths are traversed during any given execution. Therefore, a combination of a suitably sized hash table and a good hash function almost always guarantees the absence of conflicts. In the current example, if T was set to 4, the Ball-Larus profiler would use a hash table since there are 6 paths from s to t.

Let us assume that we are interested in profiling only a subset $I = \{sacdt, sact, sbct\}$ (interesting paths) of paths. The Ball-Larus identifiers for the paths $sacdt$, $sact$, $sbct$ are 0, 1 and 5 respectively. This means that one would have to allocate a hash table even though there are only 3 paths of interest. In such a scenario, it would be ideal if we could compute an edge assignment that allocates identifiers 0, 1 and 2 to these paths and identifiers > 2 to the other paths. In Section 3.3, we show that computing an edge assignment W and a number β such that (a) $\forall p \in I$, $\mathit{pathid}(p) \leq \beta$, and (b) $\forall p \notin I$, $\mathit{pathid}(p) > \beta$ is not always feasible. Therefore, we relax the constraints on this problem by eliminating condition (b) (which is a condition over uninteresting paths), and ask the question if it is possible to label the edges in G such that the paths in the set I have path identifiers in $\{0, 1, 2\}$. Figure 3.4(b) shows that such an assignment of weights to edges indeed exists, and this is precisely the assignment computed by the preferential path profiling PPP algorithm described in Section 3.3. Therefore, our profiler incurs lower overheads since we now can use an array to track frequencies instead of a hash table. Note that while the interesting paths *sacdt*, *sact*, *sbct* have been assigned unique identifiers from 0 to 2, the uninteresting paths *sabct* and *sbcdt* alias with the interesting paths *sacdt* and *sact* respectively. We resolve these "aliases" using Ball-Larus path identifiers, which are unique for every path. In PPP, edges are annotated with a second weight computed using the Ball-Larus algorithm (these weights are shown in parentheses in Figure 3.4(c)). The profiler also stores the Ball-Larus identifiers of all interesting paths along with their counters. The occurrence of an interesting path can be detected by comparing the Ball-Larus identifier computed during the traversal with the Ball-Larus identifier stored in the array – a match indicates that an interesting path was just traversed and vice versa. For example, when the uninteresting path *sbcdt* (PPP identifier is 2 and Ball-Larus identifier is 4) occurs, before incrementing the count at index 2 in the path array, the Ball-Larus identifier at index 2 is compared with the Ball-Larus identifier of *sbcdt* – since they are different, the profiler infers that this path is not interesting (or it might be a residual path not exercised by the test suite) and takes necessary action.

3.3 Preferential Path Profiling

We will now address the problem of encoding arbitrary subsets of paths over a DAG $G = (V, E, s, t)$. Informally, we wish to compute an edge assignment that allows us to uniquely identify paths as well as differentiate interesting paths from the uninteresting ones. First, consider the possibility of finding an edge assignment that *separates* interesting and uninteresting paths using a number $\beta \in \mathbb{Z}^{\geq 0}$.

Lemma 3.3.1 (Separation of paths) Given a DAG $G = (V, E, s, t)$ and a set of interesting paths $I \subseteq \mathit{paths}(G)$, a map $W : E \to \mathbb{Z}$ that satisfies the following conditions may not always exist.

1. uniqueness: $\forall p, q \in I$, $\mathit{pathid}(p) \neq \mathit{pathid}(q)$.

2. perfect numbering: $\forall p \in I$, $0 \leq \mathit{pathid}(p) < |I|$.

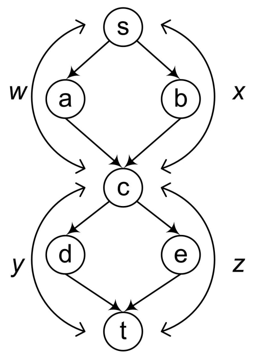


Figure 3.5: A counterexample for separation of paths.

3. separation: $\forall p \notin I$, $pathid(p) \geq |I|$.

Proof: Consider the simple DAG in Figure 3.5. Assume that we are interested in profiling paths *sacet* and *sbcdt*. Assume that there exists an edge assignment W that satisfies all conditions in the lemma. Let w, x, y and z represent the cumulative weights of the sub-paths sac, sbc, cdt and cet respectively. From condition 2, we have $w + z < 2$ and $x + y < 2$, and from condition 3, it follows that $w + y \geq 2$ and $x + z \geq 2$. This implies that

$$\begin{aligned} x + y + w + z &< 4 \\ x + y + w + z &\geq 4 \end{aligned}$$

which is a contradiction and the lemma follows. ■

This result is also related to the notion of *linear separability* and the XOR problem in pattern classification [21]. Specifically, finding an edge assignment that separates interesting paths from the others corresponds to the problem of finding a linear surface that separates a set of patterns, where each pattern has been classified into one of two classes.

In practice, we find separating arbitrary sets of interesting and uninteresting paths is almost always infeasible, primarily due to the presence of many shared edges. We therefore simplify the problem by relaxing condition 3. Edge assignments that cause the interesting paths to alias with the uninteresting paths are acceptable as long as the interesting paths are assigned minimal unique identifiers. As described in Section 3.2.3, a second counter that computes the Ball-Larus identifiers of all paths can be used to resolve the aliases. This relaxation allows us to reason about interesting paths only, an aspect critical to the solution we propose. However, it turns out the even this simplified problem may not have a perfect solution as the following lemma indicates.

Lemma 3.3.2 (Perfect edge assignment) Given a DAG

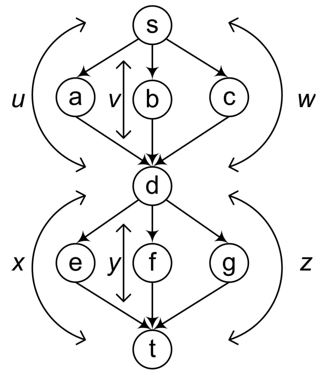


Figure 3.6: A counterexample for perfect edge assignment.

$G = (V, E, s, t)$ and a set of interesting paths $I \subseteq paths(G)$, a map $W : E \to \mathbb{Z}$ that satisfies the following conditions may not always exist.

1. uniqueness: $\forall p, q \in I$, $pathid(p) \neq pathid(q)$,

2. perfect numbering: $\forall p \in I$, $0 \leq pathid(p) < |I|$.

Proof: Consider the graph in Figure 3.6. Say we are interesting in profiling the paths $sadft, sadgt, sbdet, sbdgt, scdet$, and $scdft$. For simplicity, we represent the sum of the edges along the sub-paths sad, sbd, scd, det, dft and dgt as u, v, w, x, y and z. Consider the sum of the identifiers of all these paths.

$$\begin{aligned} sum &= (u + y) + (u + z) + (v + x) \\ &\quad +(v + z) + (w + x) + (w + y) \\ &= 2(u + v + w + x + y + z) \end{aligned}$$

Hence, the sum of the path identifiers of these paths is necessarily even. However, for a perfect assignment, these paths must be allocated identifiers between 0 and 5. Since the sum of numbers from 0 to 5 is odd, we conclude that a perfect edge assignment for this graph and set of interesting paths does not exist. ■

Since a perfect edge assignment for interesting paths may not always exist, even an optimal edge assignment may induce a path assignment with "holes" in the interval of path identifiers. In light of this lemma, we restate our problem as follows.

Problem A (Optimal Edge Assignment): Given a DAG $G = (V, E, s, t)$ and a set of interesting paths $I \subseteq paths(G)$, compute an edge assignment $W : E \to \mathbb{Z}$ that satisfies the following conditions.

1. uniqueness: $\forall p, q \in I$, $pathid(p) \neq pathid(q)$,

2. compactness: The compactness measure δ defined by

$$\delta \overset{\mathsf{def}}{=} \frac{(\max_{p \in I} pathid(p) - \min_{p \in I} pathid(p)) + 1}{|I|}$$

is minimized.

It is easy to see that $\delta \geq 1$. A perfect edge assignment W induces a $\delta = 1$. Lemma 3.3.2 shows that a solution with $\delta = 1$ does not always exist. Hence solutions with lower values of δ are preferred. We find that for arbitrary graphs and arbitrary set of paths, even characterizing the optimal δ seems to be a hard problem. In the next section, we propose an algorithm that computes an edge assignment that attempts to minimize δ, and later prove an optimality result for this algorithm by establishing a connection with arithmetic coding.

3.3.1 The Preferential Path Profiling Algorithm

The preferential path profiling (PPP) algorithm is a generalization of the Ball-Larus algorithm with the added capability of biasing the edge assignment towards an arbitrary set of interesting paths. Before we describe the algorithm, we introduce some notation and state some of the key observations that the algorithm is based on.

Let $G = (V, E, s, t)$ be a DAG, and let $I \subseteq paths(G)$ be a set of interesting paths. Consider a node $v \in V$ and an edge $e \in out(v)$. Let $paths_I(e)$ represents the set of interesting paths that traverse the edge e. Let $prefix(p, e)$ denote the sequence of nodes from s to $src(e)$ along the path p. Then $prefix_I(e)$ denotes the set of all prefixes $\{prefix(p, e)\}_{p \in I}$. Given a pair of interesting paths $p, p' \in I$, $lcp(p, p')$ represents the longest common prefix of p and p' in G. Note that $lcp(p, p')$ is trivially the start node s if p and p' are edge disjoint. We define the *critical node* for a pair of interesting paths p and p' as follows.

$$critical(p, p') \overset{\mathsf{def}}{=} \textit{the last node in } lcp(p, p')$$

For any pair of paths, the critical node is unique since the longest common prefix is uniquely defined. For example, in Figure 3.7, node a is the critical node for paths p_1 and p_2, whereas node b is the critical node for paths p_1 and p_3. Again, the critical node for a set of paths is trivially the start node s if the set of paths are edge disjoint in G.

We also define a map $pid : I \rightarrow \mathbb{Z}$ to track *partial identifiers* allocated to paths during the execution of an edge assignment algorithm. Assuming that all edges are initialized with a weight $\perp$ (this denotes the undefined value), the partial identifier of a path is defined as follows.

$$pid(p) = \sum_{e \in edges(p) \wedge W(e) \neq \perp} W(e)$$

From the statement of Problem A, we note that an edge assignment must simultaneously satisfy two constraints i.e. uniqueness and compactness of path identifiers. We now

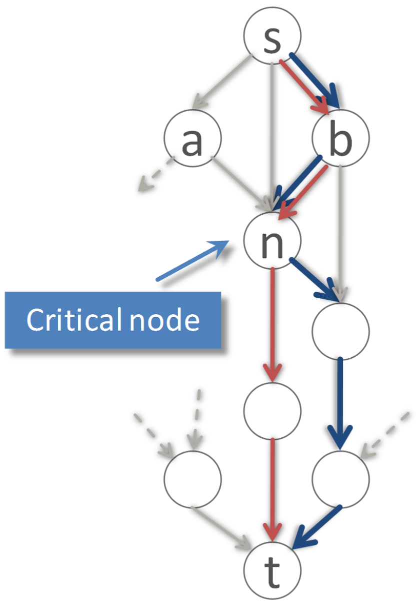


Figure 3.7: Critical nodes for pairs of paths.

describe how PPP satisfies these constraints.

Uniqueness. We first define an invariant that *any* algorithm computing an edge assignment by processing nodes in reverse topological order must satisfy in order to ensure that all interesting paths are allocated unique identifiers[1].

Lemma 3.3.3 (Invariant for uniqueness) Consider a node v being processed by the algorithm. If v is the critical node for any pair of interesting paths p and p', then p and p' should be assigned different partial identifiers after the node v has been processed.

Proof: Follows from the definition of critical nodes, and the fact that the algorithm assigns weights to edges in reverse topological order. ■

The PPP algorithm computes an edge assignment that attempts to achieve the most compact path numbering (low δ) that maintains this invariant at every node. However, to make the algorithm simpler and more amenable for analysis (Chapter 4), PPP makes the following approximation. Instead of explicitly checking the partial identifiers of paths at a critical node, PPP works over *intervals* of path identifiers. Given an edge e, the interval $int_{e,q}$ represents the range of partial identifiers allocated to all interesting paths through

[1]A similar invariant based on suffixes can be defined if the algorithm were to perform a top-down traversal, processing nodes in topological order

e that have a prefix q. Formally,

$$\begin{aligned} int_{e,q} &= [min(pid(p) \mid p \in paths_I(e) \wedge prefix(p,e) = q), \\ &\quad\; max(pid(p) \mid p \in paths_I(e) \wedge prefix(p,e) = q)] \end{aligned}$$

At every node, PPP computes an interval $int_{e,q}$ for every (edge, prefix) pair and assigns weights to the edges to maintain the following invariant.

Lemma 3.3.4 (Invariant for uniqueness over intervals) Consider a node v being processed by PPP. Assume that a prefix q induces a set of intervals $S_{v,q} = \{int_{eq} | e \in out(v)\}$ on the outgoing edges of v. To ensure uniqueness, the intervals in $S_{v,q}$ should not overlap after v has been processed. Furthermore, this condition must hold for every prefix $q \in \cup_{e \in out(v)} prefix_I(e)$.

Proof: It is easy to see that if a prefix q induces an interval at two or more outgoing edges of a node v, then v is the critical node for all paths p with the prefix q. By preventing overlap between all such intervals for a given prefix, PPP automatically ends up separating all paths with prefix q for which node v is critical. If this condition is satisfied for all prefixes, all interesting paths for which node v is critical are distinguished and hence the Lemma 3.3.3 is satisfied. ■

Compactness. With the uniqueness constraint taken care of, we now describe how PPP ensures compactness. At any node $v \in V$, consider the set of intervals $S_{v,q} = \{[min_i, max_i]_{i \in [1,k]}\}$ (where $k = |out(v)|$) induced on edges $e_1, e_2 \ldots e_k$ by a prefix q. If q is the only valid prefix at v, PPP uses *compaction* to compute a *minimal* edge assignment $W(e_i)_{i \in [1,k]}$ which ensures that these intervals do not overlap. To achieve compaction, each $W(e_i)$ is computed as follows:

$$W(e_i) = \sum_{j \in [1,(i-1)]} (max_j - min_j + 1) - min_i \tag{3.1}$$

$$= cis_{i-1} - min_i \tag{3.2}$$

where cis_{i-1} represents the cumulative interval size of all intervals induced on previous edges. However, this simple compaction method cannot be used if multiple prefixes induce intervals on the outgoing edges of a node. In such situations, PPP performs a *join* operation over the intervals at all edges with two or more intervals. The *join* operation computes the weights induced by different prefixes on the edge and conservatively assigns a weight equal to the maximum amongst all these weights. Due to the *join*, interesting paths associated with all but one of the prefixes will be assigned a weight higher than what is required to separate its intervals, creating holes in the path numbering. However, it is easy to see that this choice of weight leads to the most compact numbering feasible.

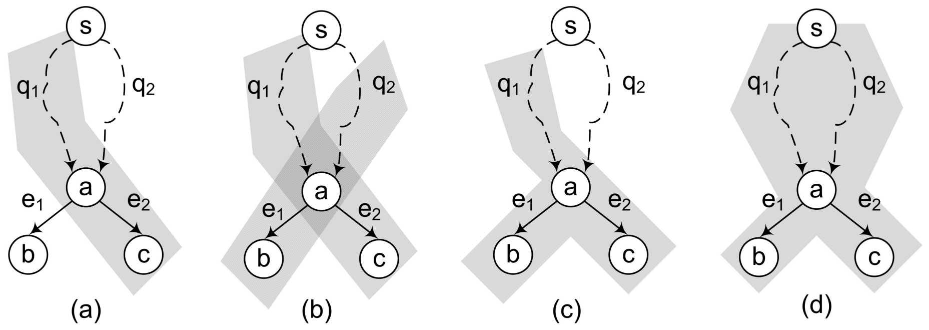


Figure 3.8: The assignment of weights to edges under four scenarios

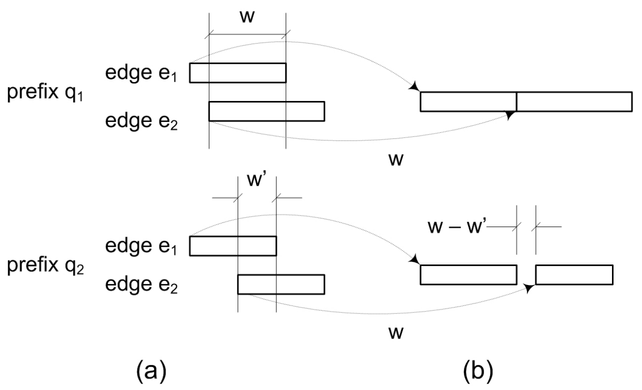


Figure 3.9: The effect of using the join *operator to conservatively assign weights to edges. (a) Intervals induced by the prefixes before the join, and (b) effective intervals after the join.*

Figure 3.8 illustrates the scenarios PPP deals with. In Figure 3.8(a), all interesting paths through the node a have the same prefix q_1 and traverse the edge e_2 (represented by the shaded region). Since the interval int_{e_2,q_1} is the only interval in the set S_{a,q_1}, no overlap between intervals exist and the invariant for uniqueness (Lemma 3.3.4) is trivially satisfied. A similar situation occurs in Figure 3.8(b), where paths through the edges do not share any prefixes. The interval sets S_{a,q_1} and S_{a,q_2} are singletons and no conflicts occur. Figure 3.8(c) represents a scenario where the interesting paths induce two intervals for the prefix q_1. PPP uses Equation 3.1 to compute weights and ensures that these intervals do not overlap. Finally, Figure 3.8(d) illustrates the scenario where the interesting paths through edges e_1 and e_2 share prefixes q_1 and q_2. Figure 3.9 illustrates the effect of a join on two sets of intervals induced by prefixes q_1 and q_2. Since we need a larger weight (say w computed by Equation 3.1) to separate intervals induced by prefix q_1 (as compared to the weight w' required to separate intervals induced by prefix q_2), PPP assigns w to e_2, leading to a hole in the interval for the prefix q_2.

Figure 3.10 describes the PPP algorithm in detail. For each node $v \in V$ and each outgoing edge $e \in out(v)$, the algorithm iterates over all prefixes and computes the beginning of the interval $int_{e,q}$ ($min_{e,q}$ at Line 5). It uses an auxiliary map *cis* to determine the cumulative interval size of intervals through previously processed outgoing edges of v with the prefix q (as per Equation 3.1) and computes the weight induced by q on the edge (Line 7). Finally, the join operation (Lines 10 and 11) selects the maximum over the weights induced by each prefix and assigns this weight to the edge. After the edge is assigned a weight, PPP updates the partial identifiers of all paths through the edge and also computes the new $cis(q)$ for the next iteration on this edge.

In summary, at every node $v \in V$, *computePPPIncrements* recursively merges intervals of each prefix q into the most compact single interval $int_{v,q}$. At the start node s, this interval defines the range of identifiers allocated to the interesting paths. The time complexity of PPP is $O(E \times P)$, where E is the number of edges and P is the number of interesting paths.

3.3.2 Example

We now walk-through an example illustrating how the PPP algorithm works. Consider the DAG in Figure 3.4; say we are interested in profiling paths *sacdt*, *sact* and *sbct*. The following steps trace the manner in which PPP assigns weights to edges of the DAG. Step i_j denotes that at step i, PPP processes node j.

Step 1_t Initialize the partial identifiers of all paths to 0 and cumulative interval sizes of all prefixes to 0.

Step 2_d Node d is not a critical node for any pair of paths since it has only one outgoing edge. The prefix *sacd* induces an interval $[0, 0]$ on the edge (d, t). Since $cis(sacd) = 0$, $W((d, t)) = 0 - 0 = 0$.

Step 3_c Node c is a critical node for paths *sacdt* and *sact*. Say the edge (c, d) is processed first. Both prefixes *sac* and *sbc* induce an interval $[0, 0]$ on this edge. Hence PPP assigns a weight $W((c, d)) = 0 - 0 = 0$ to this edge. PPP updates the map *cis* as follows: $cis(sac) = 1$ and $cis(sbc) = 1$. PPP then processes the edge (c, t). The prefix *sbc* induces an interval $[0, 0]$ on this edge. Since $cis(sbc) = 1$, PPP assigns a weight $W((c, t)) = 1 - 0 = 1$ The partial identifiers of paths *sact* and *sbct* are also updated to 1.

Step 4_b Node b has one outgoing edge (b, c). The prefix *sb* induces an interval $[1, 1]$ on this edge. PPP assigns a weight $W((b, c)) = 0-1 = -1$ to this edge since $cis(sb) = 0$. The partial identifier of the path *sbct* is now updated to $1 + -1 = 0$.

Step 5_a Node a has two outgoing edges but only the edge (a, c) has interesting paths through it. The prefix *sa* induces an interval $[0, 1]$ at this edge. PPP assigns a

```
procedure $ComputePPPIncrements$ $(G)$
Assume:
(a) $G = (V, E, s, t)$ is a DAG.
(b) a map $paths_I : E \to 2^P$.
(c) $pid : P \to \mathbb{Z}^{\geq 0}$ initialized to 0 for all interesting paths.
(d) $cis$ :$prefix \to \mathbb{Z}^{\geq 0}$, initialized to 0 for all prefixes of interesting paths.
Returns: A edge assignment $W : E \to \mathbb{Z}$.
1: for all nodes $v \in V$ in reverse topological order do
2:   for all edges $e \in out(v)$ s.t. $e \in edges(p)$ for some $p \in I$ do
3:     for all prefix $q \in prefix_I(e)$ do
4:       // compute the beginning of the interval $int_{e,q}$
5:       $min_{e,q} := min\{pid(p) \mid p \in paths_I(e)$ and $prefix(p, e) = q\}$;
6:       //compute weight induced by prefix $q$
7:       $weight_q := cis(q) - min_{e,q}$;
8:       // the join: compute the maximum weight
9:       if $(W(e) = \perp || W(e) < (weight_q))$ then
10:        $W(e) := weight_q$;
11:      end if
12:    end for
13:    // update partial identifers of all paths through $e$
14:    for all paths $p \in paths_I(e)$ do
15:      $pid(p) := pid(p) + W(e)$;
16:    end for
17:    for all prefixes $q \in prefix(e)$ do
18:      //determine new cumulative interval size for prefix $q$
19:      $cis(q) := max\{pid(p) \mid p \in paths_I(e)$ and $prefix(p, e) = q\} + 1$;
20:    end for
21:  end for
22: end for
```

*Figure 3.10: The preferential path profiling (*PPP*) algorithm for computing an edge assignment for a set of interesting paths.*

weight $W((a, c)) = 0 - 0 = 0$ to the edge.

Step 6_s Node s has two outgoing edges with three paths, all sharing a common prefix s. This prefix induces an interval [0,1] at the edge (s, a) and the interval $[0, 0]$ at the edge (s, b). PPP processes the edge (s, a) first and assigns a weight $W((s, a)) = 0 - 0 = 0$ to the edge. PPP updates $cis(s) = 1 + 1 = 2$. Next, PPP processes the edge (s, b).

Since $cis(s) = 2$, the edge (s, b) is assigned a weight $W((s, b)) = 2 - 0 = 2$. The partial identifier of the path *sbct* is also updated to 2.

At termination, the interesting paths *sacdt*, *sact* and *sbct* are allocated identifiers $0, 1$ and 2 respectively.

3.3.3 Discussion

In summary, PPP attempts to achieve a compact path numbering by (1) only numbering the edges required to distinguish interesting paths, and (2) computing the smallest weights such that the interesting paths are assigned unique identifiers. Our experiments suggest that PPP achieves good compactness measures for a vast majority of the procedures, even when a large number of interesting paths are specified. However, despite its best efforts, PPP does not always achieve the best possible compactness measure. Figure 3.11 illustrates one scenario in which PPP fails to assign an optimal numbering although such a numbering clearly exists. Consider the graph G_1 and assume that PPP has assigned identifiers to a set of interesting paths. Also assume that the interval of allocated identifiers contains two holes of size k_1 and k_2 respectively. As shown in the figure, we can construct a new graph G and select a set of interesting paths such that there exists an edge assignment which achieves the $\delta = 1$. Here, G is obtained via a parallel composition of three graphs G_1, G_2 and G_3 such that $|paths(G_2)| = k_1$ and $|paths(G_3)| = k_2$. Furthermore, the set of interesting paths now includes all paths through G_1 and G_2. An optimal numbering for this set of interesting paths is obtained by assigning a weight I_1 to the edge (s, s_2) and I_2 to the edge (s, s_3), filling up the holes in the interval of graph G_1. However, PPP fails to compute such assignments since we restrict ourselves to the class of solutions where edge intervals do not overlap. As we show in the next chapter, this constraint allows us to establish an optimality condition by making a connection with arithmetic coding.

3.4 Interprocedural Preferential Path Profiling (IPPP)

Modern software development has embraced modular programming, which increases the number of procedures in a program. As a result, many interesting program behaviors span across procedure boundaries and intra-procedural testing and profiling techniques may not suffice. Unfortunately, although inter-procedural analysis, profiling and testing techniques are desirable, they have traditionally been associated with high runtime overhead. For instance, inter-procedural path profiling is extremely expensive [33, 48]. These high overheads have limited the use of inter-procedural techniques even in testing environments where cost is usually not a prime concern. We propose to perform a simplified version of inter-procedural path profiling that has significantly lower overheads.

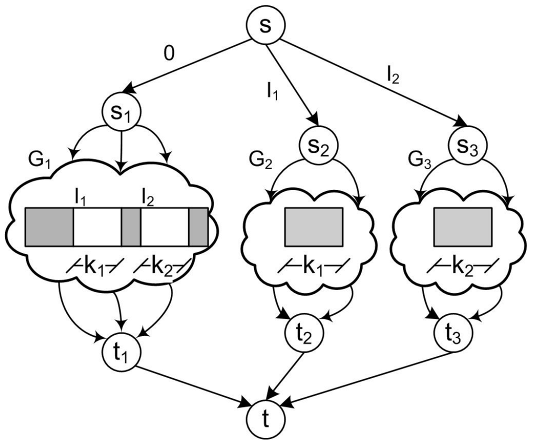


Figure 3.11: Figure illustrating a scenario in which PPP *assigns a sub-optimal numbering but an optimal numbering clearly exists.*

3.4.1 Preliminary Definitions

We define a few terms to facilitate the exposition of our technique.

Subpath: A subpath is an acyclic, intra-procedural path that terminates at procedure calls, in addition to loop back edges and function returns.

Whole Program Path (WPP): The whole program path is the entire sequence of subpaths generated during a given execution of a program. The WPP precisely characterizes the entire control flow behavior of the program [30].

Interprocedural Path Segment (IPS): An inter-procedural path segment is a sequence of subpaths that can be generated using the following grammar

$$P \to (P \mid (P) \mid PP \mid \langle f, p \rangle$$

where '(' denotes a function call, ') represents a return, and $\langle f, p \rangle$ represents the execution of a subpath p in the function f. Intuitively, an IPS is similar to the traditional notion of inter-procedurally valid paths [33], except that it does not necessarily start with a call to the main function.

3.4.2 Specifying Interesting IPSs

To perform inter-procedural preferential path profiling, we first need to specify interesting inter-procedural path segments (IPSs). While the whole program path (WPP) is a valid (though very large) IPS, for efficiency reasons we constrain the set of IPSs that can be specified. We define a depth k IPS as one that spans at most k procedure calls. For a given value of k (programmer specified), the set of IPSs exercised can be easily extracted

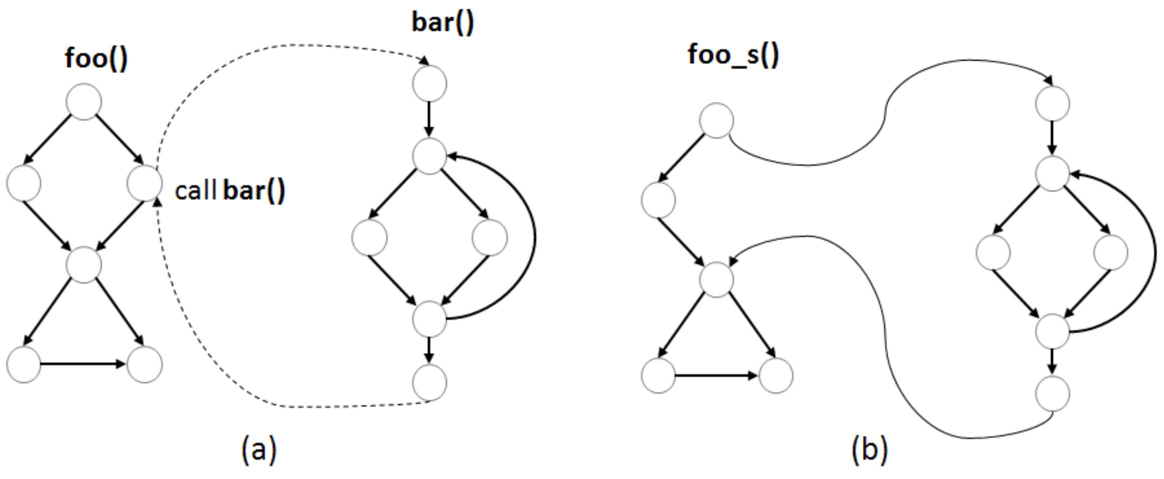


Figure 3.12: IPPP *example. (a) Interesting inter-procedural path originating in* $foo()$*, passes through the function* $bar()$*, before returning to* $foo()$*. (b) After function inlining, this path becomes an intra-procedural path in the supergraph* $foo\_s()$*.*

from the WPP. For instance, consider the follow substring of a WPP:

$$(\langle f1,p1\rangle(\langle f2,p2\rangle(\langle f3,p3\rangle(\langle f4,p4\rangle\langle f4,p5\rangle\langle f4,p6\rangle)\langle f3,p7\rangle$$
$$(\langle f5,p8\rangle)\langle f3,p9\rangle)\langle f2,p10\rangle)$$

Here, the paths $p1$, $p2$, $p3$ and $p7$ terminate on procedure calls. For $k = 1$, the set of valid IPSs is

1. $\langle f1,p1\rangle(\langle f2,p2\rangle$
2. $\langle f2,p2\rangle(\langle f3,p3\rangle$
3. $\langle f3,p3\rangle(\langle f4,p4\rangle$
4. $\langle f3,p7\rangle(\langle f5,p8\rangle)\langle f3,p9\rangle$

Note that IPS 4 includes the subpath executed after returning from procedure f5. Also, IPS 3 does not extend to the subpath $\langle f4,p5\rangle$ because the path $p4$ ends at a loop back edge and our current implementation does not profile paths that span across loop boundaries.

3.4.3 Profiling Interesting IPSs

Much like PPP, inter-procedural preferential path profiling (IPPP) achieves low overhead by exploiting knowledge about interesting paths. Given a set of inter-procedural paths for which profiling information is required, we proceed in two stages. First, assuming that the number of inter-procedural path segments (IPSs) that most consumers of profiles are interested in is small, we transform the inter-procedural path segment profiling problem into an intra-procedural path profiling problem using function inlining. This results in a set of supergraphs that are used for analysis and instrumentation. Second, since these supergraphs are likely to contain a large number of acyclic intra-procedural paths, we use PPP to compactly number interesting paths in the supergraphs, which correspond to interesting IPSs in the original graphs. This compact numbering avoids the use of hash tables required by traditional path profiling techniques, and consequently reduces the overhead of IPS profiling. We provide an overview of IPPP with an example shown in Figure 3.12. Consider two functions $foo()$ and $bar()$ as shown in Figure 3.12(a). Assume we are interested in profiling a single IPS that originates in function $foo()$ and passes through $bar()$. In this simple scenario, the function $bar()$ is inlined into $foo()$ as shown in Figure 3.12(b). The inlining transformation leads to the creation of a supergraph $foo\_s()$, in which the IPS has an intra-procedural equivalent. Standard intra-procedural path profiling techniques can now be applied to such supergraphs to profile these paths. However, the supergraphs created using this function inlining transformation are likely to contain a significantly larger number of intra-procedural paths. This leads to two problems.

- First, in several cases where an array would have been sufficient for book-keeping, path profiling will be forced to use a hash table, resulting in increased runtime overheads.
- Second, in situations where the number of paths increases beyond a threshold (for example, 2^{32}), certain paths in the graphs must be truncated leading to a loss of precision [6].

We address both of these problems. First, we use PPP to compactly number interesting paths in the supergraph. Since the number of interesting IPSs is likely to be a small subset of all possible IPSs, PPP, which provides strong guarantees about compact numbering [49], will almost always be able to use an array instead of a hash table for tracking paths. Second, we address the problem of loss of precision due to truncation by avoiding truncating edges traversed by interesting IPSs. Consequently, interesting IPSs are profiled precisely and efficiently.

3.4.4 IPPP Algorithm

This section informally describes our algorithm for profiling interesting inter-procedural path segments (IPSs). Figure 3.13 provides an overview of the technique.

1. Input: A set of CFGs and a set S of interesting depth k-limited IPSs.

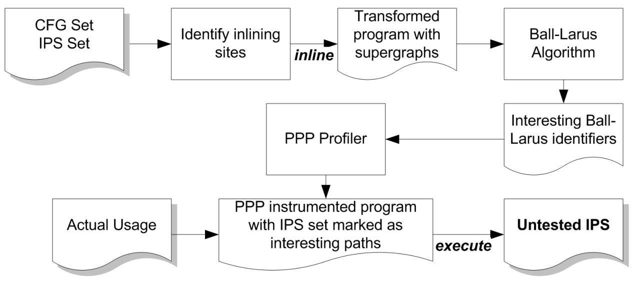

CFG Set
IPS Set
Identify inlining sites
inline
Transformed program with supergraphs
Ball-Larus Algorithm
Interesting Ball-Larus identifiers
PPP Profiler
Actual Usage
PPP instrumented program with IPS set marked as interesting paths
execute
Untested IPS

Figure 3.13: Overview of the inter-procedural preferential path profiling scheme.

2. Identify inlining sites: Based on the paths in set S, identify the set of call sites for inlining. For each IPS $\langle f1, p1\rangle(\langle f2, p2\rangle, \ldots$, consider all the subpaths that terminate at a procedure call. All such procedure call sites are identified and marked.

3. Mark all edges traversed by IPSs: Assign a globally unique identifier to all IPSs. Traverse all edges along each IPS and mark the edges with the corresponding IPS identifier. Edges that participate in multiple IPSs will have multiple IPS identifiers associated with them. This labelling serves two purposes. First, it helps create a mapping between an IPS in the original collection of CFGs and its corresponding intra-procedural equivalent in the transformed supergraph (see step 6). Second, it marks these edges as non-candidates for truncation, if truncation is necessary (see step 5).

4. Supergraph construction: Create supergraphs as shown in Figure 3.12(b) by combining the CFGs of individual procedures as determined in step 2.

5. Ball-Larus numbering: Assign Ball-Larus numbers to all paths in each supergraph. Ensure that edges traversed by IPSs as marked in step 3 are not truncated. After this step, each path in a supergraph is assigned a unique identifier.

6. Identify interesting IPSs: From the Ball-Larus numbering and the IPS edge information computed in step 3, obtain the Ball-Larus identifiers of interesting IPSs.

7. Drive PPP: Use the Ball-Larus identifier of interesting IPSs computed in Step 6 as input to the PPP algorithm.

3.4.5 Avoiding Inlining through Code Duplication

IPPP uses procedure inlining to convert an inter-procedural path profiling problem to an intra-procedural one. In IPPP, each procedure is effectively duplicated as many times as the number of unique contexts it occurs in the set of input IPSs. However, this approach may not scale if the number and/or depth of IPSs in the input specification is large or the

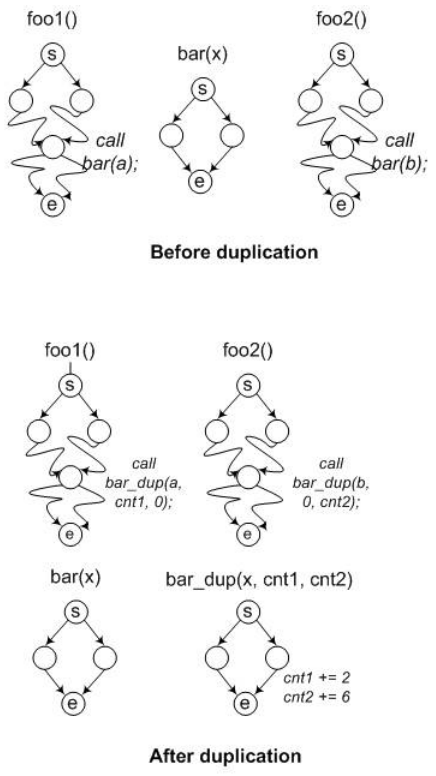


Figure 3.14: An example that illustrates an IPPP approach based on function replication that avoids increase in code size due to inlining-based IPPP.

IPS passes through procedures with large bodies. While our current implementation uses selective inlining, an alternative is to duplicate the code.

We now describe the approach based on procedure duplication using an example. Consider the program in Figure 3.14. The program consists of three functions, $foo1()$, $foo2()$ and $bar(x)$. Both $foo1()$ and $foo2()$ have calls to $bar(x)$. Assume that the input to IPPP consists of two inter-procedural path segments that pass through both call sites. As per the approach described in the previous section, we would inline a copy of $bar(x)$ in $foo1()$ and in $foo2()$ (the original copy of $bar(x)$ remains). If the resulting increase in code size is not acceptable, IPPP creates a second copy of the procedure $bar(x)$ (say $bar\_dup(x)$) and shares it across multiple contexts as shown in Figure 3.14. Here, we only inline the procedure bar into $foo1()$ and $foo2()$ for the purpose of analysis and compute an assignment of weights to the edges of $bar\_dup(x)$ independently in each context. We then map these weights back to the original edges of $bar\_dup(x)$. Each edge of $bar\_dup(x)$ is now associated with as many weights as the number of unique contexts (say N). Next, we add N counter variables as arguments to the procedure $bar\_dup(x)$, one for each context, to obtain $bar\_dup(x, cnt1, cnt2, \ldots, cntN)$. We also modify the respective call sites in each context to pass the current value of the Ball-Larus and/or PPP counter as an argument to the duplicated procedure, as shown in Figure 3.14. The edges of $bar\_dup(x, cnt1, cnt2)$ are instrumented to increment all counter variables as and when required. These increment operations are usually inexpensive and are unlikely to have an adverse effect on performance, at least for a moderate number of contexts. Finally, before returning from the procedure $bar\_dup(x, cnt1, cnt2, \ldots, cntN)$, we save the state of all counter variables so that they can be retrieved by the respective callers. In this way, both the Ball-Larus and PPP algorithms can be extended to the inter-procedural case without an excessive increase in code size.

3.4.6 Discussion

The key enabling insight that avoids an exponential blowup in code size due to inlining and code duplication is that the inlining decisions are based on concrete Ball-Larus path-profiles from a regression test suite. Typically, a test suite exercises only a small fraction of all possible inter-procedural-paths. Our experiments (Section 3.6) show that the code size increase for IPPP is only 22% higher on average than standard intra-procedural Ball-Larus PP. Recursion is handled as we are only interested in IPSs of limited depth. Function pointers are handled by doing context-sensitive inlining from concrete Ball-Larus path profiles and using runtime-checks for validation.

3.5 Residual Path Profiling (RPP)

Residual path profiling identifies the set of paths executed by deployed software that were not tested during software development [38]. This section describes how we perform residual profiling for intra-procedural and inter-procedural paths.

3.5.1 Intra-procedural Residual Path Profiling

RPP for intra-procedural paths is fairly straightforward and proceeds in two stages. First, a program is instrumented for path profiling with the Ball-Larus technique and run with its test suite inputs. The Ball-Larus identifiers of intra-procedural paths that were executed are recorded. Next, the same program is instrumented with PPP with the recorded Ball-Larus paths marked as interesting paths and this version of the program is deployed to gather real usage profiles, perhaps as part of a beta testing phase. When an untested path is executed, either its PPP identifier will exceed the size of the array used to track paths or its Ball-Larus identifier will not match the one recorded in the array entry (see Figure 3.4(c)). In this way, untested paths are detected and recorded during actual usage of deployed software.

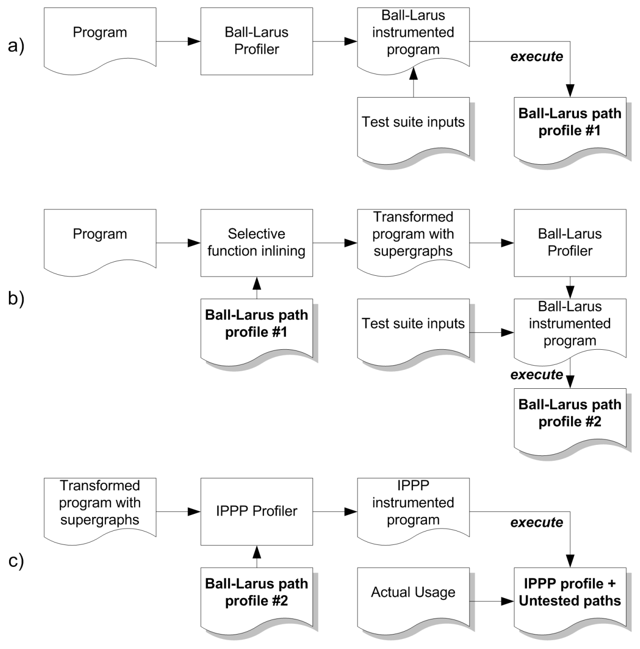

a)
Program
Ball-Larus Profiler
Ball-Larus instrumented program
execute
Test suite inputs
Ball-Larus path profile #1
b)
Program
Selective function inlining
Transformed program with supergraphs
Ball-Larus Profiler
Ball-Larus path profile #1
Test suite inputs
Ball-Larus instrumented program
execute
Ball-Larus path profile #2
c)
Transformed program with supergraphs
IPPP Profiler
IPPP instrumented program
execute
Ball-Larus path profile #2
Actual Usage
IPPP profile + Untested paths

Figure 3.15: Residual path profiling for IPPP*s. (a) Profile executed paths on test suite inputs using the Ball-Larus scheme. (b) Inline functions to convert executed* IPS*s to intra-procedural paths and re-execute on test inputs. (c) Use path profile from (b) to drive* PPP *and run program with field inputs to detect untested paths.*

3.5.2 Inter-procedural Residual Path Profiling

We can use the inter-procedural preferential path profiling (IPPP) technique described in Section 3.4 to perform residual profiling of inter-procedural paths. First, we need to specify the set of interesting IPSs that should be profiled to the IPPP algorithm. This is done by performing whole program path (WPP) profiling on the test suite inputs. Given a user specified value for k, we can identify all exercised depth-k constrained IPSs from the WPP. These depth-k IPSs are used as input to the IPPP algorithm. This generates a new binary ready for deployment. Running this binary produces a list of paths that are exercised in the current run but were not exercised by the test suite. The Ball-Larus identifiers of these untested paths can be used to generate the set of untested IPSs [6].

Alternative scheme for Inter-procedural Residual Path Profiling

An alternative technique that does not require the use of WPP profiles is illustrated in Figure 3.15. Our experimental results reported in Section 3.6 use this technique for inter-procedural residual path profiling. The scheme proceeds in three stages.

Stage I: The program is profiled on its test suite inputs to identify all acyclic, intra-procedural paths that were exercised (Figure 3.15(a)). The Ball-Larus technique is used to perform path profiling.

Stage II: The path profile from Stage I is used to identify all call sites that can occur on a depth-k IPS. Note that this is an overapproximation of the set of call sites that would have been identified given a WPP profile. These call sites are inlined and a new binary, instrumented to collect Ball-Larus path profiles, is generated. As a result of inlining, all depth-k IPSs exercised by the test suite inputs appear as intra-procedural paths in the new binary. This binary is then instrumented to collect Ball-Larus path profiles and rerun on the test suite inputs. The path profile generated effectively assigns unique Ball-Larus identifiers to all depth-k IPSs exercised. Figure 3.15(b) illustrates this process.

Stage III: The path profile generated in the previous step along with the transformed binary serve as inputs to the PPP algorithm as shown in Figure 3.15(c). This generates a new binary suitable for deployment, where all depth-k IPSs that were exercised by the test suite inputs are marked as interesting paths. When this binary is executed on field inputs the untested paths reported correspond to depth-k IPSs (and intra-procedural paths) not exercised by the test suite.

3.5.3 Discussion

Testing large software is a resource constrained activity. Consequently, the priority is to test most frequently exercised behaviors across all users (not a single user). RPP accomplishes this by recording all untested paths exercised. This information can be aggregated across users to determine priorities. A scheme that only records frequent untested paths would miss rare untested paths that all/most users execute.

3.6 Experimental evaluation

We have implemented the preferential path profiling algorithm using the Scale compiler infrastructure [32] and Phoenix, Microsoft's C/C++ compiler [35]. A few key features of our implementation are listed below.

- *Representing paths and prefixes.* While a user is free to specify the set of interesting paths in several ways, we choose to represent the interesting paths using their Ball-Larus identifiers. Similarly, we represent a prefix using the cumulative sum of the Ball-Larus weights along the edges of the prefix. It is easy to see that this sum is unique for each prefix leading to a given node.

- *Register usage.* Unlike traditional path profiling, preferential path profiling requires two registers, one for PPP counts and one for Ball-Larus counts. Our experiments suggests that the use of two registers instead of one does not add to the overheads of profiling

- *Counter optimizations.* All counter placement optimizations [6] used in the Ball-Larus algorithm also apply to the PPP counter. These include reducing the number of initialization and increment operations by placing weights only on the edges that do not belong to a maximal spanning tree of the DAG, pushing counter initialization downwards along the edges of the DAG and merging the initializations with the first increments. In our implementation, we ensure that both PPP and Ball-Larus counter updates occur on the same edges.

- *Hash table usage policy.* The default Ball-Larus profiler is configured to use a hash table instead of an array when the total number of paths through the procedure exceeds a threshold. For our experiments, we configured our profiler to switch to a hash table when the number of paths exceeds 2500. In this scenario, our profiler uses a hash table with 700 elements. However, the policy for hash table use in preferential path profiling depends on the specific scenario in which the profiler is used. For instance, in residual path profiling, where the goal is to detect the occurrence of untested (uninteresting) paths, a hash table may never be used, even when the PPP identifers allocated to the tested (interesting) paths are large. Here, the profiler implementation may decide to ignore all tested paths with PPP identifiers greater than a threshold, in essence treating them as untested paths. As a result, a few of the tested paths may appear as untested during program execution. Such false positives may be acceptable since PPP ensures that interesting paths are assigned compact numbers. In addition, they can be easily weeded out off-line. A policy that switches to using hash tables when the PPP identifiers are large may also be used in other applications. Although our implementation supports both modes, we report our results using the former policy.

- *Additional checks.* Before indexing the path array using PPP identifiers, our profiler must check for an underflow/overflow, which can result when an uninteresting path occurs. Our experiments suggest that these additional checks do not add to the cost of preferential profiling since they are highly biased and easy to predict.

We evaluated our profiler implementation using benchmarks from the SPEC CPU2000 suite. We first simulated a realistic preferential profiling scenario. We collected a path profile of the benchmarks using the Ball-Larus profiler for the standard reference input. We then assumed that all paths exercised during the reference run were interesting (including procedures where several hundred paths were exercised). These were fed to the preferential profiler, which generated a new instrumented binary. Binaries obtained from the Scale

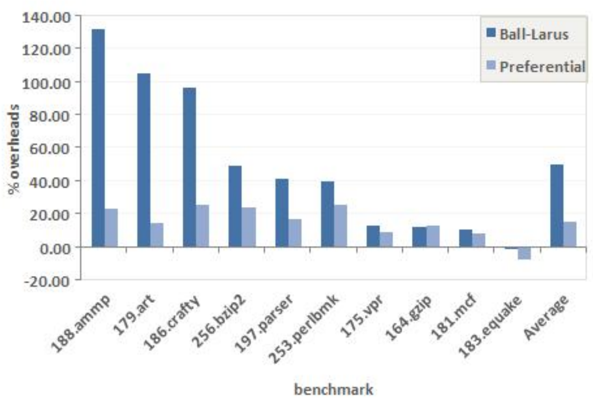


Figure 3.16: Overheads of preferential path profiling.

compiler were run to completion using the SPEC reference input on an 800Mhz Alpha 21264 processor with 512MB RAM running Digital OSF 4.0. Each binary was executed 5 times and the minimum of the execution times (measured using hardware cycle counts) was used for comparison.

Figure 3.16 shows the percentage overheads of the two schemes relative to execution time of the un-instrumented binary on the Alpha. We find that Ball-Larus profiling incurs an average overhead of 50% with a maximum of 132%. On the other hand, the preferential path profiler incurs an average overhead of 15%, with a maximum of 26% (*186.crafty*. We attribute the low profiling overheads of PPP to (a) elimination of expensive hash operations, and (b) judicious allocation of counters for profiling (the size of the counter array is proportional to the number of interesting paths and not the number of potential paths). It is therefore not surprising that PPP achieves significant performance benefits for benchmarks (such *188.ammp*, *179.art* and *186.crafty*) in which *hot* functions have a large number of potential paths but only a small number of these paths are typically exercised. PPP does not have a significant impact for benchmarks like *175.vpr* and *164.gzip* where the hot functions have a small number of potential paths to begin with. For benchmark *183.equake*, profiling reduces execution time when compared to an unprofiled binary; we attribute this anomaly to secondary micro-architectural effects.

Although reflected in the overheads, the real efficacy of the preferential profiling algorithm lies in the compactness measure δ that it achieves. We illustrate the compactness measure achieved by our algorithm in Figure 3.17, which plots the size of the interval allocated to interesting paths vs. the number of interesting paths for procedures from programs in the SPEC CPU2000 benchmark suite. As aforementioned, all paths exercised during one reference run were selected as interesting paths. The figure suggests that our profiling scheme achieves a δ close to 1 for a vast majority of the procedures, although the value tends to increase for procedures with a large number of paths (100-300). We also

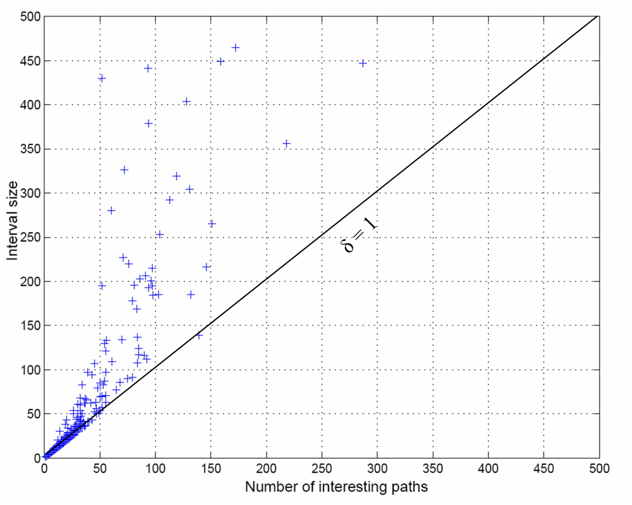


Figure 3.17: δ values achieved by preferential path profiling.

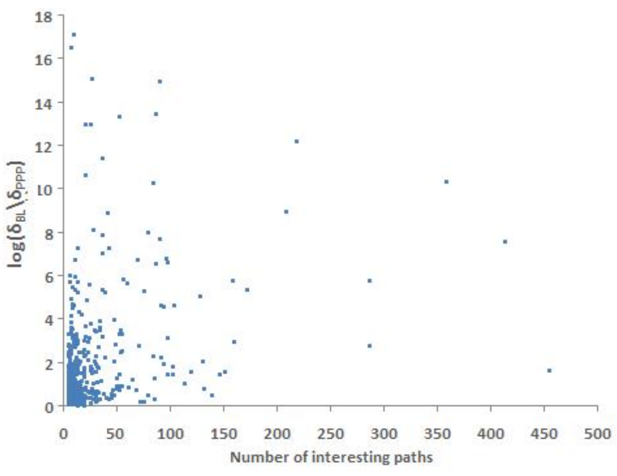


Figure 3.18: Ratio of the δ values obtained using the Ball-Larus algorithm and PPP (on a log scale).

found a very small number of cases with $\delta > 10$ (not shown in this figure), most of them in the benchmark *crafty*, a chess program known to have complex control flow.

To further illustrate the benefits of PPP, we compared the δ values obtained using the Ball-Larus profiling with the δ values obtained using PPP. Figure 3.18 illustrates the ratio of the delta values (on a log scale). As expected, the identifiers assigned to the interesting paths by the Ball-Larus algorithm are disproportionately larger than the number of paths typically traversed by programs.

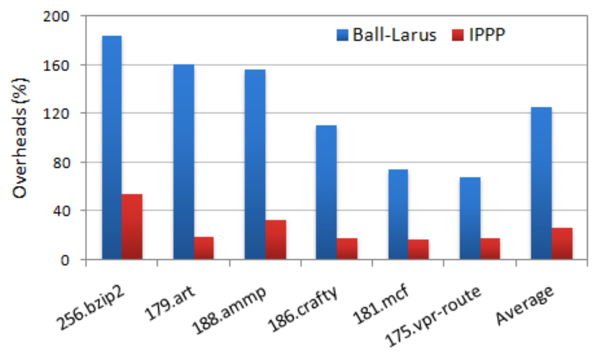


Figure 3.19: Runtime overheads of inter-procedural path profiling.

3.6.1 Inter-procedural Preferential Path Profiling (IPPP)

To evaluate our IPPP scheme, we labeled all depth 1 inter-procedural path segments (i.e., all IPSs that span a single procedure call boundary) exercised by the SPEC train inputs as interesting IPSs and produced an instrumented binary that profiles these inter-procedural paths. This binary was then run on the ref inputs. Since we perform selective inlining to convert IPSs into intra-procedural paths, we can also profile these paths with the Ball-Larus technique.

Figure 3.19 shows the overhead of these techniques on some of the SPEC benchmarks (the Scale compiler does not successfully compile all SPEC benchmarks, even without our path profiling). The Ball-Larus scheme incurs high overheads that range from 70% to 180% with an average of 125%. With the exception of `256.bzip2`, which incurs an overhead of 52%, IPPP achieves reasonably low overhead with an average of 26%. For four of the six benchmarks the overhead is less than 20%. This overhead may be low enough to permit residual profiling of IPSs during beta software testing of many interactive desktop applications, such as web browsers, email clients, and productivity software, where the slowdown is possibly below human perception threshold on fast modern machines.

IPPP is able to achieve significantly lower overheads as compared to inter-procedural Ball-Larus profiling because it is able to compactly number interesting IPSs and avoid using a hash table for recording path information. Figure 3.20, which plots the size of the interval allocated to interesting paths versus the number of interesting paths for functions in the SPEC benchmarks, substantiates this claim. For almost all procedures this ratio is very close to 1, indicating almost perfect compaction.

3.6.2 Intra-procedural Residual Path Profiling

As discussed earlier, the goal of residual path profiling is to identify paths that are exercised in the field but were not exercised by any test in the program's test suite. To evaluate

Benchmark	#untested paths	%untested paths	%freq of untested paths	#funcs with untested paths	#untested edges	#funcs with untested edges	#untested paths in funcs with no untested edges	%untested paths in funcs with no untested edges
164.gzip	80	7.2	0.0	6	3	2	77	96.3
175.vpr	274	20.9	0.0	29	147	22	13	4.7
179.art	132	50.0	47.2	12	130	10	6	4.5
181.mcf	3	1.2	0.0	3	8	1	2	66.7
183.equake	1	0.5	0.0	1	0	0	1	100
188.ammp	117	22.5	0.0	4	2	1	114	97.4
197.parser	612	13.0	0.0	61	211	29	273	44.6
256.bzip2	398	45.1	0.0	13	81	10	26	6.5
300.twolf	295	11.3	0.0	36	43	18	117	39.7
PCgame-1	970	19.8	1.5	139	248	81	502	51.8
PCgame-2	3531	16.5	0.6	248	384	143	898	25.4
Average	583	18.9	4.5	50.2	114.3	28.8	184.5	48.9

Table 3.1: Untested intra-procedural path information obtained from residual path profiling.

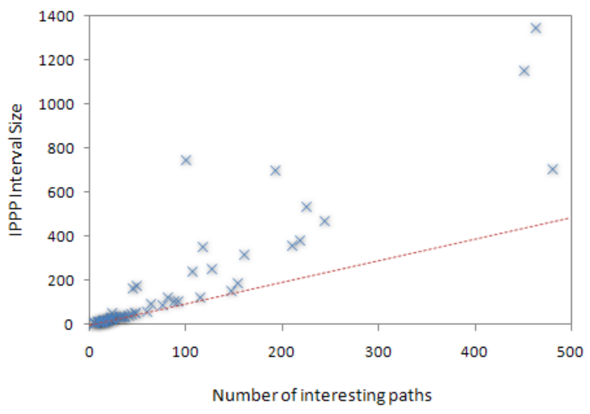


Figure 3.20: Compact numbering achieved by IPPP *for several functions from programs in the SPEC CPU2000 suite.*

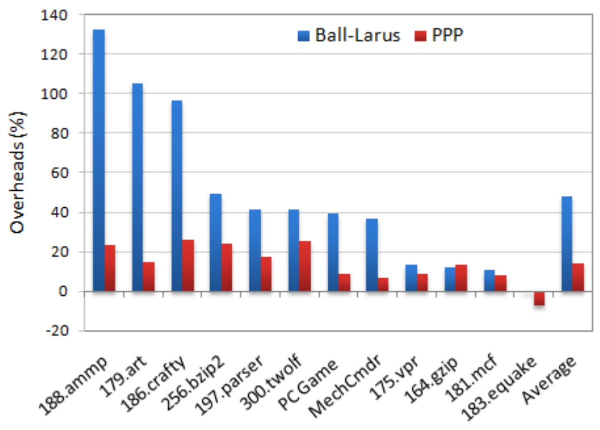


Figure 3.21: Comparison of runtime overheads of Ball-Larus and intra-procedural RPP*.*

the efficacy of PPP for residual path profiling, we used the train inputs provided with the SPEC benchmarks. First, the Ball-Larus path profiler was used to collect profiles of the benchmarks on their train inputs. The paths profiled become the interesting paths that are input to the preferential path profiler (PPP). PPP generated a new instrumented binary which was then run on the benchmarks reference input. Paths that were exercised only by the ref input are reported as untested paths.

We also included a Microsoft shared source game called MechCommander and another Microsoft game in this study. We compiled these games using the Phoenix implementation of our path profiler. We played these games several times for a duration of 5-10 minutes and assumed that paths exercised in these runs were tested. We then played the games for a much longer duration, assuming that this run was a run in the field.

Figure 3.21 reports the overhead of performing intra-procedural RPP for the scenario

Benchmark	#untested paths	%untested paths	%freq of untested paths
175.vpr	300	21.1	0.0
179.art	199	77.4	57.5
181.mcf	3	1.1	0.0
183.crafty	3262	63.5	0.0
188.ammp	123	21.5	1.4
256.bzip2	949	58.3	0.1
Average	314.8	35.9	11.8

Table 3.2: Untested IPS information obtained from inter-procedural residual path profiling.

described above relative to Ball-Larus profiling.The overhead numbers for the PC games represent percentage reduction in frame rate as a result of profiling. The overhead numbers (average of 13%) are in line with those in Figure 3.16. The numbers are lower because in this residual profiling scenario, we use train inputs to generate interesting paths while the earlier experiment used the *ref* inputs for this purpose. The numbers for the PC games are especially impressive as these are cutting-edge, resource intensive programs and indicate that RPP can be used in deployed software, at least during beta software testing.

We also performed residual edge profiling and compare the results against RPP. The results are shown in Table 3.1. Many of the SPEC benchmark ref inputs are merely larger size versions of the train inputs. Despite this, the data indicates that the ref inputs exercise many more paths. The execution frequency contribution of these paths show that, with the exception of `179.art`, which exercised new hot paths, these untested paths are rarely executed. The comparison between untested paths and untested edges is striking. For these benchmarks, a significant number of untested paths occurred in functions which reported no new untested edges. These paths would go undetected with residual edge profiling and demonstrate the advantage of RPP. Even for untested paths that exercise new edges, inferring the path from the edge profile is not always possible. The PC games, which are significantly larger than the SPEC benchmarks, show similar results. For both games, the longer scenario exercises a large number of rarely executed untested (by the shorter scenario) paths. Many of these paths do not include any new edges.

3.6.3 Inter-procedural Residual Path Profiling

We performed a similar experiment to that described in Section 3.6.2, except that we labeled all depth-1 inter-procedural path segments exercised by the SPEC train inputs as interesting IPSs and then ran the IPPP generated binary on the benchmarks ref input. Table 3.2 reports the results. Comparing entries from Table 3.2 with Table 3.1 for a few

Benchmark	#untested paths	%untested paths	%freq of untested paths
176.gcc-200	2670	4.6	0.3
176.gcc-scilab	2635	4.5	0.3
176.gcc-expr	723	1.2	0.0
179.gcc-166	2034	3.5	0.0
179.gcc-integrate	238	0.4	0.0
256.bzip2-graphic	249	6.1	0.0
256.bzip2-source	520	12.8	0.0
256.bzip2-program	49	1.2	0.0
175.vpr-place	175	1.6	0.0
175.vpr-route	30	0.3	0.0
Average	932	3.6	0.1

Table 3.3: Untested intra-procedural path information obtained using a more robust test suite.

of the benchmarks, it can be seen that IPPP exposes a larger number of untested paths.

3.6.4 Residual Path Profiling Simulation

We performed an experiment much like the one described in Section 3.6.2 for residual path profiling of intra-procedural path except that we use a larger number of train and ref inputs to simulate a residual profiling scenario. We use the MinneSPEC input suite along with the SPEC test and train inputs as representative of the test suite. The results are shown in Table 3.3. It is interesting to note that even though a large number of paths are exercised by the train inputs for these programs, we are able to detect a significant number of new paths on the ref inputs which essentially characterize the "deployed" behaviors of these programs.

3.6.5 Code Size Increase

Apart from the runtime overheads of tracking paths, path profiling schemes (Ball-Larus and PPP) also increase the size of the program binary. The reasons for the code bloat are two-fold: (a) instrumentation placed along edges of functions and at the end of every path, (b) space allocated to path counter tables and auxiliary structures. Table 3.4 shows the increase in code size (number of times relative to the unprofiled binary) caused by both the intra-procedural profiling schemes for a set of benchmarks programs. On average, the Ball-Larus profiling scheme increases the code size by a factor of 3.21. The increase in code size due to PPP is slightly lower at 3.03, primarily because a more compact numbering reduces the amount of space that must be allocated for the path counter tables. Note that

Benchmark	Ball-Larus	PPP
188.ammp	3.25	3.40
179.art	4.57	4.71
186.crafty	3.67	2.83
256.bzip2	2.60	2.40
197.parser	2.26	1.71
300.twolf	2.71	2.52
PC Game 1	5.41	5.00
PC Game 2	4.60	4.57
175.vpr	3.38	3.38
164.gzip	3.27	3.22
181.mcf	1.23	1.20
183.equake	1.56	1.44
Average	3.21	3.03

Table 3.4: Increase in code size due to Ball-Larus and PPP intra-procedural profiling.

Benchmark	Ball-Larus	IPPP
175.vpr	6.91	5.00
179.art	4.83	3.67
181.mcf	4.40	3.80
188.ammp	5.52	4.81
256.bzip2	6.62	5.38
186.crafty	3.71	3.44
Average	5.33	4.35

Table 3.5: Increase in code size due to Ball-Larus and inter-procedural path profiling.

PPP does not always result in smaller binaries, as illustrated by the *188.ammp* benchmark. This is not surprising because in several cases where the Ball-Larus scheme would use a *smaller* (700 elements) but more expensive hash table, PPP enables the profiler to use a much faster but perhaps larger array (upto 2500 elements).

We also measured the increase in code size caused by the inter-procedural versions of the path profiling schemes. One might have anticipated a significant increase in code size compared to intra-procedural profiling because of inlining. As shown in Table 3.5, this turns out to be true for Ball-Larus inter-procedural profiling scheme, which increases code size by a factor of 5.33. However, the code bloat due to IPPP is significantly lower because inlining is restricted to call-sites along a small set of interesting paths. The average increase in code size is 4.35, which is close to the increase in code size due to intra-procedural path profiling. These results show that the increase in code size due to IPPP is much lower than expected and does not limit the applicability of inter-procedural path profiling any more than the intra-procedural profiling schemes.

Chapter 4

Path Profiling - an information theoretic perspective

In this chapter, we give an information theoretic characterization of the preferential path profiling algorithm. We begin by introducing the notion of arithmetic coding and context modeling. We then show that the Ball-Larus path profiling algorithm is a special instance of arithmetic coding. After drawing this connection, we reformulate Problem A described in Section 3.3 so that it is more amenable to analysis, and provide a theoretical analysis of our algorithm for preferential path profiling.

4.1 Arithmetic Coding

Arithmetic coding [16, 42, 52] is a well-known universal, lossless compression technique that achieves close-to-optimal compression rates. Much like other compression schemes, arithmetic coding relies on the observation that in any given input stream, a small number of characters/substrings are likely to occur frequently. Arithmetic coding achieves compression by encoding these frequently occuring characters/substrings using a smaller number of bits. An arithmetic coder uses a *probability model* to identify frequent characters. In the simplest of cases, the probability model D is an assignment of probabilities to characters of the input alphabet Σ and is easily obtained from the frequency counts of characters in a representative string.

An arithmetic coder encodes strings into a single positive number less than 1. To compute this number, the arithmetic coder maintains a *range* or an *interval*, which is set of $[0, 1)$ at the beginning of the coding process. As each symbol of the input string is processed, the coder iteratively narrows the range based on the probability of the symbol. After the last symbol is read, any number within the resulting range uniquely represents the input string. Moreover, this number can be uniquely decoded to create the exact stream of symbols that went into its construction. We illustrate the encoding an decoding

Symbol	Probability	Range
a	0.2	$[0, 0.2)$
b	0.3	$[0.2, 0.5)$
c	0.1	$[0.5, 0.6)$
d	0.2	$[0.6, 0.8)$
e	0.1	$[0.8, 0.9)$
!	0.1	$[0.9, 1)$

Table 4.1: A sample probability model for the alphabet $\Sigma = \{a, b, c, d, e, !\}$.

process by way of an example (adapted from [52]).

Example 4.1.1 Let $\Sigma = \{a, b, c, d, e, !\}$ be the finite alphabet, and let a fixed model that assigns probabilities to symbols from Σ be as shown in Table 4.1. Suppose we wish to send the message *bacc!*. Initially, both the encoder and the decoder know that the range is $[0, 1)$. After seeing the first symbol b the encoder narrows it down to $[0.2, 0.5)$ (this is the range that the model allocates to symbol b). For the second symbol a, the interval is further narrowed to one-fifth of itself, since a has been allocated $[0, 0.2)$. Thus the new interval is $[0.2, 0.26)$. After seeing the first c the narrowed interval is $[0.23, 0.236)$, and after seeing the second c the new interval is $[0.233, 0.2336)$. Finally on seeing !, the interval is $[0.23354, 0.2336)$; knowing this to be the final range, the decoder can immediately deduce that the first character was b. Now the decoder simulates the action of the encoder, since the decoder knows that the interval $[0.2, 0.5)$ belonged to b, the range is expanded to $[0.2, 0.26)$. Continuing this way, the decoder can completely decode the transmitted message. It is not really necessary for the decoder to know both ends of the range produced by the encoder. Instead, a single number in the range (say 0.23355 in our example) will suffice.

Note that *the ranges for any probability model (for example, the one in Table 4.1) are non-intersecting; this condition is critical for arithmetic coding to work. If the ranges associated with the input symbols were intersecting, two or more strings could map to the same interval and the decoder has no way of distinguishing these strings.*

From the discussion, it should be clear that for arithmetic coding to be effective, the frequency of occurrence of characters in the input string must be skewed and the skew must be accurately reflected in the probability model. In other words, better compression rates are achieved if the model makes accurate predictions about the nature of the input string. We now describe a technique known as finite context modeling, which is commonly used to obtain more accurate probability models.

Finite context modeling. In a finite context scheme, the probabilities of each symbol are calculated based on the *context* the symbol appears in. In its traditional setting, the context is just the symbols that have been previously encountered. The *order* of the model refers to the number of previous symbols that make up the context. One way of compressing data is to make a single pass over the symbols to be compressed (to gather statistics) and then encode the data in a second pass. The statistics collected are the relative frequencies of occurrences of the respective symbols. These relative frequencies are then used to encode/decode the symbol as explained in earlier. Essentially, the model in this setting consists of a set of tables for every possible context up to size k for any order k model. Each context is a state and each entry corresponding to a symbol frequency is indicative of its probability of occurrence in that context. An *optimal* model is one that represents the best possible statistics for the actual data that is to be compressed. Unfortunately, computing an optimal model in general is undecidable [29].

It can be shown that arithmetic coding achieves optimal compression for a given probability model D [16]. Specifically, if X is a random variable representing events over a set $\mathcal{X}$ with a probability distribution D, then the average number of bits required to encode any event from $\mathcal{X}$ using arithmetic coding is equal to the *entropy* of D which is defined as follows.

$$\mathcal{H}(D) \overset{\mathsf{def}}{=} -\sum_{x\in\mathcal{X}} \mathsf{P}(X=x)\log_2|\mathsf{P}(X=x)|$$

Note: $\mathcal{H}(D)$ is the *binary entropy function*, and $\mathsf{P}(X=x)$ is the probability of the event $X=x$.

Note: In order to achieve optimal overall compression of data, the model must be an optimal model for that data.

Example 4.1.2 For the model D described in Table 4.1, the entropy $\mathcal{H}(D) = -\mathsf{P}(a)\log_2|\mathsf{P}(a)| - \mathsf{P}(b)\log_2|\mathsf{P}(b)| - \mathsf{P}(c)\log_2|\mathsf{P}(c)| - \mathsf{P}(d)\log_2|\mathsf{P}(d)| - \mathsf{P}(e)\log_2|\mathsf{P}(e)| - \mathsf{P}(!)\log_2|\mathsf{P}(!)| = -(0.2\log_2|0.2| + 0.3\log_2|0.3| + 0.1\log_2|0.1| + 0.2\log_2|0.2| + 0.1\log_2|0.1| + 0.1\log_2|0.1|) = 2.45$ bits.

4.2 The Ball-Larus algorithm and Arithmetic coding

We now show that the Ball-Larus profiling algorithm is in fact an instance of arithmetic coding for paths through a DAG $G = (V, E, s, t)$. We first note that both path numbering and arithmetic coding have similar objectives i.e. to *compactly and uniquely* encode strings from an input alphabet. In path numbering, the input alphabet is the set of edges through a DAG and the input strings are paths through the DAG. We also find that the process of assigning weights to the edges of the DAG corresponds to the process of computing

```
procedure $computeBLModel$ $(G)$
Assume:
(a) $G = (V, E, s, t)$ is a DAG.
(b) $\forall v \in V$, a map $D : E \rightarrow [0, 1]$ that is initially undefined.
Returns: a model $D$ such that $\forall v \in V, \sum_{e \in out(v)} D(e) = 1$.
1: $N_t := 1$;
2: for all nodes $v \in V$ in reverse topological order do
3:     $N_v := \sum_{e \in out(v)} N_{dest(e)}$;
4:     for all edges $e \in out(v)$ do
5:         $D(e) := N_{dest(e)} / N_v$;
6:     end for
7: end for
```

Figure 4.1: The Ball-Larus algorithm as a model computation process.

a probability model. However, unlike arithmetic coding where computing an optimal model is undecidable in general, an optimal model for paths through a DAG can in fact be computed for the following reasons: (a) the set of strings that can occur is known *a priori*; this is precisely the set of all paths through the DAG, and (b) the order in which edges can occur in paths is determined by the structure of the graph, which is also known *a priori*. Based on these observations, we derive a model computation procedure for paths through a DAG that is equivalent to the Ball-Larus algorithm. The procedure *computeBLModel*, shown in Figure 4.1, takes a DAG G as an input and assigns a probability $D(e)$ to every edge e in the graph. The resulting model is a finite context model, where the context of an edge is its source node and the probability assigned to an edge is the probability of the edge being traversed given that the source node has been reached. One can easily verify that $\forall v \in V$,

$$\sum_{e \in out(v)} D(e) = 1$$

For every path $p \in paths(G)$, the probability $\mathsf{P}(p)$ induced by the model D is defined as follows.

$$\mathsf{P}(p) \overset{\mathsf{def}}{=} \prod_{e \in edges(p)} \mathsf{P}(e)$$

It also follows that for every $p \in paths(G)$, $\mathsf{P}(p) = 1/N$, where $N = |paths(G)|$ and therefore,

$$\sum_{p \in paths(G)} \mathsf{P}(p) = 1$$

Denote by D_G, the probability distribution over the set $paths(G)$ – it follows immediately

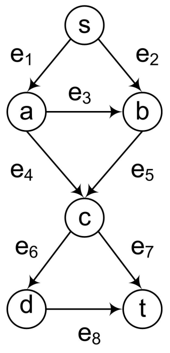


Figure 4.2: Example for the procedure computeBLModel.

Symbol	Probability	Range
e_1	2/3	$[0, 2/3)$
e_2	1/3	$[2/3, 1)$
e_3	1/2	$[0, 1/2)$
e_4	1/2	$[1/2, 1)$
e_5	1	$[0, 1)$
e_6	1/2	$[0, 1/2)$
e_7	1/2	$[1/2, 1)$
e_8	1	$[0, 1)$

Table 4.2: The Ball-Larus probability model D_G for the graph G computed by compute-BLModel

that the entropy $\mathcal{H}(D_G)$ is equal to $\log_2 |N|$.

Example 4.2.1 Consider the graph G shown in Figure 4.2. The model D computed by the procedure *computeBLModel* is given in Table 4.2.

We will now describe the procedure *pathEncoder* that takes a path $p \in paths(G)$, the model D computed by *computeBLModel* and $N = |paths(G)|$ as input, and computes its Ball-Larus identifier. This is the analogous to the procedure *computePathIdentifier* in Section 3.2.2.

Since arithmetic coding is optimal [16], that is, it achieves the entropy of the input model, *pathEncoder* (which is an arithmetic coder) is also optimal. We will make this connection explicit in the following example.

Example 4.2.2 Consider the graph G shown in Figure 4.2. Let D be the model computed by the procedure *computeBLModel* as given in Table 4.2. For an input path *sbcdt*,

```
procedure $pathEncoder$ ($p$, $D$, $N$)
Assume:
$G = (V, E, s, t)$ is a DAG and $p = (v_1, \ldots, v_k) \in paths(G)$, $\{v_i \in V\}_{1 \le i \le k}$.
Returns: path identifier for $p$.
 1: $in := [0, N)$;
 2: for all $i = 1$ to $k - 1$ do
 3:     $e := (v_i, v_{i+1})$;
 4:     $[x, y) := in$;
 5:     $n := y - x$;
 6:     let $[r, r')$ be the range for $e$ defined by $D$;
 7:     $in := [x + \lfloor rn \rfloor, x + \lceil r'n \rceil)$;
 8: end for
 9: $[x, \_) := in$;
10: return $x$;
```

Figure 4.3: The coding algorithm that takes a model D and path $p \in paths(G)$ as input, and returns the path identifier or the encoding for p.

pathEncoder(*sbcdt*, D_G, N) works as follows. We have $N = 6$, and the algorithm starts by assigning $in := [0, 6)$. The first edge encountered along this path is e_2, and therefore the interval in is set to $[4, 6)$. After seeing the next edge e_5, *pathEncoder* chooses the same interval $in = [4, 6)$. For the next edge e_6, the interval is narrowed down to $in = [4, 5)$, and finally for the last edge e_8, the interval is set to $in = [4, 5)$. Therefore, *pathEncoder* returns 4 as the path identifier for the path *sbcdt*. Note that this is precisely the Ball-Larus identifier for this path as is evident from Figure 3.4.

4.2.1 The PPP algorithm and Arithmetic Coding

In Section 3.3.1, we descibed an algorithm that compactly numbers a subset of interesting paths $I \subseteq paths(G)$ through a DAG G. We now show that the PPP algorithm is equivalent to an arithmetic coding scheme that uses a *maximal* context model for encoding paths through a DAG. As described in Section 3.3.1, the procedure *computePPPIncrements* computes for every pair (e, q), $e \in E$, $q \in prefix(p, e)$, an interval $int_{e,q}$ that represents the range of partial identifiers of interesting paths through e. At every node $v \in V$, these intervals are used to compute the weights associated with the outgoing edges of v. It can be shown that this procedure is equivalent to computing a finite context model with the prefixes as the context. Consider the simple case of a node v with two outgoing edges e_1 and e_2. Assume that a single prefix q induces intervals $int_{e1,q}$ and $int_{e2,q}$ on the edges e_1 and e_2 respectively. Define $cis_{v,q} = int_{e1,q} + int_{e2,q}$. Also define $p = \frac{int_{e1,q}}{cis_{v,q}}$. Then the model $D_{v,q}$ at node v can be defined as follows.

Symbol	Probability	Range
e_1	p	$[0, p)$
e_2	$1 - p$	$[p, 1)$

Computing the model is more involved when multiple prefixes induce intervals on the outgoing edges of a node. The problem arises because each outgoing edge may be associated with multiple probabilites, one for each valid prefix at node v. However, unlike traditional context models, an edge in the DAG cannot be associated with mutiple probabilites. We overcome this problem by using the *join* operator (defined in Section `subsec:pppalgo?`) to compute a conservative approximation of the individual models (which we refer to as D_v). Due to the *join*, certain edges may be assigned smaller probabilites than required. Consequently, the number of bits required to encode interesting paths through those edges may increase. The final model D_G is a combination of all models $D_v, v \in V$.

Finally, we note that the process of computing the PPP identifier for an interesting path $p \in I$ corresponds to calling the procedure pathEncoder with parameters p, D_G and $N = |int_{s,s}|$ assigned to the start node $s \in V$.

Example 4.2.3 Consider the graph G shown in Figure 4.2. Let the set of interesting paths be $I = \{sacdt, sact, sbct\}$. Then the probability model D_G computed by *computePP-PIncrements* is shown in Table 4.3. For the input path *sact*, *pathEncoder*(*sact*, D_G, N) works as follows. We have $N = 3$, and the algorithm starts by assigning $in = [0, 3)$. The first edge encountered along this path is e_1 (model = $D_{s,s}$), and therefore the interval in is set to $[0, 2)$. After seeing the next edge e_4 (model = $D_{a,sa}$), *pathEncoder* chooses the same interval $in = [0, 2)$. For the next edge e_7 (model = $D_{c,sac}$), the interval is narrowed down to $in = [1, 2)$, and therefore *pathEncoder* returns 1 as the path identifier for the path *sact*. Note that this is precisely the PPP identifier for this path as is evident from Figure 3.4.

This characterization of PPP as a model computer and encoder of paths works due to the fundamental invariant that the intervals in PPP do not overlap (this follows from Lemma 3.3.4).

4.3 Analysis of the PPP algorithm

The Ball-Larus algorithm computes an edge weight assignment such that δ (the objective function for Problem A) is equal to 1. Therefore, it is an optimal algorithm for those problem A instances for which the interesting paths are all the paths, that is $I = paths(G)$. In coding terms, this corresponds to saying that all paths in the graph are equally likely (and

Model	Symbol	Probability	Range
$D_{s,s}$	e_1	2/3	$[0, 2/3)$
	e_2	1/3	$[2/3, 1)$
$D_{a,sa}$	e_3	0	*empty*
	e_4	1	$[0, 1)$
$D_{b,sb}$	e_5	1	$[0, 1)$
$D_{c,sac}$	e_6	1/3	$[0, 1/3)$
	e_7	2/3	$[1/3, 1)$
$D_{c,sbc}$	e_6	1/3	$[0, 1/3)$
	e_7	2/3	$[1/3, 1)$
$D_{d,sacd}$	e_8	1	$[0, 1)$

Table 4.3: The PPP *probability model D_G for the graph G computed by* computePPPIncrements.

there is no bias towards any set of paths) – from the previous section, the entropy for such a distribution (say D) is equal to $\log_2 |paths(G)| \Rightarrow paths(G) = 2^{\mathcal{H}(D)}$.

In the previous section, we also saw that the procedure *computePPPIncrements* computes a probability model for a DAG G and a set of interesting paths $I \subseteq paths(G)$. Intuitively, this corresponds to computing a probability distribution that is biased towards the interesting paths over the uninteresting ones. Since PPP essentially mimics an arithmetic coder, the total number of bits required to represent the set of paths distributed according to the model D is equal to the entropy $\mathcal{H}(D) \Rightarrow$ the interval size that PPP computes is equal to $\lceil 2^{\mathcal{H}(D)} \rceil$. Therefore, the compactness that PPP achieves is $\frac{2^{\mathcal{H}(D)}}{|I|}$, and this is parameterized over how "precise" the model D is. In Theorem 4.3.1, we show that this model D computed by *computePPPIncrements* is indeed optimal. We now state a variant of Problem A and prove that PPP computes the optimal solution to this problem.

Problem B (Optimal Edge Assignment): Given a DAG $G = (V, E, s, t)$ and a set of interesting paths $I \subseteq paths(G)$, compute an edge assignment $W : E \to \mathbb{Z}$ that satisfies the following conditions.

1. uniqueness: $\forall p, q \in I$, $pathid(p) \neq pathid(q)$,

2. compactness: The compactness measure γ defined by

$$\gamma \overset{\text{def}}{=} \frac{2^{\mathcal{H}(D)}}{|I|}$$

is minimized, where D is any probability distribution on $paths(G)$ induced by a model D_G.

We now state and prove the main result in our analysis.

Theorem 4.3.1 Given a DAG $G = (V, E, s, t)$ and a set $I \subseteq paths(G)$, the procedure computePPPIncrements computes the optimal solution to *Problem B.*

Proof: From the discussion in Section 4.2.1, it follows that the PPP algorithm computes *maximal* context models D_G for a given set of interesting paths through DAGs. These models are the most precise context models because they use the largest context possible. Since the models D_G are optimal, $\gamma = \frac{2^{\mathcal{H}(D_p)}}{|I|}$ (where D_p is the probability model over $paths(G)$ induced by D_G) is also optimal (this follows from the optimality of the *pathEncoder* procedure, which essentially mimics an arithmetic coder), and the theorem follows. ■

From Section 3.3.3, it is clear that any algorithm (such as PPP) that maintains the invariant stated in Lemma 3.3.4 will not be able compute an optimal δ. On the other hand, our experiments described in Section 3.6 also indicate that the objective function γ minimized by our algorithm is close to the minimal δ (the objective function for Problem A) for most graphs and their associated interesting paths, and the interval size is small enough for the path profiler to use an array to track interesting paths.

Chapter 5

A Programmable Hardware Path Profiler

5.1 Introduction

In the previous chapter, we proposed a new software technique for profiling paths which reduces the overheads of path profiling from 50% to 15% on average. These overheads are similar to most point profiling techniques and likely to be acceptable for most applications of profiling. However, there exists a class of applications where even 15% overheads may not be acceptable. In runtime environments like Java, aggressive compilers often use profiles collected during the execution of a program to drive profile-based optimizations [**?**]. In most cases, the expected performance benefits of online optimizations are in the same ranage as the overheads of path profiling. Hence, most runtime systems either prefer not using path profiles or use sampling technqiues [4] to reduce the overheads of path profiling at the cost of profile accuracy. Software based path profiling techniques are also not suitable when the objective is to track micro-architectural metrics such as cache misses and branch mispredictions with paths [1]. In such a scenario, Heisenberg's uncertainity principle applies. The instrumentation introduced by path profiling (both code and data) can influence micro-architectural metrics significantly. As a result, accurately tracking these metrics with paths is hard.

In this chapter, we propose a programmable, non-intrusive hardware-based path detection and profiling scheme that overcomes these limitations. Figure 5.1 illustrates the components of the proposed profiling hardware. At the heart of the hardware profiler is a **path detector** that uses a **path stack** to detect paths by monitoring the stream of retiring branch instructions emanating from the processor pipeline during program execution. The path detector can be programmed to detect various types of paths and track architectural events that occur along the paths. The detector generates a stream of *path descriptors*, which is available to all interested hardware/software entities.

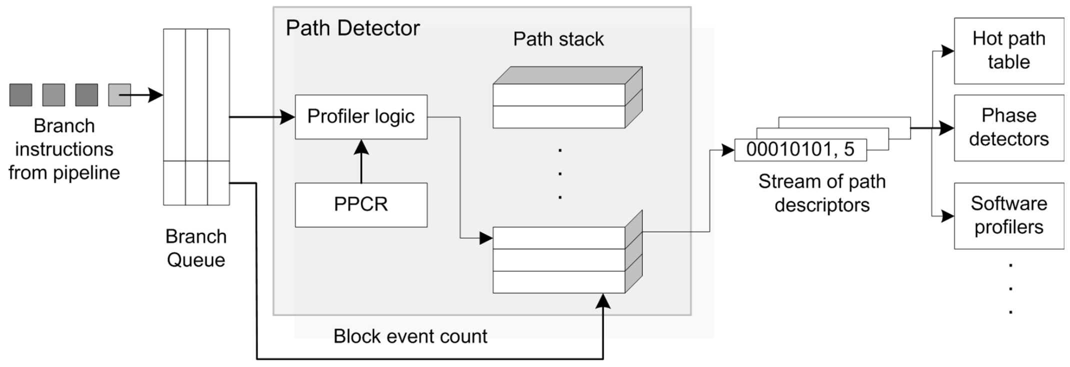
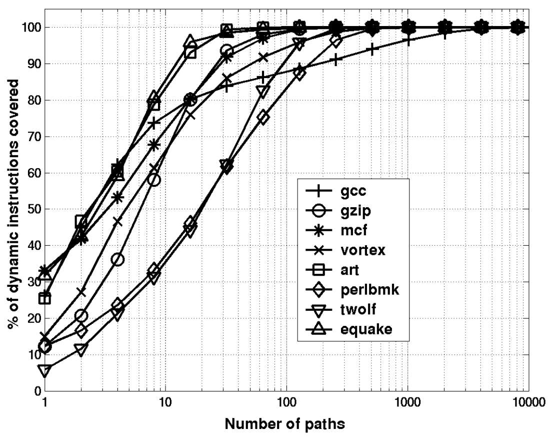


Figure 5.1: Components of the Hardware Path Profiling Infrastructure.

The second component of the hardware profiling infrastructure is a **Hot Path Table** (HPT), a compact hardware structure that processes the stream of path descriptors generated by the path detector. The HPT is designed to collect accurate hot path profiles, irrespective of the type of path being profiled, the duration of profiling and the architectural metric associated with paths. The success of a moderately-sized HPT in capturing accurate path profiles is attributed to locality exhibited by program paths i.e. programs typically traverse a small fraction of the numerous feasible acyclic, intra-procedural paths. And as illustrated in Figure 5.2, a small set of hot paths within the set of traversed paths dominates program execution. These properties continue to hold when paths are associated with architectural events such as L2 cache misses or branch mis-predictions (Figure 5.2(b)).

As previously mentioned, the path detector can track one of several architectural events that occur along paths. Events are tracked using *instruction event counters* associated with every instruction in the processor pipeline. In such cases, the HPT generated profile identifies paths that caused a significant fraction of those events, enabling context-sensitive bottleneck analysis and optimization. The generic nature of instruction event counters and our path-based event tracking mechanism can be further exploited by associating paths with more complex metrics. As an illustration, we associate paths with a power metric that provides an accurate estimate of the power consumption caused by instructions along each path in the cache hierarchy, a task hard to emulate using software profilers.

Our results indicate that our path profiler virtually eliminates the space and time overheads incurred by software path profilers. Moreover, the profiles collected in hardware, although lossy, are sufficiently accurate for program optimizers that can usually tolerate a small loss in profile accuracy. In offline environments, the hardware path profile can be serialized to a file at program completion for later use by a static program optimizer. Our profiler is all the more effective in online environments, where path profiles representative of the current program behavior can be obtained by enabling the profiler for short intervals of time. The availability of accurate profiles at low overheads helps the dynamic compiler



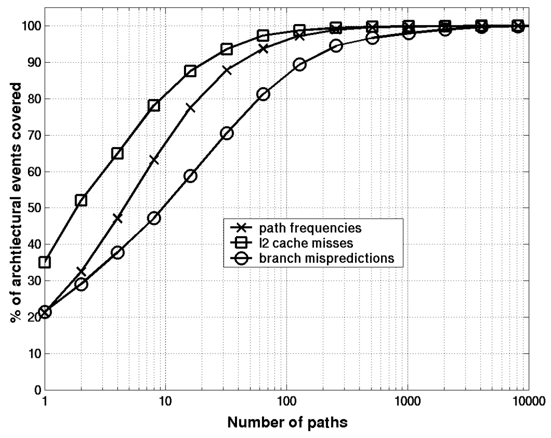


Figure 5.2: (a) Cumulative distribution of dynamic instructions along acyclic, intra-procedural paths. (b) Average cumulative distribution of path frequencies, L2 cache misses and branch mis-predictions along paths. The top 100 paths account for more than 80% of the dynamic instructions and a high percentage of L2 cache misses and branch mis-predictions in the SPEC CPU2000 programs studied.

generate optimized code efficiently and quickly, increasing the number of optimization opportunities exploited.

The rest of the chapter is organized as follows. In Section 5.2, we describe how different types of paths can be represented and detected in hardware. We propose extensions to the processor's event detection logic that enable the tracking of various architectural metrics along paths in Section 5.3. Section 5.4 discusses the design of the Hot Path Table, the hardware structure that collects hot path profiles. Section 5.5 evaluates the path profiling infrastructure.

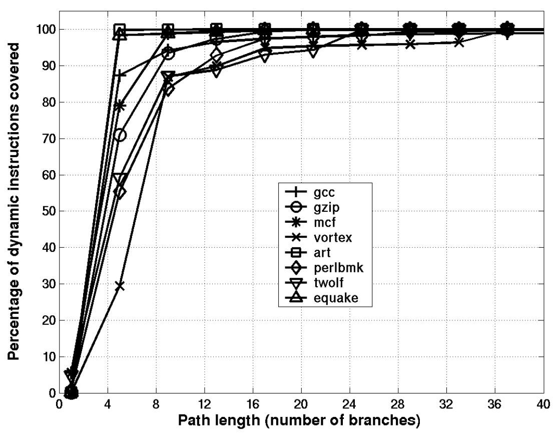


Figure 5.3: Distribution of dynamic instructions according to the length of Ball-Larus paths they execute along. Paths with at most 16 branches account for over 90% of program execution in most programs.

5.2 Representing and Detecting Paths in Hardware

Paths have traditionally been represented by the sequence of indices/addresses of basic blocks that fall along the path. However, this representation is expensive to maintain in hardware. In our profiler, a path is uniquely represented by a *path descriptor* which consists of *(1)* the path's starting instruction address, *(2)* the length of the path in terms of the number of branch instructions along the path, and *(3)* a set of branch outcomes, one for each branch along the path[1]). This representation is compact and expressive enough to describe all types of paths.

The hardware version of the path descriptor does have a limitation; it can accommodate only a predetermined, fixed number of branches per path. Our path detection hardware overcomes this limitation by splitting paths if their length exceeds this threshold. Figure 5.3 shows the distribution of dynamic instructions according to the length of Ball-Larus paths they are executed along for programs from the SPEC CPU2000 benchmark suite. We observe that paths of length less than 16 branch instructions account for over 90% of dynamic instructions in most programs. For the rest of this study, we assume a path descriptor representation that allows paths to grow up to 32 branch instructions, large enough to accommodate a majority of Ball-Larus paths and extended paths without splitting.

The hardware profiler uses a hardware *path stack* to detect paths. Each entry on the path stack consists of a path descriptor, an 8-bit path event counter, a path extension counter and other path specific information. The path profiler receives information per-

[1]This bit is set to 1 for unconditional branches.

taining to every retiring branch instruction from the processor pipeline via a *branch queue*, which serves to decouple the processor pipeline from the path profiler. Every branch read from the branch queue is decoded and classified as a call, a return, an indirect branch, a forward branch or a backward branch. The profiler then performs one or more of the following operations on the path stack depending on the type of branch being processed:

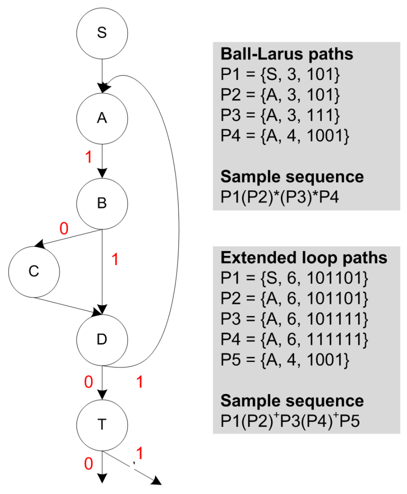
- *path_stack_push* : Pushes a new entry on the path stack with the starting address field of the path descriptor set to the target address of the branch being processed. All other fields of the new entry are reset to zero.
- *path_stack_pop* : Pops the current entry on top-of-stack and makes it available to all interested entities.
- *path_stack_update* : Updates the entry on top-of-stack with information about the branch being processed. The update involves incrementing the length of the path, updating the branch outcomes and incrementing the event counter. During update, if the length of the path on top-of-stack exceeds the predefined threshold value, the profiler logic pops the entry and pushes a new entry for a path beginning from the target of the current branch.
- *path_stack_update_count* : Increments the event counter associated with the path descriptor on top-of-stack without updating the path length or outcomes. This operation is used if it is desired that branches like calls and returns should not be explicitly recorded in the path.

The mapping between the branch type and set of operations to be performed is specified by the programmer in a *Path Profiler Control Register* (PPCR). However, the following restrictions on the mapping apply. For any branch, either one of *path_stack_update* or *path_stack_update_count* can be performed. Also, the order in which operations are performed is fixed, *viz path_stack_update/path_stack_update_count* followed by *path_stack_pop* and *path_stack_push*. These restrictions notwithstanding, the hardware path profiler can be programmed to detect several types of paths.

Detecting acyclic, intra-procedural paths: The branch type−profiler operations mapping shown in Table 5.1 enables the profiler to detect a variant of Ball-Larus paths that terminate on backward branches. Path entries are pushed on calls and popped on returns. On a forward branch, the path descriptor on the top-of-stack is updated with information about the branch. The path entry on top-of-path stack is updated and terminated on all backward branches. Paths are also terminated on indirect branches since such branches can have several targets. These profiler operations ensure that there exists exactly one entry on the path stack for every active procedure, and that the path stack grows and shrinks in the same way as stack frames on the program's runtime stack. Each entry on the path stack records the Ball-Larus path that the corresponding procedure is

Branch type	*update*	*update-count*	*pop*	*push*
call	×	√	×	√
return	×	√	√	×
forward	√	×	×	×
backward	√	×	√	√
indirect	√	×	√	√

Table 5.1: Branch type—profiler operation mapping for detecting Ball-Larus paths



Ball-Larus paths
P1 = {S, 3, 101}
P2 = {A, 3, 101}
P3 = {A, 3, 111}
P4 = {A, 4, 1001}

Sample sequence
P1(P2)*(P3)*P4

Extended loop paths
P1 = {S, 6, 101101}
P2 = {A, 6, 101101}
P3 = {A, 6, 101111}
P4 = {A, 6, 111111}
P5 = {A, 4, 1001}

Sample sequence
P1(P2)$^+$P3(P4)$^+$P5

Figure 5.4: The control flow graph, path descriptors and a sequence of Ball-Larus paths and extended loop paths detected for a sample execution of a procedure. Blocks S and C do not end with branches and block A ends with an unconditional jump to B. All other blocks end with conditional branches.

currently traversing. Figure 5.4 shows a control flow graph, the set of Ball-Larus path descriptors and a sequence of Ball-Larus paths generated by the path stack for a sample procedure.

Detecting extended paths: In order to detect extended paths, the path stack supports a *path extension* counter with every entry on the path stack. The programmer is required to specify the following options via the PPCR: (1) the type of extended paths that should be profiled i.e. paths spanning across procedure boundaries or those that extend beyond loop boundaries, and (2) a maximum extension count that indicates the number of procedure calls or backward branches that the path is allowed to span across. The mapping shown in Table 5.1 is reused.

When programmed to detect extended paths, the path detector processes forward and indirect branches in the usual manner. However, the operations performed on backward branches, calls and returns are qualified by the value of the extension counter of the path stack entry on top-of-stack. If the profiler is programmed to detect paths that span

Branch type	*update*	*update-count*	*pop*	*push*
call	×	√	√	√
return	×	√	√	√
forward	√	×	×	×
backward	√	×	√	√
indirect	√	×	√	√

Table 5.2: Branch type-Profiler operation mapping for detecting paths for a Whole Program Path.

```
compute dcache_access_power;
num_dcache_ports = 2;
if (num_dcache_access) {
  if (num_dcache_access <= num_dcache_ports)
    total_dcache_power +=  dcache_access_power;
  else
    total_dcache_power +=
      (num_dcache_access/num_ports)
           * dcache_access_power;
}
```

(a) Cycle level power model

```
compute dcache_access_cost;
num_dcache_ports = 2;
if (num_dcache_access) {
  if (num_dcache_access == 1)
    apportion_cost = dcache_access_cost;
  else
    apportion_cost = dcache_access_cost / 2;
  distribute apportion_cost to num_dcache_access instructions;
}
```

(b) Apportioning logic

Figure 5.5: (a) A power model that computes power consumption in L1 dcache and (b) corresponding apportioning logic that assigns costs to instructions that simultaneously access L1 dcache. Here, both dcache_access_power and dcache_access_cost are computed a priori; these are constants for a specific process technology.

across loop boundaries and a backward branch is encountered, the path on top-of-stack is allowed to expand if the corresponding extension counter value is less than the maximum extension count. In such a scenario, only the *path_stack_update* operation is performed and the extension counter is incremented. When the extension counter reaches the maximum count specified via the PPCR, the default set of operations are performed i.e the path is terminated. Figure 5.4 illustrates a set of extended loop paths detected and a sequence of paths recorded by the profiler for the given control flow graph.

When the profiler is tracking paths that span across procedure boundaries and a call instruction is encountered, the path on top-of-stack is terminated only if the extension counter value is equal to the maximum. Otherwise, the path is allowed to expand and the extension counter incremented. Conversely, paths are allowed to expand on procedure returns if the extension counter value is non-zero. Each return also decrements the extension counter. This set of actions generates path descriptors that represent inter-procedural paths through the program. It is important to note that with our profiler, the time complexity of detecting extended paths is similar to the complexity of detecting acyclic, intra-procedural paths.

Detecting paths for a Whole Program Path: The basic element of a Whole Program

Path (WPP) is a variant of the Ball-Larus path that also terminates at call instructions. A WPP is formed by compressing the sequence of such paths. Our profiler, configured using the mapping shown in Table 5.2, simplifies the process of constructing the WPP by generating the path sequence at low overhead and complexity. While processing a call, the profiler performs a *path_stack_pop* and terminates the current path before pushing a new entry on the stack. Similarly, the profiler performs a *path_stack_pop* followed by a *path_stack_push* on every return. Path descriptors generated by the path stack are fed to a software WPP profiler running as a separate thread, which compresses the sequence online and constructs the WPP.

Path stack consistency: For the path profiler to work correctly, the path stack must be maintained in state consistent with the program's execution. This requires special handling of certain events such as exceptions and setjmp/longjmp operations. The handlers for such events must repair the path stack, typically by popping entries of the path stack or purging the stack. Further, path stack overflows can occur due to the fixed size of the path stack. Implementations have the choice of handling overflows either by ignoring older entries on the stack, or by allowing the stack to grow into a region of memory specially allocated by the OS for the program being profiled. We find that a 32-entry path stack eliminates overflows for most of the programs we studied; we assume a path stack of this length for the rest of this study.

5.3 Associating architectural metrics with paths

Existing software-based path profilers are designed to track the frequency with which each path is traversed. However, future analysis tools and optimizations are likely to be interested in path-wise profiles of other metrics such as cache misses, branch mispredictions and pipeline stalls. Such profiles are important because the paths of interest to an optimizer could differ significantly depending on the metric associated with paths. For example, our studies using benchmarks from the SPEC CPU2000 suite have shown that path profiles for various architectural and power metrics have only small percentage of information in common with a path-frequency profile (25% for a branch misprediction profile, 32% for a L2 cache profile and 62% for a power profile) [50]. These results justify the need for flexible profiling schemes that can capture program behavior across different architectural metrics.

Extending existing profiling schemes to associate architectural metrics with paths is complicated due to the intra-procedural nature of paths and perturbation effects of the instrumented code [1]. However, our hardware path profiler can perform this task accurately and non-intrusively since precise information regarding all architectural events is directly available to the profiler. To track the occurrence of such events, each instruction in the processor pipeline is annotated with an *event counter* (an extension of per instruction tag in [46]). An instruction's event counter is incremented every time the instruction causes an architectural event of type X specified via the PPCR. When an instruction commits, the value in the instruction's event counter is used to update a *block event counter* maintained at the commit stage of the pipeline. The block event counter value is passed to the path profiler along with every committing branch, which in turn updates the event counter associated with the path on top-of-stack. When a path is popped off the path stack, its event counter value represents the number of events of type X that occurred along the path. The block event counter is itself reset after every branch instruction.

A limitation of this scheme is that architectural events caused by non-committing, speculative instructions are not accounted for since the profiler monitors committing instructions only. Apart from this anomaly, the profiler is capable of associating any architectural event with paths, thereby enabling precise path-based performance monitoring and bottleneck analysis.

Associating power consumption metrics with paths: Virtually all existing hardware profiling schemes are focused towards detecting and aggregating data pertaining to performance-related architectural events. While this was also the case with the initial design of our profiling scheme, we were subsequently interested in exploring possible reuse of the proposed hardware in tracking metrics related to power consumption. Our goal was to associate program paths with a count that provides an estimate of the power consumption caused by instructions along the path in a specific processor component/set of components. Such profiles enable power-aware compilers to identify and focus on regions of code that account for a significant fraction of the power consumption. Such profiles can also assist a programmer in analyzing the impact of traditional compiler/architectural optimizations on power consumption in specific regions of the program.

Our power profiling scheme assumes the availability of an accurate power model for each processor component of interest, parameterized by the component configuration and one or more architectural events. The power model is used to assign a *relative cost* to each architectural event related to the component. The cost associated with an event indicates the power consumed by the occurrence of the event relative to the event with the lowest consumption. For instance, the relative costs of accesses to each level of the cache hierarchy are derived from a power model for caches parameterized by the cache configuration and number of accesses. Since an instruction cache access typically consumes the lowest power, the costs of other events are relative to the instruction cache access.

Once costs are assigned to all events, the event detection logic associated with each component is extended to apportion the event cost to instructions that cause the events.

For simple analytical models, this translates to logic that increments the event counters of all event-causing instructions by an amount equal to the cost of the event. More complex cycle-level models can be implemented using logic that dynamically computes the apportioned cost based on online information. Figure 5.5 illustrates one such dynamic power model for the L1 data cache [10] and the corresponding apportioning logic implementation. Here, the actual power consumption and the apportioned cost are determined based on the number of simultaneous data cache accesses in each cycle and the number of ports available.

Note that our implementation of the power models assumes a constant activity factor, a significant parameter in power models for components such as caches, buses and register files. Rounding off errors during the process of computing relative costs, approximations in the apportioning logic and the loss of information due to non-committing but power consuming instructions also introduce inaccuracies. However, our results suggest that although these factors cause discrepancies in the total power estimates, their impact on the quality of hot path profiles is minimal. We evaluate the effectiveness of our profiling scheme in collecting power specific profiles in Section 5.5.

5.4 Collecting Hot Path Profiles

Much of the execution time overhead incurred by existing path profiling techniques is attributed to the hash-and-update operation performed when a path terminates [25]. This overhead can be reduced if the hash table that stores the path profile is maintained in hardware. Such a hardware implementation must be capable of generating a profile accurate enough to drive path-based optimizations without a loss in their effectiveness. Additionally, the quality of the profile must be independent of the duration of profiling and the metric associated with paths.

We evaluated several hardware profiler design configurations and next describe a simple, low overhead path collection scheme that meets these requirements. The path collection mechanism is based on a hardware structure called the *Hot Path Table* (HPT) illustrated in Figure 5.6. Each entry in the HPT consists of a path descriptor and a 32-bit accumulator. The HPT receives a sequence of path descriptors and associated counts from the path stack. An index into the HPT is computed from the fields of each incoming path descriptor. If a corresponding entry is found in HPT, the accumulator is incremented by the count associated with the incoming path. If the lookup fails, an entry from the indexed HPT entry set is selected for replacement and initialized with information about the incoming path descriptor.

The HPT design parameters that determine the effectiveness of the hot path collection scheme include the HPT size and associativity, the replacement policy and the indexing

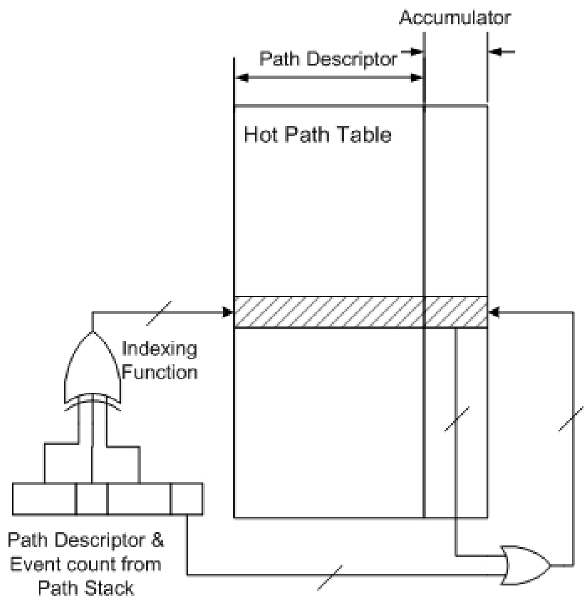


Figure 5.6: The Hot Path Table that collects hot path profiles. The HPT is indexed using bits from the incoming path's address, length and branch outcomes.

Processor core	Out-of-Order issue of up to 4 instructions per cycle, 128 entry reorder buffer, 64 entry LSQ, 16 entry IFQ
Functional units	4 integer ALUs, 2 integer MULT/DIV units, 4 floating-point units and 2 floating-point MULT/DIV units
Memory hierarchy	32KB direct mapped L1 instruction and data caches with 32 byte blocks (1 cycle latency) , 512KB 4-way set associative unified L2 cache (10 cycle latency), 100 cycle memory latency
Branch predictor	Combined: 12-bit (8K entry) gshare/(8K entry) bimodal predictor with 1K meta predictor, 3 cycle branch mis-prediction latency, 32 entry return address stack, 2K entry, 4-way associative BTB

Table 5.3: Baseline Processor Model

Parameter	Low	High
Number of HPT entries	128	2048
HPT associativity	2	32
HPT indexing scheme	Bits from path starting address	XOR(Bits from path starting address, path length, branch outcomes)
L2 cache size	256KB	2048KB
L2 cache associativity	1	8
Branch predictor	2K entry, 2-level predictor	Hybrid with 8K entry bimodal, 2K entry 2-level and 8K entry metatable

Table 5.4: Parameters considered during experiments based on Plackett-Burman design to determine an effective HPT configuration. The corresponding low and high values are also listed.

function. Our initial experiments indicate that LRU replacement, which is oblivious to the frequency of access of entries, does not capture and retain information about an adequate fraction of paths. The Least Frequently Used (LFU) policy serves the purpose of retaining frequently used entries. However, the execution time overheads of implementing LFU (*log n* for an n-way associative structure) have prevented its use in other cache structures. In the context of the HPT, a moderately expensive replacement policy does not have a significant impact on the overall execution time because *(1)* the HPT does not lie on the processor's critical path and *(2)* HPT updates are relatively infrequent (once for every path) and HPT replacements even less frequent. Our experiments with different hotness criteria and HPT configurations reveal that the LFU replacement policy outperforms others without a significant increase in execution time overheads. Moreover, an implementation of LFU for the HPT does not incur additional space overheads since frequency information is available in counters associated with every HPT entry. For the rest of the chapter, we assume an HPT implementation that uses LFU replacement. We discuss the impact of HPT size, associativity and indexing scheme on profile accuracy and profiler overheads in Section 5.5.

5.5 Experimental Evaluation

The success of a profiling technique can be measured by the accuracy of the profile, implementation costs and overheads of profiling. In this section, we evaluate our hardware path profiler on these counts.

- We explore the impact of various profiler design parameters on profile accuracy by comparing hardware generated profiles with a complete profile obtained using an infinitely large HPT. We compare profiles using the *overlap percentage* metric [4,18], which indicates the percentage of information common to the profiles. Our results show that average profile accuracy of 88% is obtained using an HPT that occupies approximately 7KB of real estate.

- We assess the quality of hardware generated profiles in a real-world application by using the profiles to drive superblock formation in the *gcc* compiler. We find that performance benefits of superblock scheduling driven by HPT generated path profiles are similar to those obtained using a complete path profile.
- We evaluate the use of our profiler in gathering profiles where paths are associated with architectural metrics such as L2 cache misses, branch mis-predictions and a metric that estimates power consumption in the cache hierarchy.
- Using a cycle-accurate superscalar processor simulator, we find that the execution time overheads of our profiling scheme are low (0.6% on average), enabling the use of the profiler in cost-sensitive environments.

5.5.1 Experimental Methodology

We performed simulation experiments using 12 programs from the SPEC CPU2000 benchmark suite *gcc, gzip, mcf, parser, vortex, bzip2, twolf, perlbmk, art, equake, mesa* and *ammp*. We extended SimpleScalar [12], a processor simulator for the Alpha ISA, with an implementation of our hardware path profiler. The baseline micro-architectural model of the processor is shown in Table 5.3.

Due to simulation time constraints, overlap percentages in Section 5.5.2 are reported from simulations of 15 billion instructions for alpha binaries precompiled at *peak* settings, running on their reference inputs. We validated our simulation methodology using complete runs of as many of the programs as possible, finding an average deviation of 6% from the reported values. We extended the *gcc* compiler (version 3.4) with a path-based superblock formation pass. Execution times for complete runs of superblock scheduled binaries optimized at level -O2 were obtained using the hardware cycle counter on an Alpha AXP 21264 processor under the OSF V4.0 operating system. Profiling overheads reported in Section 5.5.3 are estimated using out-of-order processor simulations for complete runs of the programs with the MinneSPEC inputs [28].

5.5.2 Quality of Hardware Path Profiles

We use two metrics, the *complete overlap percentage* and the *dynamic overlap percentage* to quantify the accuracy of hardware generated profiles. The complete overlap percentage is obtained when profiling is enabled for the entire duration of program execution. To assess profile accuracy under conditions where the profiler is activated for short durations

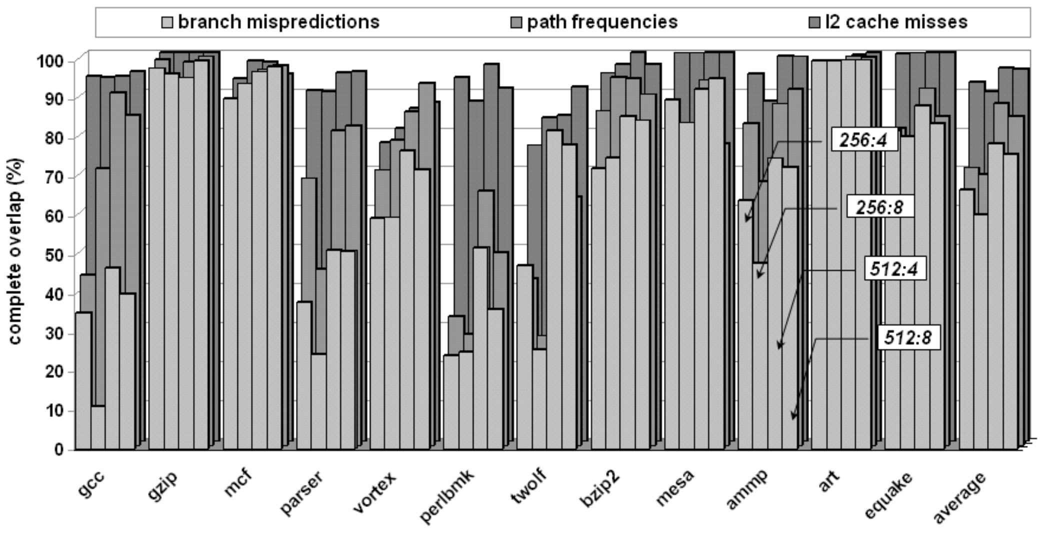


Figure 5.7: Complete overlap percentages for various profiler configurations when profiling path frequencies, L2 cache misses and branch mis-predictions.

during program execution, we define the dynamic overlap percentage as the average of overlap percentages computed for path profiles over non-overlapping 100 million instruction execution windows.

We performed a simulation study using Plackett-Burman experimental design [40, 54] to identify a path profiler configuration that realizes our design goals. Table 5.4 lists the input parameters we used in this design with their low and high values. We also conducted Plackett-Burman experimental studies where metrics other than path frequency – number of L2 cache misses and number of branch mis-predictions – are associated with paths. Our results show that the HPT size, HPT associativity and the HPT indexing scheme are the three most important profiler parameters [50]. On the other hand, the cache and branch predictor related parameters are of significantly less importance. This leads us to conclude that the hardware profile accuracy is unaffected by the underlying architecture. With these less important parameters eliminated, we next performed a full factorial study with HPT size, associativity and the indexing scheme as parameters. Our experiments reveal that an HPT indexing function that XORs bits from the path's starting address with path length and branch outcomes outperforms other functions we evaluated [50]. We use this indexing function in the rest of the study.

Figure 5.7 shows the complete overlap percentages for four HPT configurations. When profiling path frequencies, the best average complete overlap percentage of 88% is obtained using a 512 entry, 4-way set associative HPT. The overlap percentage with this configuration is 96% when the metric associated with paths is L2 cache misses. We attribute the higher coverage to the observation that L2 cache misses are concentrated along

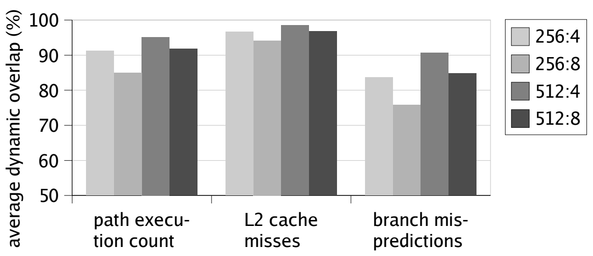


Figure 5.8: Average dynamic overlap percentages for various profiler configurations when profiling path frequencies, L2 cache misses and branch mis-predictions.

a smaller number of paths and exhibit more pronounced locality than paths themselves. On the other hand, the average complete overlap percentage dips to 78.4% when branch mis-predictions are profiled, a reflection of the fact that branch mis-predictions are usually distributed over a larger number of paths. Certain programs (*gcc, perl, twolf*) have lower overlap percentages since the working set of paths in these programs is large (Figure 5.2), making it harder for a hardware structure of limited size to capture a high fraction of execution. We also find that a further increase in HPT size leads to additional improvements in overlap percentages, although the gains taper off for HPTs with over 2048 entries [50].

The impact of HPT configuration on dynamic overlap percentages is summarized in Figure 5.8. Notice that the dynamic overlap percentages are higher than their static counterparts. This is to be expected since the dynamic overlap percentage is computed using path profiles collected over short time durations. For the 512 entry, 4-way set associative HPT, we obtain average dynamic overlap percentages of 95.5%, 98% and 91.2% when profiling path frequencies, L2 cache misses and branch mis-predictions respectively.

Although the overlap percentage metric quantifies profile accuracy, it says little about the performance impact of using a less than complete profile in a real world application. We obtain a direct assessment of the quality of hardware generated paths profiles by using them to drive superblock scheduling [23] in the gcc compiler. Our implementation of the path-based superblock formation pass identifies hot traces using path profiles followed by tail duplication, superblock enlargement and loop unrolling using heuristics similar to those used in [23]. The path-based superblock scheduler improves execution time by an average 9.3%, with a maximum of 14% over the baseline (-O2 optimized, traditional local scheduling).

Figure 5.9 shows the execution times of optimized program binaries that use path profiles generated by different HPT configurations. Program execution time is reported relative to that of a binary executable generated using a complete path profile. Observe that the difference between execution times is 0.12% on the average with a maximum of 2.2%. For certain benchmarks, superblock scheduling using hardware-generated path profiles performs marginally better than scheduling using complete profiles; this behavior

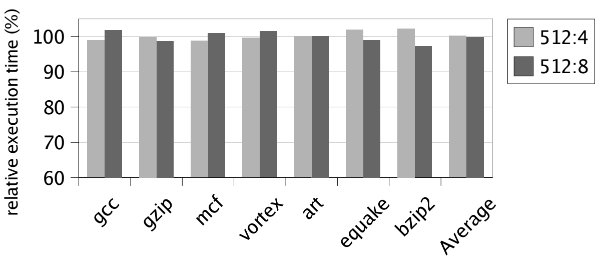


Figure 5.9: Execution times of binaries superblock-scheduled using path profiles from two HPT configurations normalized against the execution time of a binary scheduled using a complete path profile.

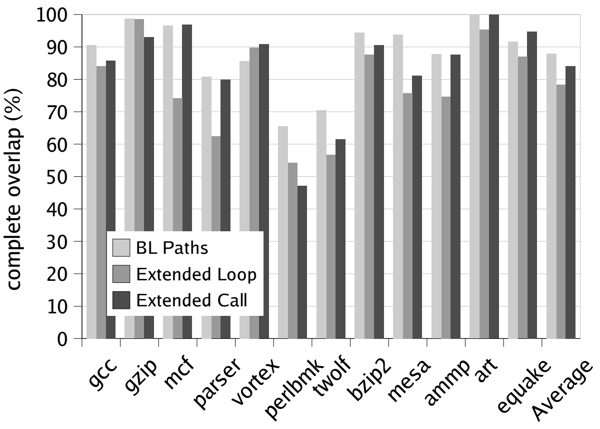


Figure 5.10: Accuracy of complete extended path profiles collected using the hardware path profiler. Paths extend across one backward branch/procedure call.

is attributed to secondary cache and branch prediction effects. On the whole, a minor change in average execution time indicates that hardware path profiles are comparable in quality to and can be used instead of complete path profiles.

Quality of extended path profiles: Next, we evaluate the effectiveness of the hardware profiler in collecting extended path profiles. Figure 5.10 shows the complete overlap percentages for extended paths that span across one backward branch and those that extend beyond one procedure call for a 512-entry, 4-way associative HPT. The average overlap percentages for such paths are 78.35% and 84.10% respectively. The reduction in overlap percentages when compared to a BL path profile is due to the increase in the number of unique paths traversed. It remains to be seen whether the reduced profile accuracy can be tolerated by real-world applications.

Quality of path-wise power profiles: Unlike other architectural metrics, the quality of path-wise power profiles cannot be assessed in isolation of the power models used to

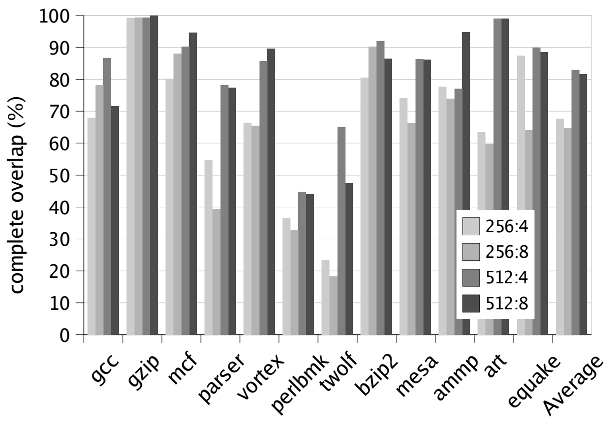


Figure 5.11: Accuracy of complete path-wise power profiles for various HPT configurations, obtained using an analytical model for power consumption in the cache hierarchy.

estimate power consumption in various components. In this section, we first determine whether the choice of a power model has an influence on the nature of path profiles. Our evaluation uses power consumption in the cache hierarchy as the metric associated with paths, primarily because a significant fraction of the overall power consumption is attributed to the caches [10]. Of the several candidate power models for caches [10,26,27], we chose two, an analytical power model from Kadayif et al [26] and a cycle level power model used by the Wattch power simulator [10]. The analytical model was used to compute the relative costs of hits and misses at each level of cache. Apportioning logic similar to Figure 5.5 was designed for the Wattch power model and integrated into a cycle-accurate processor simulator.

A comparison of the path-wise power profiles obtained using the two power models shows a strong similarity in the relative ordering of paths in the profiles despite differences in the absolute value of the associated power metric. This observation suggests that for caches, the analytical model identifies hot paths as well as the accurate cycle-level power model. Figure 5.11 illustrates the impact of various HPT configurations on the accuracy of path-wise power profiles generated using the analytical power model. An average complete overlap percentage of 82.9% is obtained using a 512 entry, 4-way associative HPT. Further investigations into the relative quality of these profiles and their impact on power-aware compiler optimizations are left for future work.

5.5.3 Profiling Overheads

Using the hardware path profiler during program execution can lead to degraded performance if the profiler does not service branch instructions faster than their rate of retirement. To study this, we incorporated our path profiler into a cycle-accurate superscalar processor pipeline simulator. Profiler operations are assigned latencies proportional

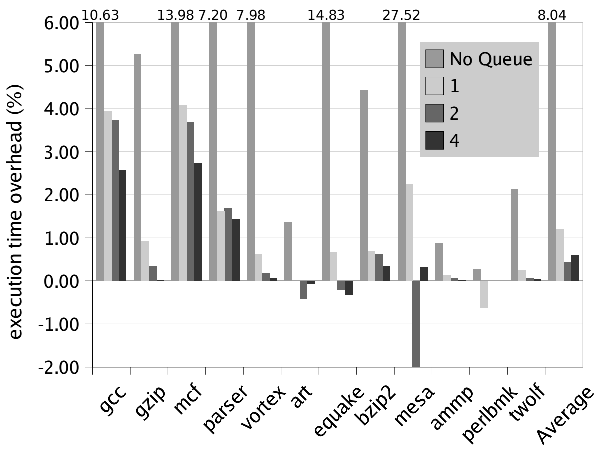


Figure 5.12: Execution time overheads incurred due to the hardware profiler while profiling BL paths for various branch queue sizes.

to the amount of work involved in carrying out those operations: the *path_stack_push*, *path_stack_update* and *path_stack_update_count* operations are assessed a latency of one cycle whereas the latency of a *path_stack_pop* is one cycle plus the latency of updating the HPT. Since the HPT uses a LFU based replacement policy, an HPT miss incurs a cost of *log(n)* cycles, where n is the associativity of the HPT. The latency of processing a branch is the sum of latencies of the profiler operations performed while processing the branch. If a retiring branch finds the branch queue full, the commit stage stalls and no further instructions are committed until the stalling branch can be accommodated in the branch queue.

The execution time overheads incurred while collecting a BL path profile using a 512-entry, 4-way set associative HPT for various branch queue sizes are shown in Figure 5.12. We observe that a 4-entry branch queue sufficiently buffers the profiler from the pipeline and reduces average execution time overheads from 8.04% to 0.6%. This also represents a sharp drop from the typical 30-45% overheads incurred by the Ball-Larus path profiling scheme [6]. Moreover, profiling overheads remain unchanged even when the profiler is configured to collect otherwise expensive extended paths. Our experiments ignore the overheads of transferring the HPT generated profile (approximately 6KB in size) to user memory. Even conservative estimates of this overhead (a few thousands of cycles) are negligible when compared to the execution time of the program.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Understanding a program's runtime behavior and ensuring that its actual behavior is in tune with the expected behavior is critical to the program's utility and effectiveness. In this thesis, we consider the problem of characterizing a program's control flow behavior, an aspect of program behavior that drives several dynamic program analysis techniques. The thesis focuses on path profiling, a profiling technique which attempts to characterize a program's control flow behavior by tracking paths exercised in a given execution.

Although paths have proven to be a powerful and pragmatic program abstraction, the widespread use of path profiling has been restricted due to the lack of efficient techniques for profiling paths. This is specially true in environments where the cost of program profiling must be kept to a bare minimum, necessitating the use of cheaper but less accurate control flow profiling techniques such as basic block and edge profiling. In this thesis, we propose two efficient path profiling schemes that overcome these problems and enable the use of path profiling in a wide variety of environments.

First, we present preferential path profiling, a new technique that profiles a specified subset of all program paths with very low overhead. Preferential path profiling labels the paths of interest compactly using a novel numbering scheme. This compact path numbering allows our implementation to use array based counters instead of hash table-based counters for gathering path profiles and significantly reduces execution time overhead (average of 14% compared to 50% overheads incurred by the Ball-Larus profiling scheme). By drawing parallels between arithmetic coding and path numbering we establish an optimality result for our compact path numbering scheme. We also show that preferential path profiling can be extended to capture inter-procedural paths using selective procedure inlining.

Apart from reducing profiling overheads, preferential path profiling has the ability to classify paths during execution. In other words, preferential path profiling can dynamically classify a path that has just occurred into one of two pre-determined classes. We find that this feature is extremely useful in applications like residual path profiling, where we are interested in identifying all paths executed by the deployed software that were untested during software development. Because of its low overheads, preferential path profiling enables residual profiling on deployed systems. Information generated by residual path profiling can be used to assess and improve the effectiveness of test suites, which are usually designed with little information about a program's actual usage.

This thesis also proposes and evaluates a hardware path detection and profiling scheme that is capable of generating high quality hot path profiles with minimal space and time overheads. The profiler derives its flexibility from a generic path representation and a programmable interface that allows various types of paths to be profiled and several architectural metrics to be tracked along paths using the same hardware. These characteristics enable the use of the profiler in a host of static and dynamic optimization systems.

6.2 Future directions

6.2.1 Algorithmic extensions

In Chapter 3, we started by defining the problem of finding an assignment of weights to the edges of a given DAG such that a given set of interesting paths are assigned unique identifiers and that these identifiers are separated from the identifiers of uninteresting paths. Using a counter-example, we proved that this problem is unsolvable in general and relaxed the requirement that the edge assignment should separate the interesting and the uninteresting paths as long as uniqueness and compactness were satisfied. However, there may be several other ways of relaxing the problem. For instance, while the finding an edge assignment that perfectly separates paths is hard, it may be possible to find edge assignments that *approximately* separates the interesting paths from the uninteresting ones. An approximate edge assignment may classify a few uninteresting paths as interesting (or vice versa), resulting in false positives (or false negatives). However, such assignments eliminate the need for a second counter and may be acceptable in certain scenarios as long as the number of wrongly classified paths is small.

In Chapter 4, we prove that the edge assignment generated using PPP is optimal in the information-theoretic sense. However, the assignment may not be combinatorially optimal, as illustrated by the discussion following Section 3.3. In our attempt to find the combinatorially optimal solution and compare the solutions generated by PPP, we formulated our problem as an ILP problem. Given a DAG and a set of interesting paths (which we identified by running the program with a test input), we expressed the requirements for uniqueness and compactness as constraints in ILP and used an off-the-shelf ILP solver. The variables in these constraints represent the weights assignment to edges of the DAG and the goal of the ILP solver was to minimize the difference between the minimum and the maximum identifier allocated to an interesting path while ensuring that each of the interesting paths is assignment a unique identifier. The ILP solver was able to find optimal solutions when the problem size was small (small number of interesting paths and edges in the DAG). For larger problem sizes (more than 20 interesting paths), the solver did not terminate for a few days. We conclude that the ILP approach to finding the optimal solution does not scale. Other techniques for finding the optimal solution are left for future work.

6.2.2 Applications of PPP

Bug isolation

We have already shown that PPP is naturally suited for residual path profiling. We observe that if a program fails, some of the paths exercised in the failing run but not exercised by a test suite are likely to be the root cause of bugs. If used with a rigorous test suite, PPP can be seen as a dynamic version of the bug isolation technique proposed by Ball et al [7]. Similarly, using paths in general and untested paths in particular can improve the precision and efficiency of statistical bug isolation techniques like co-operative bug isolation (CBI) [31], which have traditionally used individual predicates as predictors for bugs.

Improving test effectiveness

Efficient path profiling techniques have a significant role to play in improving the efficiency of testing. Path profiles can help *prioritize* test cases in regression testing. Test cases that exercise paths along which code has been modified are more likely to uncover bugs than other paths. These test cases can be assigned a higher priority in the testing cycle. If a modified program passes these high priority tests, chances of regression are less likely. PPP can also be used for eliminating redundant test cases simply by detecting whether a given test case exercises any paths that have not be exercised by other test cases.

6.2.3 Applications of HPP

We believe that the availability of semantically rich and detailed path information in hardware opens the doors to several architectural optimizations. For instance, we have proposed a phase detection scheme [50] that accurately detects phase changes using the sequence of acyclic, intra-procedural paths generated by our hardware profiler. Our technique detects subtle phase changes missed by existing phase detection schemes that rely on basic block profiles. Our phase detector can identify phase changes that occur when the set of paths exercised by the program changes but the set of basic blocks remains the same. In other words, the program starts exercising the same fragments of code differently.

Possible avenues for future work include the use of hot path information in improving the performance of trace caches and pre-computing memory reference addresses for cache pre-fetching. In addition, our low-overhead path profiling techniques can be applied to perform more informed compiler optimizations driven by real-usage path-profiles. These profiles can be also used to check hotness assumptions made by in-house optimization. They could also be used for security applications where certain program behaviors need to be restricted based on runtime criteria.

Bibliography

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Context Sensitive Profiling. In *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 85–96, 1997.

[2] Glenn Ammons and James R. Larus. Improving Data-flow Analysis with Path Profiles. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 72–84, 1998.

[3] T. Apiwattanapong and Mary Jean Harrold. Selective path profiling. In *Workshop. on Program Analysis for Software Tools and Engineering (PASTE)*, pages 35–42, 2002.

[4] Matthew Arnold and Barbara Ryder. A Framework for Reducing the cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 168–179, June 2001.

[5] Thomas Ball and James Larus. Programs follow paths. Technical Report MSR-TR-99-01, Microsoft Research, 1999.

[6] Thomas Ball and James R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO)*, pages 46–57, 1996.

[7] Thomas Ball, Mayur Naik, and Sriram Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 97–105, 2003.

[8] Michael Bond and Kathryn McKinley. Continuous path and edge profiling. In *International Symposium on Microarchitecture (MICRO)*, 2005.

[9] Michael Bond and Kathryn McKinley. Practical path profiling for dynamic optimizers. In *International Symposium on Code Generation and Optimization (CGO)*, pages 205–216, 2005.

[10] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 83–94, 2000.

[11] B. Bruegge and P. Hibbard. Generalized Path Expressions: A High Level Debugging Mechanism. *Journal of Systems and Software*, 3(4):265–276, 1983.

[12] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Toolset. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, 1996.

[13] Trishul Chilimbi, Aditya Nori, and Kapil Vaswani. Quantifying the Effectiveness of Testing via Residual Path Profiling. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, September 2007.

[14] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A Comparison of Dataflow Path Selection Criteria. In *Proceedings of the 8th International Conference on Software Engineering (ICSE)*, pages 244–251, 1985.

[15] Thomas M. Conte, Burzin Patel, Kishore N. Menezes, and J. Stan Cox. Hardware-Based Profiling: An Effective Technique for Profile-Driven Optimization. *International Journal of Parallel Programming*, pages 187–206, Vol 24, No. 2, April 1996.

[16] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., N. Y., 1991.

[17] Jeffery Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 292–302, December 1997.

[18] P. T. Fellar. Value Profiling for Instructions and Memory Locations. CS98-581, University of California, San Diego, April 1998.

[19] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path Profile Guided Partial Dead Code Elimination using Predication. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, page 246, November 1997.

[20] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path Profile Guided Partial Redundancy Elimination Using Speculation. In *Proceedings of the 1998 International Conference on Computer Languages (ICCL)*, page 230, 1998.

[21] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall PTR, 2nd edition, 1998.

[22] Timothy Heil and James E Smith. Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 281–290, December 2000.

[23] Wen W. Hwu and Scott A. Mahlke. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, pages 7(1–2):229–248, May 1993.

[24] Quinn Jacobson, Eric Rotenberg, and James E. Smith. Path-Based Next Trace Prediction. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 14–23, 1997.

[25] Rahul Joshi, Michael D. Bond, and Craig B. Zilles. Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 239–250, March 2004.

[26] I. Kadayif, T. Chinoda, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. vEC: Virtual Energy Counters. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 28–31, 2001.

[27] M. B. Kamble and K. Ghose. Analytical Energy Dissipation Models for Low Power Caches. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 1997.

[28] AJ KleinOsowski and David J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 2002.

[29] A. Kolmogorov. Three approaches to the quantitative definition of information. *Prob. Peredach Inform*, 1(1):3–11, 1965.

[30] James R. Larus. Whole Program Paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 1999.

[31] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the International Symposium on Programming Language Design and Implementation (PLDI)*, June 2005.

[32] K. S. McKinley, J. Burrill, M. D. Bond, D. Burger, B. Cahoon, J. Gibson, J. E. B. Moss, A. Smith, Z.Wang, and C. Weems. The Scale compiler. *http://ali-www.cs.umass.edu/Scale*, 2005.

[33] D. Melski and T. W. Reps. Interprocedural path profiling. In *Proceedings of the 8th International Conference on Compiler Construction (CC)*, pages 47–62, 1999.

[34] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen mei W. Hwu. A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, pages 136–147, June 1999.

[35] Microsoft Phoenix Compiler. *http://research.microsoft.com/phoenix*, 2007.

[36] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 15–23, 1995.

[37] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, June 2001.

[38] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st international conference on Software engineering (ICSE)*, pages 277–284, 1999.

[39] Erez Perelman, Trishul Chilimbi, and Brad Calder. Variational path profiling. In *Parallel Architectures and Compilation Techniques '05 (PACT)*, pages 7–16, 2005.

[40] R. Plackett and J. Burman. The Design of Optimal Multifactorial Experiments. *Biometrika*, Vol 33, Issue 4:305–325, June 1956.

[41] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The Use of Program Profiling for Software Maintainance with Applications to the Year 2000 Problem. In *Proceedings of the International Symposioum on Foundations of Software Engineering (FSE)*, pages 432–449, 1997.

[42] J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM J. Res. Develop.*, 23(2):149–162, 1979.

[43] Eric Rotenberg, Steve Bennett, and Jim Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO)*, 1996.

[44] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computuing Systems*, 15(4):391–411, 1997.

[45] Michael D. Smith. Overcoming the Challenges of Feedback-directed Optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo 2000)*, pages 1–11, 2000.

[46] Brinkley Sprunt. Pentium 4 Performance Monitoring Features. *IEEE Micro*, pages 22(4):72–82, July-August 2002.

[47] Sun Microsystems Inc. *UltraSPARC User's Manual*, 1997.

[48] Sriram Tallam, Xiangyu Zhang, and Rajiv Gupta. Extending Path Profiling across Loop Backedges and Procedure Boundaries. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 251–262, March 2004.

[49] Kapil Vaswani, Aditya Nori, and Trishul Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *ACM SIGPLAN Conference on Principles of Programming Languages (POPL)*, pages 351–362, 2007.

[50] Kapil Vaswani, Mattew J. Thazhuthaveetil, and Y. N. Srikant. Representing, Detecting and Profiling Paths in Hardware. Technical Report IISc-CSA-TR-2004-7, Indian Institute of Science, May 2004.

[51] Kapil Vaswani, Matthew J. Thazhuthaveetil, and Y. N. Srikant. A Programmable Hardware Path Profiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, March 2005.

[52] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[53] Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. Structural Path Profiling: An Efficient Online Path Profiling Framework for Just-In-Time Compilers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept-Oct 2003.

[54] Joshua Yi, David Lilja, , and Douglas Hawkins. A Statistically Rigorous Approach for Improving Simulation Methodology. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, February 2003.

[55] C. Young and M. D. Smith. Better Global Scheduling Using Path Profiles. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, November 1998.

[56] C Young and M. D. Smith. Static correlated branch prediction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):1028–1075, 1999.

[57] Craig Zilles and Gurindar Sohi. A Programmable Co-processor for Profiling. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA)*, pages 241–253, January 2001.