High Frequency Dynamic Profiling of Managed Runtimes with SHIM

Theodore T. Olsauskas-Warren

A thesis submitted for the degree of
Bachelor of Software Engineering (Honours)
The Australian National University

October 2015

Typeset in Palatino by TEX and LATEX 2$_\varepsilon$.

Except where otherwise indicated, this thesis is my own original work.

Theodore T. Olsauskas-Warren
22 October 2015

To Xi, without whom none of this is possible.

Acknowledgements

To Steve and Xi, your bottomless wealths of knowledge have been the driving force behind this work. No matter how lost I felt, a conversation with either of you left me feeling with a renewed sense of drive and direction.

Abstract

Managed runtimes rely on feedback captured during execution via dynamic profiling to guide the optimisations they perform. Such optimisations have enabled Just-In-Time compiled code to effectively bridge the performance gap to statically compiled code. Because exhaustive profiling of application execution comes with a substantial performance burden, the data which informs this optimisation processes is commonly generated via the use of interrupts, in which the execution of the application thread is suspended at regular intervals whilst brief bursts of profiling occurs.

The maximum frequency at which interrupts can occur is generally restricted by the operating system to be no greater than single digit kilohertz. Hence the temporal resolution of any interrupt based profiling technique is confined to the millisecond range. There is no quick fix to be found here, because increased timer interrupt frequency can result in system instability and severe performance penalties.

Limitations in the sampling speed available to current profiling techniques results in an inability to observe high-frequency application behaviour; such behaviour becomes blurred across the large sampling period. This affect limits the level of specificity with which observed performance characteristics can be attributed, generally to the program subsystem level.

SHIM is a new technique which leverages the existence of unutilised hardware threads to enable continuous, low overhead, high frequency sampling without a reliance on interrupts. SHIM allows semantic information and CPU performance data capture to be performed on executing applications in the micro to nanosecond range. On this time scale, behaviour changes across an individual function may be observed.

It is my thesis that the drastically increased sampling frequency enabled by SHIM can have practical benefits in the world of managed language optimisations. To demonstrate this, we apply SHIM to JavaScript, enabling micro-architectural performance statistics to be attributed to user level JavaScript functions and their calling context, thus realising greatly enhanced temporal resolution and attribution specificity.

We also apply SHIM based techniques to the generation of frequency weighted dynamic call graphs within both Java and JavaScript. We find that even with the massive increase in sampling speed afforded by SHIM, current burst sampling techniques outperform continuous sampling DCGs in both accuracy and overhead.

However, through combining the core concepts of both SHIM and burst sampling, we are able to create DCGs which significantly exceed the accuracy of those creatable with existing techniques, at only a moderate performance impact. These results confirm the practical usefulness of increased sampling frequency, encouraging further exploration of its applications, as well as provide a foundation for direct enhancements to existing adaptive optimisation techniques.

Contents

Chapter 1

Introduction

This thesis demonstrates some of the practical applications of SHIM, a profiling technique which enables low overhead sampling to occur at frequencies orders of magnitude higher than existing techniques. SHIM allows profiling to break through the frequency barriers imposed on existing interrupt based techniques, allowing developers to gain unparalleled insight into the runtime behaviour of their programs. It is my thesis that the increase in sampling frequency afforded by this new technique can have a practical impact on the optimisations performed in managed languages.

1.1 Thesis Statement

Dynamic profiling of managed languages plays a critical role in the optimisation process of their runtimes. Whether through the guiding of development optimisations in offline analysis, or by informing run time compiler choices through adaptive optimisation, the quality of the information input into these improvement techniques is a significant component of their overall efficacy. Current techniques for generating this information are limited

Current interrupt based profiling techniques are fundamentally limited in the maximum sampling frequency they may achieve. This limitation presents itself in two interlinked ways. Firstly, an interrupt based sampling technique inherently stops execution of the target program in order to perform sampling. Because execution is suspended whilst sampling occurs, sampling at high frequencies creates drastic performance overheads. Secondly, partially in response to the potential incurred overhead, the Linux kernel places a hard-coded limit on the frequency with which interrupts may occur.

This limitation in frequency constrains the temporal resolution of currently available sampling techniques. This loss in resolution results in the blurring of high-frequency behaviours present within the application. A lower resolution also constrains the ability to attribute performance characteristics to different sections of the program. Comparatively long running program subsystems, such as garbage collection, may with some difficulty be profiled in this way, but drilling down further is not possible.

The requirement to suspend thread execution to perform sampling can be circumvented by moving sampling to an alternate hardware thread. In this scenario, the profiler is no longer required to interrupt the execution of the profiled program in order to perform a sample.

This multi-threaded approach to sampling is the key idea behind SHIM, a new technique recently developed at ANU by Yang, Blackburn, and McKinley [2015]. SHIM allows profiling at sampling resolutions as fine as 15 cycles, some three orders of magnitude smaller than the minimum achievable with interrupt based techniques. A SHIM observer thread executes concurrently with the program to be profiled and periodically reads hardware performance counters, along with semantic information in the form of software tags, generated through instrumenting the profiled application. This information is used to attribute the performance counters in a way meaningful to the developer. Owing to the increased sampling frequency, this attribution can occur on a much finer structural scale than information generated through interrupt based techniques.

This thesis looks at some of the practical applications of SHIM based techniques in two popular managed languages, JavaScript and Java, in two of their high performance runtimes, SpiderMonkey and Jikes RVM respectively. Through these applications, we aim to show that the benefits of high frequency sampling are not purely theoretical. To this end, we demonstrate how SHIM enables greater temporal resolution and structural specificity than existing micro-architecture performance analysis techniques. We also show that applying SHIM to the generation of dynamic, frequency weighted call graphs, provides an overhead-viable means to improve their accuracy over existing techniques.

In this thesis we will be utilising software tags to capture managed language function identifiers, as a way of attributing collected counter information to user level functions. We will be extending the initial idea of a single software tag in SHIM, to the creation and maintenance of separate data structures to enable SHIM to capture user level calling contexts, enabling the construction of the aforementioned call graphs.

1.2 Contributions

This thesis has three primary contributions:

Implementing SHIM in SpiderMonkey We implement the primary SHIM concept presented in its initial paper within Mozillas SpiderMonkey JavaScript runtime, enabling high-frequency capture and attribution of micro-architectural performance counters to JavaScript level code. We enable finer grained attribution by extending the core, single software tag of SHIM, to a multi-tag stack structure to enable calling context information to be captured alongside the aforementioned counters.

Generating dynamic call graphs with SHIM We move away from the initial micro-architecture focus of SHIM and instead utilise the driving idea of off-thread sampling to generate frequency weighted dynamic call graphs (DCGs). We perform implementation in both SpiderMonkey and the Jikes Java virtual machine, finding that despite a massive sampling frequency advantage over existing techniques, DCGs generated in this manner fail to exceed the accuracy of existing techniques. We offer an analysis of possible causes for this deficit, finding that even at these high frequencies, the sampling period is still too long to accurately capture shortly lived methods.

Combining existing concepts and SHIM based techniques to generate highly accurate dynamic call graphs We combine existing burst sampling concepts with the high sampling frequency afforded by SHIM to create frequency weighted DCGs that exceed the accuracy of existing techniques. We find this increase in accuracy comes only at a modest execution time overhead, and thus we argue presents a better method of increasing call graph accuracy than existing techniques.

1.3 Meaning

The improved insight in runtime performance provided through the application of SHIM to SpiderMonkey allows for a more thorough examination of its performance properties. This paves the way for optimisation strategies that are more well informed, more targeted, and hence more effective. The pervasiveness of JavaScript means that even marginal performance improvements in one of its popular engines will have far reaching impacts.

Our work in the domain of frequency weighted DCGs has two immediate impacts. Firstly, as we can straightforwardly improve the accuracy of generated DCGs, we can improve the quality of information fed to current adaptive optimisation systems. Our hope is that such improvements will have easily attainable benefits for the overall performance of the runtime. Secondly, we have demonstrated that the prior discovery of accuracy deficiencies in single sampling techniques extends to frequencies much higher than those explored in previous works. Just as previous work guided sampling techniques at slower timescales, so too will this work serve as a blueprint for DCG applications in faster ones.

Taken together, the contributions made by this thesis show that the higher frequencies made available by SHIM have demonstrable, positive impacts, on the world of dynamic profiling. This work practically exhibits that the limitations imposed by a reliance on thread self-sampling, and the interrupt mechanism are just that, limitations. We build on the work presented in the original SHIM paper to show that not only is the world of high-frequency, low-overhead sampling open, but it is very promising.

1.4 Thesis Outline

This thesis is constructed of five chapters as follows. Chapter 2 provides an overview of existing sampling techniques, along with alternatives to sampling, as well as an introduction to SHIM. In Chapter 3, we introduce SpiderMonkey, Firefox's JavaScript engine and explore what can be gained through the marriage of SpiderMonkey and SHIM. Also In this chapter, we extend the single tag concept of SHIM to incorporate calling context through the maintenance of a separate stack structure.

In Chapter 4 we move away from capturing performance counters to exclusively focusing on capturing calling context with a view to creating dynamic, frequency weighted call graphs. We achieve this through several different implementation approaches, each is considered in a different section within the chapter with an analysis of their benefits and drawbacks, both in terms of their implementation and performance.

Firstly, we look at a simple extension of the aforementioned separate stack structure in SpiderMonkey. Secondly, we introduce Jikes RVM, a high performance Java research virtual machine, and within it implement call graph generation using direct observation of the target thread. Lastly, we implement a ring buffer sampling structure in both Jikes RVM and SpiderMonkey to emulate the known effective burst sampling technique to improve the accuracy of generated call graphs.

Finally, in Chapter 5 we conclude the thesis. We reflect on the overall results, address any pertinent omissions, and offer what we believe to be the next logical steps to build on this work and continue demonstrating the practical usefulness of high frequency sampling.

Chapter 2

Background and Related Work

In this chapter we will look at the existing body of work surrounding dynamic profiling, we will explore the three primary dynamic sampling techniques and introduce SHIM, a new method for sampling that alleviates some of the problems associated with existing profiling techniques. We will also introduce the key types of data that we will be looking at throughout this thesis, hardware performance counters and dynamics call graphs.

2.1 Dynamic Profiling

Dynamic profiling is a form of program analysis that occurs during the execution of a program, with the goal of measuring some characteristic of the programs execution. Program analysis offers several benefits to developers, it can help to identify undesirable behaviours, such as memory leaks, which may be otherwise difficult to test for. It also allows a deeper inspection of program performance behaviour that simple wall clock analysis allows.

The performance insights gained by dynamic profiling are key to the development of managed runtimes. Profiling allows for detailed scrutiny of optimisation techniques used within these, such as the effectiveness of garbage collection or JIT compilation. In contrast to static analysis, which performs analysis on programs without executing them and is often built on abstract analysis techniques, dynamic profiling is heavily correlated with actual, rather than expected, program behaviour.

2.2 Existing Profiling Techniques

There are many different methods for carrying out dynamic profiling, each with their own particular set of benefits and drawbacks. Depending on the application, there is generally only a single profiling type that is able to satisfy any particular set of needs. This is partially a property of them all having significant drawbacks. Rather than selecting a profiling methodology that excels in the required area, typically one that merely has an acceptable drawback is chosen.

2.2.1 Simulation Based

Tools such as Valgrind [2015] and its derivatives completely abstract away the computer architecture. This enables incredibly precise analysis of program behaviour, but offers limited correlation to real hardware. Because of the complete abstraction, simulation based tools are exclusively used for offline analysis.

2.2.2 Interrupt Based

Utilised by such tools as perf_events [Linux 2014] and Intel's Vtune [Intel 2015b], interrupt based sampling is a statistical sampling technique that uses the interrupt mechanism to perform sampling of the target application. During an interrupt, the profiler records the relevant information whilst the execution of the target program is halted.

There is a fundamental problem with this approach, as the sampling frequncy increases, so too does

Limitations in sampling frequency with this approach limit the temporal resolution of the data they generate. High frequency program behaviours are thus blurred behind the averaging that occurs over the sample period. This limits interrupt based techniques ability to determine the root cause of performance behaviour.

2.2.3 Instrumentation Based

Instrumentation based profiling techniques use code segments inserted into program code to perform sampling actions. This has a benefit over interrupt based techniques in that the mechanism for performing the sample is usually lighter-weight, and does not require the explicit halting of execution and the associated accounting.

Despite this, the overhead experienced through instrumenting the code can in some cases be high, especially if the instrumentation is extensive. To combat this, code patching may be used. Code patching is a term used to describe the runtime addition and removal of sampling instrumentation code. One such usage is to perform runtime analysis to determine hot methods for JIT compilation. Once the compilation has been completed, the instrumentation may be removed [Suganuma et al.].

This technique introduces significant amounts of complexity to a system, requiring careful interaction with existing code. Generating statistical data with instrumentation based techniques is also challenging, as it is hard to guarantee when the sample code will be run.

2.3 Introduction to SHIM

Continued difficulty with simple clock rate increases has led to processors across a wide variety of environments to embrace multiple cores and the associated multiple hardware threads. Whilst the addition of multiple cores can theoretically increase performance by an equal factor, obtaining such gains often requires inherently parallelisable tasks. Such parallelism is easily achieved when tasks require little to no interaction with each other, as complex concurrency structures can be avoided.

SHIM [Yang et al. 2015], makes the argument that dynamic profiling is a task that can benefit heavily from parallelisation. The core idea behind SHIM is to move the responsibility of sampling away from the program thread and onto a separate observer thread. In doing this, SHIM removes the requirement that the application threads execution be suspended whilst a sample is performed.

This approach addresses the interlinked problem associated with interrupt based techniques; in order to sample faster it is required that the thread be interrupted more regularly, which in turn causes larger overhead. Without this requirement SHIM can sample at much higher frequencies, without running into prohibitive performance boundaries.

By utilising Simultaneous Multithreading (SMT), through implementations such as Intel's HyperThreading, SHIM can read core local performance information generated by the target application without interfering in its execution. By recording its own impact, and using a specialised kernel to accurately identify which threads are executing on which cores, SHIM can negate most of it's observer effect. This gives the performance data SHIM generates high levels of correlation to unobserved execution [Yang et al. 2015].

SHIM makes use of software tags, which can be thought of as a high frequency signal read by SHIM and created by the profiled application. SHIM borrows from instrumentation based techniques and requires the developer to attach the creation of this signal to events they are interested in attributing measured performance to. The work in this thesis will use the tag idea presented in the original SHIM paper, which is to instrument method prologues and epilogues to attribute performance to methods. This is by no means the only tag that may be created, and it depends entirely on the needs of the developer.

2.4 Hardware Performance Counters

The primary performance indicators collected by SHIM are micro-architectural hardware performance counters. These counters give insight into the operation of the CPU at the hardware level. They are often used to confirm the effectiveness of native code optimisations, as optimisations at this level are designed with the purpose of improving this low level performance. There is a wide range of performance counters exposed by modern CPUs. Some examples include the number of instructions execute, the number of cache misses and the amount of memory bandwidth being used [Intel 2013].

In this thesis, we will demonstrate the use of these counters to evaluate JIT compilers within managed runtimes. Because JIT compilers compile down to native code, they can apply optimisation techniques much like native compilers. As such, it is important to their overall performance that the code they generate behaves in a manner that allows the CPU to operate with maximum efficiency. As many managed runtimes have many stages of JIT compilation, it can be hard to attribute performance information to a particular compiler implementation. This is because during the longer sampling period used by current profiling techniques, code generated by different compilers may be executing. The increased sampling frequency that SHIM brings, combined with the software tag method of attribution, allows much finer grained analysis.

2.5 Dynamic Call Graphs

A call graph is a directed graph that indicates calling relationships between functions. Each node in the graph represents a function, whilst the edges between functions indicate that one function calls another. Edges can be weighted in various ways, the two most prevalent are by the number of times one function calls another, or by the amount of execution time spent in one function called from another. These are referred to as frequency weighted and time weighted respectively.

A *dynamic* call graph (DCG) is a call graph that is generated through running the application for which it is to represent. This is opposed to static call graphs which generate graphs solely through inspection and analysis of the source code. DCGs can be considered a specific variety of information captured through dynamic profiling, and as such are created through the same profiling methods.

Generating a true dynamic call graph, in which edges are weighted with the true value of their occurrence, is exceptionally expensive, either in time, memory or both, as a record outlining the caller and callee must be created and catalogued appropriately for every function call. Such overhead is tolerable for offline analysis, and is thus primarily used in simulation or instrumentation based profiling techniques.

The use of DCGs is varied, they can be investigated to reveal structural program behaviour, such as the time spent in various calling contexts or analyse where a program spends the majority of its time. They can also be useful in program maintenance, to lay bare the workings of a program without requiring in-depth examination of the source code.

A key usage of frequency weighted DCGs is to inform inlining decisions within managed runtimes. Inlining is considered to be a critical performance optimisation within managed languages. It can significantly reduce the overhead associated with the heavy decompositional style favoured by programs written in object-orientated languages [Holzle and Ungar 1994]. It does this by eliminating the overhead associated with performing function calls, such as the creation of a new stack frame. Performing inlining compilation is however an expensive operation, especially in managed runtimes where the JIT must contend with the executing program for resources.

Frequency weighted DCGs are thus used to decide which inlining actions are expected to have a net positive result on performance, this is generally dependent on how frequently a particular call relationship occurs. Because the DCGs that inform these decisions must be generated at runtime, significant emphasis is placed on the performance of the profiling technique used to create them. Typically, this means that the only appropriate technique is that of interrupt based sampling.

DCGs generated through interrupt mechanisms are subject to the same limitations as other forms of performance data captured in this method. Within the domain of

DCGs, poor temporal resolution manifests itself as a reduction in accuracy. The magnitude of the benefits which frequency weighted DCGs provide to adaptive inlining actions is heavily dependant on their accuracy, which is a measure of how closely they represent the actual call relationships. An inaccurate DCG may lead the compiler to poorly select inlining candidates, and thus fail to realise any overall performance improvement.

2.5.1 Burst Sampling

Because of the reliance on their accuracy, but also the strict overhead limitations associated with their creation, there exist many attempts to improve the fundamental sampling technique used to created frequency weighted DCGs.

The most successful of these is burst sampling, which serves to remove a major source of inaccuracy present in frequency weighted DCGs generated through regular spaced, interrupt based samples. When sampling with single, approximately evenly spaced samples the data collected about calling relationships is representative of the *time* spent in a particular relationship, rather than the *frequency* of that occurrence. This leads to methods with short execution times, but high occurrences, to be under-represented in the created DCG. They are thus less likely to be selected as inlining candidates, despite benefiting from inlining actions the most.

Arnold and Grove [2005] present what can be considered to be the seminal work in burst sampling for the generation of frequency weighted DCGs. They adjust interrupt based techniques to capture a series of method calls with each sample, the eponymous burst, instead of a single call relationship. In doing so the DCG generated is less directly tied to the execution time of a method relationship, and better represents the frequency of that relationship.

Significant amounts of work builds on that performed by Arnold and Grove [2005], with various sampling techniques demonstrating benefits from incorporating concepts associated with burst sampling [Zhuang and Serrano 2006]. Each of these is however still constrained by their fundamental reliance on interrupt based mechanisms.

2.6 Summary

In this chapter, we have given an overview of dynamic profiling, outlined the existing sampling techniques used in its practice, and explored the some of the information that it is typically employed to generate. Each of the profiling techniques explored has some fundamental limitation associated with it. Whether it be the extreme overheads that simulation imposes, the un-attractive frequency to overhead relationship inherent to the interrupt mechanism, or the significant complexity of dynamic instrumentation.

We introduced SHIM, a profiling technique used to remove the limitations present in existing techniques through leveraging the multiple hardware threads commonly present in the environments programs execute in today. In doing this, SHIM enables high frequency, low overhead profiling of applications. Finally, we explored the types of data we will be generating with SHIM throughout this thesis and their importance to both the development and optimisation of managed runtimes.

Chapter 3

High Frequency Hardware Performance Counter Capture and Attribution

In this chapter we will demonstrate how SHIM can be integrated into SpiderMonkey to enable hardware performance counter collection and attribution to JavaScript level code. We show the overheads associated with differing sampling frequencies with this method across the Octane benchmarks. We also demonstrate how through extending the basic concept of a single software tag to that of an external tag stack structure, the current JavaScript calling context can be made easily visible to the observing SHIM thread. We show that this additional contextual information comes at little overhead, and enables increased specificity when attributing performance counters.

3.1 Introduction to JavaScript & SpiderMonkey

JavaScripts usage in the web is ubiquitous. Its ability to underpin both server and client side applications has lead to some dubbing it the assembly of the web[Hanselman 2013]. JavaScript is a fundamental component in the HTML5 stack, which enables everything from basic webpage interactivity, to the ability to run large server applications such as the entirety of PayPal's public facing web applications [Harrel 2013].

SpiderMonkey is Mozillas JavaScript runtime, which while forming part of the Firefox browser, is also a standalone runtime used to power server side applications such as CouchDB and node.JS [Mozilla 2015]. SpiderMonkey features numerous optimisations, including garbage collection and a multi-stage JIT compiler. The multi-stage JIT comprises of an initial interpretation stage, followed by the Baseline compiler, which performs a direct translation from JavaScript to machine code. Finally, the optimised compiler named IonMonkey uses both dynamic and static analysis techniques on intermediary code representations before generating assembly code [Pierron 2014].

Despite the languages ubiquity, and its widespread runtime optimisations, JavaScripts performance still lags behind that of other managed languages [Zakas 2010]. The performance environments in which JavaScript code must run is also widely varied, from mobile phones to large scale servers. This places increased pressure on JavaScripts performance, and any ability to further analyse and optimise said performance will have far reaching consequences.

3.2 Testing Methodology & Benchmarks

The performance figures presented in this chapter are generated through use of the Octane benchmarks. The Octane benchmarks are a series of JavaScript benchmarks developed as part of the V8 project by Google [Google 2015]. The goal of the benchmarks are to replicate real-world scenarios for JavaScript. For example, one of the included benchmarks is PdfJS, which is based on the actual Pdf file renderer present in Firefox.

To gather performance information, each benchmark was run ten times in the altered SpiderMonkey and the average of all the results was recorded. This was then normalised against the average time taken for a production SpiderMonkey on the same benchmark across another ten runs. It should be noted that the Octane benchmarks deliver a score value, rather than reporting the time taken for the benchmark to complete. This score is the quotient of some selected, constant time value, and the actual execution value. Because of this, despite reporting a unitless score, the quotient of two Octane scores is still representative of execution time overhead.

All tests were performed on desktop computer running Ubuntu 14.04 with the modified SHIM Linux kernel, based on kernel 3.17.0. The processor used was an Intel Core i7-2600K running at 3.4GHz. This processor has 4 cores and 8 hardware threads, and comes equipped with 1 and 8 MB of level 2 and 3 cache respectively. 4GB of DDR3 RAM operating at 1333MHz was also used.

Unfortunately, SpiderMonkey offers little capability to lock down the execution environment to improve the repeatability of tests. As such, variations of up to 10% in terms of execution time can be present across invocations of the same benchmark, even with the production implementation. Some of this can be attributed to the slight non-determinism present within the JIT compilation and garbage collection systems.

We will be evaluating the performance of SHIM at three different sampling periods: As small as possible, approximately 200,000 cycles, and one millisecond. The first of these is an interesting data point to understand the limits of the system. The second is for comparison with existing interrupt based techniques, since 200,000 cycles is approximately the minimum sampling period achievable with interrupt mechanisms, allowing a more direct comparison. The third is primarily used to understand where the incurred overheads drop to a marginal percentage. It should be noted that whilst all figures have precise sampling period labelling, all periods are only an approximation.

No data points exist between those at the minimum sampling period and 200,000 cycles, because modulating the sampling speed to fall in this range requires the use of busy waiting. This is required because the minimum period for which a thread can sleep is determined by the maximum interrupt frequency, as timer interrupts are used

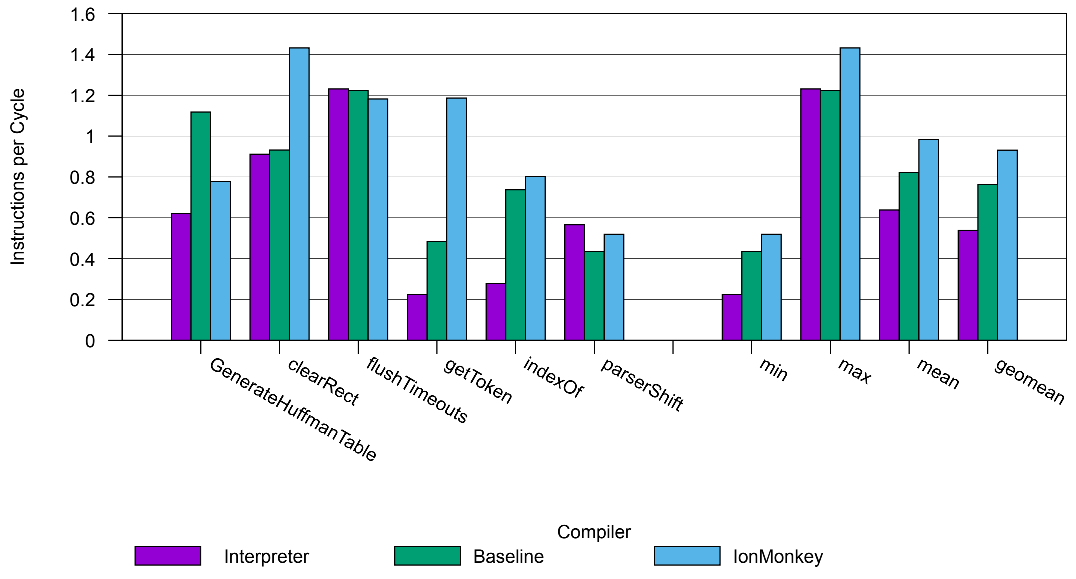

Figure 3.1: IPC across different compiler implementations of a selection of JavaScript methods from the PdfJS benchmark

to schedule threads. Because we are primarily interested in the overheads incurred, and because to measure hardware counters SHIM must execute on the same core, busy waiting serves to simply reduce temporal resolution with no appreciable difference in overhead.

3.3 Single Tag Attribution

This section will look at a direct application of the style of performance profiling outlined in the original SHIM paper to SpiderMonkey. In this implementation, a single software tag comprising of a function identifier and compiler information is generated by the profiled program in every JavaScript function prologue and epilogue. This is read along with enough hardware performance information to calculate the instructions per cycle (IPC) of the JavaScript execution thread.

Figure 3.1 demonstrates the kind of data that can be captured with this rudimentary marriage of SHIM and SpiderMonkey. This figure shows how the measured IPC differs across different compiler implementations of the same JavaScript function, across a run of the PdfJS benchmark. We can see that in general, the higher optimising compilers within SpiderMonkey generate code that has a higher IPC, which is constant with expectations since higher IPC indicates better quality code generation. With existing techniques, this kind of information would be muddied in both temporal resolution and structural specificity; deeply specific attribution such as this is simply infeasible.

3.3.1 Implementation

The implementation in this section closely follows that outlined by Yang, Blackburn, and McKinley [2015] and recounted in Section 2.3. The SHIM source code and modified kernel are used to enable access to the process ID map. The SHIM library is invoked with the desired performance counters, in this case we collect instructions retired for the whole core, instructions retired for the current thread, and the number of CPU cycles. The current thread in the context of reading the performance counters is the SHIM observer thread, and as thus the number of instructions executed by the target thread can be determined through the difference of these two counts. Since cycle counts remain constant across the core, only one cycle value for this need be collected.

SpiderMonkey is altered such that prior to the commencement of interpreting the JavaScript code, it begins and detaches a SHIM observer thread. This thread performs three tasks continuously, reading the performance counters, reading the software tag, and then recording both. The SHIM observer thread, and the main SpiderMonkey execution thread, are then pinned to different hardware threads on the same core. Although SpiderMonkey enables multi-threaded compilation of methods, the primary execution logic, and the target of our profiling, remains single threaded.

To generate the software tags, we instrument the three different compiler stages of SpiderMonkey such that they insert a single move instruction into the prologue and epilogue of each JavaScript function they compile. This move instruction stores a 32 bit integer to a memory location shared with the SHIM observer thread. Into these 32 bits, we pack information used to identify which JavaScript function stored the tag, which compiler was used to generate the currently executing code, and which optimisation level the compiler was running at when it generated the code. We also include a flag to identify whether the tag was stored from a function prologue or epilogue.

3.3.2 Overheads Incurred

Figure 3.2 shows the overheads incurred for generating and attributing IPC information with SHIM in SpiderMonkey across the Octane benchmarks. On average, the minimum attainable sampling period is 200 cycles. It is important to that this number is an approximate average across all recorded samples. Overhead seen here approximately mirrors that seen in the original SHIM paper with regards to sampling frequency against overhead.

The overhead at high sampling frequencies is considerable, and is primarily through contention for resources within the core. For SHIM to read the retired instructions it is required to be executing on the same core as the target program, these counters are core local and as such cannot be read from a different core.

Decreasing the sampling frequency to the maximum interrupt frequency drastically reduces the overhead, whilst a one millisecond sampling period eliminates it entirely. For reference, a no sampling variation is included to measure the overhead of the compiler instrumentation, which is found to be negligible as expected. Each

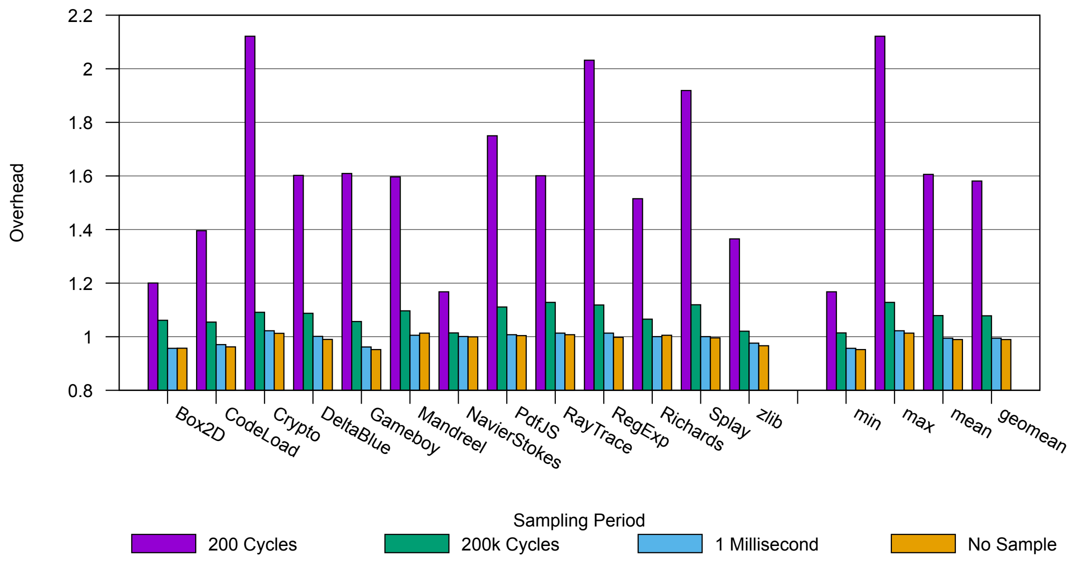

Figure 3.2: Overhead of generating and attributing IPC values across Octane benchmarks

prologue and epilogue requires only one instruction, and across the octane benchmark the number instrumented is in the order of 10^6. At a CPU frequency in the order of 10^9Hz this only imparts a penalty in the order of 10^{-2} seconds. Across a benchmark with an execution time in the order of 1 second this is simply not visible.

At the minimum sampling period of 200 cycles, the average overhead induced by observing with SHIM is approximately 40%. Whilst this is certainly too high for any kind of online analysis, it is similar to existing interrupt based techniques [Weaver 2015], but with several orders of magnitude increase in sampling frequency. Through offloading the sampling task, SHIM has enabled a significantly higher sampling frequency without the overhead increase associated with existing techniques.

3.4 Contextual Information

The performance properties of an executing function are heavily influenced by the context in, and the parameters with, which they are called. By virtue of these differences, different function executions may exhibit different performance characteristics. As statistics are collated across all calling contexts, the method used in the previous section which relies on a single method identifying tag, does not grant the ability to determine the context in which a function is executing; all contexts are blurred together.

Having an understanding of the user level context in which a method was called allows differentiation between method invocations with different input parameters or program state. This understanding provides for a deeper analysis of potential causes of observed behaviour.

In this section, we will demonstrate how through extending the idea of a singular

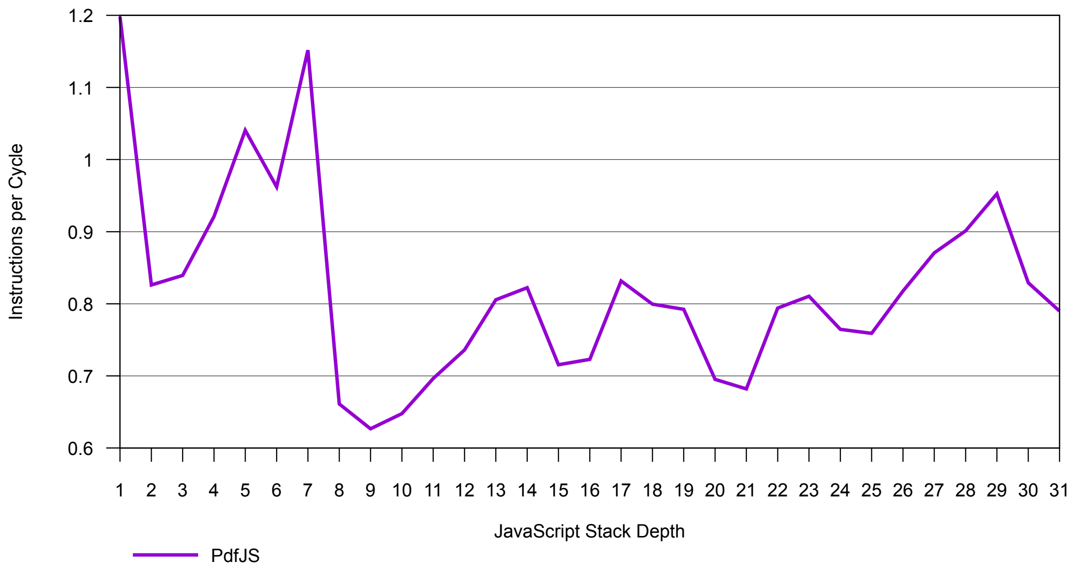

Figure 3.3: IPC recorded at different stack depths during execution of PdfJS

software tag to the maintenance of a separate stack data structure, SHIM can be extended to capture JavaScript calling context to enable differentiation between contexts as well as provide information about the state of the JavaScript stack.

In practice, this allows the generation of information such as that presented in Figures 3.3 and 3.4 in addition to the information in the previous section. This information reveals some interesting behaviour, despite touting an aggressive optimisation strategy, execution time across stack depths is dominated by functions compiled with the Baseline compiler or simply interpreted. We can also see generally poorer IPC performance as the stack depth increases and becomes predominantly Baseline compiled methods, perhaps indicative of poor Baseline compiler performance.

3.4.1 External Stack Structure

The collection of context information can be achieved by either observing the state of the program stack, as it is done in interrupt driven profilers, or by creating a separate specialised data structure to capture this information. The latter must be realised through additional instrumentation of the runtime, as it relies on extra data being generated. The single tag mentioned in the previous section can be considered a rudimentary example of this.

Observing the program stack with a view to extract the current calling context requires an understanding of the exact layout of the stack. It also has the drawback of requiring the information captured in the software tag, which is generated at function compile time, to be made available at run time.

Creating a separate data structure has the primary advantage that is requires no such understanding of the stack layout. Instructions to build and maintain the sepa-

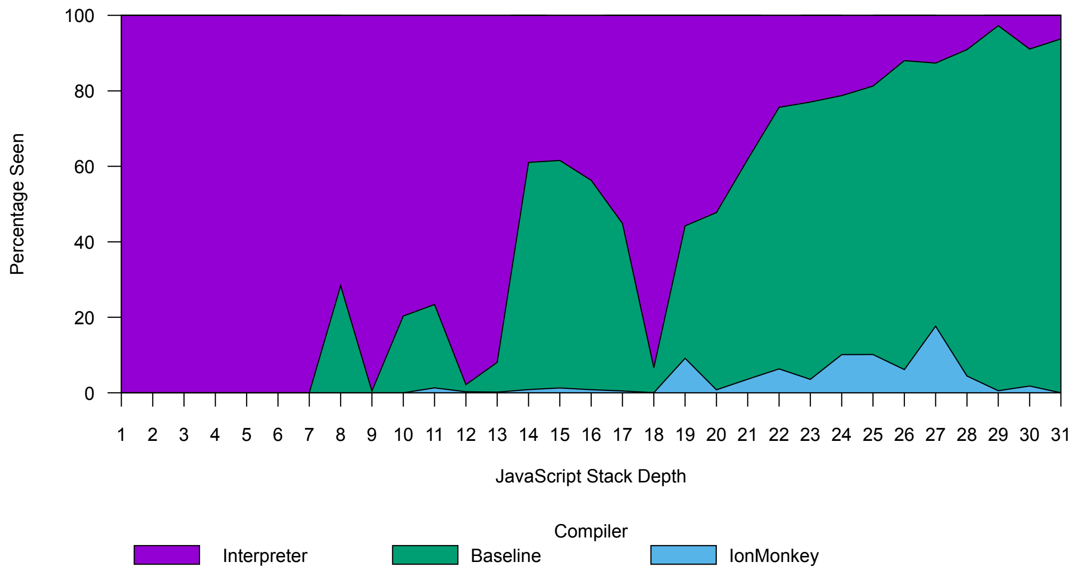

Figure 3.4: Percentage of compiler type present at different JavaScript stack depths of PdfJS

rate structure are simply inserted into the generated method prologues and epilogues.

This has the primary drawback of imposing an overhead on each generated function. In the previous section, the overhead of instrumenting each JavaScript code prologue and epilogue with a single move function instruction resulted in minimal overhead. As the data structure grows in complexity, so too will this overhead.

3.4.2 Implementation

With a goal of capturing calling context, an external stack structure was implemented in SpiderMonkey. In essence, rather than moving a single software tag value to a specified location, the tag would be pushed to this external stack in the prologue, and then popped from the external stack in the epilogue. This means that at any point in time, the external stack represented the current JavaScript calling context, inclusive of any other compile time information stored in the tag. Rather than reading a single tag, the observer thread can walk the stack and record the level of calling context desired.

The management of the external stack construct occurs in instrumentation inserted into the prologue and epilogue of each JavaScript function (along with the interpreter). Pushing to the stack requires a handful of instructions, inflated by the fact that a register must be used for an indirect load to determine the location of the top of the stack. Popping from the stack can in theory be achieved with a single subtraction instruction, simply moving the pointer to the available slot downwards.

Unfortunately, owing to the bailout mechanism present within SpiderMonkey, it is possible to reach the epilogue of a Baseline compiler compiled method without having gone through the corresponding prologue. This occurs because IonMonkey aggressively assumes type information when compiling. If a function is called with different type information, the optimised code must be discarded and execution must resume in Baseline compiled code. If this occurs after functions have been inlined by the optimised compiler, then the prologues for these methods will not have been called. Yet because execution has resumed in Baseline code, which does not perform any inlining, their epilogues will be invoked even if their prologues were not.

This creates a situation where invalid pops are attempted on the stack; to maintain stack validity each method should pop only that which it pushed. Combating this requires instrumenting extra code within the epilogue of Baseline compiler compiled methods to check that the JavaScript function identifier of the function attempting the pop matches that of the tag on the top of the stack. If they do not, the pop attempt is ignored.

Whilst this requires more instrumentation in the epilogue in Baseline compiler compiled methods, in practice this has little impact on performance. This is for two reasons, firstly, the Baseline compiler is generally only used to quickly determine type information before the Optimised compiler takes over, thus methods are generally not executing in Baseline generated code [Pierron 2014]. Secondly, across the Octane benchmark suite bailouts are fairly uncommon. Generally only a handful occur during a single benchmark run, allowing branch prediction to work effectively as the secondary path is taken extremely rarely.

3.4.3 Concurrency Considerations

Despite utilising the concept of producer and consumer threads, there are no formal concurrency mechanisms accompanying this implementation. Because of this, it is possible that the stack will change as the SHIM thread is reading it, leading to the generation of invalid information. Invalid information in this setting refers to the reading of a calling context that does not exist, or does not occur in any temporal proximity to when the read occurred. Due to the statistical nature of the data generated, reading an old, but recent, context is acceptable.

Reading an invalid context requires that between the reading of two stack elements, before the second element is read, at least both are popped and at least one new element is pushed. For example, consider a stack, from top to bottom, consisting of the software tags A B C. As the stack is read downwards, assume the following interleaving of actions:

1. Read A
2. Pop A
3. Pop B
4. Push E

The stack is now Empty E C, the SHIM thread then reads the next element down the stack which is now E. This generates the calling context whereby A is called from E, when in reality E has not and may never call A. Alternatively, E may call A in some completely different context, in which case performance attributes will be incorrectly attributed to that context. Obviously the injection of more stack operations between each read exacerbates this issue.

Because values of the buffer are read and copied, with no other analysis in between reads, and as stack operations are tied to the execution of entire JavaScript methods, thus reducing the rate at which they may occur, we assume that the impact of this to be low.

In practice, this effect has little to no bearing on the overall validity of the contexts read. By comparing the complete set of calling contexts with those generated by this method, Sub-Section 4.2.5 of this thesis finds that reads generating fictitious contexts account for less than 0.1% of all reads across all of the Octane benchmarks.

3.4.4 Overheads Incurred

In this section we show overheads incurred for generating various levels of JavaScript calling contexts in addition to recording enough hardware counter information to discern the IPC of the primary JavaScript execution thread.

Figure 3.5 displays overheads for simply capturing the currently executing JavaScript method, alongside its caller. This is achieved by sampling only the top two elements of the external stack. We can see that overheads are very similar to those generated through usage of a single software tag, and that the additional instrumentation required to maintain the stack structure has no appreciable impact.

Figures 3.6 and 3.7 show the overheads for a context of 10 function calls and the full JavaScript calling context respectively. Overheads are again extremely similar to those of the single software tag, the key difference is in the different minimum attainable sampling period. Because at each sample additional information must be collected, the frequency with which is can be collected decreases.

Overall, we can see that providing large amounts of calling context information along side captured performance counters does not result in an appreciable increase in overhead. It does however limit the maximum sampling frequency, as more data must be collected with each sample. Depending on the behaviour attempting to be observed, a balance between these two can be achieved.

3.5 Summary

In this chapter we have demonstrated how SHIM can be incorporated into SpiderMonkey to allow for the high frequency capture of hardware performance counters, and their attribution to user level JavaScript code. We then extended the basic SHIM software tag to a tag stack structure, enabling detailed calling context information to also be captured, finding that doing so required no measurable increase in overhead. We offered some examples of how this novel insight could be used to target performance issues within SpiderMonkey.

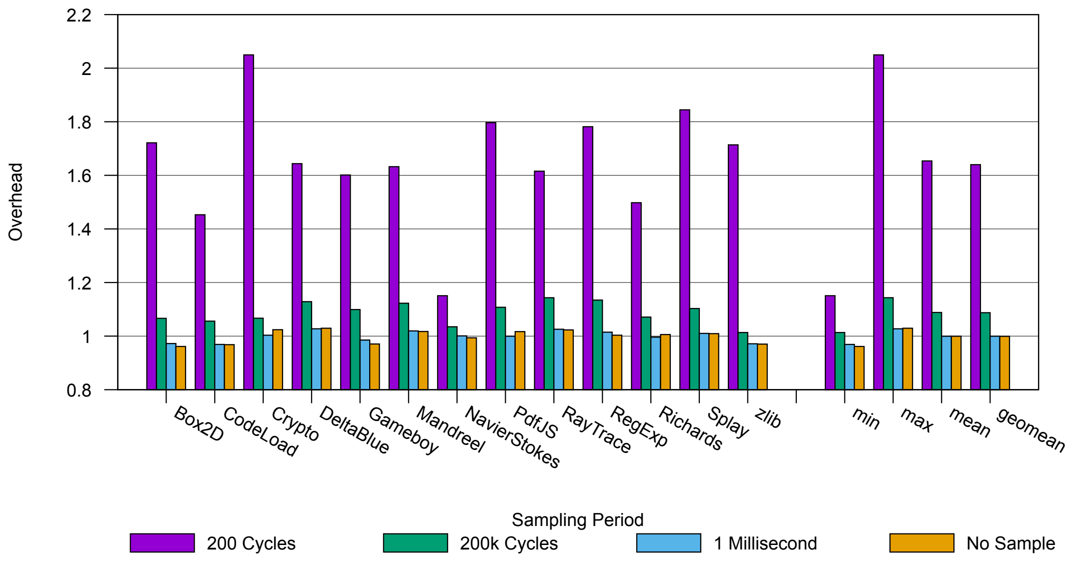

Figure 3.5: Overhead of capturing single layer JavaScript calling context at various frequencies

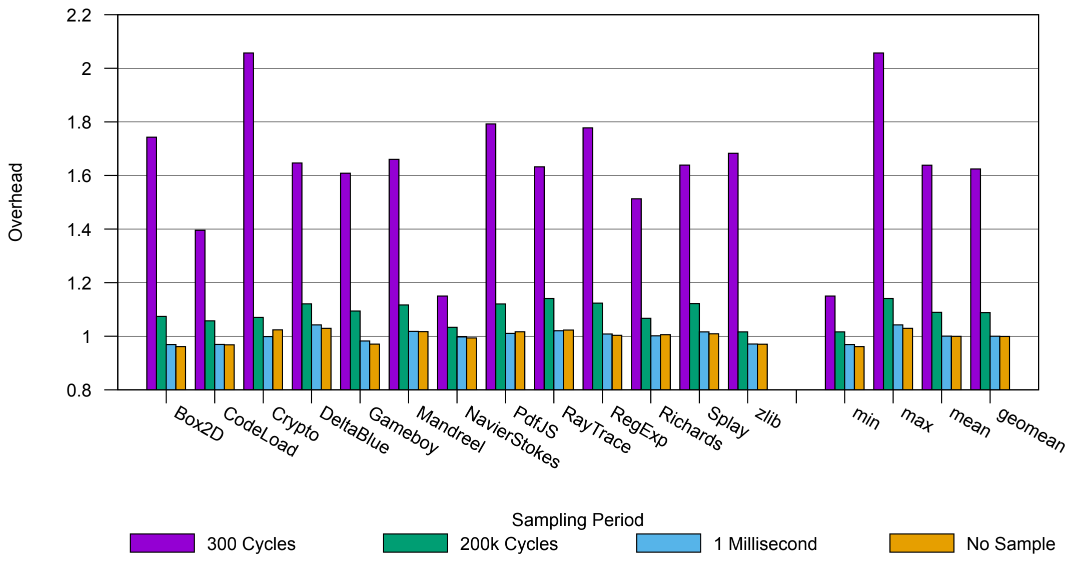

Figure 3.6: Overhead of capturing 10 layer JavaScript calling context with each sample

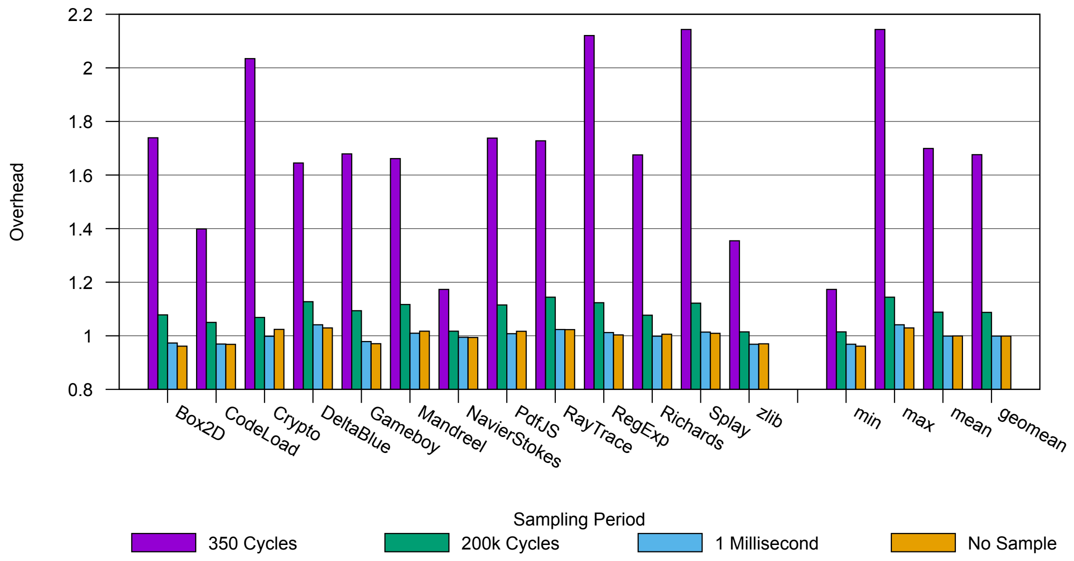

Figure 3.7: Overhead of capturing full JavaScript calling context with each sample

Level of Context	Sampling Period	Average Overhead
Caller / Callee	200 Cycles	65.4%
	200k Cycles	8.8%
	1 Millisecond	0.0%
10 Deep	300 Cycles	63.8%
	200k Cycles	8.9%
	1 Millisecond	0.0%
Full Depth	350 Cycles	69.9%
	200k Cycles	8.8%
	1 Millisecond	0.0%
No Sample	N/A	0.0%

Table 3.1: Average overhead for various context levels and sample periods across the Octane benchmarks

Chapter 4

High Frequency Call Graph Generation

4.1 Introduction

A dynamic call graph (DCG) is a directed graph that captures calling relationships between functions. Each node in the graph represents a function, whilst edges represent that one function calls another, with the direction describing the caller and callee relationship. In this chapter, we will look at the generation of frequency weighted DCGs, in which edges are weighted according to the frequency with which the call relationship occurs. This is in contrast to time weighted DCGs, where edges are weighted according the amount of execution time spent in a particular relationship.

Frequency weighted DCGs are commonly used to inform inlining decisions performed by JIT compilers in managed runtimes. In this environment, they are created through the use of interrupt based profiling, which as discussed in Chapter 2, offers only limited sampling frequency.

Just as SHIMs usage of multiple hardware threads enables high frequency, low overhead capture of hardware performance counters, so too can it be used to drive the sampling process utilised to record calling contexts and build DCGs.

This chapter will explore the generation of frequency weighted dynamic call graphs through several SHIM based techniques. We will first look at a simple extension of the tag stack structure used in the previous chapter. Following on, we implement a direct observation technique, to extract the calling context directly from the executing applications stack. Finally, we incorporate burst sampling concepts with a ring buffer implementation, enabling high accuracy call graph generation at overheads conducive to adaptive optimisations. We perform an analysis of the accuracy of the DCGs generated through these various methods, alongside their respective incurred overheads, and offer a comparison against existing techniques.

The usefulness of a call graph is almost exclusively determined by its accuracy. Likewise, the cost of it can be captured by its overhead. Thus, throughout this chapter we will consider three classes of sampling period, from which one data point will be taken. The first, is simply as small as possible, ”Minimum with SHIM”, utilised to give an understanding of the maximums of the implementation. The second, will be

"Minimum with Interrupts", which will sample at the maximum allowable interrupt frequency to generate the smallest possible sampling period in this manner. This will be achieved by calling sleep() with the minimum time period. This allows for comparison with the maximum frequency of traditional interrupt based sampling methods. Lastly, "1 Millisecond", a self-explanatory sampling period, will be used primarily as comparison for existing techniques which perform online analysis, as they generally sample in this range [Arnold and Grove 2005].

4.1.1 Call Graph Accuracy

The accuracy of a DCG can be determined by the degree to which it represents that of the complete DCG. A complete DCG contains all of the relationships that exist within a program, weighted at the amount to which they actually occur during execution. We can capture this degree of representation by looking at the amount of shared data between the DCG generated through sampling, and the complete DCG. This shared data is a measure of how much information is present within both graphs.

To formalise this, we use the definition of accuracy from Arnold and Grove [2005].

$$Accuracy(DCG_{sampled}) = Overlap(DCG_{sampled}, DCG_{complete})$$

Where the overlap is a measure of the shared information between two DCGs.

$$Overlap(DCG_1, DCG_2) = \sum_{e \in CallEdges} min(weight(e, DCG_1), weight(e, DCG_2))$$

Where $CallEdges = DCG_1 \cap DCG_2$, and $weight(e, DCG_x)$ is the percentage of samples associated with edge e in DCG_x

Comparing weighted overlap, rather than raw sample overlap, is key to obtaining a proper gauge of DCG usefulness. This is because we are generally more interested in the relationship *between* call edges, rather than their absolute values. Consider the case of adaptive inlining, it is less important to know exactly how many invocations a particular function relationship has, rather than to know one function relationship is dominating over all others. Should raw values be desired, a simple scaling factor based on sampling period can be applied to create a good approximation.

Conveniently, this method for calculating accuracy always generates a figure between 0 and 100 percent, with 100 percent indicating that the graphs contain exactly the same weighted information, and 0 percent indicating no mutual information whatsoever.

4.1.2 Requirements of Frequency Weighted Call Graphs

Simply sampling and recording all of the calling contexts seen creates a time weighted, rather than frequency weighted DCG. As we are interested in the latter, we make a slight adjustment to our sampling strategy. We incorporate the idea of having seen a call, such that if we sample the same calling context twice in succession, we have a way of determining whether this is a newly created context, that happens to have the same caller and callee, or the same context seen prior. In doing this, we reduce the impact of over attribution to call relationships with long execution time, but low numbers of occurrences. Each of the various implementations explored in this chapter have their own way of performing this differentiation, detailed in their respective implementation sections.

4.2 External Stack

In the previous chapter, we look at how by maintaining a separate stack structure of software tag information, the current JavaScript calling context could easily be discerned at any time. This information was used to differentiate between calls to a single function from different contexts, allowing more granular attribution of micro-architectural performance counters.

In this section, we will demonstrate how this structure can straightforwardly be used to generate accurate, frequency weighted call graph information within Spider-Monkey. Firstly, we will look at the implementation details surrounding this, followed by how the complete JavaScript call graph can be generated, whilst finally the accuracy and overhead of varying sample speeds is presented.

The primary advantage that the maintenance of a separate stack structure has over other popular methods of generating context, such as gleaning it from the program stack, is that it requires no knowledge of the layout of the program stack. This is particularly useful within SpiderMonkey, as the implementation of the JavaScript stack is quite removed from that of the native one, making its navigation complex. A separate stack does however introduce an instrumentation overhead, as it must be maintained in addition to the normal execution stack.

4.2.1 Implementation Details

In Section 3.4, we outlined the implementation of a tag stack structure which could be read at any point in time to discern the entire JavaScript calling context. To generate DCGs within SpiderMonkey, we will utilise the same basic implementation with some minor changes.

Firstly, we remove any compiler information present within the stored tag as we are only interested in the function identifiers. Secondly, to focus on the creation of frequency weighted DCGs as per the requirements in Sub-Section 4.1.2, we specify a flag bit within the stored tag to indicate whether it has changed since last being read. When the stack is pushed to, this flag is set to indicate that the pushed tag is new.

At each sample, the SHIM observer thread walks the stack from top to bottom, inspecting each element to determine whether it has changed. If it has, then a call relationship is recorded comprising of the function whos tag is currently being inspected, and that of the tag below the current one. The flag is then set on the current element of the stack to indicate it has been read. This procedure continues until the observer thread arrives at a stack element which it determines has not changed. In this situation, due to the nature of the stack, the observer thread can be sure that no elements lower on the stack have changed either, and thus the sample is concluded.

Lastly, because we are no longer concerned with capturing core-local performance data, the SHIM observer thread is relieved of its requirement to execute on the same core as the primary JavaScript thread. To achieve maximum performance, we pin each thread to different cores to ensure they are not scheduled onto the same one.

4.2.2 Creating the Complete Call Graph

To create the complete DCG with which to derive the accuracy of the generated DCG, we make a slight modification to the instrumentation. We continue to push a tag to the stack in each function prologue, however in the epilogue, rather than performing a pop, we push the same identifier with a flag indicating it is an epilogue. In performing this, we change the stack to a log of function prologues and epilogues. By walking this log, we can see every call that occurred during program execution. This obviously has a massive memory impact, but in practice with 4GB of memory it is manageable across the benchmarks used.

4.2.3 Testing Methodology

The testing methodology and hardware remains the same as that outlined in Section 3.2. In brief we use the Octane JavaScript benchmarks, running each benchmark ten times and taking the average value for overhead and accuracy across all runs. We normalise overhead to the execution time of a production SpiderMonkey implementation, whilst accuracy is determined by the overlap metric described in Sub-Section 4.1.1.

4.2.4 Accuracy and Overheads

In this section we present the resultant accuracy and overhead figures for the three sampling periods outlined in Section 4.1, across various levels of stack inspection. We modulate the maximum depth that the SHIM observer thread will read to, to observe if this has any appreciable impact on the accuracy of the generated DCG. Figures 4.1, 4.1 and 4.1 show the overheads for inspecting the top two elements, ten elements, and the full stack respectively. We find that the incurred overheads across stack inspection depths to be very similar, and thus only graph that of the full inspection in Figure 4.4, whilst we tabulate all overall results in Table 4.1.

The most striking component of these results is the vast increases in accuracy that accompany reduced sampling periods. In comparison to the fastest sample frequency attainable through the interrupt mechanism, sampling at frequencies enabled by SHIM results in an improvement in accuracy of approximately 40 percentage points. Yet, because we can perform the sampling without causing a major disruption to program execution, the overheads incurred, even at maximum sampling frequency, are on average roughly 6%. It should be noted that this massive reduction in overhead

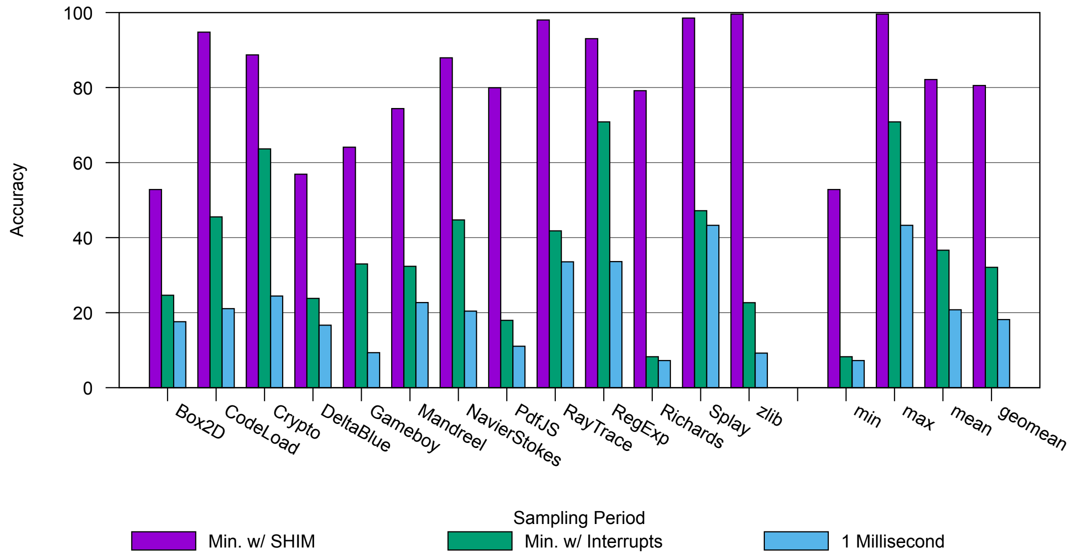

Figure 4.1: Accuracy of Dynamic Call Graph generated through inspecting only the top two elements of the external stack

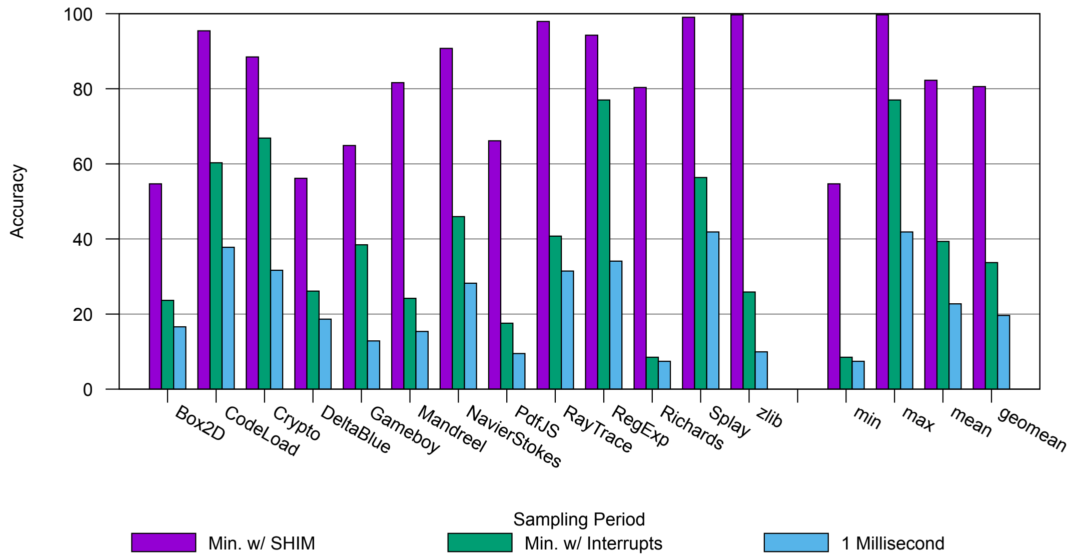

Figure 4.2: Accuracy of Dynamic Call Graph generated through inspecting only the top 10 elements of the external stack

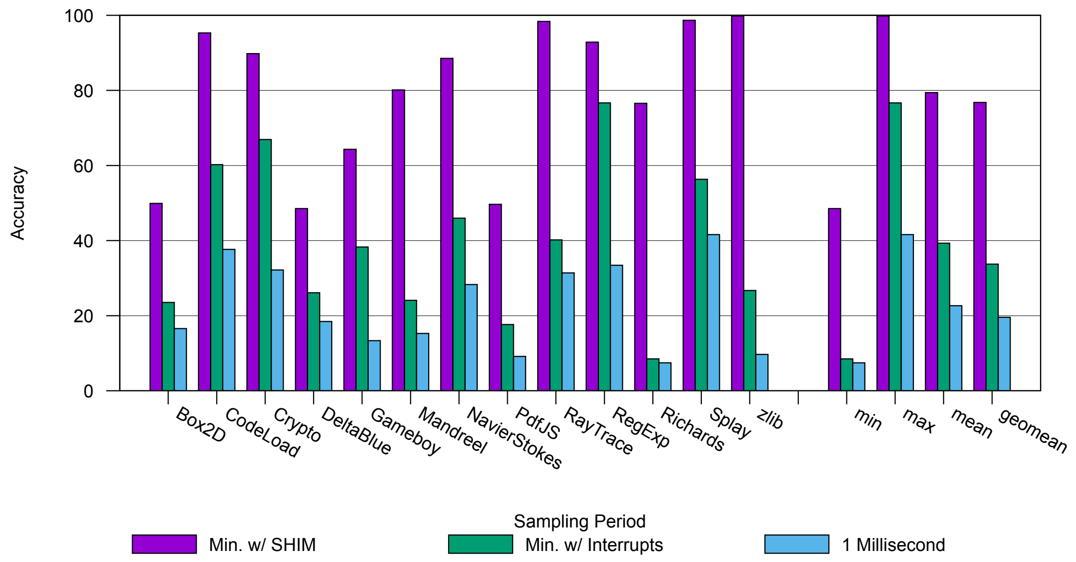

Figure 4.3: Accuracy of Dynamic Call Graph generated through inspecting all of the elements in the external stack

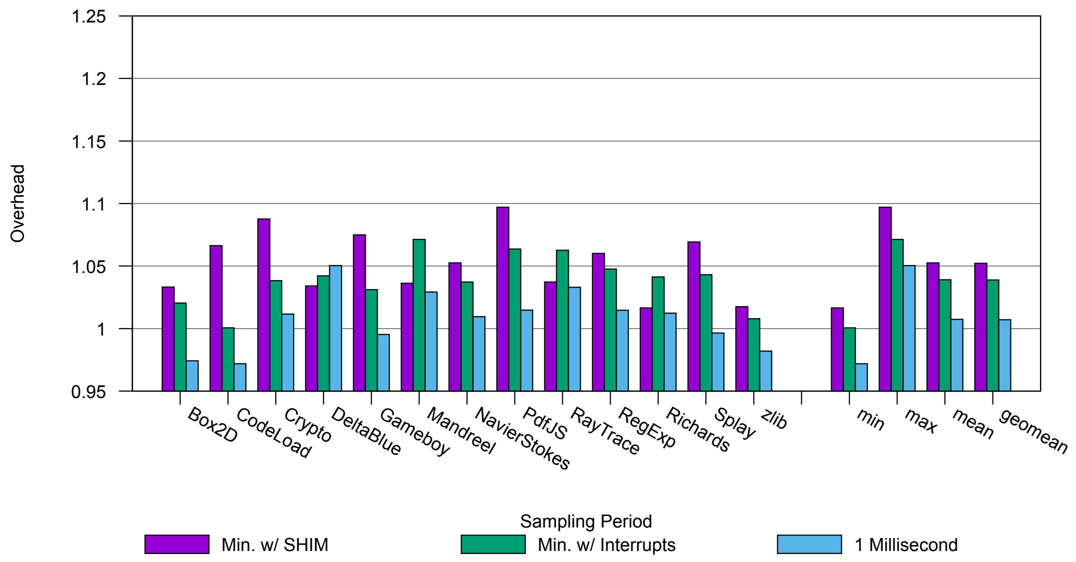

Figure 4.4: Overhead of generating a dynamic call graph by sampling the entire stack structure

Inspection Depth	Sampling Period	Average Accuracy	Average Overhead
2 Deep	Min. with SHIM	82.15%	7.0%
	Min. with Interrupts	36.65%	2.4%
	1 Millisecond	20.78%	0.2%
10 Deep	Min. with SHIM	82.27%	5.7%
	Min. with Interrupts	39.35%	4.2%
	1 Millisecond	22.72%	0.1%
Full Depth	Min. with SHIM	79.42%	5.3%
	Min. with Interrupts	39.33%	3.9%
	1 Millisecond	22.66%	0.1%

Table 4.1: Average overhead for various context levels and sample periods across the Octane benchmarks

in comparison to that incurred in the previous chapter is a property of removing the requirement to execute on the same core.

We can see that there is little accuracy difference across the varying inspection levels at the maximum sampling frequency. This is primarily because even though high inspection levels can sample deeper into the stack, they still stop when then encounter an unchanged element. Because the sampling frequency is so high, few elements on the stack change in-between samples. Thus at high sampling frequencies, varying inspection depths are primarily recording the same data. As the sample rate slows, we can see that the accuracy of inspecting only the top two elements decreases in comparison to the others. As the stack will have changed much more during the longer period, the higher inspection depths will kick in.

Overheads also remain fairly constant independent of inspection depth. For the reason outlined above, at high sampling frequencies different stack inspection depths are generally performing the same work, resulting in similar performance penalties. As the sample rate slows, the differences in overhead grow, however only marginally. We also note that in keeping with the results in Section 3.4, the performance penalty of the instrumentation is unnoticeable.

4.2.5 Concurrency Issues

In Sub-Section 3.4.3 we discussed the concurrency issues in the implementation of the external stack, namely that because there are no mutual exclusion mechanisms in place it is possible that invalid information may be read. This invalid information manifests itself as incorrect calling context information. In the previous chapter, this was simply accepted and assumed to be of low impact as it required a specific interleaving of reads and writes to occur.

Given that the complete graph contains all valid calling contexts, we can garner some information about the impact of this issue. If a call relationship exists in the data gathered from sampling the stack, but does not exist in the complete graph, it can be assumed to have been created by an invalid read. In practice, across the Octane benchmarks, the weight of such edges represents less that 0.1% of the total edge weights. This holds across all sampling frequencies and stack depths.

The result that it holds across frequencies and stack depths is unsurprising. Invalid data is generated during the reading of the stack, which, as frequency is determined by the delay between entire sets reads, takes the same amount of time regardless of the sampling frequency. Holding across different stack depths can be attributed to the fact that invalid reads are most likely to occur at the top of the stack, where the structure is changing most rapidly. All stack depths read at least the top two elements of the stack, and are thus all subject to the majority of the inconsistency.

This method of error detection will only allow for detection of invalid reads that produce relationships outside of the set of valid relationships. It is imaginable that through chance, an invalid read produces a relationship which does fall within this set. We postulate that as the recorded occurrences of reads outside the valid set is so low, even an order of magnitude larger comparative number of undetectable invalid reads would not have a significant impact on overall accuracy.

4.2.6 Summary

In this section we have shown how the external tag stack presented in the last chapter can be modified to generate frequency weighted DCGs through high frequency sampling with SHIM. This technique creates DCGs with good levels of accuracy, but more importantly highlights the extreme advantage that SHIM provides in this domain. In comparison to DCGs generated at sampling frequencies available to interrupt based techniques, those generated with frequencies afforded by SHIM present significantly higher accuracies. Because SHIM does not require suspending program execution in order to perform a sample, these large increases in accuracy do not come with equivalently large increases in overhead.

4.3 Direct Observation

At any point during program execution, the current calling context can be determined by careful inspection of the program stack layout. This method is used in conjunction with interrupts to capture the calling context in current dynamic profilers. During the inspection of the stack, as with all interrupt based techniques, execution of the profiled application is effectively suspended.

In this section, we will look at implementing a similarly inspired system with a separate SHIM observer thread, enabling inspection of the stack to occur without interrupting the executing program. We term this direct observation, as rather than using a software tag intermediary to communicate between the SHIM observer thread and the profiled application, the observer thread is able to simply directly inspect the profiled application.

In the previous section, we demonstrated how the creation and maintenance of a separate stack structure could also be used to communicate the calling context. We argued that such a structure is less complex to implement than direct observation, as it requires no understanding of the program stack layout. The direct observation method however does not require extensive instrumentation to be inserted into methods in order to maintain the separate data structure. This independence from the program stack was particularly useful in the context of SpiderMonkey, as its stack layout is not particularly conducive to extracting the relevant information.

For direct observation, we move to an environment more receptive to this approach, Jikes RVM. We first give an introduction to Jikes RVM, and look at the performance of its exiting approaches to generating DCGs. We then implement the direct observation technique and examine its performance comparative to these existing strategies. We find its performance to be sub-optimal, failing to outperform the current technique, and thus conclude with an analysis of why we believe this to be so.

4.3.1 Introduction to Jikes RVM

Jikes RVM is high performance, Java-in-Java research virtual machine. Its primary purpose it to serve as a test bed for novel virtual machine ideas [Alpern et al. 2005]. Owing to this, it contains a plethora of performance optimisations, including a multi-stage JIT and advanced garbage collection techniques.

Of particular interest to the work in this thesis is the usage of dynamic profiling within the RVM. To drive adaptive optimisation inlining decisions, Jikes RVM performs dynamic profiling of the currently executing program to build a frequency weighted DCG. Hot edges within this graph are then used to determine which calls are likely to yield a performance increase when inlined.

4.3.2 Dynamic Profiling in Jikes RVM

Whilst not explicitly an interrupt based technique, dynamic profiling within Jikes RVM utilises a mechanism with very similar properties. All sampling within Jikes RVM is performed through the use of yield points. A yield point is a section of code that can be taken during method prologues, epilogues and on loop backedges. They are not always taken, instead when an executing thread reaches one of these points its yield point flag is checked to determine whether is should take the yield point. Yield points are used for many different purposes within Jikes RVM, including thread scheduling and synchronisation.

To perform a sample, an external timer thread regularly sets the yield point flag on each running thread, such that they will execute the next available yield point and take a sample. This primarily differs from a purely interrupt based mechanism in the fact that rather than being immediately stopped, the thread runs until it reaches a location where it may take a yield point. This mechanism does not however address the sampling frequency limitations associated with interrupts, the executing thread must still suspend execution of program code, just at a more convenient opportunity.

Yield points also straightforwardly enable a sampling technique optimisation called burst sampling. As discussed in Chapter 1, burst sampling is a technique used to increase the accuracy of frequency weighted DCGs generated by dynamic profiling. Instead of taking approximately evenly spaced samples, burst sampling has periods where full profiling occurs in which all method calls are recorded. This allows for better representation of short execution time methods, which may be frequently called, but owing to their short duration, are underepresented within the graph.

By indicating to the sampled thread that rather than taking just one yield point, it should take the next several, Jikes RVM implements burst sampling. Arnold and Grove [2005] demonstrate that this leads to a significant improvement in DCG accuracy, and indeed their work is the basis for the system currently used within Jikes RVM.

Burst sampling is however primarily a response to the low sampling frequencies provided through interrupt like mechanisms. Because direct observation with SHIM is not limited to these frequencies, this chapter is built on the conjecture that single samples at much higher frequencies can outperform the existing low frequency burst sampling technique.

4.3.3 Testing Methodology and Benchmark

To evaluate the accuracy of our implementation, along with that of the existing one, we utilise the DaCapo benchmark suite [Blackburn et al.]. The DaCapo benchmarks are a series of client side Java benchmarks specifically designed to test the properties of Java runtimes. This is achieved through workloads targeted at stressing the complex interactions between runtime subsystems not present in native language implementations. Due to limitations in the Jikes RVM Java libraries, the 2006 version of the DaCapo benchmarks is used. All benchmarks are run on the large setting to give a good variation in execution times, from sub one second to over ten. We exclude the eclipse benchmark from testing as its size and complexity makes generating and validating its complete call graph prohibitively time consuming.

To improve testing repeatability, we make use of several mechanisms present within both DaCapo and Jikes RVM to increase determinism across benchmark runs. Firstly, we generate a set of compiler advice files for use with Jikes RVM. These files are a record of a set of compilation decisions made for a specific invocation of a benchmark. This record of decisions can be fed back into Jikes RVM for future benchmark runs to ensure that the same compilation strategy is used. This eliminates any variations due to the inherent non-determinism present in the adaptive optimisation system. Providing compiler advice also turns off any dynamic profiling or analysis, allowing for a baseline measurement unaffected by the existing profiling techniques.

Secondly, we make use of the replay compilation feature made available through the union of DaCapo and Jikes RVM. This mechanism causes a complete recompilation to occur after an initial warmup period. After this, recompilation is disabled, putting the runtime into a steady-state with respect to compilation actions. Thirdly, each run makes several warm-up passes over the benchmark to warm up caches before performing the final, measured run.

These mechanisms serve to eliminate most other sources of overhead variation, allowing highly repeatable, finely focussed overhead measurement.

All tests are performed on the same physical hardware as outlined in Section 3.2, the frequencies tested with the same as detailed in Sub-Section 4.2.3, and for determining generated DCG accuracy, we used the overlap measure described in Sub-Section 4.1.1. Overheads reported are normalised to execution time with no profiling system running.

4.3.4 Generating the Complete Call Graph

Creating the complete call graph in Jikes RVM is achieved by altering the runtime such that yield points are always taken, and in all prologue yield points a sample is performed. This is achieved by modifying the existing checks for yield points such that that the yield point is always taken. To avoid concurrency issues, each thread maintains its own set of call relationships which are combined at the end of execution to form the full trace.

As a method must take a yield point to be sampled, methods that do not take yield points, such as those marked as uninterruptable, will not form part of the complete call graph. This is in keeping with the existing sampling method, which is also yield point based and hence has this property.

It should be noted that this limitation does not apply to the novel sampling methods present within this chapter, as they do not require yield points to be taken in order generate a sample. For ease of comparison with existing sampling techniques, during sampling, methods which are determined to not take yield points are simply discarded.

Complete call graphs are generated under the same testing methodology presented in Sub-Section 4.3.3. As such they exhibit high levels of consistency across different runs. For all utilised DaCapo benchmarks, each of the ten generated complete call graphs exceed 98% similarity with each other.

4.3.5 Overhead and Accuracy of Existing Implementations

Before exploring the properties of our implementation, we first conduct an examination of the existing sampling technique present in Jikes RVM. We perform this both to generate a baseline for comparison, as well explore possible alternatives for increasing DCG accuracy within the existing system.

The default profiling system used to generate DCGs within Jikes RVM uses a sampling period of one millisecond. This period is far above the minimum period attainable with interrupt based techniques. It is thus likely that we can achieve an increase in accuracy simply through increasing the existing sampling frequency.

To demonstrate the benefits of burst sampling, and to give another point of comparison, we also examine the performance of the existing implementation with burst sampling turned off. Doing this gives us four variations of the existing system; the default setting, which takes eight sample bursts with a millisecond sampling period, the default setting without burst, which takes a single sample every millisecond, and our two adjusted systems, which are the previous ones with the sampling period reduced to the minimum possible whilst using interrupts.

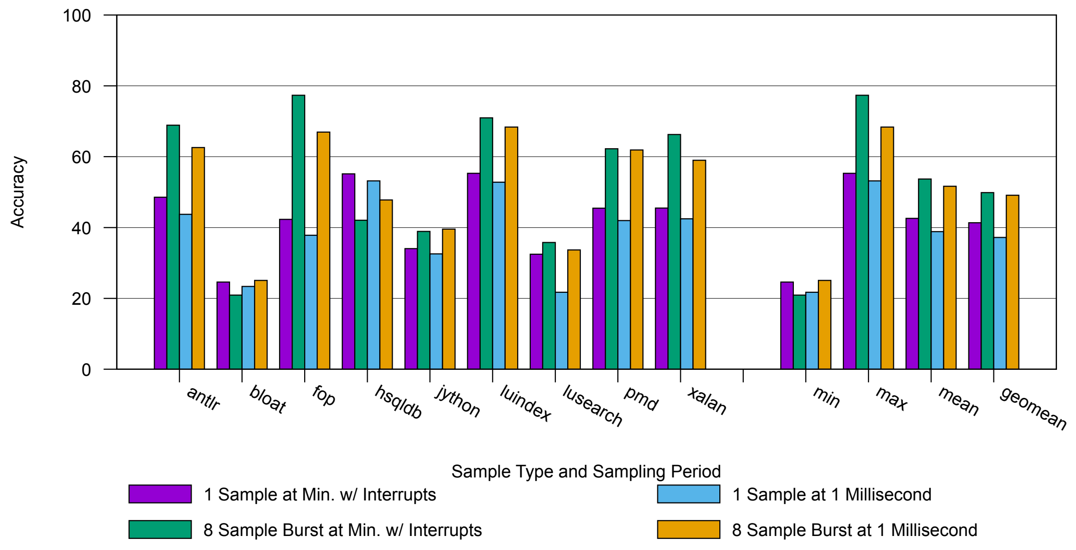

Figure 4.5: Accuracy of existing techniques in Jikes RVM for DCG generation

Figures 4.5 and 4.6 display the accuracy and overhead of these variations respectively. These demonstrate two unsurprising, but important results. Firstly, burst sampling does indeed offer a good increase in accuracy over single sampling at only a moderate increase in overhead. Secondly, we can see that because the executing thread is required to suspend program execution to sample itself, increased sampling frequency results in large performance overheads. As the data generated is to be used at run time, these kinds of overhead increases must be carefully weighted against their expected benefit. As increasing the sampling frequency on the existing implementation resulted in only a minor accuracy increase at a large overhead penalty, it is unlikely to offset its performance cost.

It should be noted that whilst it is possible to remove the delay entirely from the existing system, the incurred overhead is extreme. In this case, the yield point flag is simply continually set for every thread, and the performance penalties approach those exhibited when generating the complete call graph, or approximately 10,000%. As such, this is clearly not appropriate for online usage. Whilst it is possible to busy weight to delay the yield point flag being set, we find the two data points selected for each sampling type to be telling enough.

As the sampling system implemented within Jikes RVM is the same as that presented and examined in Arnold and Grove [2005], we can compare our results with theirs. Overall, we see that accuracies are within similar ranges, although our recorded overheads significantly exceed theirs. We believe the cause of this to be two-fold. Firstly, Arnold and Grove [2005] use the Spec98 [SPEC 1998] benchmarks, which were designed without a focus on stressing managed languages runtimes. This is in comparison to the DaCapo benchmarks used here, which do place such a focus and thus may be more sensitive to performance variations [Blackburn et al.].

Secondly, the overhead measured by Arnold and Grove [2005] was on a earlier

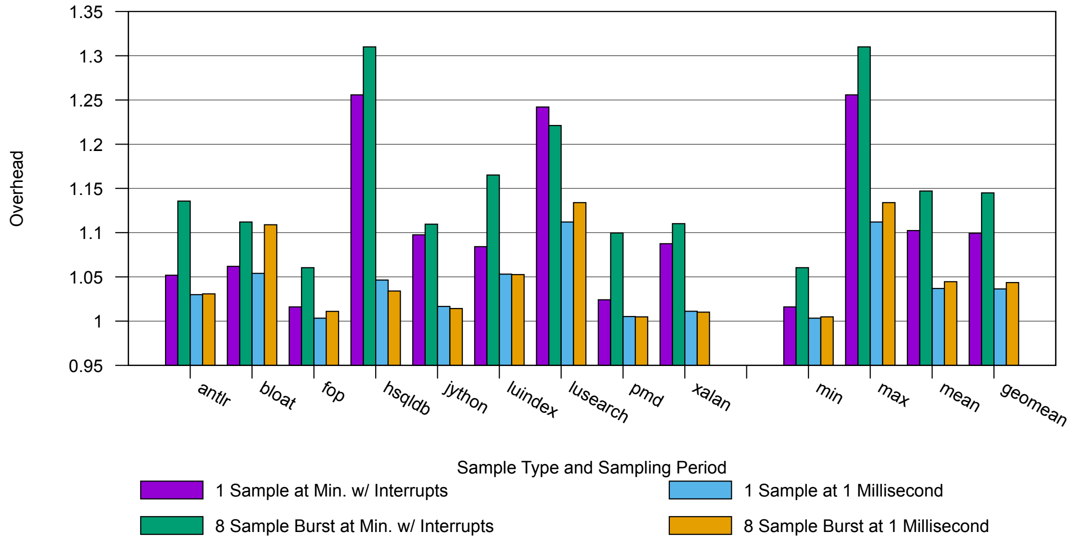

Figure 4.6: Overhead of existing techniques in Jikes RVM for DCG generation

version of Jikes RVM that hadnt benefited from their work, whilst the version we tested on has and is thus overall higher performing. It should be noted that because of the compiler advice mechanic mentioned in Sub-Section 4.3.3, whilst these tests examine only the overhead of capturing data with the existing method, the compiler is being informed as if the default burst sampling is running.

Overall, these results serve to re-enforce the idea that interrupt based sampling techniques have a fundamentally unattractive property. Namely, that increases in their frequency can have a large negative impact on performance. This places a limit on the quality of information that may be collected with them, if it is hoped that said information can be feasibly utilised at runtime.

4.3.6 Implementation Details

To implement direct observation in Jikes RVM we make use of the virtual machine level threading system in a similar manner to the existing sampling infrastructure. Whilst implementing this in native code may result in the ability to sample at higher frequencies, we feel for an appropriate comparison with existing techniques implementing it in a similar manner is important.

One of the aspects that makes this approach so appealing in Jikes RVM, is the layout of the frame. By default, when a call occurs, and a new frame is being set up, the Compiled Method ID (CMID) of the called method is stored in a specific location within the frame. This makes determining the current Java calling context extremely easy, as the CMIDs are unique to Java methods, we simply match the CMID to the method.

To match the current profiling system, we require three pieces of information to define a call edge: the caller CMID, the callee CMID, and the bytecode index of the call instruction. At each sample, the SHIM observer thread iterates through all of the executing Jikes RVM threads. For each thread, the observer thread looks up the current frame pointer for that thread, reads the CMID in that frame, finds the return address in the frame, uses that address to find the callers frame, then reads the callers CMID. Fortunately, Jikes RVM includes existing tools to determine the bytecode offset of a call instruction given the callers CMID and the return address. Thus we simply use this to generate the bytecode index.

Once a relationship has been established, it is stored in an open addressing scheme hash map. In practice this presented the fastest option for storing the graph and enabled a smaller sampling period. This structure is quite different to the tree like structure used by the existing implementation to store and access call graph data. We believe however that data extraction from the hash map to be no more difficult, as the hash used is a combination of the relevant properties of the call edge. Namely, the caller and callee CMIDs, as well as the bytecode offset of the call. As this is the information required by the optimising compiler when it makes inlining decisions, accessing the information is straight forward, it can simply reconstruct the hash. This is an important facet to consider, as it is ultimately hoped that information generated by this new method can be easily plugged into the existing system.

Unfortunately, direct observation does not completely absolve the requirement to perform some instrumentation inside method prologues. To implement the important concept of having already seen a call relationship, as explained in Sub-Section 4.1.2, an increment instruction is inserted into each compiled method prologue. This instruction changes the value of counter local to each thread, such that the SHIM observer thread can compare the count it saw during the last sample of the thread, with the current count to determine whether a new call has occurred.

This is not strictly required, as the observer thread could set a bit somewhere in the currently executing programs frame, to indicate that it had been read already. A prime candidate for this would be one of the high bits in the CMID stored in the frame. Unfortunately, this would require altering other infrastructure to ignore this bit. In comparison to a simple increment, setting and clearing a bit also has the potential to dirty cache lines in both the observer and program thread.

4.3.7 Concurrency Properties

As with the external stack structure, there are no mutual exclusion mechanisms in place to prevent invalid reads by the SHIM observer thread. As the observer thread is interacting directly with the execution stack, such mechanisms would be prohibitively complex and introduce significant overhead.

Three separate pieces of information must be read from the current frame layout without it changing; the callee CMID, the return address, and the callerCMID. If the layout were to change in between these three actions, an invalid read would occur

A series of checks occurs to ensure that the information gathered is at least plausible. Firstly, the CMIDs are checked to see that they are within the appropriate range. As CMIDs are simply integers assigned in order from zero, this is a simple compar-

Sampling Type	Sampling Period	Avg. Accuracy	Avg. Overhead
Existing 1 Sample	1 Millisecond	38.84%	3.7%
	Min. with Interrupts	42.60%	10.2%
Existing 8 Sample Burst	1 Millisecond	51.65%	4.5%
	Min. with Interrupts	53.70%	14.7%
Direct Observation	1 Millisecond	37.36%	3.8%
	Min. with Interrupts	41.07%	5.5%
	Min. with SHIM	49.45%	7.6%

Table 4.2: Average accuracy and overheads for sampling techniques in this section

ison with the maximum assigned CMID. Secondly, the return address for the callee method is checked to determine whether or not it falls within the code range of the caller method. Finally, the byte code index in the caller method which corresponds to the location of the call instruction is checked to confirm that it is in fact a call instruction.

In practice, we record the number of reads that fail some check to garner an idea of how large this impact is. We find that less than 1% of all reads fall into this category. As discussed in Sub-Section 4.2.5, even if a read passes all checks, it may still be an invalid read that just so happens to have the same properties as a valid read. As the chances of landing on a valid combination are less than that of landing on an invalid one, the percentage of such reads is likely much smaller than the 1% of identified invalid reads, and thus is unlikely to have a noticeable impact on accuracy.

As it is possible that an invalid return address is de-referenced in an attempt to determine the caller CMID, we must have some method for dealing with possible segmentation faults. To enable the observer thread to safely check the read addresses, the procedure for handling segmentation faults within Jikes RVM is used. Attempting to access an object through a null pointer is required by the Java language standard to generate a NullPointerException. Within the Jikes RVM, references are machine addresses whereby a null reference takes on the value of 0. Thus, an attempted access to a null reference results in a hardware trap that must be handled by the runtime [Oracle 2015].This results in control being returned to the most appropriate catch block on the stack. Thus we can simply wrap any de-referencing of potentially unsafe addresses in a try/catch block, and masquerade it as an attempted object reference to make use of this in built mechanism.

4.3.8 Overhead and Accuracy of Direct Observation

Figures 4.7 and 4.8 display the generated DCG accuracy and the incurred overheads for the three selected sampling periods of our direct observation implementation, whilst Table 4.2 tabulates these results against that of the existing implementations.

To begin our analysis, we perform some sanity checks on similar techniques. We find that the accuracy for direct observation and the single sample variety of the existing technique to be very similar at both one millisecond and minimum interrupt

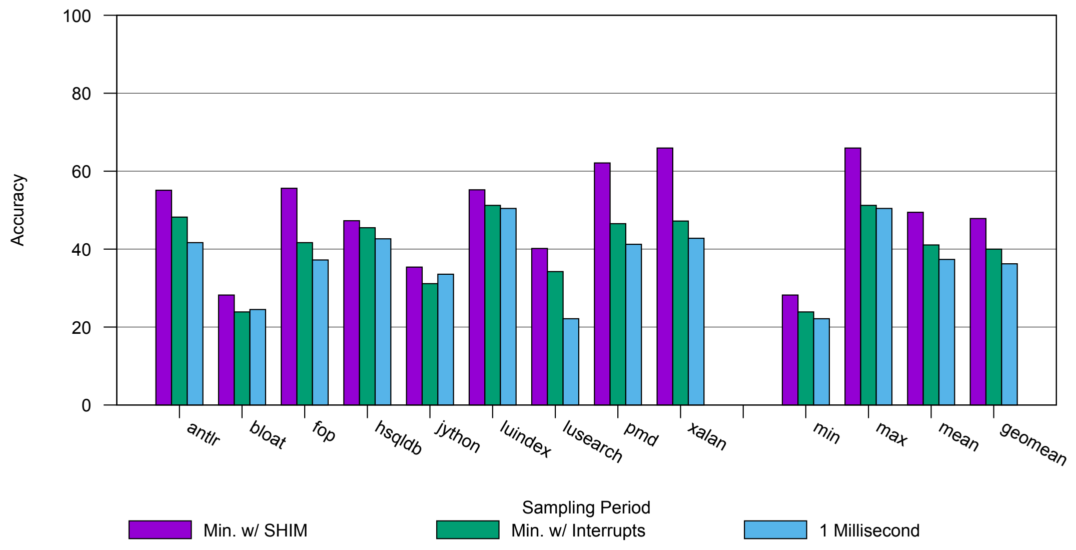

Figure 4.7: Accuracy of Direct Observation at various frequencies across the DaCapo benchmarks

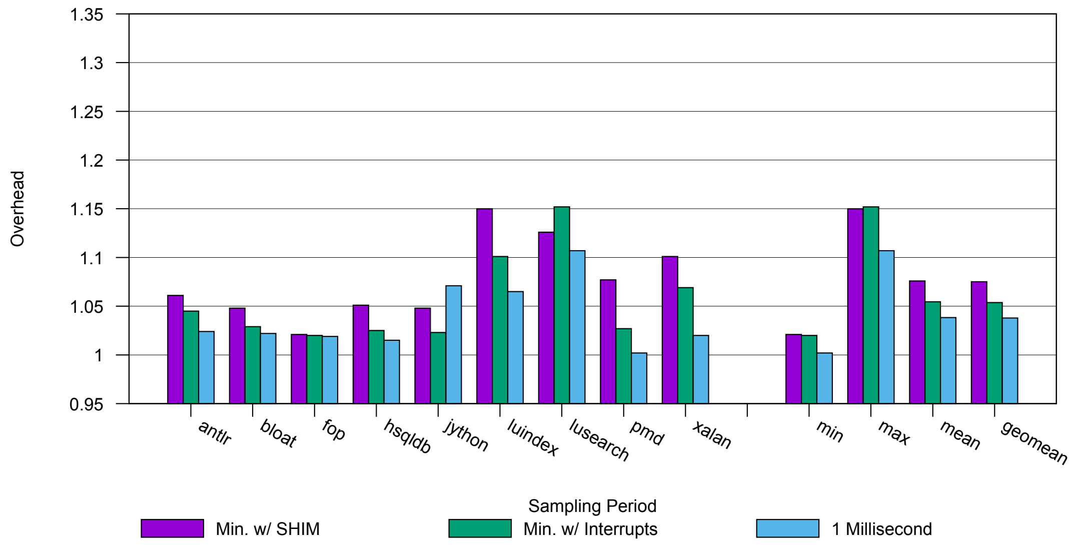

Figure 4.8: Overhead of Direct Observation at various frequencies across the DaCapo benchmarks

sampling periods. This is to be expected, as both are performing roughly the same amount of sampling. We can see however that direct observation is not subject to the same performance penalty when reducing the sampling period from one millisecond. Again, this is to be expected, as we have eliminated the reliance on interrupts to perform sampling. We also note a large jump in accuracy as we move to the maximum frequency afforded by this new technique in comparison to other single sample methods.

An issue arises however when comparing the default, eight sample burst technique with that of direct observation. The existing technique offers a higher accuracy call graph at a lower incurred overhead. Whilst our implementation out performs other single sample techniques, even with a huge increase in sampling frequency it cannot overcome the advantage of the burst sampling technique.

4.3.9 Under-Performance Analysis

Despite allowing sampling to occur at much higher frequencies, frequency weighted call graphs generated through direct observation with a SHIM observer thread fail to improve on the overhead or accuracy of existing techniques within Jikes RVM.

To explore this issue further, we first look at the amount of data generated by each sampling technique. Figure 4.9 displays the total weight of the DCGs generated by the various sampling techniques across different sampling periods, normalised to the execution time of the benchmark. The primary comparison here is between the default eight sample burst approach with a one millisecond sample period, and our direct observation approach at with the minimum sampling period. Note the log scale present on the y axis in Figure 4.9, this is required to offer an effective visual comparison between these techniques as on average the amount of data they generate per millisecond is two orders of magnitude apart.

Peaks in the other implementations are caused by heavily multi-threaded benchmarks, the existing implementation sets the take yield point flag on every thread before sleeping. Thus the amount of data generated scales linearly with the number of concurrently executing threads.

We can clearly see that the frequency advantage has transferred into a data advantage, but this fails to be realised as an increase in DCG accuracy. Our original conjecture that we could overcome the advantage presented by burst sampling through raw sampling frequency evidently seems not to hold.

A perfectly accurate frequency weighted DCG is guaranteed to be generated by sampling at a frequency high enough that every call relationship is marked as seen. Thus, we can garner some idea of whether we are approaching such a frequency by looking at the percentage of calls that the observer threads detects have not changed since its last sample.

Figure 4.10 shows the results of altering our implementation to instead of ignoring unchanged call relationships, to record them and determine the what percentage of the overall call relationships they would make up. We find that at the minimum sampling period, on average less that 10% of calls fall into this category. Other sam-

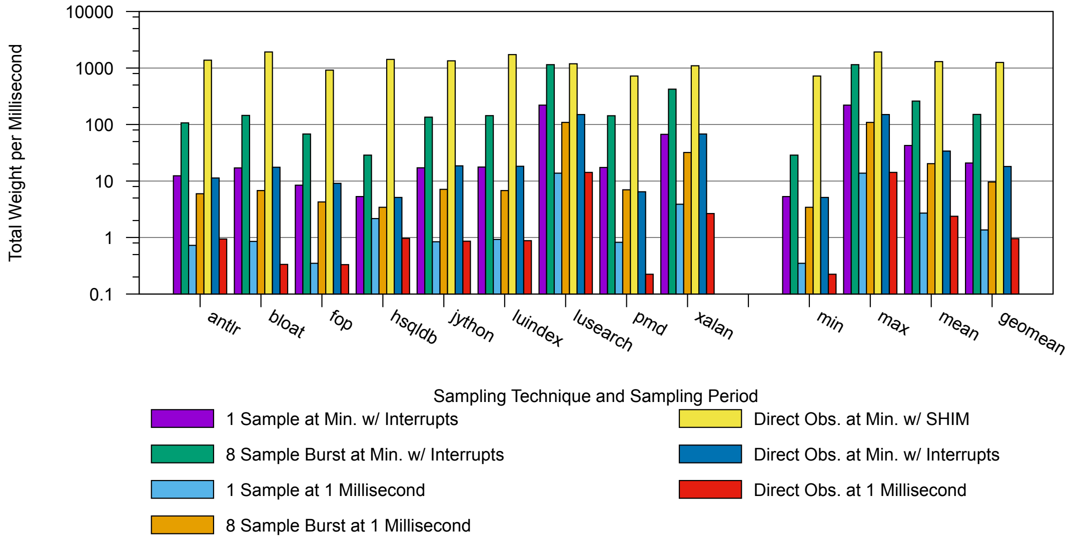

Figure 4.9: Total edge weight of produced call graphs normalised to benchmark execution time

pling periods are ignored as either their percentage of seen calls is zero, or some minor percentage caused by long running synchronisation methods present in certain benchmarks.

Given we have a fixed data point at one end of 100% seen calls resulting in perfect DCG accuracy, we can see the unsurprising general positive correlation of DCG accuracy and the percentage of calls marked as seen displayed in Figure 4.11.

With both a theoretical and practically demonstrated relationship between the percentage of calls and DCG accuracy, we can conclude that as our direct observation implementation only samples fast enough to observe a small number of seen calls, that the speed with which we are sampling is still insufficient to generate highly accurate frequency weighted DCGs through single sample techniques.

Whilst it is undoubtedly possible to squeeze a higher frequency sample rate out of our implementation through optimisations, given the growth of accuracy with respect to sample period displayed in Figure 4.7, and the low percentage of seen calls at the current sample rate recorded in Figure 4.10, we believe that another order of magnitude increase in speed would be required to consistently outperform burst sampling techniques. It is unlikely that under the constraint of having the data be both validated, and usable at runtime, that this could be achieved.

4.3.10 Summary

In this section we presented direct observation, a technique utilising a SHIM observer thread to directly inspect the runtime stack of Jikes RVM with a view to generating frequency weighted DCGs. We first analysed the existing profiling techniques present within Jikes RVM. We concluded that owing to their interrupt based implementa-

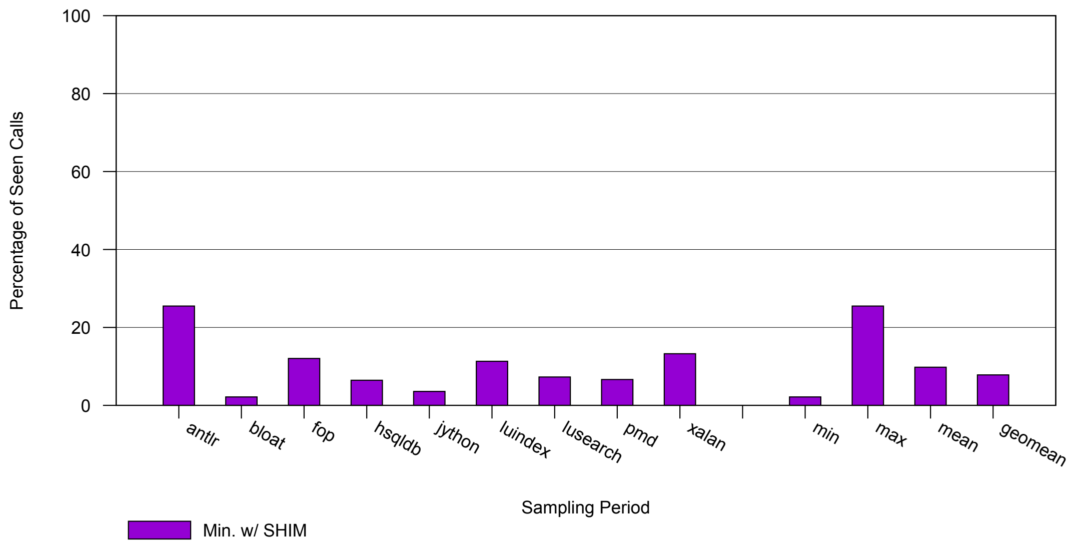

Figure 4.10: Percentage of samples that read an unchanged call relationship

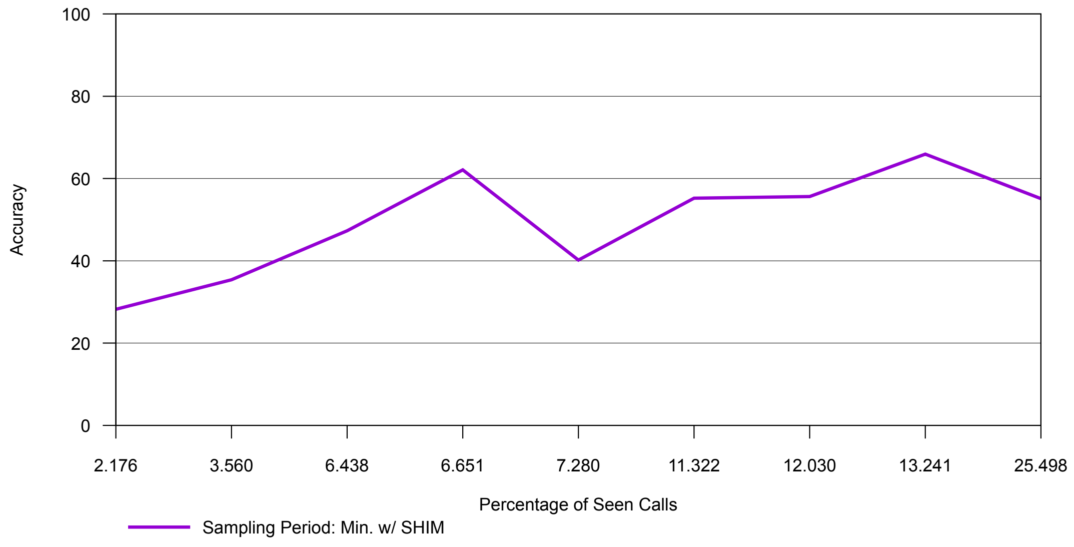

Figure 4.11: Percentage of seen calls vs. generated call graph accuracy

tion improving their accuracy through frequency increases resulted in unsatisfactory overhead. We then discussed the details of our implementation and analysed its performance, finding that in comparison to equivalent single sample techniques, direct observation provides similar accuracy at reduced overhead.

When compared against burst sampling however, despite highly increased sampling frequency and the generation of vastly more data, the direct observation technique outlined in this section fails to provide a better accuracy overhead tradeoff. We attribute this to a sampling period still too long to capture short execution time methods with accuracy, and find it unlikely that without another major increase in sampling frequency that DCGs generated in this manner will outperform their burst sampling generated brethren.

4.4 Ring Buffer

In the previous section, we discovered that even at vastly increased sampling frequencies, single sample techniques fail to outperform burst sampling based ones. In this section, we will combine the high frequency, low overhead sampling afforded by SHIM, with burst sampling techniques demonstrated to significantly improve DCG accuracy.

To achieve this, we will make use of a ring buffer intermediary data structure. Similar to the tag stack presented in Section 4.2, this data structure is maintained through instrumentation inserted into method prologues. In this section we will outline how the ring buffer structure incorporates concepts of burst sampling, give an overview of its implementation within both Jikes RVM and SpiderMonkey, and perform an analysis of its performance comparative to existing techniques and those presented thus far in this thesis.

4.4.1 Achieving Burst Sampling

The key idea that enables burst sampling to improve on the accuracy of single sampling techniques, is its increased resistance to inaccuracies caused by short lived methods. In a single sampling technique, a sample must occur within the executing method for it to be recorded, hence methods which may be called frequently but have short execution time are underrepresented. Burst sampling alleviates this requirement, it must simply instigate a sampling burst *near* a short lived method for it to be recorded.

Within Jikes RVM the yield point mechanism lends itself well to the implementation of burst sampling, as a thread can simply be instructed to take a series of yield points at each sample. This benefit is not afforded to the direct observation technique, even if periods of extreme sampling frequency could some how be created, they would not guarantee that every method during the burst had been recorded.

To emulate the benefits of burst sampling, we use a ring buffer intermediary structure. Similar to the tag stack structure outlined in Section 4.2, this is used for communication between the SHIM observer thread and the executing program. In contrast to the tag stack however, which was designed to represent the current calling context, the ring buffer is designed to capture the most recent series of calls. Rather than pushing and popping from a stack, methods simply place their identifier in the buffer. Thus, by simply reading through the buffer the observer thread can see the exact sequence of previous calls. A ring buffer is this case necessary to avoid the massive memory usage of an unbounded buffer.

4.4.2 Jikes RVM Implementation Details

In contrast to direct observation, in which the observer thread was responsible for walking the stack to find the relevant information, when utilising a ring buffer the producer thread extracts this information from its own stack and writes it to a buffer. Instrumentation is inserted into each method prologue to record the current methods CMID, return address, and to determine and record the CMID of the calling method by inspecting the stack. Because the entire call can be captured in the method prologue, there is no need to instrument the epilogue.

To simplify concurrency considerations, each thread maintains an individual buffer, through as relationships are read they are collated by the observer thread into a single hash map.

At each sample, the SHIM observer thread checks to see whether the target threads buffer is full. If it is, then it simply reads through the buffer recording each call within it. Once it has finished, it resets the buffer to the start and the thread continues to write to it.

The choice to have the observer thread reset the buffer is a deliberate one to improve the concurrency properties of the system (this is explored further in the next sub-section). This does however increase the computational requirements of maintaining the buffer, as the application thread must check whether or not it is full before writing to it, rather than just automatically looping around.

4.4.3 Concurrency Benefits

The specific implementation details of the ring buffer structure provide some inherit concurrency benefits. Because the observer thread is responsible for resetting the producer threads pointer, after the producer thread has detected the buffer is full is will not write to it again until the consumer thread has finished reading the buffer. As the consumer thread does not attempt to read from the buffer until it is full, mutual exclusion is achieved.

This is in comparison to a continuously running ring buffer, which does not inherently provide mutual exclusion. In this format, the producer thread simply resets its own pointer when it reaches the end (or uses some form of bit masking). Without additional mutual exclusion mechanisms in place, the producer thread may write to a location currently being read from. As three buffer slots are used to capture the call relationship, this may lead to invalid relationships being read.

The continuous type of ring buffer can achieve thread safety in the case whereby each relationship can be captured in a single word sized value, and atomic word sized reads and writes are available (these are supported on all recent Intel hardware)[Intel 2015a]. In this scenario, because each element in the buffer is complete relationship and are atomically accessed, there is no chance that an invalid read will occur. However, the recorded information must contain at least a word's worth of address information to identify the byte code offset of the call instruction in the caller, and thus the relationship cannot be fully captured in a single word.

Because of the mutual exclusion provided by the implementation, many of the validity checks present within the direct observation implementation can be forgone. In addition, as extraction of information from the stack is performed by the producer thread, the observer need not attempt to reference addresses which have been read from the stack and may be invalid. This reduces the reliance on the runtime exception handling, as the only addresses that are attempted to be referenced are guaranteed to fall inside of long lived objects, namely the compiled methods.

4.4.4 Accuracy and Overhead in Jikes RVM

In this subsection we evaluate the overhead and accuracy of several different ring buffer sizes across three sampling periods. The testing methodology and benchmarks used are the same as outlined in Sub-Section 4.3.3. We use the same sample periods as the rest of this chapter, and select three sizes of ring buffer; 10KB, 100KB, and 1MB. These sizes are selected to give a good understanding of performance across buffer sizes, whilst maintaining practical memory footprints. Figures 4.12, 4.13 and 4.14 display the accuracy of each of these implementations, whilst Figure 4.15 shows the overheads incurred for the 100KB sized buffer only, whilst all results are tabulated in Table 4.3.

As overheads across buffer sizes are highly similar, we elect to only graph the overhead for highest accuracy buffer size. This similarity in terms of overhead is to be expected as the primary overhead is incurred during the maintenance of the ring buffer by the application thread, which remains constant independent of buffer size.

From these results, we can see that although the 100KB buffer is best performing with regards to accuracy, the difference is only slight in comparison to those an order of magnitude larger or smaller. This echoes the results of [Arnold and Grove 2005] that find after a certain burst sampling duration, accuracy remains fairly constant.

We also note that the accuracy between the smallest sampling period attainable with SHIM, and that attainable with the interrupt mechanism, to be similar. This is because despite the large increase in sampling frequency, sampling only occurs when the buffer is detected to be full. At these buffer sizes, this means that the fastest sampling speed is only performing around twice as many reads of the buffer than the next fastest, despite checking whether it is full significantly more times. Unfortunately we cannot really capitalise on this property as the primary cause of overhead is the instrumentation, rather than the speed at which the SHIM observer thread is attempting to sample.

Despite not requiring the high sampling speed enabled through using a separate SHIM observer thread, offloading all of the online analysis to the observer thread en-

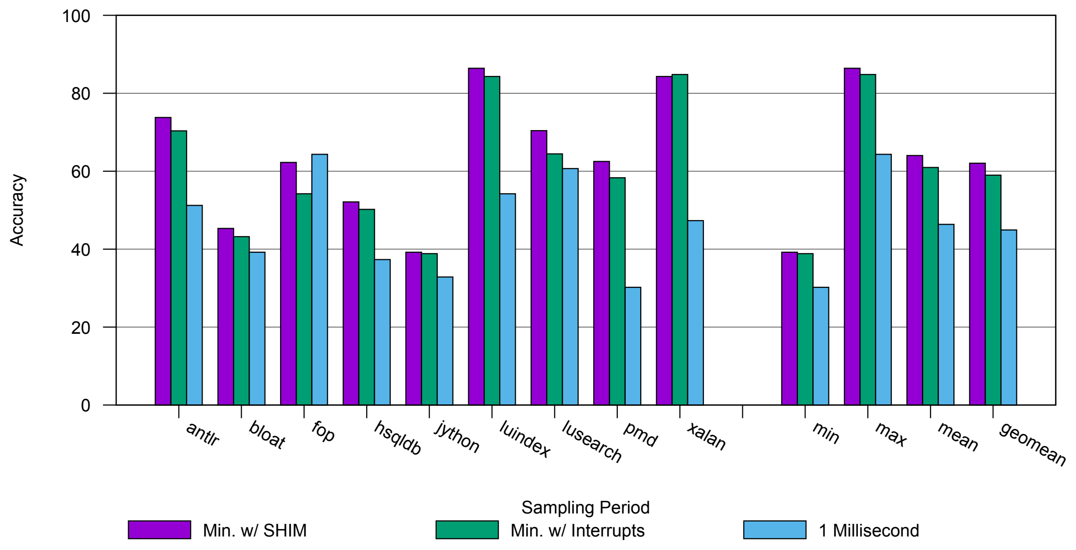

Figure 4.12: Accuracy with SHIM utilising a 10KB ring buffer in Jikes RVM

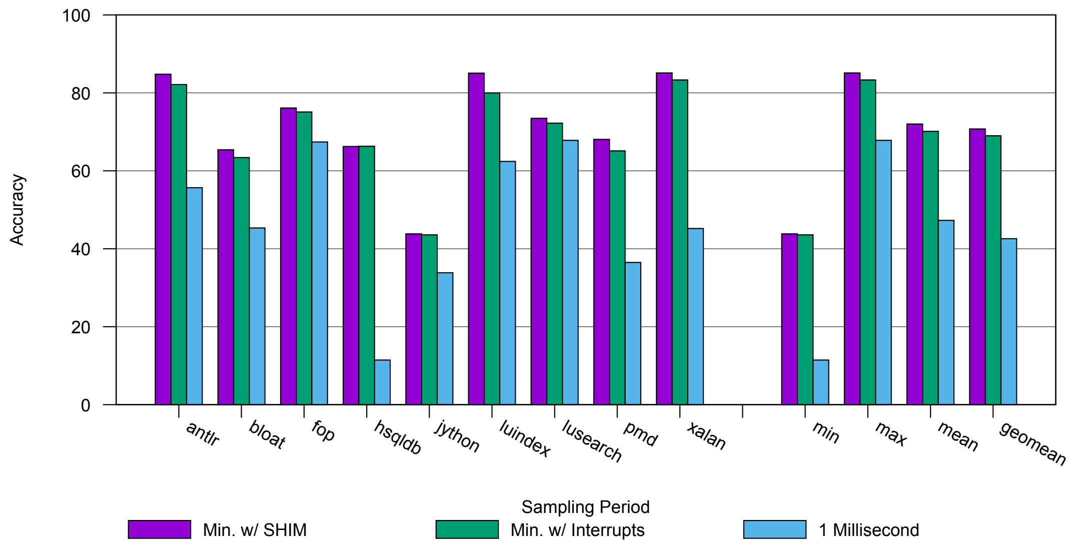

Figure 4.13: Accuracy with SHIM utilising a 100KB ring buffer in Jikes RVM

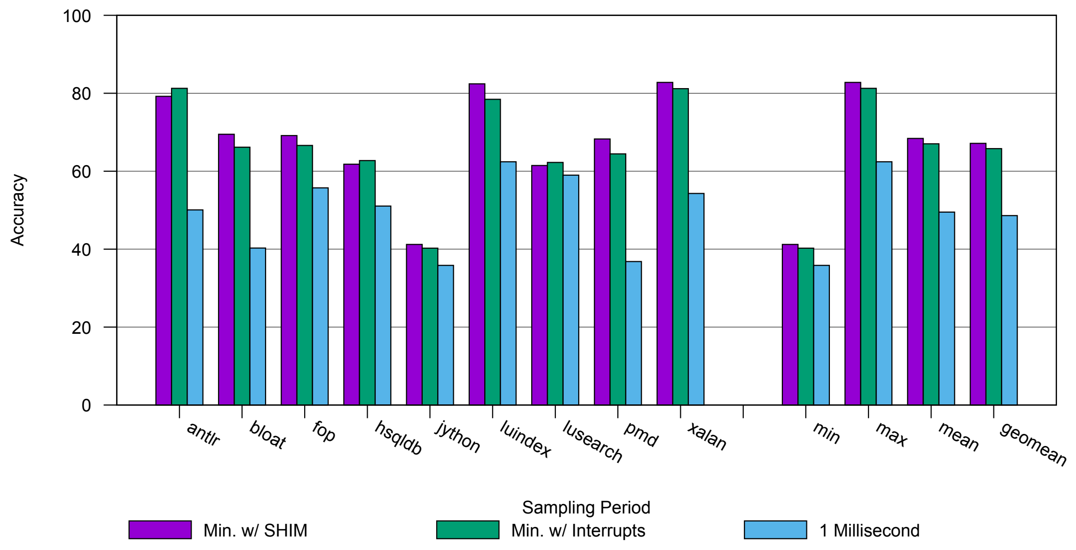

Figure 4.14: Accuracy with SHIM utilising a 1MB ring buffer in Jikes RVM

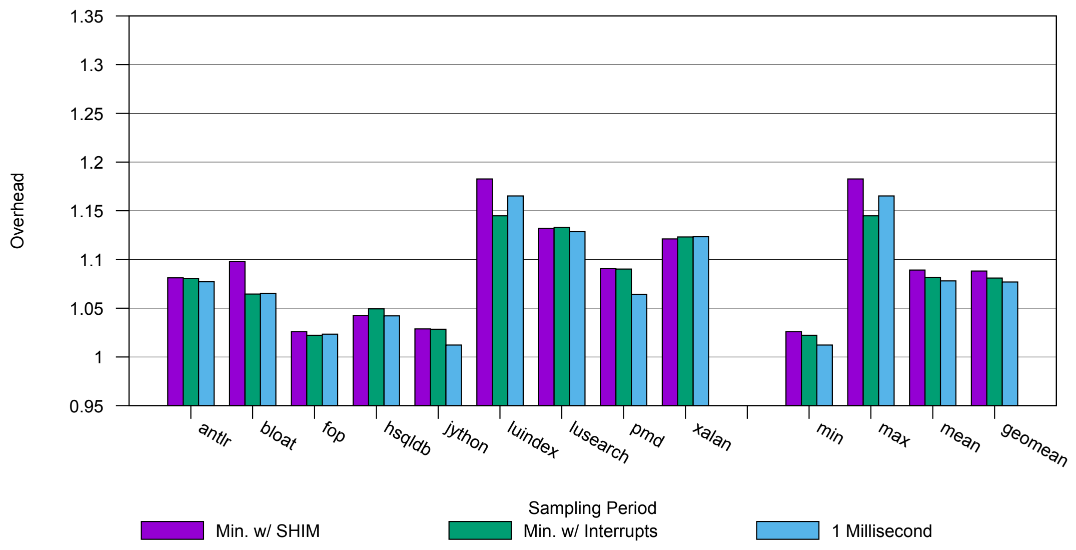

Figure 4.15: Overhead of SHIM utilising a 100KB ring buffer in Jikes RVM

Sampling Type	Sampling Period	Avg. Accuracy	Avg. Overhead
Existing 8 Sample Burst	1 Millisecond	51.650	4.5%
	Min. with Interrupts	53.70%	14.7%
Direct Observation	Min. with SHIM	49.45%	7.6%
10KB Ring Buffer	1 Millisecond	46.37%	7.0%
	Min. with Interrupts	60.97%	8.6%
	Min. with SHIM	64.03%	9.1%
100KB Ring Buffer	1 Millisecond	47.30%	7.1%
	Min. with Interrupts	70.133%	8.2%
	Min. with SHIM	72.01%	8.9%
1MB Ring Buffer	1 Millisecond	49.50%	7.0%
	Min. with Interrupts	67.04%	8.1%
	Min. with SHIM	68.42%	9.0%

Table 4.3: Average accuracy and overheads for ring buffer based sampling techniques, as well as best performing alternatives, in Jikes RVM

ables lower overhead. It may seem that the extra speed is superfluous, but additional speed is only one aspect of what SHIM provides. In the existing interrupt based system, because the thread must not only perform a sample, but also update the existing DCG, the sampling process takes significantly longer. As updating the ring buffer only takes around 10 instructions, it can be completed significantly more times (in this case, at every method call) before reaching the same overhead.

Given the large increase in accuracy experienced by utilising a ring buffer, when compared simply to direct observation, we can conclude that emulating the benefits of burst sampling has indeed improved the results significantly. We also note that as expected, the increased amount of instrumentation required to facilitate its use has had a performance impact.

Comparing the existing default implementation to the best performing ring buffer based SHIM approach, we find that for an overhead increase of around 4 percentage points we can increase accuracy by nearly 20 percentage points. If increased DCG accuracy is desired, then the approach presented in this section results in a better overhead to accuracy trade-off than can be obtained by simply increasing the sampling of frequency currently implemented techniques.

For the particular use case of guiding inlining optimisations, it remains to be seen whether this increase in accuracy can result in an overall speed up of execution, which offsets the slightly increased cost. To give some context, [Arnold and Grove 2005] found the extra accuracy afforded by burst sampling, over single sampling, at millisecond timescales, resulted in an overall performance increase of around 4%. As displayed in Figure 4.5, burst sampling provides around a 10 percentage point gain over single sampling. Given that the technique outlined in this section managed to almost double that accuracy gain, we believe it likely that it will be able to offset its increased cost by enabling improved performance.

4.4.5 SpiderMonkey Implementation Details

Given the positive result experienced in Jikes RVM, we implement the ring buffer intermediary structure in SpiderMonkey. Implementation follows the same basic principles as outlined in Sub-Section 4.4.2. To maintain the same amiable concurrency properties, we again choose to have the observer thread reset the buffer pointer after finishing its read.

Where the implementations differ is the data stored to the buffer. Whilst to compare accuracy with existing implementations, DCGs generated in Jikes RVM differentiated between different call offsets between two methods, which we do not in SpiderMonkey. In Jikes RVM, this required storing the return address which hence we may forgo in SpiderMonkey. The previous implementation also required that the application inspect its own stack in order to extract the caller and callee CMIDs. Because the stack layout in SpiderMonkey is less receptive to inspection, we must alter what information we store.

During the method prologue, a method identifier is written to buffer, whilst in the epilogue the same method identifier is written with a bit set to identify it as an epilogue. This method means that the executing program need not inspect its own stack to determine the current calling context, but rather this task is offloaded to the observer thread. The SHIM observer thread is able to walk through the buffer and by interpreting the sequence of prologues and epilogues, determine which calls were made.

4.4.6 Accuracy and Overhead in SpiderMonkey

We generate accuracy and overhead figures for the same buffer sizes and the same sampling frequencies as Sub-Section 4.4.4, and with the same testing methodology as outlined in Sub-Section 4.2.3. Figures 4.16, 4.17 and 4.18 display the accuracy of DCGs generated with these various buffer sizes and sample periods. Table 4.4 tabulates all of the results from this section, with the key results from Section 4.2.4. We again find that overheads across different buffer sizes to be fairly consistent, and thus only plot the overhead of the most accurate buffer size of 1MB in Figure 4.19.

Here we see some distinct similarities, and key differences, between the behaviour of the ring buffer implementations within SpiderMonkey and Jikes RVM, across buffer sizes and sample periods. We again see a high level of consistency between the accuracy of the minimum sample period with SHIM, and of that attainable with interrupts. Inspection reveals the same root cause, namely that there is little difference in the number of buffer reads performed by these sampling periods. Interestingly, this also generally extends to sampling at the millisecond timescale. This is a property of the JavaScript call graphs being significantly simpler than that of those in Java, this is both caused by the language usage in general and the elimination of call offset differentiation. This also causes the DCGs generated within SpiderMonkey to show exceedingly high accuracy.

Where these result differ significantly from those of the last section, is in the resistance to buffer size. We find that buffer size greatly affects the performance of the

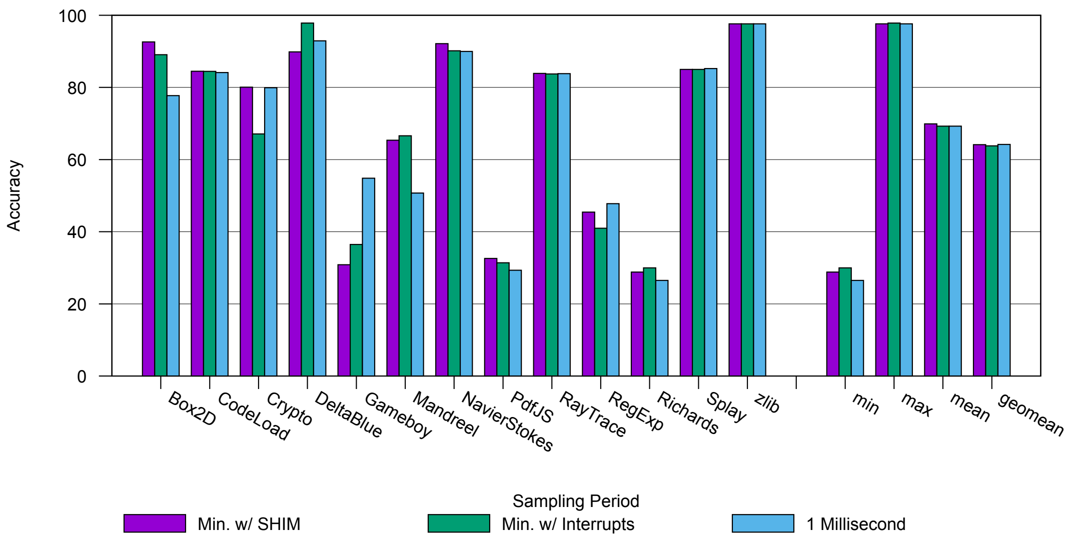

Figure 4.16: Accuracy with SHIM utilising a 10KB ring buffer in SpiderMonkey

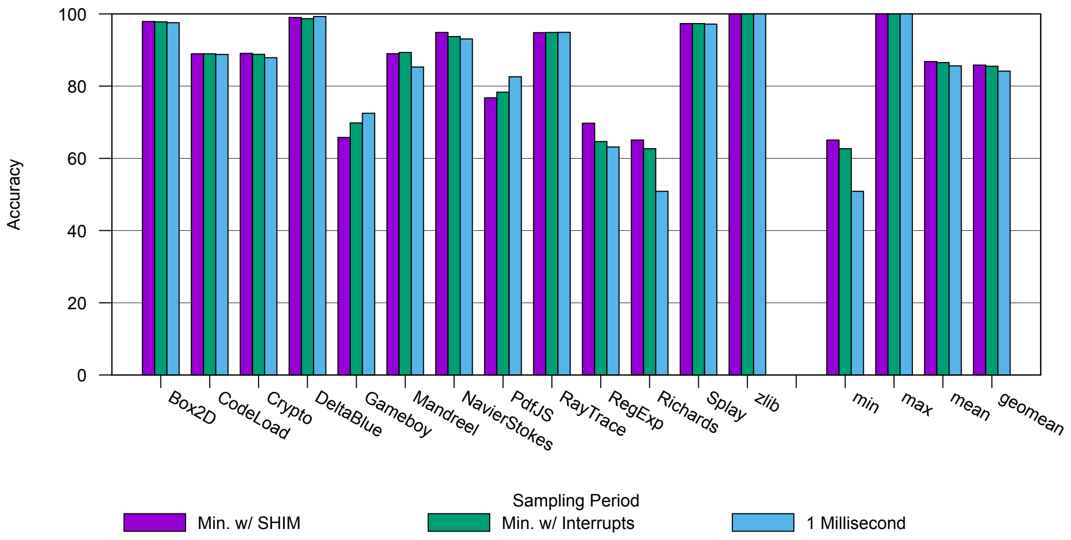

Figure 4.17: Accuracy with SHIM utilising a 100KB ring buffer in SpiderMonkey

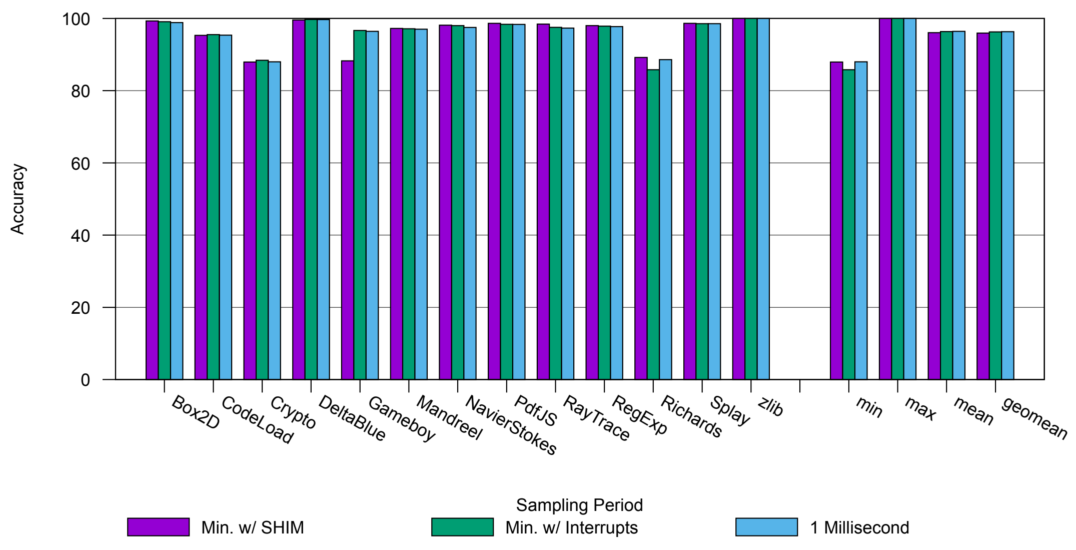

Figure 4.18: Accuracy with SHIM utilising a 1MB ring buffer in SpiderMonkey

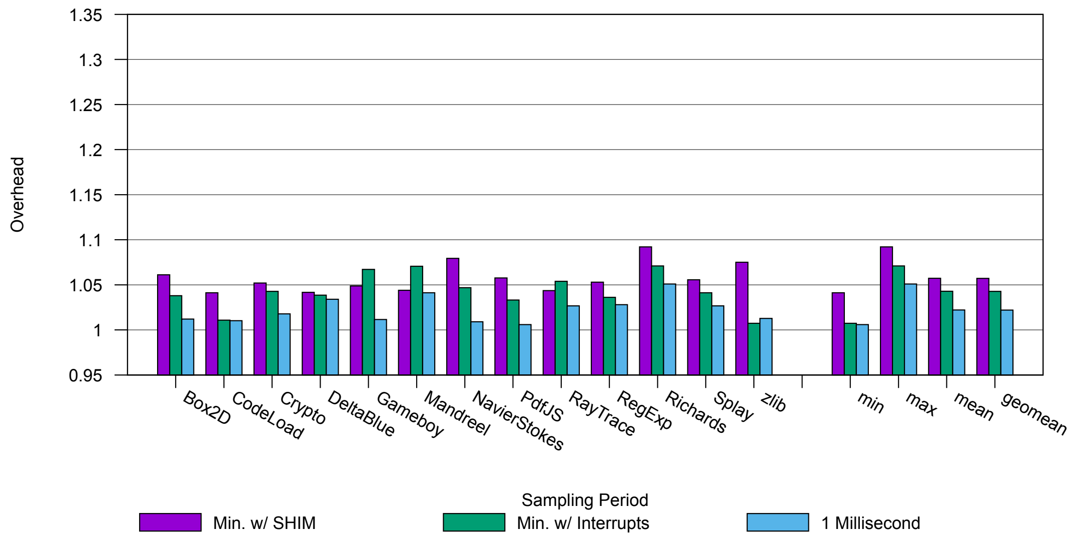

Figure 4.19: Overhead of SHIM utilising a 1MB ring buffer in SpiderMonkey

Sampling Type / Buffer Size	Sampling Period	Avg. Accuracy	Avg. Overhead
10KB	Min. with SHIM	69.91%	5.1%
	Min. with Interrupts	69.26%	2.1%
	1 Millisecond	69.27%	2.2%
100KB	Min. with SHIM	86.80%	5.3%
	Min. with Interrupts	86.62%	3.2%
	1 Millisecond	85.62%	2.1%
1MB	Min. with SHIM	96.06%	5.7%
	Min. with Interrupts	96.37%	4.1%
	1 Millisecond	96.42%	2.4%
Single Read Full Stack Depth	Min. with SHIM	79.42%	5.3%

Table 4.4: Average accuracy and overheads for sampling techniques in this section, along with key comparison technique

system, with large buffers resulting in strictly better accuracy.

This is due to the differences in implementations. Because the Jikes RVM implementation samples its own stack, there are independent calling contexts present in the ring buffer. Each set of writes to the buffer is enough for a call to be recorded. This is not the case in SpiderMonkey, in which the buffer is just a record of prologues and epilogues. Thus the observer thread must attempt to gain some idea of context by reading the buffer. For example, if the first element in the buffer is the prologue of function A, we know that if the next element is a prologue then it will have been called from A. However, we cannot tell what called A, and thus cannot record a call relationship.

We record the number of these contextless calls as a percentage of the total number of calls, and display them relative to buffer size in Figure 4.20. We find that with smaller buffer sizes, larger numbers of these calls occur, and note a strong inverse correlation between the performance in individual benchmarks and their occurrence.

With an appropriately sized buffer, we can see that the ring buffer described in this section greatly improves upon the external stack structure implemented in Section 4.2. This echoes the results of Sub-Section 4.4.4, reenforcing the idea that even high sampling frequencies can benefit from burst sampling concepts. We find that in SpiderMonkey this increase in accuracy comes at no extra performance cost over the stack structure, as the complexity of instrumentation to maintain them is approximately similar.

4.4.7 Summary

Burst sampling techniques were first developed primarily in response to the limitations of interrupt based sampling mechanisms. In this section, we have demonstrated how even with this limitation removed, high frequency sampling can still benefit significantly from incorporating burst sampling concepts.

To incorporate these concepts with SHIM based sampling, we implemented a ring

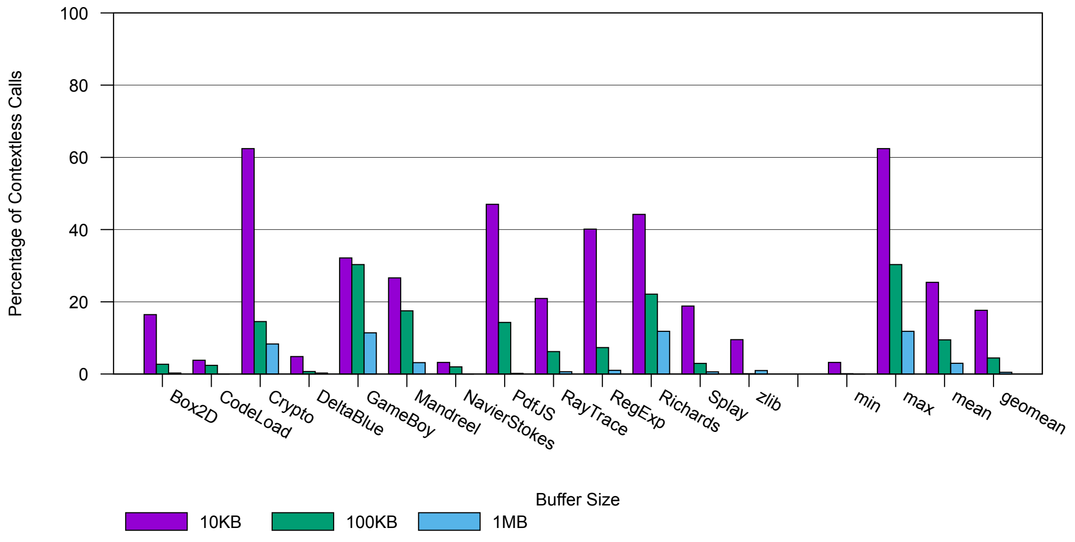

Figure 4.20: Percentage of contextless calls by buffer size at maximum sample rate

buffer intermediary structure to record a sequence of calls, rather than sampling a single call at a time. This results in a reduction in the impact on accuracy short execution time methods have. It achieves this by alleviating the requirement to sample within their short execution window.

We find that our implementation vastly outperforms the single sample technique we presented in Section 4.3, and offers a large accuracy increase over the default sampling technique present within Jikes RVM. This increase comes at only a moderate growth in overhead, which past work indicates may possible be overcome through the improvements to inlining optimisations that increased accuracy brings. We then implemented a similarly inspired ring buffer structure in SpiderMonkey, finding again that performance improved over the single sample technique presented in Section 4.2.

Chapter 5

Conclusion

The optimisation of language implementations, whether it be at runtime through adaptive processes, or as part of the development cycle, is heavily dependant on information generated through dynamic profiling. Existing interrupt based profiling techniques are fundamentally limited in the frequency with which they may perform sampling actions. Because the thread to be sampled must stop program execution during the interrupt, high interrupt frequencies have extreme overhead penalties, to the point where hard caps are enforced by the Linux kernel.

Limitations in sampling frequency experienced by current interrupt based profilers have a detrimental impact on the temporal resolution of the data they generate. This loss in resolution results a concealment of high frequency program behaviour as performance characteristics are averaged across long sampling periods.

SHIM utilises multi-threaded hardware to remove a reliance on interrupts, enabling high frequency, low overhead profiling of applications. With sampling periods several orders of magnitude smaller than current techniques, SHIM reveals performance behaviour down to the function level. This specificity is achieved in part due to high frequency sampling, but also through the use of software tags generated by instrumenting the profiled application.

We initially implement SHIM within SpiderMonkey to practically demonstrate its benefits to language runtime development. We achieve this through allowing high frequency capture and attribution of micro-architectural performance counters to JavaScript level code. We demonstrate how this can be used to measure hardware performance across compiler implementations, giving unprecedented insight into their effectiveness. We extend the single software tag concept initially presented in SHIM to a tag stack, facilitating the capture of the entire JavaScript calling context enabling even greater specificity when attributing hardware performance properties.

In Chapter 3, we demonstrated three methods for generating frequency weighted dynamic call graphs (DCGs) enabled through usage of SHIM; a separate stack structure, direct observation and an intermediary ring buffer. We find that the first two of these methods are competitive with existing single sample techniques, but are bested in both accuracy and overhead by current burst sampling approaches. We analyse this result, and conclude that even the extreme sample frequencies afforded by SHIM are not enough to create highly accurate frequency weighted DCGs with single sample based techniques.

Finally, we combine the key concepts of burst sampling, with the frequency and overhead improvements provided by SHIM, in the form of a ring buffer intermediary. Despite requiring more complex instrumentation, this novel way of communicating calling context information from the profiled application to the SHIM observer thread yields a large improvement in accuracy over existing burst sampling techniques. This improvement comes at a comparatively small performance overhead, providing confidence that the improved inlining decisions enabled by utilising this information can result in a net performance gain.

5.1 Omissions

In this section we will discuss elements that we believe are of some importance, but have not been discussed in any detail in the preceding chapters.

5.1.1 Properties of Statistical Sampling

In Chapter 2, we demonstrate examples of the kinds of data that may be generated at high-frequency with SHIM integrated with SpiderMonkey. We fail to provide any indication of the accuracy of the performance data collected. As the collected data is a statistical, rather than a complete, representation of the actual data, it must be interpreted as such. The validity of statistical data generated in the interrupt fashion is a topic that is already well explored [Schnieder 2000]. As sampling with SHIM generates data with fundamentally similar properties to those generated with interrupts, albeit at much higher frequencies, validity analysis of interrupt generated samples is applicable here.

5.1.2 Time Weighted Dynamic Call Graphs

In Chapter 3, we have exclusively looked at the generation of frequency weighted DCGs, ignoring their brethren, the time weighted DCG. In the latter, the edges are weighted according to the amount of execution time spent in the relationship represented by the edge. As discussed in Chapter 2, these types of DCGs can be used to assess which methods are hot in terms of execution time.

There are two reasons we chose not to focus on these types of DCGs. Firstly, generating the exact DCG of this variety is exceedingly difficult [Chang et al. 1992]. This is primarily because of the large impact measurement may have on execution time. This is in stark contrast to frequency weighted DCGs, where there is little impact outside of non-deterministic function calls. This property is what allows simulation based techniques to generate highly accurate frequency weighted DCGs.

Secondly, we make the claim that there is a more direct relationship between sampling frequency and the accuracy of time weighted DCGs, than that of frequency weighted DCGs. There are no competing techniques for the generation of time weighted DCGs, such as burst sampling for frequency weighted DCGs. Thus, the claim that increased sampling rate improves the accuracy of time weighted DCG's is not particularly controversial; increased sample frequency is simply strictly better with regards to accuracy.

Should a time weighted DCG be desirable, it is trivial to modify the methods presented in Sections 4.2 and 4.3 to generate them; we may simply ignore the flag which indicates whether the relationship has already been recorded. In doing this, overheads are unlikely to be affected as the core functionality remains similar.

5.1.3 Other Forms of Overhead

In our measurement and discussion of overhead, we have exclusively focused on execution time overhead. This is of course not the only form which overhead may take.

The retention of data collected will incur some kind of memory overhead. In Chapter 2, because data is not collated at run time, and samples are simply recorded one after another for offline analysis, the memory overhead is quite large. It remains practically small enough however, to capture all samples at maximum frequencies across the Octane benchmarks on a computer with 4GB of RAM. In this scenario the information gathered is expected to be both unaffected by large memory usage, as well as used for offline analysis, thus large memory usage is forgivable. In the generation of the DCGs in Chapter 3, we move exclusively to online analysis, whereby data is collated after it is captured. Because of this, memory usage is comparable to existing techniques. The external stack and ring buffer structures also have their own memory footprint, the largest tested of these is 1MB, which may or may not be important depending on the application. If memory is at a premium, their size may be reduced incurring effects explored in this thesis.

At maximum sampling frequencies, a SHIM based technique is making full use of another hardware thread. This allows low execution time overhead, but increases the overall load on the system. This increased load presents itself as increased thermal output and energy consumption. For offline profiling purposes as part of the development cycle this may be acceptably ignored, however if these techniques are to be integrated into the adaptive optimisation of a long running system, some consideration may need to be given to these factors.

5.2 Future Work

SHIM has opened a door to a wealth of information previously hidden behind low sampling frequencies. Discussing all the possible practical applications of SHIM is beyond the scope of this section, rather we examine some of the possible direct continuations of the work presented in this thesis.

Replace Burst Sampling with Ring Buffer SHIM in Jikes RVM

There is no more potentially visceral example of the benefits of high-frequency sampling than a reduction in overall execution time. We believe that the the improved accuracy of DCGs generated through the implementations discussed in this thesis will allow for more informed inlining decisions, potentially resulting in an overall improvement in performance for Jikes RVM.

Utilise Selective Instrumentation with Ring Buffer SHIM
The implementation of the ring buffer in Section 4.4 lends itself to selective instrumentation. As each relationship recorded in the buffer is essentially atomic, it does not require that every method be instrumented in order to continue to function correctly. As the data generated by SHIM in this manner is designed to inform inlining decisions, if the method for some reason will never be inlined, and this can be determined at compile time, then the instrumentation can safely be left out. Performing this kind of selective instrumentation can reduce the overall overhead without impacting inlining performance, and is an approach used succesfully in other managed runtimes.

Replace Static Inlining in SpiderMonkey with Adaptive Inlining using SHIM
Currently the optimised JIT compiler within SpiderMonkey makes inlining decisions based primarily on static information (the only dynamic information used is type related information). Large performance gains were witnessed in Jikes RVM by moving away from static inlining, which can make naive and thus poor inlining choices, to inlining driven by dynamic information. This thesis demonstrated how high levels of DCG accuracy can be achieved with overheads low enough to drive runtime inlining decisions within SpiderMonkey. It is believed that replacing the current naive strategy with a dynamic one, fuelled with information gathered by SHIM is likely to result in overall improved performance.

Utilise Hardware Performance Counters for Adaptive Optimisation
Because of the limitations in temporal resolution of interrupt based techniques, their use is essentially non-existent in the realm of adaptive optimisation. It is difficult to marry the information they generate in this time scale with existing optimisation techniques. In Chapter 2, we demonstrated how with increased temporal fidelity we could attribute hardware performance counters to JavaScript level functions and their respective compilers. This information could be used to assess the performance of compiler actions at runtime, potentially triggering a change in compiler tack and recompilation of poorly performing functions. This is a bold goal, and in order to achieve an overall performance gain would require careful weighing of the sampling and recompilation overhead against possible performance gains.

Theoretical Analysis of Sampling Techniques
In this thesis we have shown that even at sampling speeds prohibited to interrupt based techniques, the accuracy of frequency weighted DCGs is still heavily reliant on capturing some concept of sequential method calls. At lower sampling frequencies, it is easy to hand wave this away as a property of sampling at frequencies many magnitudes slower than that of which functions execute. However, even as sampling periods approach that with which functions execute we see no drastic improvement in accuracy.

We find it unsatisfactory that discovering the properties of a well defined system, such as an executing program and its sampling methodology, requires practical intervention. We feel that if equipped with information such as method execution time distributions, that a theoretical analysis could be used to give a better understanding of what kind of sampling strategy is best suited to the target application. This kind of information may eventually be used to drive alterations in profiling strategies based on observed properties.

Appendix A

Code Patches

A.1 Obtaining Code Patches

All code used through this thesis is publicly available online at: `https://github.com/Sauski/high-frequency-profiling`. Two patch files are available, one for the work performed in SpiderMonkey, and one for Jikes RVM. Each of these files requires a specific revision of their respective main branches. As updates are expected to these files, detailed instructions are available in the readme file at the above location.

All of the work on each system has been condensed to a single patch file for each. Different configurations are either accessed through compile time options in SpiderMonkey, or through run time options in Jikes RVM. If for any reasons you are interested in the project, and wish to know more about its development or simply need guidance installing and running, please feel free to contact me at tto.warren@gmail.com.

Bibliography

ALPERN, B., AUGART, S., BLACKBURN, S. M., BUTRICO, M., COCCHI, A., CHENG, P., DOLBY, J., FINK, S. J., M., D. G., HIND, MCKINLEY, K. S., MERGEN, M., MOSS, J. E. B., NGO, T., SARKAR, V., AND TRAPP, M. 2005. The jikes rvm project: Building an open source research community. *IBM System Journal*, 399–418. (p. 31)

ARNOLD, M. AND GROVE, D. 2005. Exploiting high-accuracy call graph profiles in virtual machines. *International Symposium on Code Generation and Optimization*. (pp. 9, 24, 32, 34, 44, 47)

BLACKBURN, S., GARNER, R., HOFFMAN, C., KHAN, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., D.FEINBERG, FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIC, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The dacapo benchmarks: Java benchmarking development and analysis. *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. (pp. 32, 34)

CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., AND HWU, W. W. 1992. Profile-guided automatic inline expansion for c programs. (p. 54)

GOOGLE. 2015. Octane: The javascript benchmark suite for the modern web. https://developers.google.com/octane/?hl=en. (p. 12)

HANSELMAN, S. 2013. Javascript is web assembly language and that's ok. (p. 11)

HARREL, J. 2013. Node.js at paypal. https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/. (p. 11)

HOLZLE, U. AND UNGAR, D. 1994. Optimizing dynamically dispatched calls with run-time type feedback. *ACM SIGPLAN Notices*, 326–336. (p. 8)

INTEL. 2013. Intel 64 and ia-32 architectures developer's manual: Vol. 3a. (p. 7)

INTEL. 2015a. Atomic operations. https://software.intel.com/en-us/node/506090. (p. 44)

INTEL. 2015b. Vtune amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe/details. (p. 6)

LINUX. 2014. Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial/Event-based$_{s}ampling_{o}verview.(p.6)$

MOZILLA. 2015. Spidermonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. (p. 11)

ORACLE. 2015. Class nullpointerexception. http://docs.oracle.com/javase/7/docs/api/java/lang/NullPointer (p. 37)

PIERRON, N. 2014. Ionmonkey: Optimizing away. https://blog.mozilla.org/javascript/2014/07/15/ionmonkey-optimizing-away/. (pp. 11, 18)

SCHNIEDER, S. 2000. Statistical profiling: An analysis. http://www.embedded.com/design/prototyping-and-development/4018371/Statistical-Profiling-An-Analysis. (p. 54)

SPEC. 1998. Specjvm98 benchmarks. http://www.spec.org/jvm98. (p. 34)

SUGANUMA, T., YASUE, T., AND NAKATANI, T. An empirical study of method in-lining for a java just-in-time compiler. (p. 6)

VALGRIND. 2015. Valgrind. http://valgrind.org/. (p. 6)

WEAVER, V. 2015. Self-monitoring overhead of the linux perf event performance counter interface. *ISPASS*. (p. 15)

YANG, X., BLACKBURN, S., AND MCKINLEY, K. 2015. Computer performance microscopy with shim. (pp. 2, 7, 14)

ZAKAS, N. 2010. *High Performance JavaScript*. O'Reilly Press. (p. 11)

ZHUANG, X. AND SERRANO, M. 2006. Accurate, efficient, and adaptive calling context profiling. (p. 9)