Parallelism Detection using Dynamic Instrumentation in a Virtual Machine

Christopher Edward Atkin-Granville

Master of Science
Computer Science
School of Informatics
University of Edinburgh

2013

Abstract

Contemporary computers are increasingly based around highly parallel architectures through chip multiprocessors, instruction-level parallelism and graphics processing units with potentially thousands of cores. Despite this, many popular programs are based around a sequential programming paradigm.

This project investigates the use of the Graal compiler infrastructure in order to dynamically profile running programs within the Java Virtual Machine and determine which hot loops are good candidates for automatic parallelism transformations, possibly JIT recompilation to an OpenCL target. We show that, currently, Graal is not yet mature enough to support the transformations required. However, in the future, it will becomes mature enough and our framework is a simple 'drop-in' to the Graal compilation procedure to add instrumentation to programs.

We consider two main approaches to trace collection: exact approaches and probabilistic approaches based on bloom filters. A new benchmark has been created that allows for fine-tuning of the number and types of dependencies, as well as the amount of computation associated with each operation (in the form of a number of floating-point operations) which allows us to analyse the efficacy of each approach without the disadvantages of real-world benchmarks (i.e., where we cannot know ahead-of-time how many dependencies there are, and so cannot evaluate the performance of each approach).

The findings presented by this dissertation show that not only is it feasible to use bloom filters for these kinds of analysis, but it also advantageous to do so over hash sets. With the correct parameters, bloom filters show a small memory use increase over uninstrumented programs, and a relatively small increase in the execution time of between a factor of 1.4 and 2 times slower.

Acknowledgements

I wish to thank my supervisors, Dr. Christophe Dubach and Dr. Björn Franke for their insightful and valuable contributions to the project.

Also in need of thanks are my friends and especially my parents, Sandra and Ian who have supported me throughout my entire University career.

Lastly, I would like to thank all those folks on the Graal mailing list who answered all my questions regarding the system.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Christopher Edward Atkin-Granville*)

To my grandfather, Leslie.

Table of Contents

List of Figures

Chapter 1

Introduction

In this chapter, some background to the problems of paralleism expression and detection are discovered. We cover why the problem is important, and why a general-purpose solution has not yet been found, despite decades of research into the area. The concepts of static and dynamic analysis are introduced, along with a critical analysis of their advantages and disadvantages. Lastly, we draw a list of contributions which this dissertation provides, as well as outlining the rest of the dissertation.

Ever since the introduction of the first microprocessors in the early 1970s, there has been a trend within the microprocessor industry affectionately called Moore's Law. Although not a law in the proper scientific sense (rather, it is more of an observation of the behaviour of the industry), it does accurately describe the trend of the number of transistors which can be placed in a given area[1]. The trend so far has been that this number double every 18 months. Altogether, Moore's Law successfully described the behaviour of the semiconductor industry until roughly five years ago.

1.1 Golden Age

During this time of rapid advancement, programmers had to expend very little effort in order to improve performance of their programs on newer hardware. In the best-case scenario, literally no change was required whatsoever - not even a recompilation of the program. The underlying hardware was improving, and when one combines this fact with the separation of concern between user-level applications and the underlying hardware (i.e., the abstraction layer that compilers introduce, with a special focus

[1]Which is emphatically *not* the idea that processor speed doubles every 18 months - which is a common misconception. Moore's Law is also applicable to other VLSI products, such as memory.

on ISA abstraction) meant that developers could simply urge their users to buy new hardware for a performance improvement.

In a slightly less-than-ideal scenario, the higher transistor counts being allowed would allow semiconductor designers to add new features to the 'base' instruction set of their choice - for example on x86 there have been several additions over the years (MMX, 3DNow!, SSE, PAE, x86-64 etc). In these cases, programmers would simply need to recompile their programs with a compiler that could take advantage of the new extensions. Platforms supporting just-in-time (JIT) compilation such as Java, C# etc would need to replace existing virtual machines (VMs) with ones capable of using the new instruction sets.

In many ways, this time could be seen as a golden age of computer architecture. Transistors were cheap and plentiful and the promise was always there that next year transistors would be even cheaper and more plentiful. Semiconductor manufacturers started experimenting with radical new designs (not all of which were successful, for example Intel's NetBurst which promised speeds of up to 10GHz by, amongst other techniques, utilising an extremely long pipeline). Consumers were confident that a new machine would be significantly faster than the machine they purchased a mere 12 months prior. Enabled by the new-found performance of processors, application developers would start to introduce many new layers of abstraction (and indirection), which would allow for safer, stabler programs to be written using high-level languages such as Ruby, Python, Perl and PHP. These extremely-high-level languages (sometimes called scripting languages) commonly sacrificed execution speed for programmer ease of use, safety, new features and other such advantages. Indeed, this phenomenon even became widespread in lower-level languages via Java and C#, both of which introduced a virtual machine between the application and the hardware. In many cases, these virtual machines were specifically designed (at least initially) for the languages for which they were designed (in that they were not initially designed to be 'language agnostic'), meaning they may have allowed features that are difficult to implement lower in the stack. For example, the Java Virtual Machine (JVM) includes opcodes such as `invokespecial` (which calls a special class method), `instanceof` and other such codes specifically designed for an object-oriented language[2]. These features are enabled via high-performance processors, and would likely not exist (or certainly, not be mainstream) without these processors.

[2]There is currently an effort to add new instructions to the JVM designed to ease execution of languages with non-object-oriented paradigms

1.2 Cheating the System

However, these increases could not occur indefinitely. There exists not only a fundamental lower-bound on the size of an individual semiconductor-based transistor (as a result of quantum tunnelling), but also the extent to which contemporary techniques can provide performance improvements. For example, many processors exploit instruction-level parallelism (ILP) by executing several instructions at the same time - pipelining. This is achieved by effectively duplicating many stages of the pipeline and the supporting infrastructure. Besides the standard issues with pipelining (data, control and structural hazards spoiling issue flow, multi-cycle instructions spoiling commit flow and the like which can be solved via trace caches, as done in the Pentium 4), there exists a larger problem. As the degree to which ILP is exploited in a processor increases, the complexity of the supporting infrastructure increases combinatorially. Hence, this is clearly not the 'silver bullet' which ILP was once thought to be. The extent to which current processors exploit ILP are not likely to increase significantly in the next several years, barring a revolutionary breakthrough in processor manufacturing, ILP detection/exploitation etc.

About a decade ago, it was a commonly held belief that the path to improving processor performance was to make a single-core processor increasingly powerful [Nayfeh and Olukotun, 1997], through a combination of higher clock speeds (which manifested itself as the so-called 'Megahertz War') and architectural improvements. Although this did come true to an extent (eventually culminating in the 3.8GHz Intel Pentium 4), this period did not yield the kind of performance that was expected (see above). The main reason for this was a simple one - transistors with higher switching frequencies produce more heat. This, when combined with the fact that Moore's Law would allow higher transistor counts per unit area meant that around 2006 to 2007 manufacturers were unable to improve performance much simply through increasing the clockspeed - architectural advances would also be required.

1.3 Hello, Parallelism

There existed no simple solution to this problem. For decades developers were used to having to expend little to no effort to realise potentially significant performance improvements. The solution that industry converged upon was that of parallelism - to improve performance not by increasing the performance of a single processor, but to provide many processors each of which may be slightly slower when taken individu-

ally. When combined together (with a multi-threaded program), the culmination of these processors would be more performant than a single processor could ever be.

Parallelism (and concurrency) was not a new idea. For decades parallelism had been used for the most compute-intensive problems. These kinds of problems are usually 'embarrassingly parallel' - each unit of work is totally independent from all other pieces of work. Example of this include ray-tracing, where each ray can be simulated independently; rasterisation, where each pixel can be computed in parallel and distributed scientific problems such as SETI@Home. Indeed, concurrency has been part of developers standard toolkit for many years since the advent of GUIs. In Java, developers commonly use helper classes such as `SwingWorker` to run compute-intensive GUI tasks in a thread independent from UI event processing in order to prevent the UI 'hanging' when performing long-running computations.

However the level of parallelism present in most applications is fairly superficial. Even using tools such as `SwingWorker` does not introduce a significant level of parallelism. For example, imagine a button that invokes a `SwingWorker` which executes a loop for many iterations. Although that loop is running on a different thread, that loop is *still* executing sequentially. A significant performance improvement could be realised if the developer had introduced structures and processes that allow the loop to be executed in parallel; unfortunately these transformations are non-trivial and hence are usually not performed.

Regardless of the main reason that parallelism hasn't been introduced to any significant degree in programs (i.e., there was not a pressing need to), there are still many barriers to introducing parallelism. The main problem is likely that most developers simply do not have the required education or experience to do so. Parallelism and concurrency introduces many subtle timing errors that appear transiently. Scheduling algorithms are usually non-deterministic, which makes reasoning about them (either formally or informally) difficult. The behaviour of multi-threaded programs can change with varying number of processors.

1.4 Parallelising Compilers

The result of increasing level of parallelism of programs going from simply computing UI events in parallel to performing actual results in parallel is that it becomes substantially harder to write the parallel algorithms. Indeed, many universities do not cover parallel algorithms until final year undergraduate, postgraduate level or in

some cases not at all. This is because designing, analysing and implementing parallel algorithms is much harder to do than writing sequential ones.

In order to overcome this problem, research into the possibility of automatically parallelising compilers was started. Unfortunately, designing a compiler that is capable of parallelising arbitrary loops is arguably an intractable problem; the semantics of imperative/procedural languages do not convey enough semantic information. Additionally, the problem of data dependence became important - if an iteration is dependent on the iterations previous to it, then the iterations cannot be executed out-of-order.

Dependency detection is of critical importance to this field, as a compiler can only parallelise when iterations can be executed out-of-order. The first works in this field were in the area of static analysis. Static analysis is a compile-time process where the compiler attempts to introspect the dependencies between iterations, parallelising if it is able to determine that there is no dependency. Although this approach would appear to be adequate, in-fact it has a significant disadvantage. Static analysis can only reason about dependencies which are present at compile-time – run-time dependencies are not possible. Although for simple programs this is likely not an issue, for more complex programs – which are more likely to benefit from parallelisation – this becomes a significant disadvantage.

An alternative to this approach is *dynamic* analysis – analysis which occurs at run-time. This approach has the advantage of being able to detect dynamic dependencies, but at the cost of increased execution time and memory usage. This dissertation investigates this dynamic analysis through *instrumenting* programs dynamically. This instrumentation collects information about each loop and iteration, and builds data structures for suitable dependency analysis.

Dynamic dependency analysis requires storing the address of each access per iteration, in order to ensure that no two iterations access the same location. Keeping a store of these addresses can require a significant amount of space. This dissertation investigates the use of exact approaches, in the form of sets based on hash tables, against probabilistic approaches based on bloom filters. The use of bloom filters is advantageous because unlike hash tables, they have a space complexity of $S = O(1)$, as opposed to $S = O(n)$ (where n is the number of accesses) for hash tables. However, bloom filters are probabilistic in nature, and this is taken into consideration in our analysis.

1.5 Contributions

- **Comparison between hash sets and bloom filters for dependency analysis**

  The first contribution is the comparison between exact approaches (hash sets) and inexact approaches (bloom filters). The use of bloom filters was not found in a the literature review for this dissertation (section 2). We compare and contrast the relative overheads of using exact and inexact approaches, as well as the impact on the systems ability to detect parallelisable loops.

- **Implementation of a parametric benchmark**

  Also introduced is a parametric benchmark that allows the statistic generation of hazards. This benchmark has been used to not only measure the correctness of the framework, but also measure the relative overhead of the instrumentation. The parametric benchmark will be useful for testing the performance of similar approaches in the future.

- **Dynamic dependency analysis framework**

  Lastly, a framework for dynamically proving dependencies between loop iterations is presented. Although the current framework is 'stand-alone', the framework has been designed to 'drop-into' the just-in-time compiler in the Java Runtime Environment (JRE). An analysis of possible techniques for JRE-hosted runtime detection is also presented.

1.6 Outline

Chapter 1 outlines the problem background and context, including an overview of the current most common approaches to parallelism expression. Chapter 2 describes the previous work both the areas of dynamic parallelism detection and parallelising compilers/runtime systems. Chapter 3 is an outline of the Graal compiler infrastucture, the main tool used in the project. Chapter 4 provides an overview of the possible approaches to instrumenting Java bytecode where appropriate. Chapter 5 introduces the approaches to trace storage from both theoretical and practical perspectives. A software engineering-based overview of the approaches used is also included. Chapter 6 describes the experimental design, configuration and other parameters. Chapter 7 presents the findings and a critical analysis of the work. The last chapter, chapter 8 draws final conclusions about the work, and suggests possible areas of future work in this area.

Chapter 2

Related Work

The idea of an automatic parallelising compiler is not a particularly new one, and indeed has been the focus of much research since the dawn of structured programming with Fortran [Backus, 1979].

In this chapter, some background of both parallelising compilers and parallelism detection will be presented. The areas have a rich and full history spanning many decades (indeed, the objective of automatic parallelising compilers has been sought for many years), so this is a somewhat brief introduction; only the recent major results are considered.

2.1 Dynamic Parallelism and Parallelism Extraction

Runtime systems with this capacity are advantageous because they require no access to the source code. In 2011, Yang et al. [2011] introduced one of the main advances in the field, *Dynamic Binary Parallelisation*. It requires no access to the source code, and instead operates only using object code. The mechanism through which it operates is by detecting hot loops – loops where the program spends a majority of execution time – and parallelises them, executing the parallel versions speculatively. This speculation is likely the cause of the inefficiencies of their approach - using 256 cores they achieved a somewhat negligible performance improvement of just 4.5 times, and the reason for this is likely hazards spoiling the speculation frequently, which requires expensive rollbacks. However, one of the advantages of the approach presented in this dissertation is that it does not require the use of speculative execution - a loop is only executed if it can be proven to contain no inter-iteration dependencies.

The main difference between Yang et al.'s work and the work outlined here is the nature of the dynamic detection. Yang et al. used dynamic trace analysis, which can only identify the hot loops within a program, and not whether the iterations within those loops have dependencies. The work presented in this dissertation *does* determine whether there are inter-loop dependencies, therefore the use of speculative execution is not required. Although we have not yet done so, the addition of hot-loop detection would be a trivial addition to our framework.

Wang et al. [2009] used a technique called *backwards slicing* [Weiser] in order to preserve essential dependence and data flow. The advantage of an approach based on slicing is that it can detect parallelism regardless of the granularity. This is in contrast to the work presented in this dissertation which can currently only detect parallelism at a loop-level. Wang et al. [2009] represents the first work in the area of dynamic parallelisation using a splicing-based approach. An approach based on slicing can determine different 'sections' of a program (such as blocks, if-then-else statements and so on) which can be executed in parallel, allowing the runtime system to execute in parallel. The algorithms in this paper are speculative - if a dependency is detected at runtime, the pararallelisation for both blocks must be rolled back and instead executed in parallel. Wang et al. present a new set of algorithms for extraction of parallelism. These algorithms can:

- Determine loop-unrolling factors to expose maximum parallelism
- Identify backwards slices for hot regions
- Exploit speculation in program slicing
- Group the large number of slices

The approach to slicing is based on static analysis, with an additional speculative execution model. The speculations performed are:

- **Memory speculation**: memory dependencies on operations that occur infrequently (<0.1% according to profiling) are ignored
- **Control speculation**: control flow edges that are rarely executed (<0.5%) are cut
- **Branch prediction**: highly identifiable branches are cut

Performance of the system was adequate, with a roughly 3x performance improvement with unlimited threads, and 1.8x parallelism with four threads. The major disadvantage to this approach is that the complexity of the slicing algorithms increases exponentially with the length of each slice, meaning that only relatively small slices

can be detected and hence parallelised.

An alternative approach was taken by Ketterlin and Clauss. Their technique 'raises' binary code into an intermediate form (similar to how Graal converts bytecode, see chapter 3 for more details), and then applies a well-known parallelising transformation.

Dong et al. present a framework that is capable of dynamic parallelisation for general-purpose graphics processing cards (GPGPU technology). GPGPU technology allows the large number of processors in GPGPUs to be used by the user for executing large programs. These processors are simpler than the general-purpose cores found in normal CPUs, and so there are some constraints placed on programs.

The framework presented takes a binary loop body and generates IR from it, similar to both Graal and Ketterlin and Clauss. Then, the IR is combined with statically-determined loop information to perform static analysis on the code. A translator is then used to transform the sequential C code into C code that can be executed in parallel using CUDA. The framework is advantageous in that it can parallelise multiple nested loops, increasing performance for many applications. However, it relies on static analysis for dependency analysis, and it does not use speculation to increase performance. However, despite these limitations, the results are acceptable, showing performance improvements of 18x for large matrix multiplication, and betwen 15x and 32x for MRI scan analysis. Importantly, such a framework – or indeed, any framework based around executing using GPGPUs – has a efficacy lower-bound placed on it by using a GPGPU. Transferring data to a GPU for execution has large overhead, with a large startup cost. The result of this is that only large loops should be effectively parallelised using GPGPUs; if this is not the case then an otherwise effective parallelisation framework may result in performance decreases.

Another approach taken is the one taken by Tournavitis et al. [2009]. Instead of applying more traditional algorithms to the problem, Tournavitis et al. instead use machine learning, specifically Support Vector Machines. The aims and philosophy behind the paper are similar to this project: Tournavitis et al. argue that static parallelism detection has failed because there is not enough information available at compile-time - the same reasoning that this project assumes.

The paper presents an approach to auto-parallelisation based around a combined profile of statically and dynamically generated metadata, which is then fed into a pre-trained support vector machine (SVM). This SVM provides information about how the loop should be parallelised. This approach is significant because it compromised three approaches - static analysis, dynamic analysis and machine learning. The paper

presents result of up to 96% that of hand-tuned OpenMP code, which shows clearly that the use of machine learning in compilers can result in significant performance advantages.

2.2 Instrumentation

Although in this dissertation we consider dynamic instrumentation, a significant amount of work is available in the literature describing *static instrumentation*. Static instrumentation modifies bytecode directly at load-time and is 'dumb' in the sense that it is possible to only gain access to information available statically. This is unlike dynamic instrumentation which has access to the full control flow graph and other such information.

Here, we consider some of the various different approaches to static instrumentation.

2.2.1 Bytecode Instrumentation

The first approach uses so-called *bytecode instrumentation* (BCI) - that is, modifying the bytecode directly, either at compile-time or runtime. This approach is advantageous in that arbitrary commands can be inserted, and is only subject to the limitations of the bytecode format. In this sense, arbitrary functionality can be inserted into `.class` files. However, it is complex (requiring advanced knowledge of the JVM and bytecode formats), as well as being difficult to use. Graal already performs a lot of the work that would be required with this kind of approach, in that it detects control flow and memory dependencies from bytecode. Such systems require a large degree of programmer effort.

2.2.1.1 Java Agents

At the heart of BCI is the idea of Java Agents [Javabeats.com, 2012]. In order to understand them, however, one must first understand some details of the Java platform.

Unlike some other languages (for example, C and C++[1]), Java is a dynamically linked language. That is to say that the various different libraries (JARs) that Java programs used are linked at run-time, rather than compile-time. The advantage to this approach is that it allows distributables to be smaller in size (recall Java Network Launching

[1] Although note that C and C++ *also* support dynamic linking

Protocol, a method for launching (and therefore, distributing) Java applications over the Internet). The disadvantage to this approach is that it can lead to 'dependency hell', although through the combination of versioning metadata in JARs and the extreme backwards compatability mantra in Java, this is not currently a significant issue.

There are three main class loaders in Java:

- The system class loader loads the classes found in the `java.lang` package
- Any extensions to Java are loaded via the extension loader
- JARs found within the class path are loaded with the lowest precedence, these include the majority of user-level libraries

Java Agents manipulate class files at load-time, through the `java.lang.instrument` package. The package defines the `ClassFileTransformer` interface, which provides implementations of class transformers. An advantage of this approach is that since Java Agents are included in the core Java package, developers wishing to use the framework would not need to download any additional libraries or frameworks (i.e., Graal).

This mechanism allows agents to dynamically alter the bytecode at load-time, a kind of polymorphism. Such transformations could be utilised in modifying the bytecode of, e.g. array accesses in loops to include calls to instrumentation.

There are several different libraries which provide an abstraction layer for such bytecode transformations. The following sections describe various different libraries that could be used (or use themselves) for agent-based bytecode instrumentation.

2.2.1.2 ASM

Despite Java Agents providing the capability to manipulate raw Java bytecode (indeed, the bytecode is made available as a `byte` array), performing such transformations is difficult and awkward. For this reason, there exists many different libraries for manipulating Java bytecode, ObjectWeb ASM being one of them.

ASM [Bruneton et al., 2002] is a simple-to-use bytecode manipulation library, itself written in Java. It uses a high-level abstraction for the bytecode, which is advantageous because it allows developers to remain unconcerned with the specifics of control flow analysis, dependency analysis and other such concerns.

Although now common, when ASM was first developed it was considered particularly innovative because it allowed the use of the visitor pattern [Gamma et al., 2012, p. 331]

for traversing bytecode. The visitor pattern 'allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure' [McDonald, 2008]. The advantage to this approach is that it allows a user to walk a serialized object graph *without* de-serialising it or defining large numbers of classes (for reference, an alternative to ASM for bytecode generation, BECL [Apache Foundation, 2013a] contains 270 classes for representing each bytecode). Additionally, it allows users to also reconstruct a modified version of the graph (in ASM, graphs are immutable).

Several well-known existing projects use ASM already for bytecode generation, including the Groovy programming language [Strachan and The Groovy Project, 2013].

ASM was selected as a possible alternative for this project for several reasons. Firstly, its visitor pattern-based approach to bytecode generation/modification is high-level and easily understandable. It also has high performance, being a factor of 12 more performance than BECL for serialisation/deserialisation, and a factor of 35 times more performance than BECL for computing maximum stack frame sizes. ASM is also superior to BECL for performing modifications, although with a significantly lower margin of just a factor of four.

The reason for this performance improvement is likely the way ASM and BECL are designed. BECL follows a strict, classical interpretation of object-oriented design principles. Although 'good' software design, it is well known that object models have considerable overhead.

2.2.1.3 Javassist

Javassist [Chiba, 1998] is similar to ASM in that it is also a library for manipulating bytecode, but its method of operation is significantly different. It allows for run-time polymorphism, by dynamically switching implementation of classes at run-time.

There are also additional libraries for manipulating Java bytecode, but due to their features (or lack of), they were not considered for this project.

2.2.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [Kiczales, 1996] is another dialect of object-oriented programming that aims to significantly increase separation of concern within programs, so that programs are more loosely coupled. AOP is a direct descendent from object-oriented programming as well as reflection. Reflection allows programmers to

dynamically introspect classes at run-time; changing values and so on. AOP takes this to another level, by allowing so-called *advice* to be specified (essentially the additional behaviour to be added) and added to *join points*, which are arbitrary points of control flow within the program.

When combined, aspect-oriented systems add these behaviours to the program in question at compile-time through a process called *weaving*.

To illustrate this concept, consider the problem of logging method calls. In traditional systems, at each function/method definition the programmer would need to add specific logging code:

```
def function name(...) {
    if (DEBUG)
        println("function called at " + time());

    // other statements
}
```

Listing 2.1: Traditional use of advice in programs

This behaviour is called an *aspect* (an area of a program which may be repeated several times which is unrelated to the purpose of the program). If the behaviour is to change (e.g. for example, changing the call to `date()` to a call to `time()` instead), each method declaration must be changed manually - a time consuming and potentially error-prone task.

Instead, the use of aspects allows the programmer to remove this functionality, and combine a pointcut and advice into an *aspect*:

```
def aspect TraceMethods {
    def pointcut method-call: execution.in(*)
        and not(flow.in(this));

    before method-call {
        println("function called at " + time());
    }
}
```

Listing 2.2: AOP-based advice equivalent to listing 2.1

Although that from a software engineering perspective this is clearly a superior

solution (as it decreases coupling, increases reuse, increases separation of concern), the use of aspects has not been widely adopted. There are several likely causes for this, such as:

- **Lack of education**: widespread adoption of new programming constructs requires that the average programmer can understand the feature without in-depth education in the model. AOP is somewhat counter to intuitive definitions of imperative or procedural languages, which hampers their adoption
- **Lack of language support**: no widely adopted programming language comes with AOP included, or with an AOP library included in the standard library. Standard licensing issues also apply to third-party additions (e.g. GPLv2/3 differences)
- **Unclear flow control**: perhaps the single largest issue with AOP. As noted by Constantinides et al. [2004], aspects introduce effectively unconditional branches into code, mimicking the use of `goto` which Dijkstra famously considered harmful [Dijkstra, 1968]
- **Unintended consequences**: defining aspects incorrectly can lead to incorrect (global) state, e.g. renaming methods and so on. If a team of developers are unaware of each other's modifications at weave-time, there may unintended consequences and subtle (or substantial) bugs introduced

2.2.3 AspectJ/ABC

AspectJ [Kiczales et al., 2001] is an extension to the Java language that adds aspect-oriented features. It is a project of the Eclipse Foundation (of Eclipse IDE fame). The usage of AOP within Java is a somewhat natural extension as aspects can be seen as the modularisation of behaviour (concerns) over several classes - and not to forget that AOP was originally developed as an extension to object-oriented languages.

2.2.3.1 Array and Loop Pointcuts

However, the limitations of the AspectJ join-point model are somewhat obvious for this project. To be specific, 'vanilla' AspectJ cannot define point cuts for neither array accesses or loops - a combination of which would be required for this project. In addition, the vanilla AspectJ implementation is not particularly extensible, which means that defining new point-cuts is somewhat difficult.

There is, however, an implementation of AspectJ which *is* designed to be more extensible and compatible (mostly) with the original AspectJ implementation - abc, the AspectBench Compiler for AspectJ [Allan et al., 2005].

Although abc itself does not include point-cuts for either array access or loops, there exists two projects which, if combined, could offer the required features for this project.

LoopsAJ [Harbulot and Gurd, 2005] is an extension to abc that adds a loop join point. This is not a trivial addition - when loops are compiled, they are compiled to forms that lose loop semantics (and instead use `goto` instructions). There are several forms that a loop can take, and a significant proportion of Harbulot and Gurd's work is in the identification of loops from the bytecode.

For array access, the ArrayPT project [Chen and Chien, 2007] adds additional array access capabilities to abc. Although the included point cut does include array access, it is somewhat limited and cannot determine either the index, nor the value to be stored. ArrayPT adds these capabilities to abc. ArrayPT defines two new point cuts, `arrayset(signature)` and `arrayget(signature)`. ArrayPT relies on the `invokevirtual` bytecode in the JVM.

It is anticipated that, if these projects are combined, it would present a feasible approach to instrumentation.

2.2.4 Hybrid Models

2.2.4.1 DiSL

Recently, there has been renewed interest in Java bytecode instrumentation. Clearly, the use of aspect-oriented techniques is advantageous, but the current implementations (AspectJ/abc) are deeply flawed. In a sense, they are *static* - they rely on predefined join and point-cuts before any aspect definitions can be constructed. DiSL is considered a hybrid approach because, unlike AspectJ which relies on access to source code, it uses an agents-based approach to aspect-oriented programming.

DiSL (*Domain Specific Language for Instrumentation*) [Marek et al., 2012] is a new approach to a domain-specific language (which incidentally, implies that DiSL is declarative) for bytecode instrumentation. It does rely on the use of aspects, but it instead uses an open join-point model where any area of bytecode can be instrumented.

There are several key advantages to DiSL over other, existing AOP implementations.

- Lower overheads
- Greater expressibility of aspect and join-point definition
- Greater code coverage
- Efficient synthetic local variables for data exchange between join-points

As opposed to AspectJ, which requires compile-time definition of join-points, DiSL uses an open-ended join point format which can be evaluated at weave-time. This allows arbitrary regions of bytecodes to be used as join points. *Markers* are used to specify such bytecode regions (markers are included for common join points, such as method calls and, unusually, exception handling - a novel addition to aspect-systems in Java although control-flow analysis can be used to implement user-defined markers), while *guards* allow users to further restrict selected join-points. Guards are essentially predicates which have access to only static information which can be evaluated at weave-time.

DiSL implements advice in the form of *code snippets*. Note the distinction between DiSL snippets and Graal snippets - although they are similar, DiSL snippets allow arbitrary behaviour to be inserted whilst Graal snippets are used to mainly lower complex bytecodes into simpler ones. Unlike other aspect-systems, DiSL does not support 'around' advice. However, this is not usually regarded as a disadvantage per-se as synthetic local variables mitigate this.

The semantics of snippets and guards is novel in DiSL. Both have complete access to local static (i.e., weave-time) reflective join-point information, meaning they can make (theoretically) unbounded numbers of references to static contexts. In addition, snippets have access to dynamic (i.e., run-time) information, including local variables and the operand stack.

Marek et al. present benchmarks of overheads with DiSL versus AspectJ, and their results are promising - a factor of three lower overhead, yet DiSL manages greater code coverage than AspectJ (the number of join-points captured is greater).

In conclusion, DiSL represents a significant advancement in aspect-systems in general. DiSL allows many semantics of dynamically-typed languages to be expressed in the (statically-typed) Java language.

2.2.4.2 Turbo DiSL

An extension to DiSL, Turbo DiSL has been proposed by Furia and Nanz [2012, p. 353-368]. Turbo DiSL is essentially an optimiser for DiSL which processes the bytecode produced by 'vanilla' DiSL.

There are several advantages of Turbo DiSL over DiSL. For example, instead of requiring expressions to be placed into separate classes, Turbo DiSL allows these expressions to be placed in the same class, increasing maintainability. Turbo DiSL also performs some standard compiler optimisations on DiSL-generated code, such as pattern-based code simplification, constant propagation and conditional reduction. These are supported by a novel partial evaluation algorithm.

Turbo DiSL implements conditional reduction using partial evaluation. Many conditional control-flow statement expressions can be evaluated at weave-time – Turbo DiSL removes these dead blocks. DiSL replaces these with `pop` commands[2], resulting in program correctness remaining unchanged.

In addition, an approach similar to peephole-based optimisation. For example, Turbo DiSL reduces unrequired instruction such as jumping to the next instruction, or optimising the conditional reduction effects. For each `pop` instruction found, the source bytecodes are found (i.e., which bytecodes push the to-be-popped operands). If those bytecodes are side-effect free, they are both (the pop and the source) removed.

The authors present an analysis of Turbo DiSL performance characteristics. The benchmarks selected were from the DaCapo benchmarks [Blackburn et al., 2006]. There is a considerable increase in weave-time of a factor of 7.64 above the baseline, which clearly shows the drawbacks of partial evaluation. However, Turbo DiSL outperforms DiSL by a factor of 5.18 and 13 for startup and steady-state respectively - a considerable improvement.

The authors present several uses cases where Turbo DiSl is superior to DiSL (dynamic instrument configuration, tracking monitor ownership, field access analysis and execution trace profiling). However, this author speculates that, in spite of the aforementioned increase in weave-time, Turbo DiSL will completely supersede DiSL in all situations.

[2] `http://homepages.inf.ed.ac.uk/kwxm/JVM/pop.html`

2.3 Summary

In this chapter, we have given a review of the previous work related to this dissertation. Some recent examples of systems utilising various forms of parallelism detection have been introduced, as well as a critical analysis of their strengths and weaknesses.

Secondly, we included an overview of the various (alternative) approaches that could have been used to implement instrumentation.

In the next chapter, we will move away from the static instrumentation presented in this chapter, and on to dynamic instrumentation using a new compiler framework called Graal.

Chapter 3

The Graal Compiler Infrastructure

Graal is a new approach to Java compiler engineering. Here, Graal is introduced and technically assessed. It's strengths and weaknesses are also evaluated.

3.1 Background

The basis of the project is the Graal compiler infrastructure [Oracle and OpenJDK, 2012]. Graal is an experimental project developed mainly at Oracle Labs (although there are some additional collaborators at AMD) under the OpenJDK programme.

Graal is, in essence, a Java compiler written in Java. However, this doesn't fully explain the Graal project - '*a quest for the JVM to leverage its own J*'.

Virtually all languages used commonly in industry have had their compilers go through a so-called 'bootstrap' process. This bootstrapping process involves writing the compiler for a language in the language it is intended to compile. In many cases the first version of a compiler is written in a different language - commonly used languages include the standard C and C++ due to their performance. There are many examples of this in the real-world - GCC is written in a combination of C and C++[1], LLVM/Clang is written in C++ etc. There are several advantages to this approach - in essence, this process is a kind of informal proof that the language has matured to a level that is capable of supporting a program as complicated as a compiler. In effect, bootstrapping a compiler shows that a language (and associated platform) has a certain level of 'maturity'; that it is now ready for large software projects (or at least, not totally unprepared for them).

[1] Although the project is currently converting all C code to C++

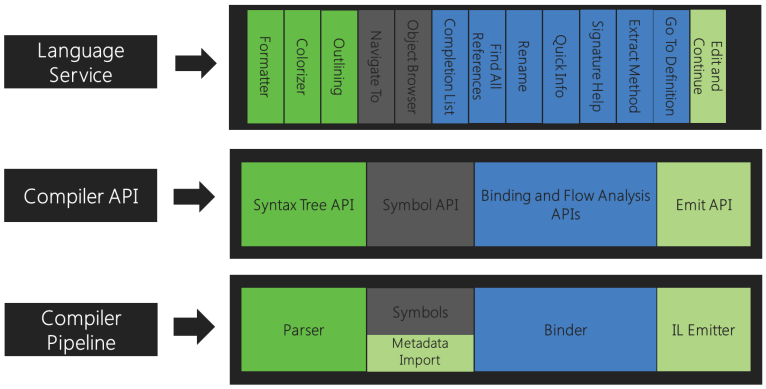

Figure 3.1: The compiler pipeline in Microsoft Roslyn, a similar project to Graal. Roslyn operates at a higher level than Graal, instead manipulating abstract syntax trees. [Figure Ros]

Graal is an attempt to bring this approach to the Java Virtual Machine. Note that there are other projects with attempts to bootstrap parts of the Java platform - for example, the Maxine VM is a Java virtual machine written in Java. However, because Java is unlike most other platforms (in that it not only requires a compiler, but also a virtual machine, class library and such), the compiler has, until the advent of the Graal project, remained written in C++.

Another feature of Graal is that it allows users (of the compiler) to interface directly with the compiler. Common compilers (GCC, ICC etc) are seen as 'black-boxes', where a user invokes the compiler, waits for a while, and then a resulting object file or binary is produced. Graal is a part of a new generation of compilers that expose APIs to users, which means users can change parts of the compilation process to suit their needs, ease debugging and other such advantages. With modern languages and platforms being required to target multiple different machine classes (module, desktop, laptop and server/cloud), this is a crucial advantage over more conventional languages and platforms. There are only a few examples of this new generation of compiler, but another - somewhat more mainstream example - is Microsoft's Rosyln project [Ros] for their .NET platform. The new Windows Runtime includes deep metadata integration into the platform (which is the basis for, amongst other things, the Common Object System in .NET languages[2]); this metadata is available to users of the compiler through Roslyn.

[2]Which allows inter-language types to be considered equivalent - a C int is semantically equivalent to a C++ int, a C# int, a JavaScript int and so on

3.2 Introduction

Graal is somewhat different than other compilers. As opposed to other compilers, which use a combination of parsers and lexers to produce their IRs from source code, Graal builds the IR from Java bytecode instead. This approach has several advantages for this project, the main being that we cannot assume that the source code is available to many legacy programs. Another advantage to this approach is that it would allow the detection mechanism to not only be performed upon user-provided programs, but also system-level libraries as well (for example, the Java Collections Framework). Lastly, it would allow the instrumentation of all JVM languages, not just Java. There are many languages which have been designed for the JVM, such as Scala, Groovy and Clojure, and there are also some languages which have been ported to the JVM such as Python (through Jython), Ruby (through JRuby) and PHP (through Quercus).

3.3 Intermediate Representations

Like many compilers, Graal uses several different internal representations at different stages of compilation. Each of the representations has a distinct, non-overlapping use case; despite this the graphs are somewhat similar in structure.

As is common in many compilers, Graal uses graphs for intermediate representations. These graphs combine several different kinds of graph together into a single form, such as control flow and memory dependency (data flow).

To illustrate this, consider figure 3.2, created using the Ideal Graph Visualiser (*IGV*).

The format for graph visualisations is the following:

- Red edges represent control flow
- Blue edges represent memory dependence
- Black edges are defined as edges which are not control flow or memory dependence. In reality, they are mainly used for associating `FrameState` nodes where appropriate

The text inside nodes uses the following format:

`<node-id> NodeName <additional>`

In some cases, `<additional>` contains the value associated with the code (for subtypes of `ValueNode`). Node IDs are unrelated to the ordering within the graph. Time is

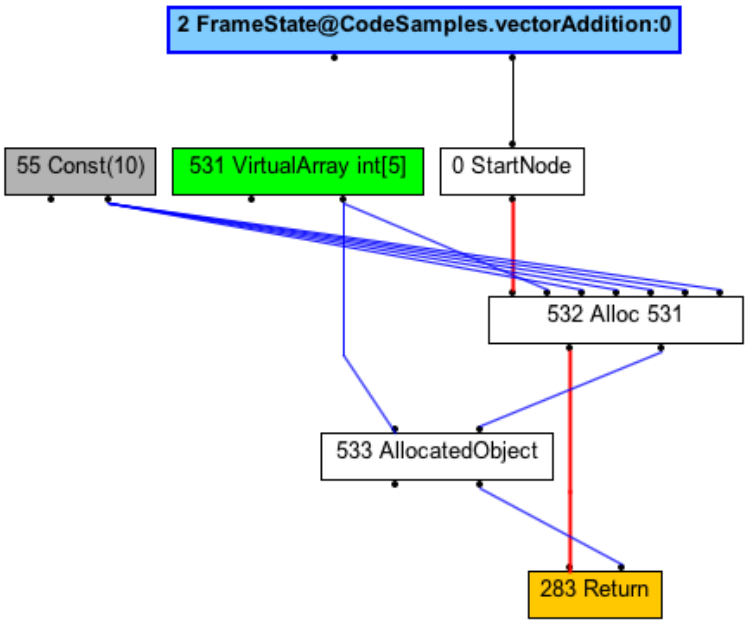

Figure 3.2: Graal HIR created for a vector addition using two array literals

represented on the y-axis of the graph, flowing down the page (so in the case of figure 3.2, node 0 (`StartNode`) is executed first, followed by node 532 and 283 in that sequence).

Figure 3.2 is a representation of a vector addition using two `int` array literals as arguments. The original source code contains a loop, which itself contains two array load operations, an addition and an array store operation. From it, we can see clearly the different kinds of node relationships in Graal's IR. The `StartNode` is the start of the method, which is followed by an allocation (notice that the loop was been optimized away by Graal), the result of which is then returned from the method.

The allocate has two dependencies:

- A `VirtualArray` node is used to create an array. We can see that the type of this array is `int[]` with length five.
- A `ConstantNode` is referenced five times, one for each position of the array.

Without both of these dependencies, the compiler cannot create the array, and assign the correct values. Note that, because the JVM allocates default values of zero to declared-yet-unassigned variables, if the `ConstantNode` was not used, the array would consist of zeros.

3.4 Graph Transformations

One of the ways the abilities of Graal are manifested is through it's capacity to apply transformations to the various intermediate representations. The main use for this is to allow users of the compiler to add custom behaviour at the various stages of the compilation process, but the mechanism extends to additional uses. For example, custom behaviour can be inserted into programs, as well as the graphs dumped for inspection in external tools.

As of the time of writing, Graal uses a hybrid approach to transforming graphs between the IRs - suites and phases. They are, in effect, essentially the same thing - they both consist of a sequence of transformations (in Graal terminology, a *phase*) which are applied to a graph in a well-defined order (although the user can change the ordering if required, as well as disable certain phases[3]. The result of a sequence of phases is the representation used at the next lower-level of abstraction. The three phases in Graal are high-level, mid-level, and low-level.

Todo: clear up the difference between runtime-specific lowering and target-specific lowering.

The high-level phase is used mainly for the graph-building phase. Graal uses the concept of `ResolvedJavaMethod`s, which are internal 'linkages' to constant pools found within `.class` files. It is also used for runtime-specific lowering[4], for example to handle multiple machine instruction set architectures - although the Java language is unconcerned with JVM implementation details such as the endianness of the machine's CPU, the runtime must, by definition, be aware of this fact. Examples of runtime-specific lowering are found throughout Graal, but examples include multiple implementations of the `GraalRuntime` interface - one for each ISA that Graal supports. Single Static Assignment (SSA) form is used at the high-level in order to allow for optimisations to be performed.

Mid-level phases remove the SSA form, and includes target-specific lowering. The low-level phases are analogous to the backends of more traditional compilers, and deals with low-level issues such as register allocation and code generation.

It is important to note that lowering is an iterative process, because a given lowering phase may result in further 'lowerable' nodes being added to the graph.

There are two main classes of nodes in Graal, and they are split into nodes which implement `Lowerable` and those which implement `LIRLowerable`. `Lowerable` nodes

[3] A common usage for this is to disable the inlining optimisation.

[4] *Lowering* is a mechanism to convert a complex bytecode into a simpler one, much like the difference between RISC and CISC architectures

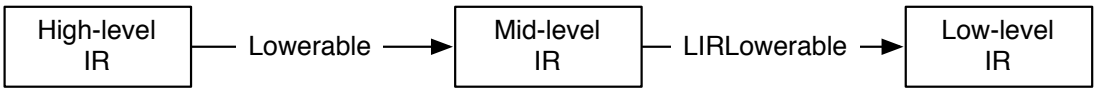

Figure 3.3: Relationships between IR levels and lowering types in Graal

are transformed between the high-level and mid-level, and `LIRLowerable` nodes are transformed between the mid-level and low-level – figure 3.3 shows this diagrammatically.

3.4.1 The `.class` File Format - Constant Pools

In order to properly understand `ResolvedJavaMethod`s, one needs to understand some parts of the `.class` file format. The source for this section is the Java Virtual Machine specification [Lindholm et al., 2013, p. 69].

`.class` files are structured containers for Java bytecode streams. However, they are not 'plain old data structures', as would be indicated by that description. Instead, they are laid out in such a way to increase performance for the JVM.

Unlike some other executable file formats, the JVM does not rely on the (relative) positions of the various kinds of definition permitted. In this context, constants refer to *all* immutable identifiers - and not simply to the language-level construct of constants (e.g., `public static final int THOMAS_KLAUS = 9;`). Each constant has an entry in the *constant pool* - a table containing `cp_info` structures. A `ResolvedJavaMethod` contains a link to the offset of a `cp_info` structure for a method.

3.5 Snippets

In order to engineer the lowering and optimisations mechanisms in a way that follows 'good'[5] manner that follows solid software engineering principles (low code reproduction, cohesion and linkage etc.), Graal uses the snippets mechanism.

Snippets are Graal graphs represented as Java source methods. They are used for lowering bytecodes with runtime-dependent semantics, such as `CHECKCAST`[6]. Snippets allow certain complex bytecodes to be replaced with simpler ones. Furthermore, snippets have the additional constraint that they must be deoptimisation (see section 3.7) free.

[5] For some definition of 'good'

[6] `http://homepages.inf.ed.ac.uk/kwxm/JVM/checkcast.html`

Each class of snippets has an associated template, which provides the mechanism through which snippets are instantiated, and the graph modified. These are usually made available through a static nested class of the snippet class. Every time a snippet is activated, the template class creates a template for the snippet, adds arguments (via the `Arguments` class), and then modifies the graph, replacing the target bytecodes with the snippet.

Snippets are highly related to lowering, as they are used to lower nodes between the different IRs.

3.6 Replacements

In this feature, which is perhaps unique amongst compiler implementations, Graal allows the user to dynamically change implementations of methods at compile-time – in other words, Graal allows compile-time method polymorphism. In a sense, replacements are similar to snippets (section 3.5), in that they both represent graph replacements. They differ in that snippets are a low-level abstraction, used for replacing bytecodes with other bytecodes, whereas replacements are for replacing method implementations.

The idea is simple: to replace the body of a method dynamically, at compile-time. This requires performing a compilation on both the replacer and replacee. The name and signatures of the methods must be identical.

This mechanism is exposed via both the `Replacements` interface, and the `@MethodSubstitution` annotation. This annotation includes various metadata, such as whether the replaced method should be substituted in all cases, and the identifier and signature of the method that will be replaced. The class which this is defined is then passed to an implementation of the `Replacements` interface, which performs a runtime-appropriate transformation.

In all cases, however, the basic idea of the transformation is the same. Both methods are compiled, as well as the surrounding pre-call and post-return of the replacee[7]. The nodes for the method starts and returns (or end nodes) are replaced until the transformation is complete. Figure 3.4 displays this mechanism.

[7] Of course, at this stage this compilation should have *already* been performed

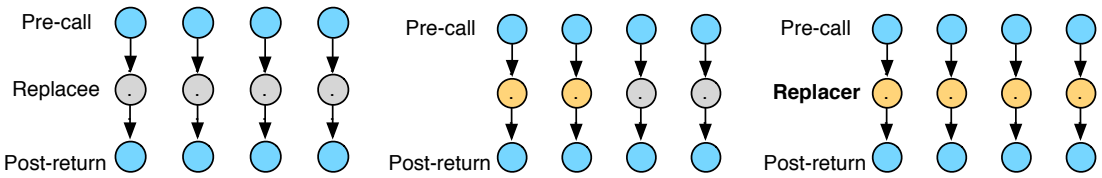

Figure 3.4: Method substitution in Graal

3.7 Optimisations and Deoptimisations

For maximum performance, Graal applies optimisations aggressively and speculatively. The potential performance advantages of speculative optimisation outweighs the possible downside of occasionally requiring deoptimisation. In addition, probability of certain blocks being executed can be computed at compile-time, which in-essence provides Graal with additional metadata regarding which blocks it may prove fruitful to optimise. If a loop is to be executed 100 times, with a low probability of exception-causing errors being thrown per iteration, it makes sense to optimise the loop for the cases where no exception will be thrown. The performance advantage of doing so will be greater than the performance decrease as a result of deoptimisation in the few(er) cases where it will be required.

Some of the optimisations that Graal applies are described here.

- **Dead and unreachable code elimination**

  Dead code is code that may be executed, but has no effect on program correctness or semantics (i.e., it has no meaning [Aho et al., 2007, p. 533]). Code is said to be unreachable if there are no possible inputs which would result in execution of the block the statements exist in.

- **Type-checked method inlining**

  Method inlining (also called *procedure integration*) replaces calls to procedures with copies of their bodies [Muchnick, 1997, p. 465]. However, in some cases the result of these transformations may not be type-safe, even in statically-typed languages such as Java [Glew and Palsberg, 2002]. However, transformations are possible that ensure these method inlings are type-safe.

- **Probability of exception throwing**

  Many nodes in Graal have an associated *probability* - i.e., the probability of their execution, and hence, the basic block within which they lie.

  To illustrate, imagine the following loop:

```
for (int i = 1 to 100) {
    if (i == 100)
        doSomethingStrange();
    else
        doSomethingNormal();
}
```

In 99% of cases, `doSomethingNormal()` will be executed, and in 1% of cases `doSomethingStrange()` will be executed.

This behaviour can be used to speculatively merge blocks into *extended basic blocks* - a maximal sequence of instructions beginning with a leader that contains no join nodes other than its first node [Muchnick, 1997, p. 175]. These extended basic blocks can then be optimised in the same way that 'normal' blocks can be.

If the above example were used, it could optimise for the `doSomethingNormal()` calls, and use a deoptimisation for the call to `doSomethingStrange()`.

Probabilities are computed by post-order graph traversal. When it encounters control flow nodes, Graal uses information about the split to divide probability between the successors. At merge nodes, Graal sums the probability of all predecessors. Lastly, loop frequencies are propagated and each fixed node within loops have their probabilities multiplied by the loop's frequency.

Note: Graal also includes references to '`UseExceptionProbabilityForOperations`', but this feature has not yet been completed.

- **Loop limit checks**

  This refers to the practice of "inserting a predicate on the entry path of a loop and raising a common trap if the check or condition fails" [Peng, 2010]. Loops are split from a single block into several blocks containing a pre-, main and post-loop.

  The advantages of approaches based around this technique are three-fold:

  - Loop predication can be applied to outer loops without code size increment
  - Checks can be eliminated from the entire iteration space
  - The approaches can be applied to loops with calls

However, there are some cases where an optimisation has been applied, but it leaves programs in an inconsistent state. In this case, a deoptimisation occurs. More specifically, deoptimisation [Hölzle et al., 1992] is the process of converting an optimised stack

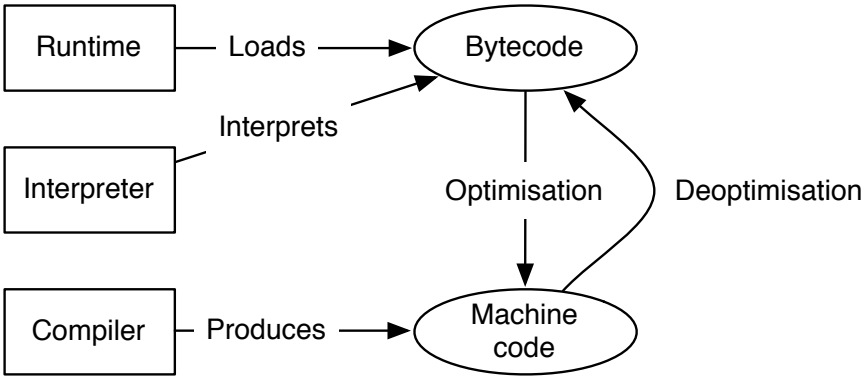

Figure 3.5: Relationship between optimisations and deoptimisations [adapted from Schwaighofer, 2009, p. 24]

frame into an unoptimised one. In HotSpot, an optimised frame is the result of the JIT compiler and unoptimised frames run using the interpreter. Figure 3.5 shows the interplay between optimisations and deoptimisations. Optimisation-deoptimisation is controlled through an assumption mechanism - in order to be (or remain) optimised, a set of assumptions must remain true. If any of the assumptions becomes false, the optimisation is broken and deoptimiastion occurs. When deoptimisation occurs, the runtime returns to the bytecode index associated with the optimisation, and executes using the interpreter instead. Thus, a bytecode index is critical to the correct functioning of the deoptimisation mechanism (and, transitively, program correctness). This technique is used because Graal currently relies on the deoptimisaton framework in HotSpot™, which uses the approach described here.

Deoptimisations can occur for several reasons:

- **Exceptions**: certain kinds of exceptions can trigger deoptimisations as they disrupt control flow. These exceptions are `NullCheckException`, `BoundsCheckException`, `ClassCastException`, `ArrayStoreException` and `ArithmeticException`. Additionally, if an exception handler has not been compiled, a deoptimisation will occur.

- **Violated assumptions**: if any of the assumptions made during optimisation are violated (i.e., their state is modified), the optimisation is marked as invalidated. One way these assumptions can be broken is via class loading.

- **Unreached code**: if there is unreached code that could not be determined statically, this code is not optimised.

3.8 Summary

In this section, we have introduced the Graal compiler infrastructure and some of the advantages that it brings over more traditional compilers. Graal is at the forefront of compiler technology, and is part of a new generation of compilers that provide user-facing APIs. Graal also optimises speculatively and aggressively through a deoptimisation-based model; some of the optimisations and possible reasons for deoptimisations have been introduced.

Graal is distinguished from other compilers by its flexibility. It is designed to allow users to use the compiler-as-a-service, a new paradigm in compiler technology. Compilers-as-a-service promise a new level of abstraction for compiler technology, with use cases such as code analysis, online refactoring and code generation. Until now, compilers have been a black box which users have little say over the internal workings of. In the future, this will no longer be the case - compilers will adapt to suit the users and programs requirements, and Graal is the first step in this area for the Java world.

Chapter 4

Dynamic Instrumentation

4.1 Introduction

There are two main approaches to dynamic instrumentation: automatic, and manual. We consider both of them, using Graal for an automatic, dynamic approach. We conclude by showing that, currently, it is not yet possible to use Graal for automatic instrumentation – the reason for this is explained. As an alternative, manual instrumentation is used, and we explain the disadvantages and impact of this approach.

4.2 Automatic Approaches - Graal

In this context, an automatic approach to instrumentation is a technique that can be applied directly at compile-time (or just before run-time in an agent-like fashion). Such approaches require no human intervention whatsoever, and have the additional advantage of having access to additional compile-time meta-data which is not possible to gain at run-time or using manual instrumentation.

An automatic approach based on Graal was investigated. Graal was chosen because it is the most flexible approach: it allows the combination of compile-time evaluation, as well as run-time evaluation of the dependency algorithms.

As mentioned, Graal uses various different forms of intermediate representation, based on graphs (see section 3.3). In principle, it should be possible to modify these graphs in order to invoke the instrumentation (described in chapter 5).

Modifying graphs in Graal is made easy through a simple API. A user can define custom phases (see section 3.4). Pseudocode for this is as follows:

```
1    // create the phase
2    public class MyPhase extends Phase {
3        @Override
4        protected void run(StructuredGraph graph) {
5            // modify the graph
6            for (LoadIndexedNode node:
7                graph.getNodes(LoadIndexedNode.class)) {
8                graph.addAfterFixed(node,
9                    graph.add(new MyCustomNode()));
10           }
11       }
12   }
13
14   // add the phase
15   Suites s = Graal.getRequiredCapability(SuitesProvider.
     class)
16       .createSuites();
17   s.getHighTier().appendPhase(new MyPhase());
18
```

Listing 4.1: Sample code for adding a phase and manipulating a graph

The call to `getHighTier()` can be replaced with equivalent methods for the other IRs (i.e., `getLowTier()` and `getMidTier()`.

As we can see in figure 4.1, which displays the high-level graph for a simple method that takes an array actual parameter and returns index 0 of that array, there are specialised nodes for array accesses: `LoadIndexedNode`. Note that there is also an equivalent for array store operations, `StoreIndexedNode`.

This is the basis for selecting the appropriate nodes to instrument. One additional advantage of Graal is the richness of the information available at compile-time. Figure 4.2 shows all the available information, as seen in Eclipse's debugger.

The availability of predecessor and next nodes, as well as nodes representing indexes (in this case, a `ConstantNode`) highlights another advantage of using Graal for these transformations - static analysis is possible, which means that instrumentation can be disabled if dependencies can be proven statically.

There are several possibilities for adding instrumentation in Graal.

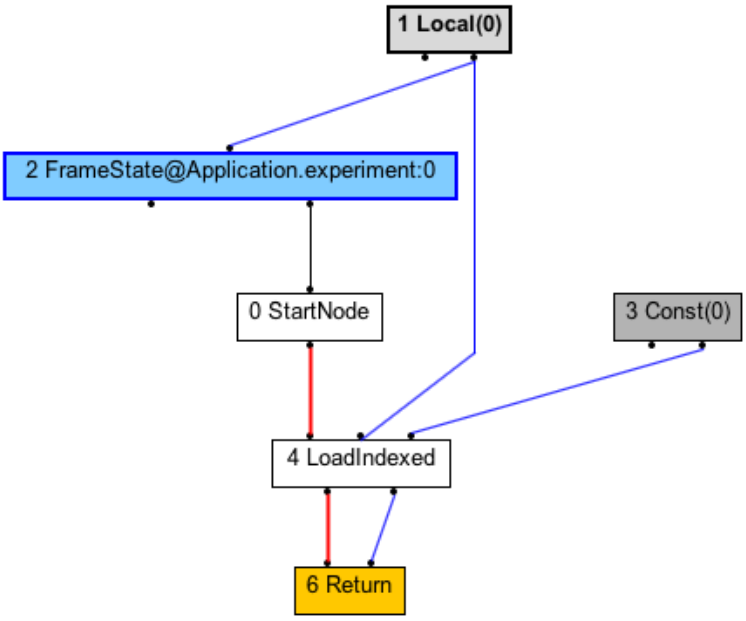

Figure 4.1: A high-level graph (with inlining disabled) for a simple method taking an array actual parameter, returning index 0

1. Add a custom node type, which is lowered to an invoke instruction after each applicable node.

2. Replace the nodes in question with a custom node that implements the same operation in addition to the required behaviour.

3. Javassist includes a method for replacing all array accesses with method calls [Javassist Project, 2013]. It may be possible to use Graal to replace the method call targets where instrumentation is required. In cases where no instrumentation is required, the method would return the array value (or perform the store), which would then be inlined by the compiler.

In cases 1 and 2, then node would need to implement `Lowerable`, an interface in Graal that allows nodes to be lowered between IR levels.

However, there is currently a limitation in Graal which means that it is not currently possible to insert calls to (static) methods. This is because there is no *bytecode index* (or BCI) for the interpreter to return to if a deoptimisation occurs (details on Graal's deoptimisation mechanisms are available in section 3.7). Since methods calls cannot be guaranteed to be deoptimisation-free, they cannot be inserted into graphs. Any static methods that modify abstract data types (i.e., the storage formats described in section 5.2) cannot be assumed to be deoptimisation free. Modifying a data structure

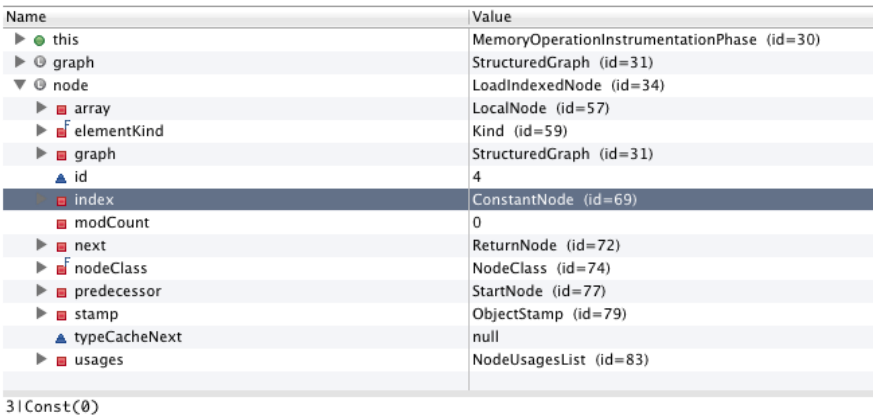

Figure 4.2: Information available at compile-time for the `LoadIndexedNode` shown in figure 4.1

violates the optimisation assumptions, causing a deoptimisation. The interpreter then tries to resume from the BCI of the invoke. However, there is no BCI associated with inserted invocation, meaning that the interpreter cannot resume in the event of the inevitable deoptimisation. The end result is that insertation of arbitrary behaviour through invoke nodes to static methods is not currently possible in the Graal system.

To illustrate the concept of BCIs, consider the following example. In Java bytecode, each instruction has an associated index (this example was compiled using `javap -c` from a basic 'hello, world!' application):

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2    // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3    // String Hello, world
       5: invokevirtual #4    // Method java/io/PrintStream.println:
       (Ljava/lang/String;)V
       8: return
}
```

We can clearly see the BCIs of the various operations: the `invokespecial #1` has a BCI of 1, and so on.

This is a limitation of the Graal platform. In the coming months, the Graal core developers are adding this required feature. Once the feature has been added, it is a simple modification (as the infrastructure has already been created) to add this into Graal. Indeed, the required infrastructure for this transformation has been created – once the support for it is available, only would need to enable the transformations.

As an alternative, we used manual instrumentation (section 4.3). Although this does have the disadvantage of requiring both source-code access as well as human effort, this will not affect the results or correctness of the evaluation as the semantics are equivalent. The results and conclusions of this report will not be affected by this shortcoming of Graal. The required framework and infrastructure has already been developed, the mechanism through which the framework is used cannot affect the results in this case.

4.3 Manual Approaches

The major alternative to automatic instrumentation is manual instrumentation, where the user manually performs the required transformations in the source code.

These transformations consist of replacing array access and store operations with method calls, whilst adding relevant metadata if required. For example, consider the following array access:

```
1 int c = a[b];
```

Listing 4.2: Standard array access in Java

Manual instrumentation refers to replacing this operation, and all such operations, with the following (or equivalent):

```
1 int c = access-array(a, b);
```

Listing 4.3: Instrumented array access

The `array-access` method (and the implied `array-store` method) performs the dependency checking algorithms using the techniques as described in section 5.3.4.

4.4 Summary

In this section, we have considered the use of Graal for automatic instrumentation. We have seen that, due to a limitation in the current version of Graal, it is not possible to add arbitrary static method invoke instructions. Regardless of this limitation, there are other possible approaches that could also (in principle) be used for automatic instrumentation.

Lastly, we have seen how manual instrumentation is possible, and the reasons why the alternative approach used (manual instrumentation) will not affect the results or conclusions of this dissertation.

Chapter 5

The Runtime Library

5.1 Introduction

In this chapter, the work performed towards implementing the runtime library is presented. This library handles several core functions critical to the infrastructure of the application:

- Functions for collecting trace analyses
- Implementations of trace storage backends
- Algorithms for offline and online dependency analysis

The programs described within this chapter are open-source, released under The University of Edinburgh GPL license. They are available at `https://github.com/chrisatkin/locomotion`.

The fully-qualified package identifier for the runtime library is `uk.ac.ed.inf.icsa.locomotion.instrumentation`.

5.2 Trace Storage

Generating traces for large programs – the kind of programs which would benefit from hot-loop analysis – requires a large amount of storage as they scales linearly with the number of memory operations ($S = O(n)$). Although the number of storage operations conforming to the requirements in a program may be relatively small, this number is increased when the standard library is included.

The main problem that the storage format must be able to determine is this: given trace T and access α, is $\alpha \in T$?

5.2.1 Exact Approaches: Hash Tables and Sets

Exact approaches provide an accurate deterministic response to this question. They are not probabilistic or statistical in nature.

A hash function maps keys to indexes, where buckets are stored. Buckets contain all items with the same key. Ideally, a lookup of $T = O(1)$ is possible (i.e., a hash function with ideal distribution and a satisfactory number of buckets). Worst-case cost is $T = O(n)$, in the case of multiple collisions or a low load factor, as the bucket will need to be traversed sequentially to find the item.

It is common to use hashing by division for the hash function:

$$f(k) = k \bmod D$$

Where k is the key and D is the size (i.e., number of positions of the hash table) [Sartaj, 1998], although other functions are possible. For example, Java uses the standard `hashCode` method.

The constant-time lookup assumes that the *load factor* L is bounded below some constant. The load factor is computed by $L = \frac{n}{k}$.

Figure 5.1 demonstrates a simple hash table without collisions.

5.2.2 Probabilistic Approaches: Bloom Filters

The main disadvantage to using exact approaches is that the storage required scales linearly such that $S = O(n)$. For anything but the most trivial programs, this means that it becomes infeasible to store traces for all memory operations.

One alternative is the use of Bloom Filters [Bloom, 1970]. A Bloom Filter is a randomised data structure which supports membership queries, with the possibility of false positives. In the context of parallelism detection, this means that we may conclude that a loop is not parallelisable when in reality, it is.

The operation of a Bloom Filter is simple: there exists a bit vector of size m and a number k of hash functions (which could use universal hashing [Carter and Wegman, 1979]). Upon insertation of an item i, for each hash k_n the value of $v_n = k_n(i)$ is

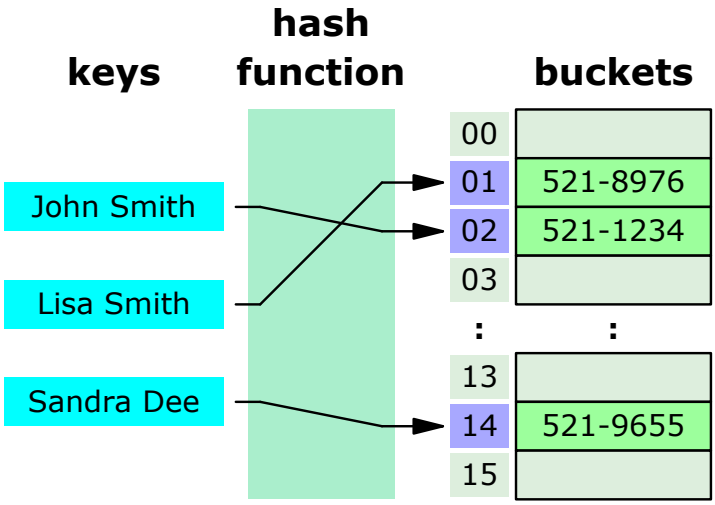

Figure 5.1: A simple hash table for a phone book (figure Wikipedia [2013a])

computed, which is an integer in the range $0..m$. The corresponding index of the vector is then set to 1.

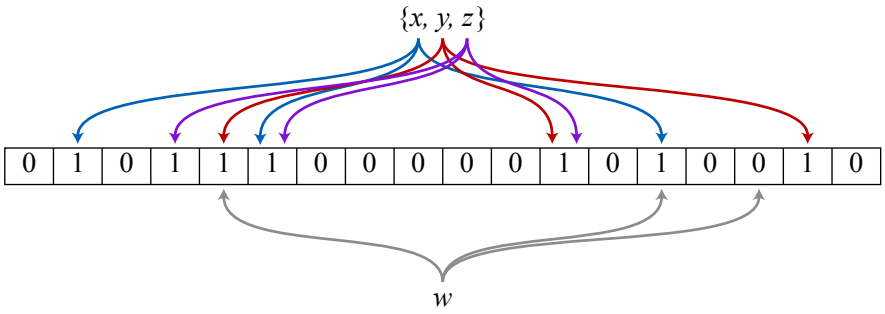

Figure 5.2: Bloom filter operation with $m = 18$ and $k = 3$ [Wikipedia, 2013b]

To test membership for an item, feed the item to each hash function. If all the corresponding indexes are 1, then the item *may* be contained within the filter - if any of them are 0, then the item is definitely not.

For a given number of entries s and a bit vector of size m, we need to use k hash functions such that:

$$k = \frac{m}{s} \ln 2 \tag{5.1}$$

The error rate is defined as:

$$\sigma = 0.5^k$$

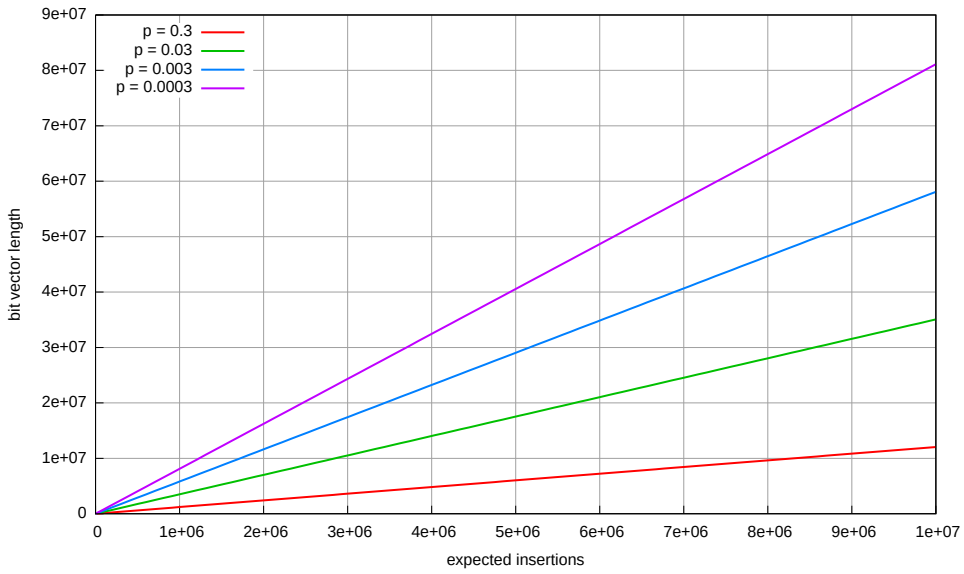

Figure 5.3: Plot showing bit vector length versus expected insertations for various false probability rates

In essence, the longer the bit vector, the more accurate the filter becomes - at the expense of increasing space requirements. If $m = \infty$, then there are no false positives - the filter becomes accurate and deterministic (for a fixed set of k). In essence, it becomes a standard set - although it is important to realise that this can never be achieved.

Swamidass and Baldi [2007] show is that the number of elements within a Bloom Filter can be estimated by:

$$E = -n \ln \frac{1 - n/n}{k}$$

It is possible to calculate the theoretical optimal length of the bit vector m from the number of expected insertions n and the *false positive rate* p. This is defined as

$$m = -n \times \log(p) / (\log(2) \times \log(2)) \tag{5.2}$$

It is possible to use equation 5.2 to compute the memory usage for given n and p, this is shown in figure 5.3.

The main advantage of bloom filters is that the space complexity is constant $S = O(c)$, where c is the size of the bit vector. This will allow memory overhead to be substantially lower, whilst still retaining $T = O(1)$ lookup time complexity. As mentioned above, this is a significant (factor n) improvement over the exact approach.

However, lookup time in bloom filters is asymptotically greater than the hash-based alternative. Unlike the hashing-based approach which has, with an optimal load factor, a lookup time of $T^{hash}_{lookup} = O(1)$, a bloom filter has a lookup time of $T^{bloom}_{lookup} = O(k)$ (where k is the number of hash functions). This is unlike amongst array/vector-based approaches, which in many cases are $T = O(1)$. As the number of optimal hash functions is dependent on the number of expected entries and the bit vector size (see equation 5.1), it is predicted that execution time will increase slightly as a result, with the assumption that the hashing-based approach is operating efficiently (i.e., with an adequate load factor).

5.2.2.1 Implementations

Unlike hash sets, where a canonical implementation exists in the form of `HashSet` within the Java API, there is not a single canonical implementation of bloom filters for Java. Naturally, however, there are many third-party implementations, each of which have different properties. Although where appropriate, these properties will still conform to the theoretical time and space complexities presented above, asymptotic performance is only one aspect of 'real-world' performance (due to differing hidden constants in the asymptotic costs).

The first implementation that exists is from the Google Guava project, which is a collection of APIs that applications commonly require. The Google implementation is a general-purpose bloom filter implementation, based around the concept of 'Funnels'. Funnels stream primitive values into `PrimitiveSinks`, of which a bloom filter is an example. In this way, a Funnel is a kind-of interface to the bloom filter, allowing the user to specify only certain values which should be considered for hashing. Additionally, they permit high performance as string-based representations, another 'general-purpose' representation, require additional overhead.

The alternative implementation used is taken from the Apache Cassandra project. Cassandra is a distributed, highly-scalable database management system, and it uses bloom filters to improve performance on disk lookup [Apache Foundation, 2013b]. Although this implementation is not considered a standalone implementation in the same way the Google Guava implementation is, it is still easy to remove from Cassandra.

The key difference that this implementation is the difference in hash function. Instead of a custom function used in Guava (which is based around Swamidass and Baldi [2007]), Cassandra instead uses the MurmurHash function [Appleby, 2013]. Mur-

murHash is known to be a high-performance, general-purpose hashing mechanism, and may prove superior to the custom methodology use in Guava. The reason that MurmurHash has higher performance is because it operates on 4 bytes per operation, rather than the single byte used in many other hash functions. However, this implementation is optimised towards strings, requiring all objects to be converted to string representation before hashing. This may be a source of overhead. The Guava implementation also uses a mathematical 'trick' presented in Kirsch and Mitzenmacher [2006] in order to reduce the number of hashes required.

In chapter 7, we will begin with a comparison between these bloom filter algorithms, continuing with the rest of the analysis with the implementation with the highest performance.

5.3 Dependency Analysis Algorithms

Computing dependency between two operations is of critical importance to this project; an efficient algorithm is important.

5.3.1 Why are dependencies important?

Dependencies, also called *hazards* (especially in processor architecture literature) are of critical importance to automatic parallelisation as they determine whether (or not) a loop can be automatically parallelised. If there is a dependence, the statements cannot be executed in parallel without additional transformation of the program[1]. However, such transformations are likely not possible for compilers to perform automatically in procedural/imperative as there is not enough semantic information available at the source-code level; in effect the compiler cannot reason about the program.

5.3.2 Dependency Theory

From Allen and Kennedy [2000, p. 37], there is a data dependence from statement σ_x to statement σ_y if and only if:

1. both statements access the same memory location and at least one of them stores into it

[1] One example of where such a transformation is possible is a summation of a counter: `for int i = 0 to n: i++`. This program can be broken down into a tree of tasks and executed in a divide-and-conquer fashion, which is easily parallelisable

2. there is a feasible run-time execution path from σ_x to σ_y (in which case, $\sigma_x \rightarrow \sigma_y$)

There are several kinds of inter-iteration dependency [Ibbett, 2009; Stallings, 2013, p. 526]. In this section, $R(\alpha_x)$ denotes the set of addresses/locations read by access α_x, and $W(\sigma_y)$ denotes the set of addresses read by access σ_y.

- **Write-after-write** or *output dependency*: given two statements σ_x and σ_y, there is a write-after-write dependency if a total ordering is required for σ_x and σ_y. For example:

```
1 R3 = R1 + R2
2 R3 = R3 + R4
```

  If the second statement were executed before the first statement, program correctness would be violated, so speculative execution is not possible.

  Formally, there is an output dependence iff $W(\sigma_x) \cap W(\sigma_y) \neq \varnothing$.

  Output dependence for two statements σ_x and σ_y is denoted as $\sigma_x \delta^0 \sigma_y$.

- **Write-after-read** or *antidependency*: given two statements σ_x and σ_y, there is a write-after-read dependency if evaluation of σ_y is dependent on the evaluation of σ_x:

```
1 R1 = R2 + R3
2 R2 = R4 + R5
```

  In this case, if the statements are re-ordered, the wrong value of `R2` will be used.

  There is an antidependency iff $R(\sigma_x) \cap W(\sigma_y) \neq \varnothing$.

  Antidependence for two statements σ_x and σ_y is denoted as $\sigma_x \delta^{-1} \sigma_y$.

- **Read-after-write** or *true dependency*: given two statements σ_x and σ_y, there is a read-after-write dependency if σ_y stores the value (or derivative value) of σ_x:

```
1 R1 = R2 + R3
2 R4 = R1 + R5
```

  If the statements are reordered, either a stale value of R1 will be used, or R1 may not have been initialized.

There is a true dependency iff $W(\sigma_x) \cap R(\sigma_y) \neq \varnothing$.

True dependency for two statements σ_x and σ_y is denoted as $\sigma_x \delta \sigma_y$.

Combining these, we gain the **Bernstein Condition**. There is a dependence between σ_x and σ_y iff:

$$[R(\sigma_x) \cap W(\sigma_y)] \cup [W(\sigma_x) \cap R(\sigma_y)] \cup [W(\sigma_x) \cap W(\sigma_y)] \neq \varnothing \tag{5.3}$$

For an arbitrary loop in which the loop index I runs from L to U in steps of S, the iteration number i of a specific iteration is equal to the value $\frac{I-L+S}{S}$, were I is the value of the index on that iteration.

Given a nest of n loops, the *iteration vector* i of a particular iteration of the inner-most loop is a vector of integers that contain the iteration numbers for each of the loops in order of nesting level. i is given by $i = \{i_1, i_2, ..., i_n\}$ where ik, $1 \leqslant k \leqslant n$, represents the iteration number for the loop at level k.

There exists a loop dependence from statement σ_1 to σ_2 in a common next of loops iff there exists two iteration vectors i and j for the nets, such that:

1. $i < j \vee i = j$ and there is an execution path from σ_x to σ_y in the body of the loop
2. Statement σ_x accesses memory location m on iteration i (access α_x) and statement σ_y accesses location m on iteration j (access α_y)
3. Either α_x or α_y is a store

Thus, the following problem declaration can be constructed for computing dependencies:

Let i_n^l be the set of memory accesses for iteration n in loop l. For a memory access $\alpha \in i_n^l$, determine whether there exists a previous iteration such that $\alpha \in i_{n-c}^l$ for some integer c. Additionally, for an access α_i and previous access α_{i-c}, $kind(\alpha_i) \neq read \wedge kind(\alpha_{i-c}) \neq read$.

In addition, there are two kinds of loop dependence.

- **Loop-carried dependence**: σ_x can reference a common location m_c on an iteration, σ_y can reference the same location m_c
- **Loop-independent dependence**: σ_x and σ_y can both access m_c on the same iteration, but with σ_x preceding σ_y during execution

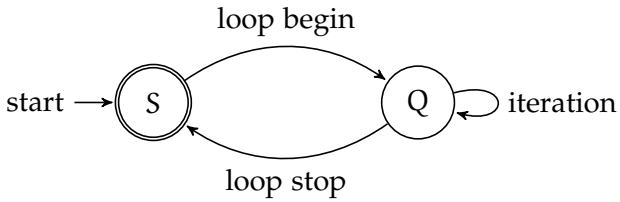

Figure 5.4: Finite state machine for the online algorithm

5.3.3 Offline Algorithms

An offline algorithm means that any dependency checking is performed after the trace has been collected [Knuth, 1997, p. 525-526]. It is the simplest form of dependency analysis algorithm, but it show poor performance. For a number of loops l with an average of i iterations each, where each iteration has an average of o operations, then $T_{offline} = O(lio)$.

Algorithm 1 Offline dependency algorithm

```
1:  d ← ∅
2:  for all loops l do
3:     p ← ∅
4:     for all iteration i ∈ l do
5:        for all access α ∈ i do
6:           for all p_α ∈ p do
7:              if α ∈ p_α then
8:                 d ← α
9:              end if
10:             p ← α
11:          end for
12:       end for
13:    end for
14: end for
```

This offline algorithm has several disadvantages. Not only is its runtime particularly poor (we can achieve at least a factor l speed-up using an online algorithm), but because it requires a complete trace of accesses per iteration, it is unsuitable for detecting dependencies at run-time. Algorithm 1 outlines the offline algorithm.

5.3.4 Online Algorithns

An online algorithm for this problem is one that runs with a sequential input of values. An online algorithm is superior because it shows better performance characteristics asymptotically, $T_{online} = O(io)$, and it runs in conjunction with the program. This allows Locomotion, in theory, to advice any JIT compilers of optimisations to perform. Figure 5.4 represents the finite state machine for the algorithm.

Algorithm 2 Online dependency algorithm

```
d ← ∅
for all loops l do
  p ← ∅
  for all iteration i ∈ l do
    for all access α ∈ i do
      for all p_α ∈ p do
        if α ∈ p_α then
          d ← α
        end if
        p ← α
      end for
    end for
  end for
end for
```

5.4 Implementation Details

In this section, we consider the techniques used to implement the runtime library and online dependency algorithm from a software engineering perspective.

5.4.1 Entry Point

The main entry point to the instrumentation is the `Instrumentation` class. It includes various static methods, which operate on an instance of itself. These static methods implement a simple state machine, defining the allowable operations for the instrumentation. Figure 5.5 shows the class diagram with public methods for `Instrumentation`.

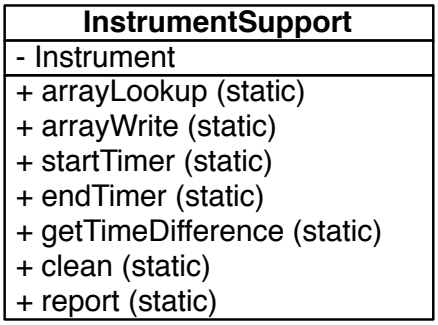

Figure 5.5: Class diagram for `InstrumentSupport`

5.4.2 Instrument Implementation

`Instrumentation` is initalised at load-time with an instance of itself (via static blocks). `Instrumentation` has a configuration. This configuration contains parameters such as whether the instrumentation should be enabled and so on.

The user-facing methods for trace collection have the following signatures:

```
1    public static void
2    setConfiguration(Configuration c);
3
4    public static void
5    startLoop(String id);
6
7    public static void
8    enterIteration(int number);
9
10   public static <T> T
11   load(T[] array, int index);
12
13   public static <T> T[]
14   store(T[] array, int index, T value);
15
16   public static void
17   endIteration();
18
19   public static void
20   endLoop();
21
```

```
22    public static long
23    memoryUsage();
24
25    public static void
26    clean();
27
28    public static List<LoopDependencyException>
29    getDependencies();
```

Listing 5.1: Method signatures for instrumentation methods

The arguments are:

- `T[] array`: the array upon which the operation is occuring
- `int index`: the array index in question
- `int i`: the value of the iteration variable
- `T value`: the value being written to the array at the index
- `String id`: a unique loop identifier

The Java generics system was leveraged in order to reduce the amount of code required to implement the collection; it is important to recognise that the Java type system allows the trace collection mechanism to be unconcerned with the type of the array being accessed (and the type of the item being inserted if appropriate).

Although it would be possible to use the above generics-based methods for all data types, this would require the use of the object-oriented wrapper libraries (also called the reference types) for `int` → `Integer`, `float` → `Float` and so on. This would have the disadvantage of incurring the overhead of autoboxing [Oracle, 2010] in the library.

5.4.2.1 Java Auto(un)boxing

Despite being an object-oriented language, Java has several non-object-oriented types. These are known as *primitive values*. They are not part of the standard object-hierarchy, and no methods can be called upon them. The primitive types are `int`, `long`, `char`, `float`, `double`, `boolean` and `short`.

Each primitive type has a corresponding reference type (named as such because Java uses pass-by-reference for objects), which is an object and hence part of the class hierarchy.

Although this approach does have some advantages – primitive types require less over head than the object-oriented value types – there are some disadvantages – namely, enable interoperability between the two.

This process is called auto boxing (in the case of primitive-to-reference type occurs) or auto unboxing (where reference-to-primitive type occurs). The advantage of this approach is that it simplifies programming and reduces 'crufty' code. For example, adding two `Integer`, `a` and `b` doesn't require `Integer c = a.add(b);`, but instead the more natural `Integer c = a + b` can be used. In this case, `a` and `b` will be auto-unboxed to primitive types, the addition is performed and the result is autoboxed back to `Integer`.

However, this automatic casting operation does occur a slight performance overhead, both in terms of memory and execution time. Although in most cases the overheads are small, for benchmarking a runtime system with somewhat significant overheads, another solution was opted for. Instead of only supporting value types, we *also* support primitive types by overloading the appropriate methods.

In order to overcome this performance problem, the instrumentation has been optimised to consider both primitive types and reference types.

5.4.3 Trace Storage and Configuration

The instrumentation includes implementations of both exact and inexact approaches to trace storage.

The `Access` class provides an abstraction for an access. An `Access` represents an array access with an associated kind, index, number ("*this iteration represents the* n*th access this iteration*").

5.4.3.1 Exact - Hash Set

For the exact implementation, I chose to use the standard Java Collections Framework `HashSet` class. This is for several reasons, such as:

- **Performance**: array lookup in hash maps is $T = O(1)$, assuming an equal distribution of hash codes. For this reason, `Access` also overrides the standard `hashCode()` implementation, so that any two accesses are equivalent iff the array ID and indexes match.

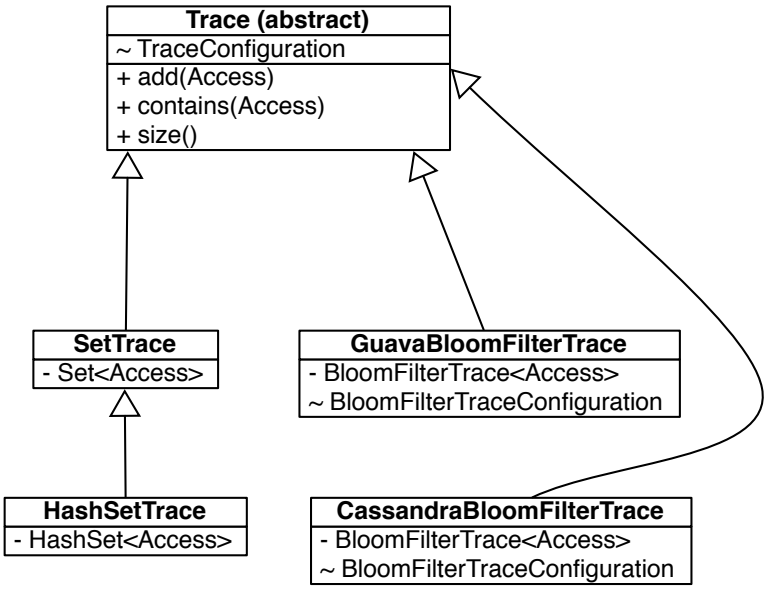

Figure 5.6: Trace class hierarchy

- **Uniform interface**: any member of the Collections framework includes by-default various capabilities, such as being `Iterable`, meaning enhanced for-each loops can be used.
- **Standard implementation**: the built-in implementation has been tested for bugs, and is (mostly) bug-free. A custom implementation could not be subjected to this same rigorous testing.

5.4.3.2 Inexact - Bloom Filters

There is a parent abstract class, `Trace` which specifies several standard methods that all interfaces must define (`add()`, `contains()` and `size()`). All implementations of traces inherit from this superclass. Figure 5.6 shows this hierarchy in detail.

Both implementations for the Guava and Cassandra bloom filters are included in this hierarchy.

A `TraceConfiguration` class is used to provide configuration details for the trace. There are no configuration variables used for hash sets, but for bloom filters this includes the initial size and the `Filter` used (an implementation detail as a result of the use of Google Guava). Dynamic configuration was preferred to constants as many experiments will need to be run, often with different variables.

When a new access α in iteration `i` and loop `l` is detected, the library checks if `l` is a new loop, or an existing one. If the loop is new, it instantiates a new `Loop` object, which

holds the traces, as well as detecting whether an access is dependent. `Loop` stores two instantiations of the supplies `Trace` class - one for read and one for write operations. This is necessary in order to detect $\sigma_x\delta\sigma_y$, $\sigma_x\delta^0\sigma_y$ and $\sigma_x\delta^{-1}\sigma_y$ dependencies, but not read-after-read dependencies. In order to use a single trace implementation, the semantics of the `hashCode()` and `equals()` methods, along with the `Comparable` access would need to be violated as a temporal dependence is introduced. Both bloom filters use `hashCode()` to determine an item is contained. It is not possible to use a $\sigma_x\delta\sigma_y$, $\sigma_x\delta^0\sigma_y$ and $\sigma_x\delta^{-1}\sigma_y$ but not read-after-read dependencies.

In order to detect dependencies, `Loop` checks all *previous* iterations, using a $T = O(1)$ check on for containment within both read and write traces. Overall, for n iterations the solution has a time complexity $T = O(n)$ and a space complexity $S = O(a)$, where a is the number of accesses. Once a loop has been completed as detected by the finite state machine (see figure 5.4), the accesses are deleted in order to reduce memory location.

Dependencies are reported to the runtime via an exceptions mechanism. When a dependence is detected a special exception, `LoopDependencyException` is thrown, which is initialised with hazard metadata (which access is dependent, the kind of dependence, and the two iterations which the access occurs within). `LoopDependencyException` is a checked exception. The exception traverses up the stack until it reaches `Instrument`, which stores all exceptions of the same type in a linked list. This list is available to users of the framework, meaning full dependency information is available.

5.5 Summary

In this chapter, the theory of dependence analysis has been introduced, as well as some practical approaches to implementing them. Both kinds of dependency checking algorithm (offline and online) have been covered, as well as implementation details of online algorithms in our framework.

Lastly, a software engineering-based approach to the design of the instrumentation framework was discussed.

Chapter 6

Methodology

6.1 Introduction

In this chapter, the experimental set up and technique is outlined, as well as the specific benchmarks that were used. Rationales for each benchmark are also presented.

6.2 Experimental Setup

The hardware used for the experiments is a mid-2009 MacBook Pro with the following specifications:

- Intel Core 2 Duo Esomething
- 8GB 1067MHz DDR3
- Integrated Intel graphics
- AFS filesystem

The software configuration is as follows:

- Scientific Linux 6.3 `DICE` 64-bit
- Java 7 update 21

All software used was the latest version available at the time of writing.

In silico experimental design is notoriously difficult within the context of standard scientific procedure. Reproducibility of experiments is not always guaranteed, as slight and sometimes un-noticed machine intricacies and flaws can affect experimental

results. Because of this, all benchmarks were run on a separate user account (in order to reduce the number of running processes which may affect results, perhaps by consuming additional resources).

6.3 Experiment Descriptions

In this section, we describe the various experiments which will be performed, the results of which will be presented in chapter 7.

6.3.1 Implementation

The first experiment which was conduced was an experiment to determine the optimal bloom filter implementation. There are several bloom filter implementations, and when one considers the higher time complexity associated with bloom filter operations, a high-performance operation is vital.

In this experiment, n randomly generated memory operations (the same objects used in the actual instrumentation) are generated, and the median operation time for both bloom filter implementations as well as an optimally configured hash set and a hash set with default configuration.

An optimally configured hash set is one where the initial capacity is set to the actual number of insertions. The load factor is left at the default value of 0.75, as recommended by the Java API documentation. The hash set with default configuration uses an initial capacity of 16. Linear hashing is used to increase the capacity of the set as-and-when required.

6.3.2 Precision

The next experiment was to measure the precision, or detection rate, of the storage structures selected with highest performance as a result of the experiment presented in section 6.3.1.

This is important as dependency detection is the primary objective of the framework. We used the benchmark presented in section 6.6 in order to determine the accuracy, in terms of detecting 'correct' dependencies as well as any false positives which may occur.

6.3.3 Overhead

However, it is well documented that adding instrumentation to programs does come at a cost, both in terms of execution time and memory usage. In the design of the instrumentation (see chapter 5 for details), care was taken to ensure that the overhead was as minimal as possible.

In this set of experiments, we test the impact that the instrumentation has on the execution time and footprint of our parametric benchmark.

In addition to testing the overhead on the benchmark without any operations being performed, we also conducted testing where there were a number of floating-point computations associated with each memory operation. These computations were designed in such a way that they cannot be removed by the optimiser in either the compiler or the runtime environment. For each memory operation that is performed, a single floating-point operation is performed in a loop. The variable was the number of times that the loop runs.

6.3.3.1 Execution Time

6.3.3.2 Memory Use

6.3.4 Multiples

Early in testing we found that, as expected, there was no single size of bit vector which would cover all possibilities; each loop is unique and requires

6.3.5 Online Instrumentation Disabling

6.4 Repeats

In order to improve the results, each experiment was repeat ten times. The median of the ten repeats was used for the rest of the analyses.

6.5 Measurement Methodology

6.5.1 Execution Time

Execution time was measured by taking the difference between `System.nanoTime()` before and after the experiment was run. This is superior to using other methods, such as the Unix `time` program because it computes an accurate value, instead of elapsed user-space CPU time.

6.5.2 Dependencies

When a dependency is detected, an exception is thrown to the caller, which collects dependencies in a linked list. This technique was chosen because it allows the most flexibility and separation of concern.

6.5.3 Memory Usage

The difficulties of measuring memory usage in Java programs due to the non-deterministic nature of the garbage collector are well documented in the literature [Kim and Hsu, 2000; Ogata et al., 2010]. Despite this, the Java 7 API presents several techniques [Oracle Inc, 2013] of measuring memory within the JVM:

- `freeMemory()`: the amount of free memory in the virtual machine
- `maxMemory()`: the maximum amount of memory that the virtual machine will attempt to use
- `totalMemory()`: the amount of memory currently in use by the virtual machine

In addition, there is a Java Agent for measuring memory usage of an object - the Java Agent for Memory Measurements [Ellis, 2011] (JAMM). JAMM is essentially a wrapper for the `java.lang.instrument.Instrumentation.getObjectSize()` method. There are several methods available, and the framework uses `measureDeep()` for the greatest accuracy.

`measureDeep()` crawls the object graph, calling `getObjectSize()` on each object it encounters. An `IdentityHashMap` is used to detect loops in the object graph. Unfortunately, this does affect execution time - but memory usage is recorded after execution time has been recorded. Ellis does suggest investigating the possible use of bloom filters to overcome this memory usage, but this it outside the scope of this project.

In order to take the most accurate readings possible, the JAMM library was used to measure the overhead. As an implementation detail, this led to an interesting bug in measuring the usage, which is worthy of mention.

As previously mentioned, JAMM walks the object graph, which is created by instantiation within classes - this forms an implicit directed edge between the calling context and the instantiated object. Such link is used for determining scope visibility.

Initial implementations of the instrumentation used an anonymous class for the funnel (see section 5.2.2.1) in a fashion similar to:

```
1 Instrument.setContainer(new BloomFilter<Access>(new Funnel<
     Access> {
2     public void funnel(Access from, PrimitiveSink to) {
3         to.putInt(from.getArrayId())
4           .putInt(from.getIndex());
5     }
6 }));
```

This, however, results a bug in measurement of memory usage. Anonymous inner classes have an implicit edge on the object graph from their lexical context to the context of the scope they were declared in.

```
1  public class OuterClass {
2      public void doSomething() {
3          // things
4      }
5
6      public class InnerClass {
7          public void test() {
8              OuterClass.this.doSomething();
9          }
10     }}
```

As the funnel is stored as a field in the bloom filter, when the code to measure the size of the bloom filter was called:

```
1 public class BloomFilterTrace extends Trace {
```

```
2     BloomFilter<Access> bloom;
3
4     public long getMemoryUsage() {
5         return new MemoryMeter().measureDeep(bloom);
6     }
7 }
```

The memory meter would not only walk the bloom filter, but *also* the object graph of the main testing class, which contains hundereds to thousands of experiment configurations – resulting in significant memory measurements for bloom filters, despite the memory usage being very low.

6.6 Parametric Benchmarks

As we have already investigated in section 5.3, there are multiple kinds of dependency and hazards. In order to complete a thorough analysis, a new kind of benchmark has been developed in order to analyse the effect of the three kinds of dependency.

6.6.1 Motivation

In order to critically analyse the effectiveness of the bloom filter versus hash set, a controlled (i.e., non-deterministic) environment is required. In addition, a custom benchmark allows us to control the non-instrumentation workload; in order to determine an 'upper-bound' on the instrumentation overhead a benchmark where *only* instrumentation operations are performed.

6.6.2 Problem Statement

The problem statement is somewhat simple. Given n accesses, generate patterns of access such that there are n^{δ}, n^{δ^0} and $n^{\delta^{-1}}$ dependencies. These values are by definition probabilistic in nature, as the issue of array aliasing is important.

In this context, we consider an *access pattern* to be two sequential accesses , α_x and α_y, to the same index in an array. It is given that $x < y$ (i.e., $T(\alpha_x) < T(\alpha_y)$)1, but they may or may not be immediate. In other words, there is a viable code path between σ_x and σ_y.

6.6.3 Algorithm Design

There are five parameters to the main algorithm:

- Length of resulting arrays, l
- Percent of dependencies (expressed as a boolean in the range $0.0 <= x >= 1.0$), d
- Of d, what percentage of n^{δ}, n^{δ^0} and $n^{\delta^{-1}}$

The results of the algorithm are:

- An array, `a`, of random numbers within the range $0...(l-1)$
- A second array, `b`, containing the indices of `a` to access at access i
- An operation array, determining which operation should be performed

To illustrate, the usage of the results are:

```
int[] a = getA();
int[] b = getB();
Operation[] ops = getOperations();

int temp;

for (int i = 0 to m) {
    if op[i] == Load
        temp = a[b[i]];
    else if op[i] == Store
        a[b[i]] = randomInteger(0, m);
}
```

This structure allows the algorithm to generate configurations of `b` and `op` to produce each kind of dependency.

The algorithm is split into three stages, one for each array. `a` is an array of random numbers.

6.6.3.1 Generating b

`b` can be split into two conceptual sub-arrays: the section for dependent operations, and the section for independent operations. The length of these sub-arrays depends

on the parameter d. The independent sub-array contains simply the values $0..l_{indep}$ in incremental order, and in this dependent sub-array each value must have exactly one duplicate (in theory a value could have more than one duplicate, but this would not affect the results of the instrumentation).

To illustrate, this is an array of `b` with no dependencies:

```
0  1  2  3  4  5  6  7  8  9
```

Whereas this one has 100% dependency:

```
0  0  1  1  2  2  3  3  4  4
```

The dependent sub-array is generated using an incremental loop with a simple state machine.

After this, the arrays are merged and shuffled using the Yates-Fisher algorithm [Knuth, 1997, p. 145].

As a result, this stage operates in $T = O(n)$ time.

6.6.3.2 Generating `ops`

Again, a state machine is used in order to determine that exactly two operations are dependent.

As the dependency kinds can be weighted with different values, a discrete probability distribution over the hazard types is used. The algorithm performs a scan over `b`, at each index performing a second scan over the array to kind an equal value. If one is found, a probability function is created, with the appropriate values being written. If a pair is not found, an operation is randomly chosen.

6.6.4 Implementation Details

The algorithm is presented as a class, with the following public interface:

```
1 public class HazardGenerator implements Generator {
2     public HazardGenerator(RandomGenerator gen);
3
4     public Generator configure(int length, int dependencies,
    long seed, double prob_writewrite, double prob_writeread,
    double prob_readwrite);
5
```

```
6     public Generator generate();
7
8     public int[] getA();
9
10    public int[] getB();
11
12    public AccessKind[] getAccessPattern();
13
14    public static String statistics(int[] a, int[] b,
      AccssKind[] k);
15 }
```

The Apache Commons mathematics library is used to compute the probability distributions [Apache Commons, 2013].

Several methods return instances of `Generator` in order to fit into the rest of the instrumentation framework, as well as to allow method chaining.

6.7 Test Harness Design and Implementation

The main experimental class is `Experiments`, which contains the entry point. `Experiments` takes the parameters for the experiment, and generates an appropriate number of experiments matching the configuration. The configuration variables include the length start, end and step size; the expected number of insertions start, end and step size for the bloom filter; the test name and so on.

There is a generic `Experiment` interface, which all experiments must implement. An experiment in this context is a specific benchmark class, not a configuration. An `Experiment` must be able to take an arbitrary number of arguments (through polymorphism to `Object[]`) and return an identifier.

An experiment class (i.e., `Class<? extends Experiment>`) is packaged with a configuration, an output format and parameters through the `Test` class, which handles instantiation of the experiment classes and so on. In addition, `Test`s can be marked to run in parallel using an `ExecutorService`, but since this may impact execution times, this feature was not used for the data collection runs.

Output is handled via the `Output` interface, for which two implementations are provided. The `Console` implementation pipes results to standard output for debugging purposes, whilst `File` uses a `PrintWriter` to pipe to a file, which can later be analysed. For performance reasons, output lines are buffered into a linked list (with $T = O(1)$ node addition time complexity) and only written to file after the experiment is over.

6.8 Result Collection

Each experiments writes it's identifier, configuration, execution time, memory usage and number of detected dependencies to a file. A separate program is then used to collate all similar results (i.e., same configuration with varying numbers of iterations and bit vector lengths if appropriate) into a single file for plotting and analysis.

6.9 Summary

In this chapter, the experimental setup for the data collection has been outlined, as well as the measurement technique for the various results. A parametric benchmark has been introduced, including a methology, set of algorithms and some implementation details. Lastly, we covered the design of the test harness and the methodology used to collect, collate and analyse the results.

Chapter 7

Results

In this chapter, the results of the various experiments that were conducted are presented and analysed. The experimental procedure described in chapter 6 was followed.

The results are split into several sections. Section 7.1 presents the experiments with regard to the optimal selection of bloom filter and hash set implementation. Then, section 7.2 analyses the precision of the chosen bloom filter implementation, and draws conclusions to hash sets. Section 7.3 analyses the memory usage required for both schemes. Finally, section 7.4 compares the execution time required for each scheme with a break down into 'raw' overhead costs (ie, where no work is performed per access) and 'real-world' costs, where a defined number of computations occur per memory operation.

7.1 Bloom Filter Implementation

As shown previously, the main overhead for bloom filters is the cost of computing the k hashes required for their operation. It is well-documented that hash functions have a relatively high computational time complexity, and so the cost of hashing in a bloom filter is quite high. Reducing this time complexity is therefore crucial to improving the performance of the bloom filter. In this section, we perform an analysis of the implementation currently used, against another implementation in order to evaluate the importance of the hashing algorithms.

The first comparison that we make is to simply determine the raw overhead of insertion of values. This will allow us to determine an upper bound of the expected performance if the mechanism was inserted into the instrumentation framework. Two

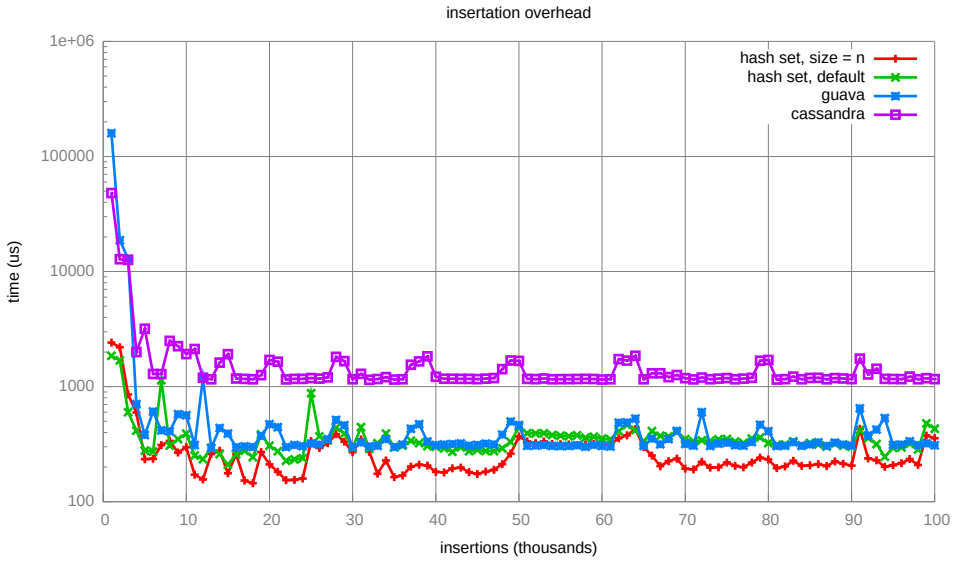

Figure 7.1: Median insertion costs for each data structure tested. (Notice the use of log scale on the y axis)

operations were benchmarked, the insertion operation and the membership operation. We measured for a number of accesses ranging 1000 to 100,000 (in steps of 1000), the median execution time for each operation for each of the storage formats. Note that the data was not collected on a 'rolling' basis, for each length an entirely new experiment was run. We considered the following storage structures:

- A `HashSet` with a predefined size of n, where n is the number of accesses and a load factor of 0.75
- A `HashSet` with default settings (i.e., an initial length of 16 and a load factor of 0.75)
- The aforementioned Google Guava implementation of bloom filter, with insertions set to n and a false positive probability of 0.03
- Lastly, the Cassandra bloom filter implementation with the same settings as the Guava implementation

7.1.1 Insertion

The insertion operation is the process of adding an object into the data structure in question. In both tests, the load factor of the hash set was 0.75, as recommended by the Java documentation.

As we can see from figure 7.1, the Cassandra implementation of bloom filters car-

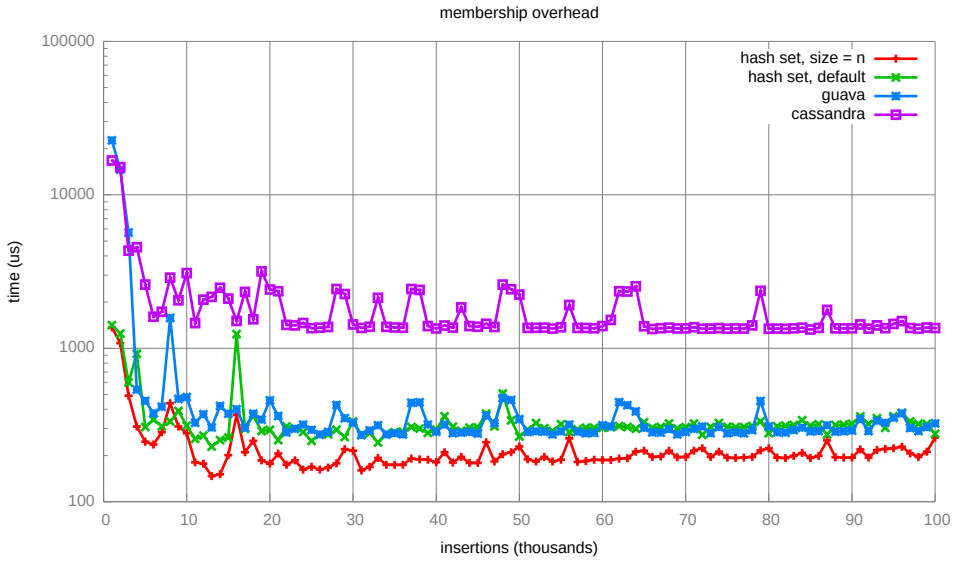

Figure 7.2: Median membership query costs for each data structure tested. (Notice the use of log scale on the y axis)

ries a significant overhead when compared to all other structures, with roughly a 50% higher execution time than the Guava bloom filter. This can be attributed to the aforementioned (section 5.2.2.1) pre-requisite 'stringification' required for this implementation.

The structure with the highest performance is clearly a hash set with a capacity set to the number of iterations. Unlike bloom filters, hash sets use only one hash function, whereas bloom filters use k. This, when combined with an optimal number of buckets results in significantly higher performance than the other structures.

When the hash set is not optimally configured (i.e., initial capacity set to the default of 16), it performs on average at the same performance as the Guava bloom filter. This is because whilst the bloom filter requires k hash functions, the set will often require rehashing in order to increase the underlying data structures.

The initial spikes in execution time followed by a 'levelling out' of the times are mainly a result of the JIT compiler - initially, the interpreter is executing the code. When the JIT recompiles to native code, performance increases significantly. There may also be some additional factors, such as caches still being initialized.

7.1.2 Membership

Figure 7.2 presents the results of the membership query testing.

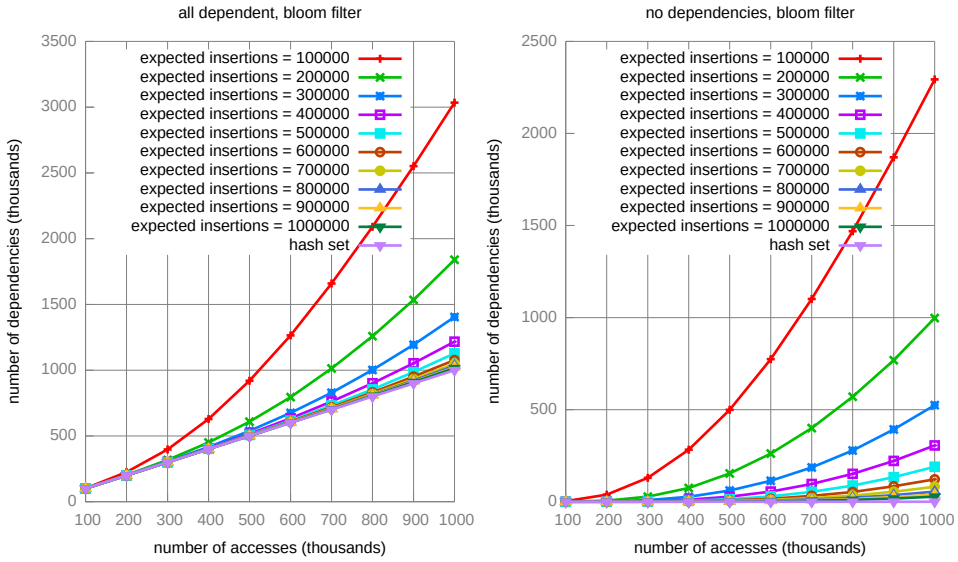

Figure 7.3: Precision results for all dependent and none dependent

As with insertion costs, the stringification required for Cassandra has a significant impact on the execution time.

The results for the other structures largely mirror those for insertion. The optimally configured hash set performs best, with a roughly 20 to 30% performance improvement over both the Guava bloom filter and a default hash set. Again, this is to be expected as a result of the lower time complexity associated with the optimal hash set.

Overall, we observe that execution times for both operations are similar. There appears to be no additional overhead for insertion above the overhead required for membership querying.

7.1.3 Conclusions

From these results, we can immediately disregard the Cassandra bloom filter for further testing, as it has been optimised for strings and, although it may be used as one, it is not a general-purpose bloom filter. This is unlike the Guava implementation, which is both high-performance *and* general-purpose.

This high-performance is a result of the design of the Guava implementation. In addition to using funnels instead of other forms of serialisation (see section 5.2.2.1), it also employs a mathematical 'trick' presented in Kirsch and Mitzenmacher [2006] in order to reduce the amount of hashing required below that of a standard bloom filter – although we must emphasis that $k >= 2$ (i.e., there are still more hash functions

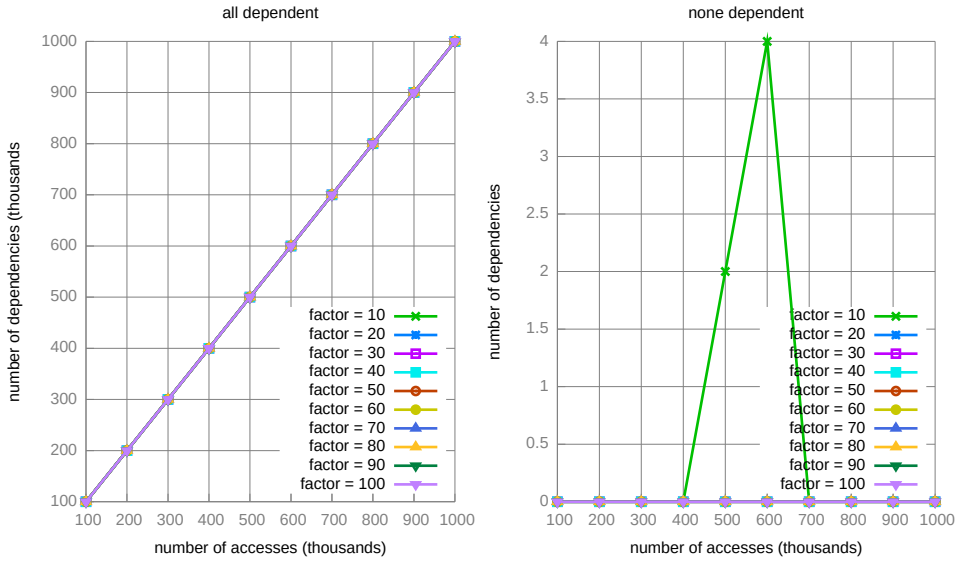

Figure 7.4: Precision rates for various factors of iteration counts

required than hash sets).

For future testing, we will use the Guava implementation for bloom filters, as well as optimally-configured hash sets, as these showed the highest performance characteristics. Note that our choices here cannot impact the precision/accuracy of future experiments. Although we did not consider accuracy in these experiments, both bloom filters are capable of attaining 100% accuracy; we were simply seeking the implementation with the lowest overhead.

7.2 Precision Testing

Now that we have chosen the optimally-performing bloom filter and configured hash set, we can now consider the accuracy which can be attained through their use.

For space reasons, we present figures for 100% dependencies and 0% dependencies only. Figures for other dependency rates (20%, 40%, 60% and 80%) are available in appendix A.

Figure 7.3 presents the precision results for the chosen structures. The charts show both the advantages and disadvantages of using bloom filters - in some cases, they were as precise as hash sets.

In both cases, the error rate decreased logarithmically as the number of expected insertions increased, as fitting with the theoretical results presented in section 5.2.2.1.

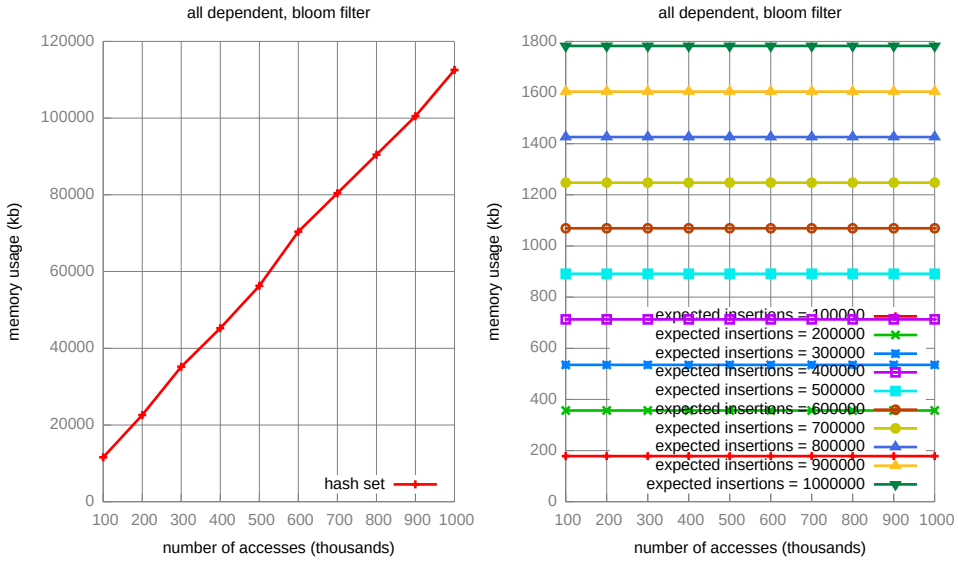

Figure 7.5: Memory usage for all dependent, hash set and bloom filter

We can also see that the bloom filter hashing mechanism is indeed evenly distributed. This is evidence through when the number of expected insertions equals the number of actual insertions, the bloom filter correctly determines that there are no dependencies.

The reason that the number of detected dependencies increases to the value that it does is the nature of the detection. As mentioned previously, there are two bloom filters used per loop - one for reads, and one for writes. However, because there are only three kind of dependencies, upon a store operation both filters must be checked, whereas the loads filter must only be checked one - a total of three checks (we note that each dependency type has equal weighting). As each filter becomes saturated, they report a 100% dependency rate, which results in 3x higher dependency detection rate above the actual.

Generally speaking, the bloom filters were as accurate as the hash set when the expected insertion count was roughly 10 times higher than the number of actual insertions. In order to test this hypothesis, we ran several more experiments where instead of using pre-determined bit vector lengths (as in these experiments), we instead used lengths of various multiples of the number of accesses. These results are presented in figure 7.4.

As we can see, when the expected insertions is increased to a factor, the results are much more promising. We can see that for all factors, the precision for all dependent was 100% (i.e., all operations were detected with no false positives).

Although the case is similar for no dependencies, the were two false positives detected

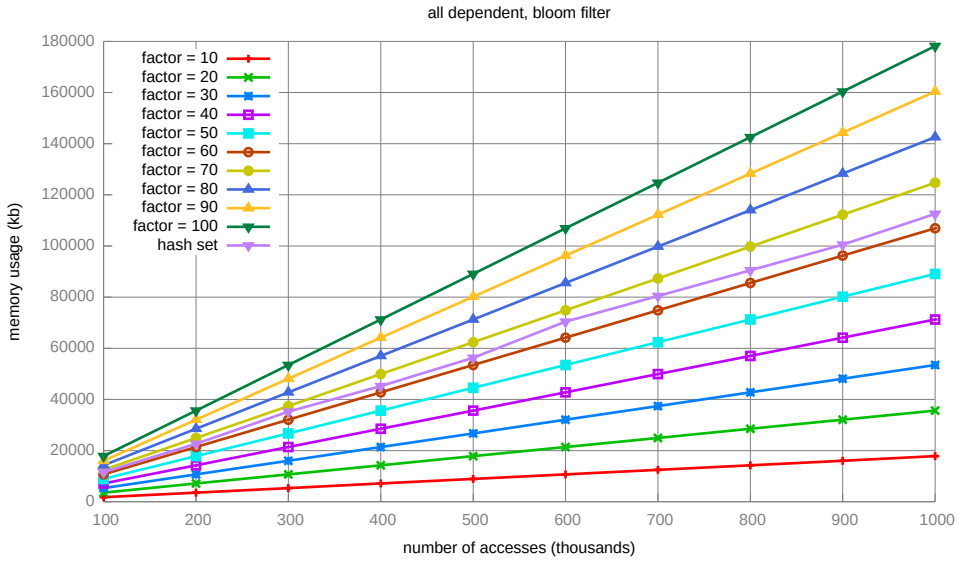

Figure 7.6: blah

at 500,000 iterations and four detected at 600,000 iterations when the factor was 10. Increasing the factor above 10 resulted in no false positives. This again highlights the possible disadvantage of using a probabilistic data structure.

We note that although it is not possible to reduce the false positive probability rate to 0% in bloom filters, we can modify the parameters such that they can be minimized. Modifying the expected insertions (and hence, the bit vector length) is, in our case, superior to modifying the false positive probability rate since doing so modifies the number of hash functions required, therefore increasing execution time.

In conclusion, we observe that when the number of expected iterations is set to 20 times greater than the number of actual insertions, bloom filters offer accuracy equal to that of hash sets. Although the false positive probability rate cannot become zero, the use of sufficiently large bit vector lengths can mitigate the risk substantially enough that there is effectively no difference (in terms of detection rates) between bloom filters and hash sets.

7.3 Memory Usage

We next consider the memory usage of each approach.

Figure 7.5 presents the memory usage, in kilobytes, versus the number of accesses when all operations are dependent. Note that the memory usage for all other configurations of dependencies is the same, and hence not shown here.

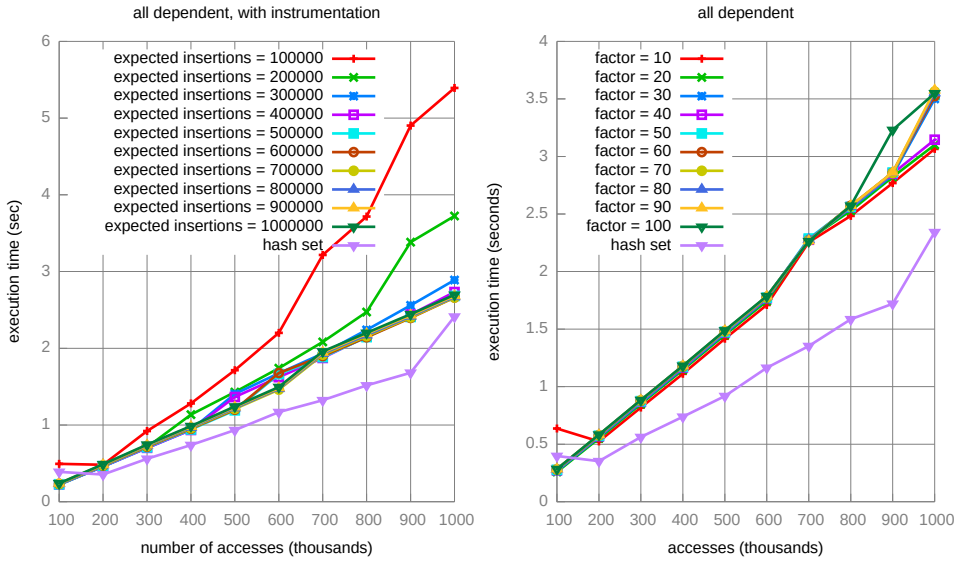

Figure 7.7: Execution time for synthetic benchmark

There are several observations which can be made immediately. The first is that the required memory for hash set is significantly higher than that for bloom filters. It is somewhat difficult to compare directly between the two mechanisms since memory usage increases as a function of number of accesses for hash sets (and indeed, we see that the result is the expected linear increase), whilst as a function of expected insertions (and not actual insertions) for bloom filters. Despite this, we can see that for 100,000 accesses, the hash set required roughly 12MB, whilst the corresponding bloom filter with 100,000 expected insertions (which gained a 100% accurate rate), required just 200KB - a factor of 60 improvement.

As expected, the memory usage for bloom filters also follows a linear relationship: double the number of expected insertions and the memory requirement also doubles.

We now consider the case when multiples are taken into consideration; figure 7.6 shows this. We also include hash set for comparison purposes.

We observe that the memory required for factors increases substantially, in some cases it is above that required for hash sets. The this occurs when the multiple is set to 70 times or higher. When we take this, and the preciseness of both methods, we can conclude that these is no net benefit to using factors greater than 70. However, given the results presented in section 7.2, we can *also* conclude that it is not necessary to use a bit vector of above a factor of 20.

When a multiple of 10 is used, memory usage for 1,000,000 accesses increases to 20MB. This is considerably lower than that required for hash sets, which is 11MB, leading

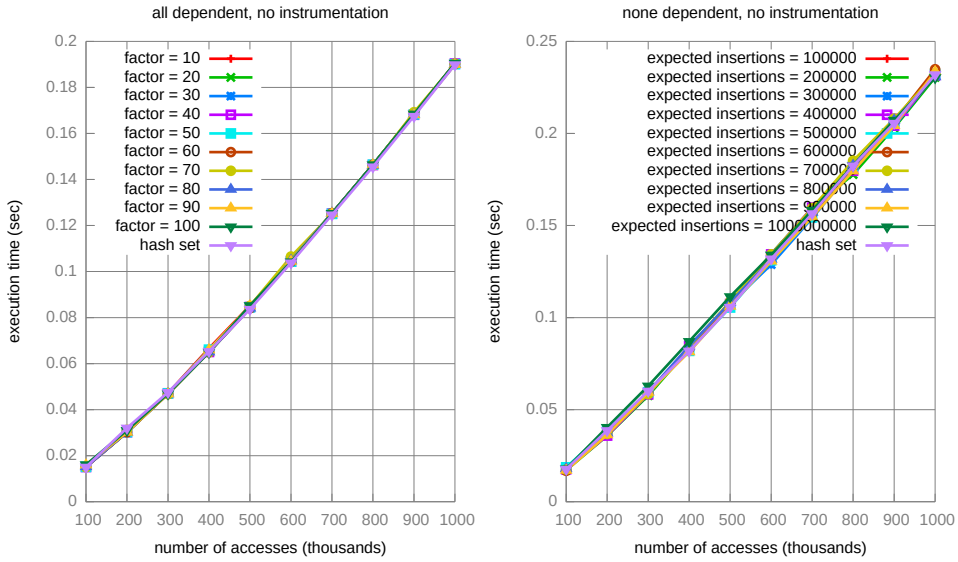

Figure 7.8: Execution time for synthetic without instrumentation

to an improvement of 5.5x lower memory usage. If the same analysis is performed except with a factor of 20, the improvement decreases to roughly 2.75x.

7.4 Execution Time

We next consider the case of execution time within a synthetic environment. In other words, the impact on execution time when no other operations are taken into account.

From figure 7.7, we can clearly see the impact of instrumentation, with there being little difference between the times recorded without instrumentation. Slight fluctuations are expected as a result of experimental noise.

We can also observe the impact of using a bloom filter over a hash set, with execution times required for bloom filters being roughly 20 to 40% higher than that those required for hash set.

This can be attributed to the additional cost of k hash functions (leading to time complexity $T = O(k)$ as opposed to $T = O(1)$ for the hash set). With the additional hash requirements of the instrumentation (as explained in section 7.3) leads to a higher overhead for execution. Additionally, this result has been found by previous authors.

When we consider the relative execution times for the various bloom filter configurations, we see that as the error rate increases, so does the execution time. This can be explained by the mechanism for reporting dependencies. When a dependency

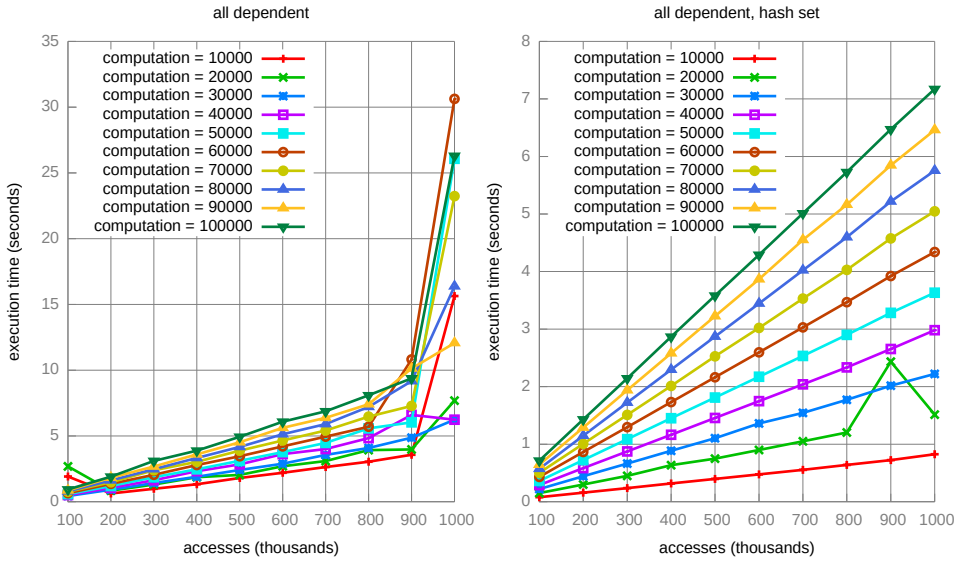

Figure 7.9: Execution times for bloom filters and no instrumentation when all operations are dependent

is detected, an exception is thrown. Although this provides the caller with semantic information regarding the dependency (possibly allowing further statistic to be computed), the overhead of throwing exceptions is high, as the runtime must walk the stack performing pattern matching as to when the exception can be caught. This results in the additional overhead of inadequate bloom filter configurations; increasing evidence for the need of well-configured bloom filters.

Figure 7.8 shows the same programs being run without instrumentation. As we can see, without instrumentation any fluctuation is minimal, with what little there is being explained by noise in the results (and it is not significant).

Execution time increases as a function of accesses, independent of any other variables; there is a highly linear relationship.

We can observe that with this synthetic benchmark, instrumentation increases execution times between a factor of 20 and 30 times. Although quite large, this is a relatively strong result for instrumented programs, which are often hundreds [Uh et al., 2006], [priv. comm.].

7.5 Real Impact

In this section, we consider the 'real-world' impact of instrumentation, as opposed to the benchmarks in section 7.4, as those benchmarks did not perform any additional

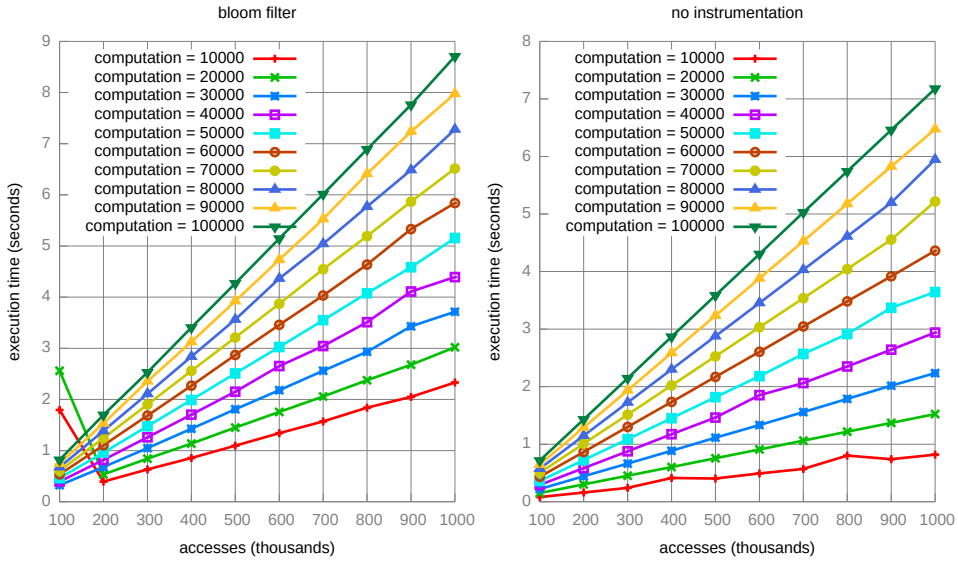

Figure 7.10: Execution times for bloom filters and no instrumentation when no operations are dependent

computation.

In these experiments, we tested the optimal bloom filter configuration (20x multiple) with operations that were 100% and 0% dependent. The variable in this experiment was the number of floating-point iterations per memory operation.

As we can see from charts 7.9 and 7.10, the addition of computation results in significant time increases, and the time required for the computation becomes the dominant factor. In other words, the overhead required as a result of the instrumentation is relatively negligable compared to that required for the computation. As a result, we can calculate that the overhead of the instrumentation if computation is included results in execution times of between a factor of 1.5 and 2 times greater than the uninstrumented programs. This is true for both hash sets and bloom filters, with the difference being negligible between the two.

7.6 Performance Tuning

Chapter 8

Conclusion

8.1 Concluding Remarks and Contributions

In this dissertation, several key contributions to the field have been made.

- We have investigated the use of bloom filters as an alternative to hash sets for dependency storage. Despite their probabilistic nature, we have shown that is is both feasible and advantageous to use bloom filters instead of hash sets for dependency analysis. Bloom filters significantly outperformed hash sets when memory usage is taken into consideration, and they outperform hash sets by roughly 10 to 15% when execution time is taken into consideration.
- A parametric benchmark has been developed, which enabled the creation of access pattern with well-known properties. This allows for 'apples-to-apples' comparisons to be made between various different approaches. Additionally, the use of this benchmark allows for a controlled setting within which the benchmarks can take place – one can only evalulate the efficacy of dependency detection algorithms if the number of dependencies is known. Unlike other (perhaps real-world) benchmarks where this value may not be known, with our benchmark it is known.
- Lastly, an instrumentation framework with an overhead of 20-30x over the baseline has been developed. We have shown how this framework performs with regard to the number of computations performed with each access.

We can draw the final conclusion that it is both possible and advantageous to use bloom filters in the pursuit of an automatic parallelising runtime. The use of bloom filters allows for both a negligible increase in memory size (on the order of tens of

megabytes versus hundreds for hash sets) and a relatively small increase in execution times of between a factor of 1.5 and 2 times slower than uninstrumented programs if a full dependency analysis is sought.

In the case of a binary decision regarding parallelisation (i.e., 'can this loop be executed in parallel or not?'), the speedup is potentially even larger; in our experiments the performance is comparable to that of uninstrumented programs.

We found that the optimal size for the bloom filter is between 10 and 20 times the number of actual insertions. This allows for the optimal trade-off between dependency detection (which was 100% accuracy for 20x in our testing) and memory usage, which was a factor of 3 lower than that required for hash sets.

8.2 Unsolved Problems

There are, however, several problems with remain with our implementation. These are mainly in the form of the lack of automatic instrumentation. We attempted to use the Graal compiler infrastructure to add these instructions, but unfortunately Graal does not yet support them. The largest issue was that in the case of a deoptimisation occuring, the interpreter does not have a bytecode index to commence execution from because the instruction does not exist within the bytecode itself.

In the coming weeks and months, the Graal development team will be adding new features and capabilities and soon will support this feature. Although we are currently unsure of the form this will take, it will in all likelihood be in a form that is easily compatible with our framework. Indeed, if it takes the form that we expect, no modifications will be required to the existing solution.

8.3 Future Work

The work that has been presented in this dissertation is a step forwards in dynamic parallelism detection. The framework correctly identifies data-parallel loops, and we have investigated whether the use of bloom filters affects the detection rate. However, there are still significant areas of future work that are possible.

Additional methods for trace storage could be analysed. Perhaps one such example is hash compaction, which instead stores compacted states in a hash table.

The framework presented is only currently capable of detecting parallelism at the level of loops (i.e., loop-level parallelism). The approach used could be extended towards a slicing-based approach, where instead of loops being instrumented, it could be possible to instrument given blocks. Whilst the approach is feasible, the framework would need to be modified in the sense that it can currently only instrument array accesses. Although a simple addition as the underlying mechanisms would not need to be modified, only their presentation API, this work is outside the scope of this dissertation. Additionally, such an approach would face the same problems as Wang et al. [2009] - as the slice coverage increases, the complexity increases combinatorially.

Additionally, although the framework *does* correctly detect parallelism at run-time, this information is current under-exploited by the runtime. It is possible, at least theoretically, that the framework could be combined with fellow student Ranjeet Singh's Java-to-OpenCL compiler, or perhaps utilise the already existing PTX or HSAIL backends in Graal in a JIT setting to produce a runtime system that dynamically detects parallel loops, recompiles then and executes using OpenCL (either on a CPU or GPU). However, there is an additional downside to this: the advantage (i.e., performance increase) of dynamic recompilation, moving execution to the CPU and then gathering the result must be greater than that of not doing so.

There are several overheads of doing so however. Although thread creation on CPUs is expensive in Java [Oracle Corporation, 2004], thread creation on GPUs is low [Mueller, 2009] (indeed, GPUs need 1000s of threads to operate efficiently in CUDA [Nvidia, 2011]).

Additionally, the overhead of data marshalling for GPGPUs (in general terms, the overhead of moving the data to the GPU and back again if required) is high, as the data does not exist within the cache hierarchy. In order to overcome this overhead, the advantage of performing the optimisation must be significant. A cost model could be performed, similar to Tournavitis et al. [2009], which combines static and dynamic analysis in order to provide intelligent run-time information to the JIT compiler regarding possible parallelisation for loops.

Appendix A

Precision Figures

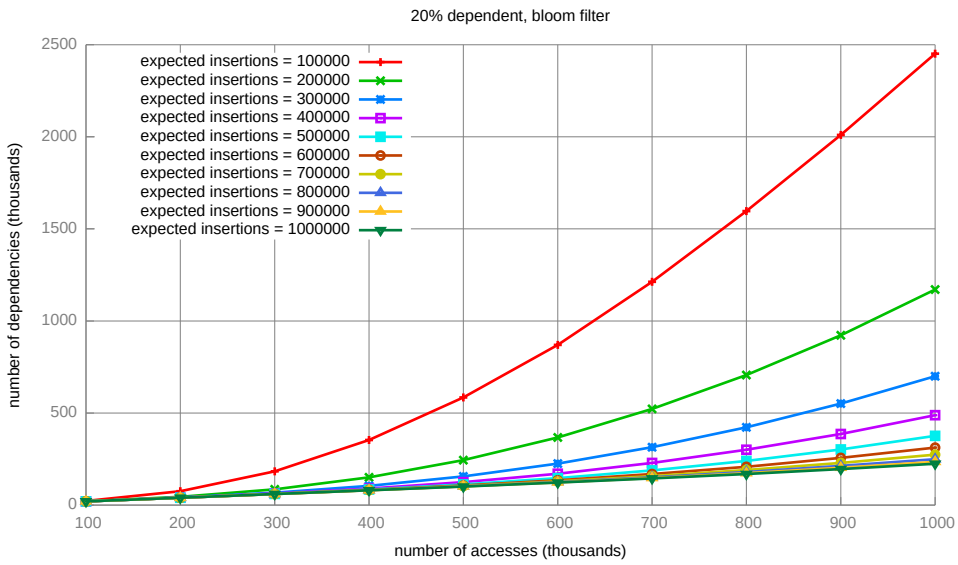

Figure A.1: Precision results for 20% dependent

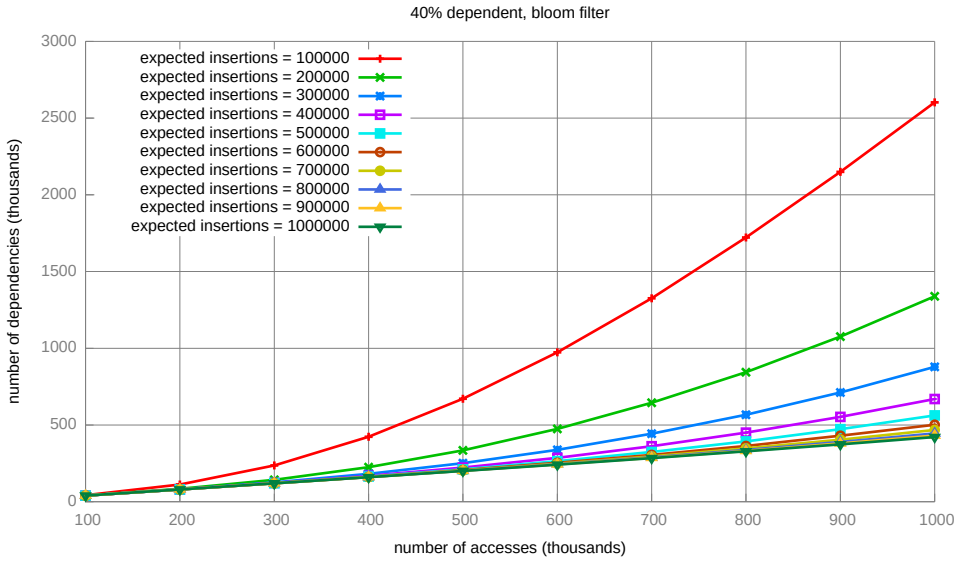

Figure A.2: Precision results for 40% dependent

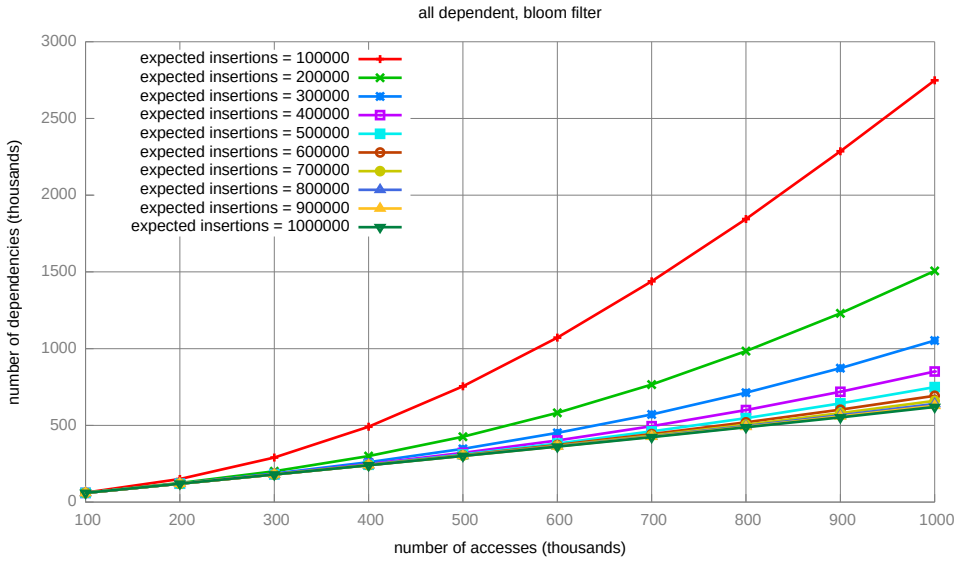

Figure A.3: Precision results for 60% dependent

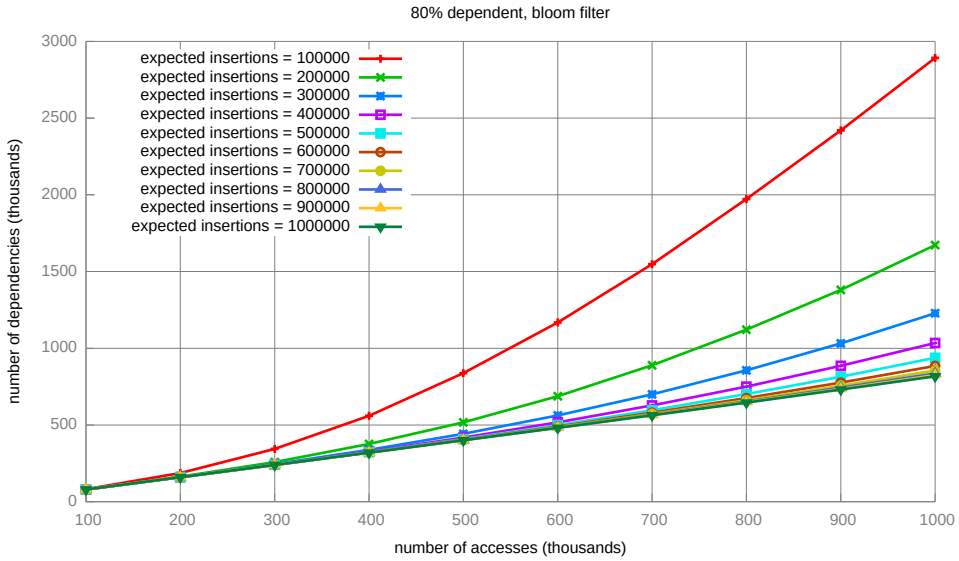

Figure A.4: Precision results for 80% dependent

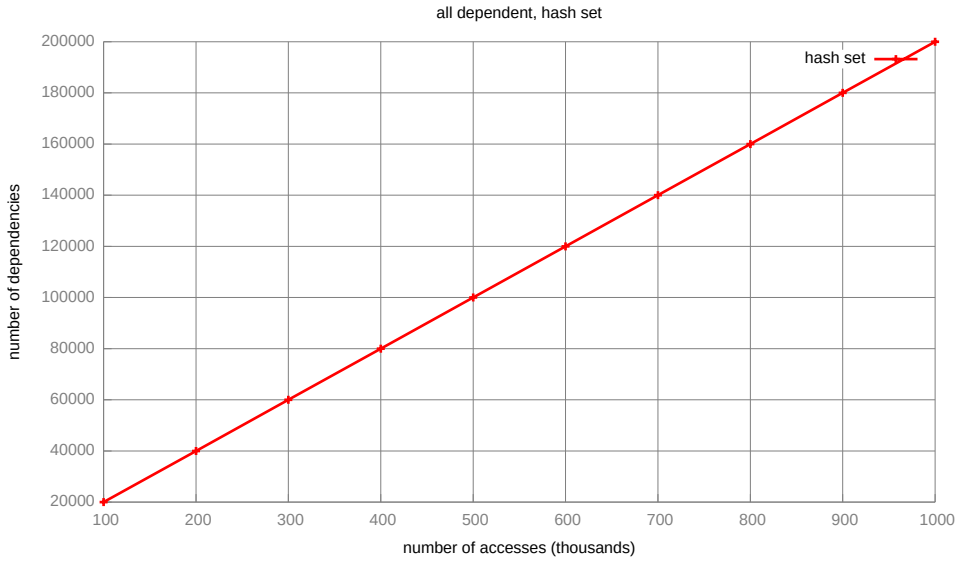

Figure A.5: Precision results for 20% dependent

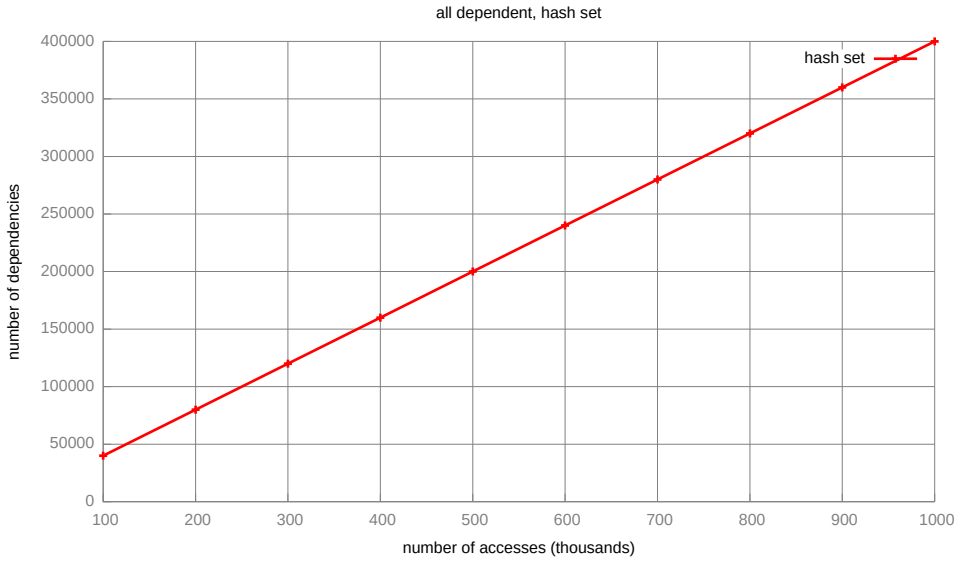

Figure A.6: Precision results for 40% dependent

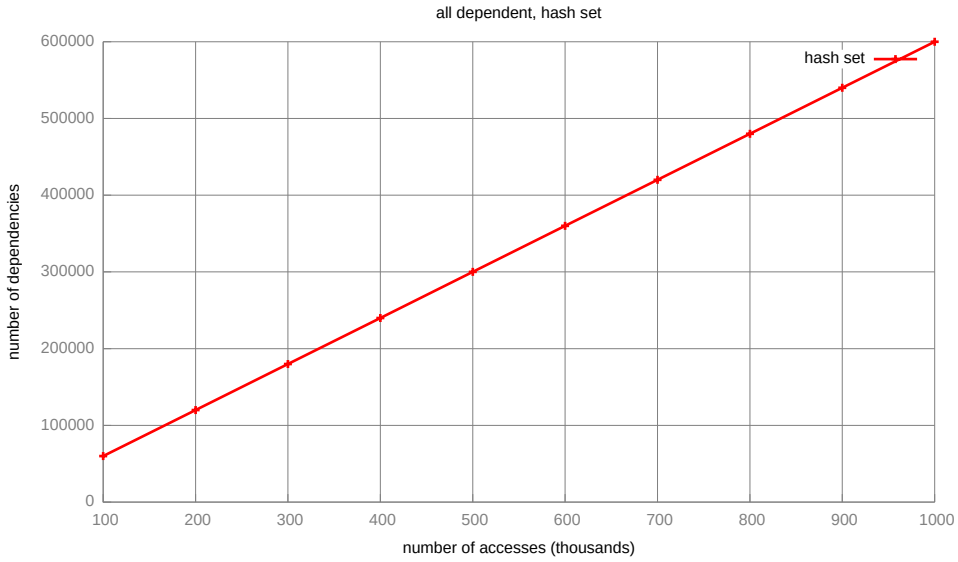

Figure A.7: Precision results for 60% dependent

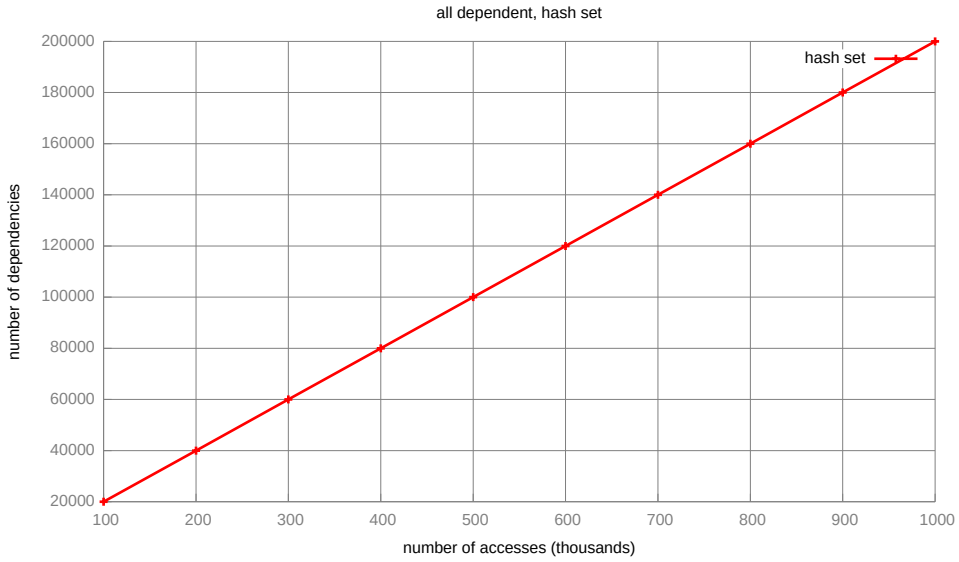

Figure A.8: Precision results for 80% dependent

Appendix B

Execution Time Figures

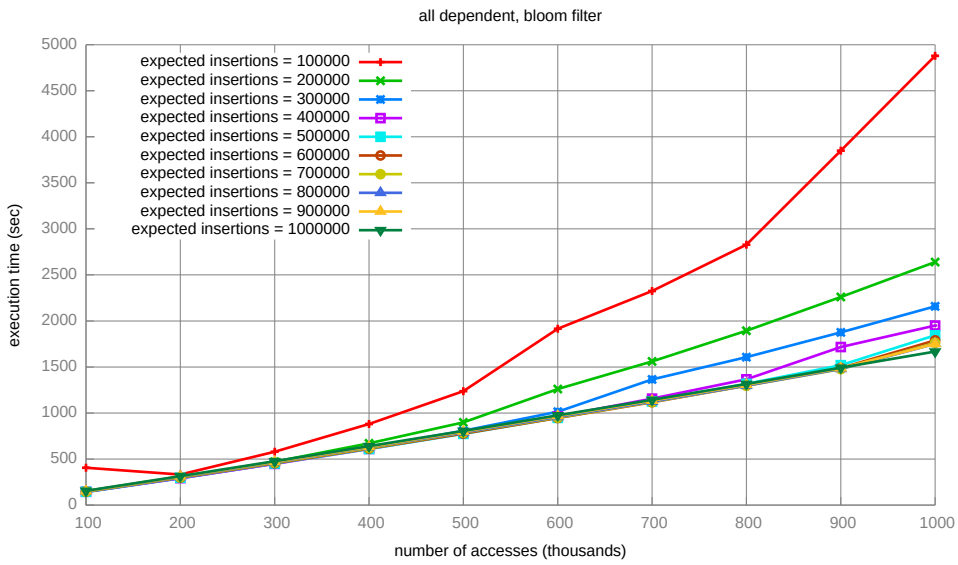

Figure B.1: Precision results for 20% dependent

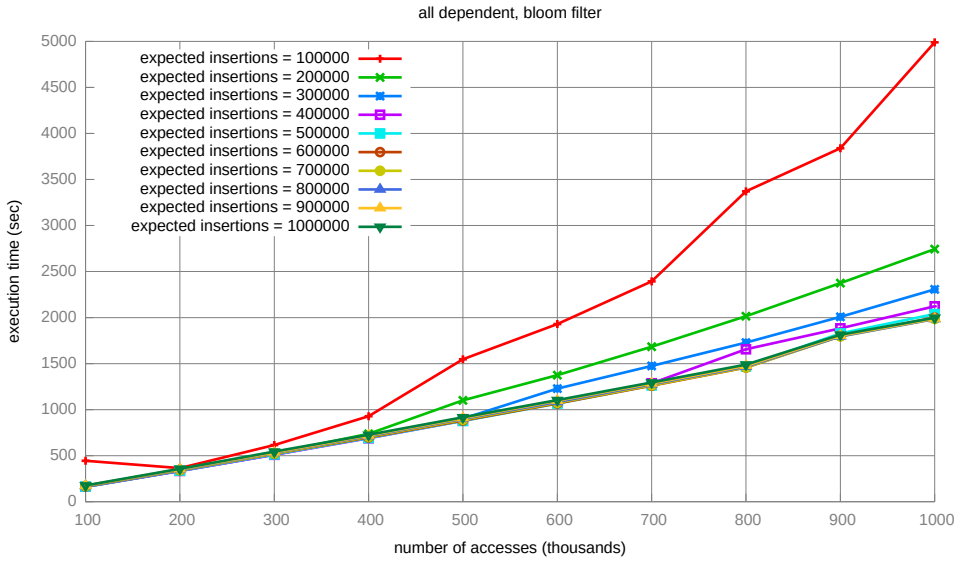

Figure B.2: Precision results for 40% dependent

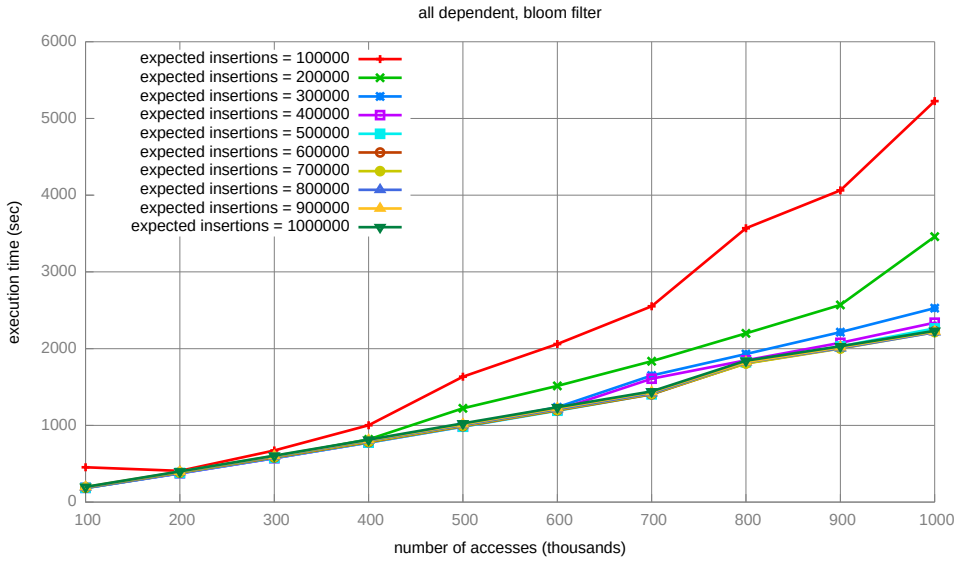

Figure B.3: Precision results for 60% dependent

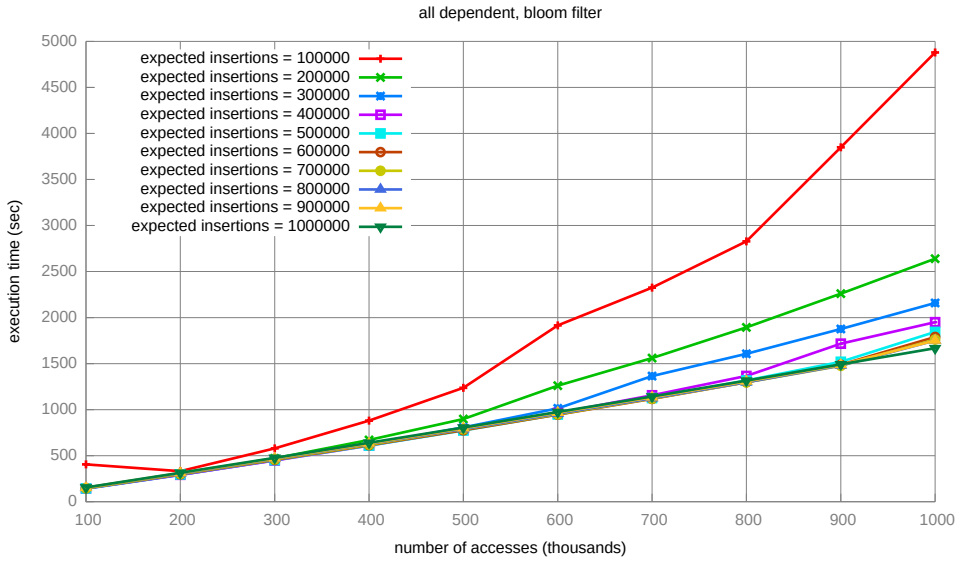

Figure B.4: Precision results for 80% dependent

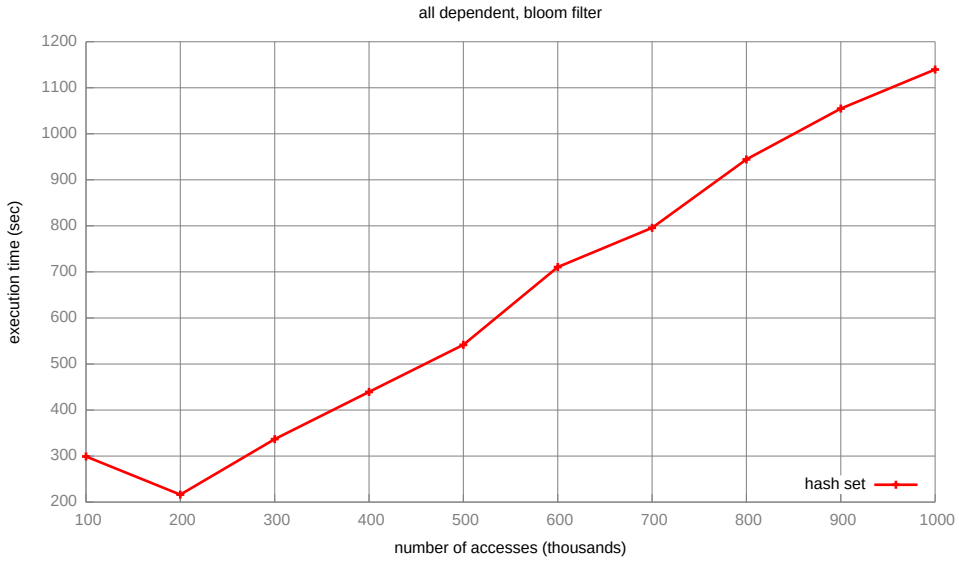

Figure B.5: Precision results for 20% dependent

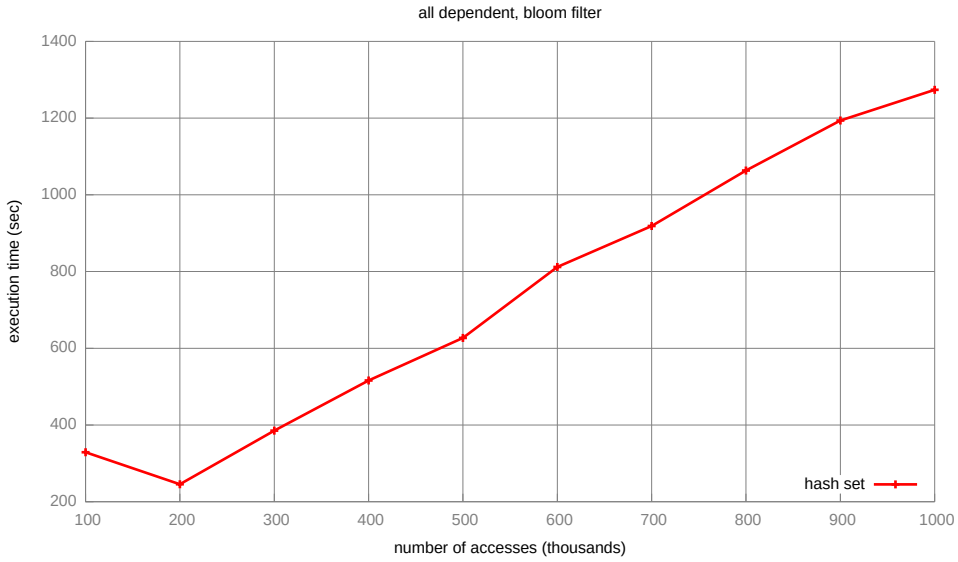

Figure B.6: Precision results for 40% dependent

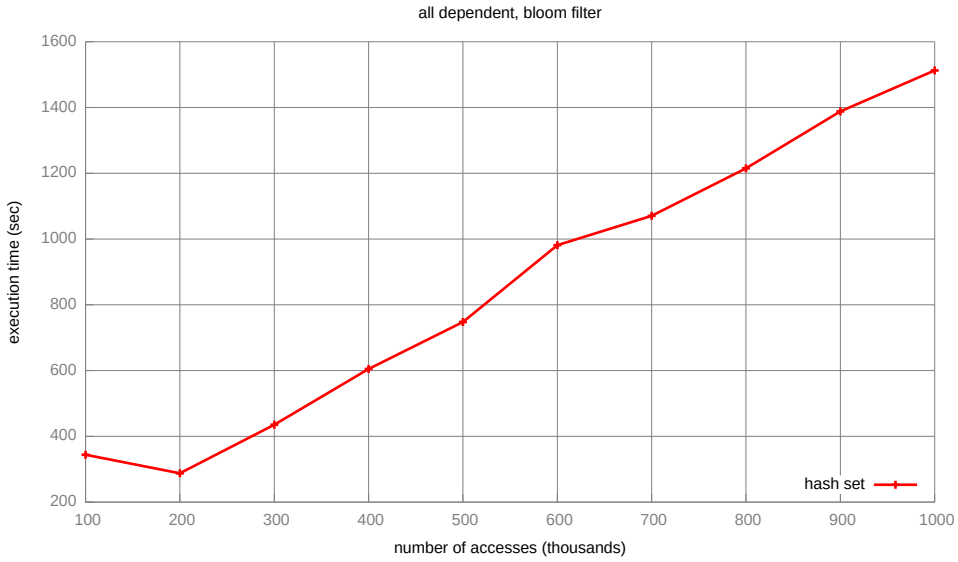

Figure B.7: Precision results for 60% dependent

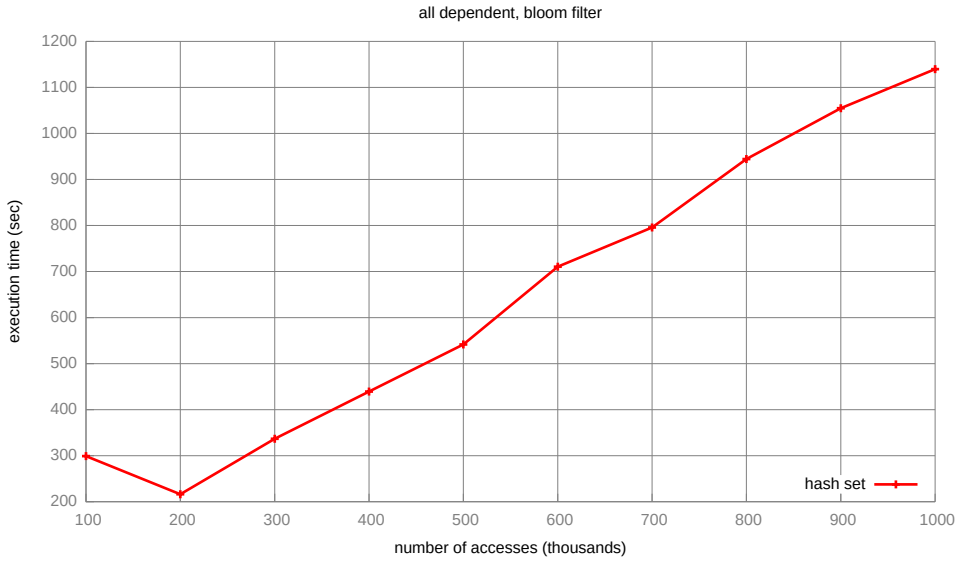

Figure B.8: Precision results for 80% dependent

Appendix C

Memory Usage Figures

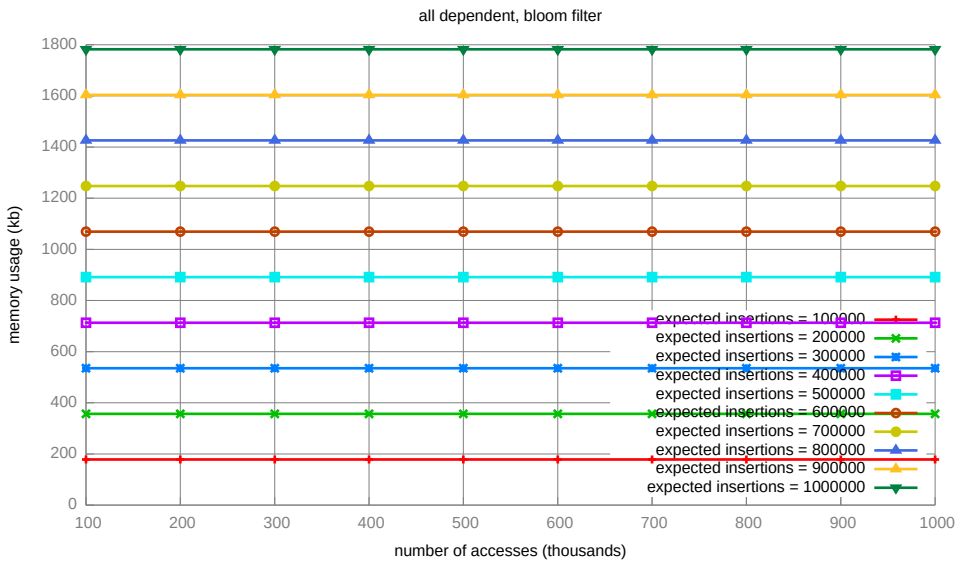

Figure C.1: Precision results for 20% dependent

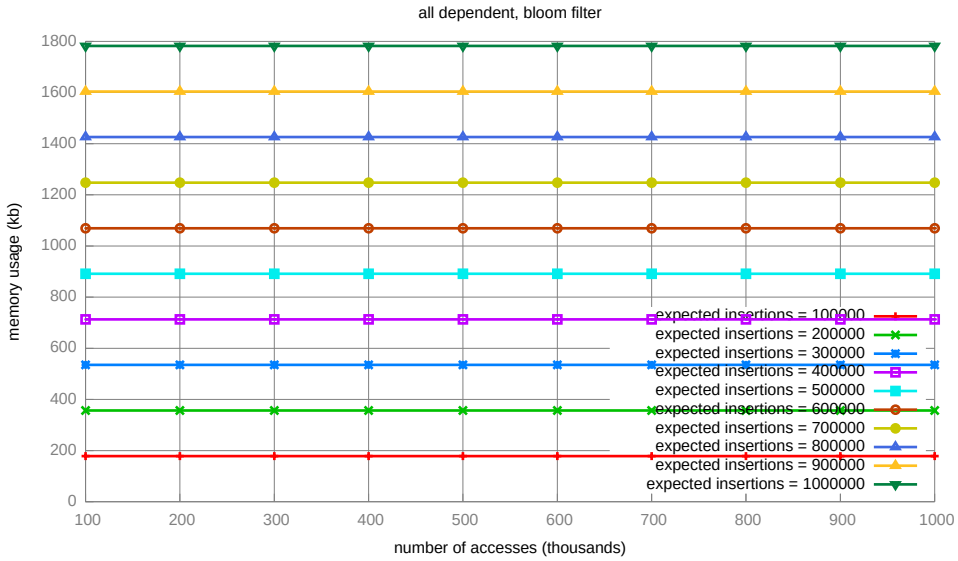

Figure C.2: Precision results for 40% dependent

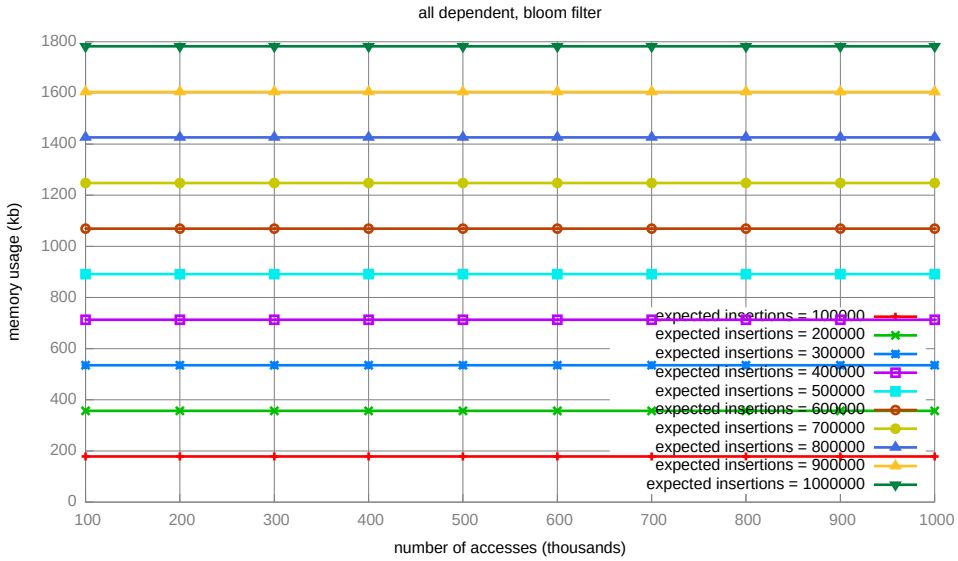

Figure C.3: Precision results for 60% dependent

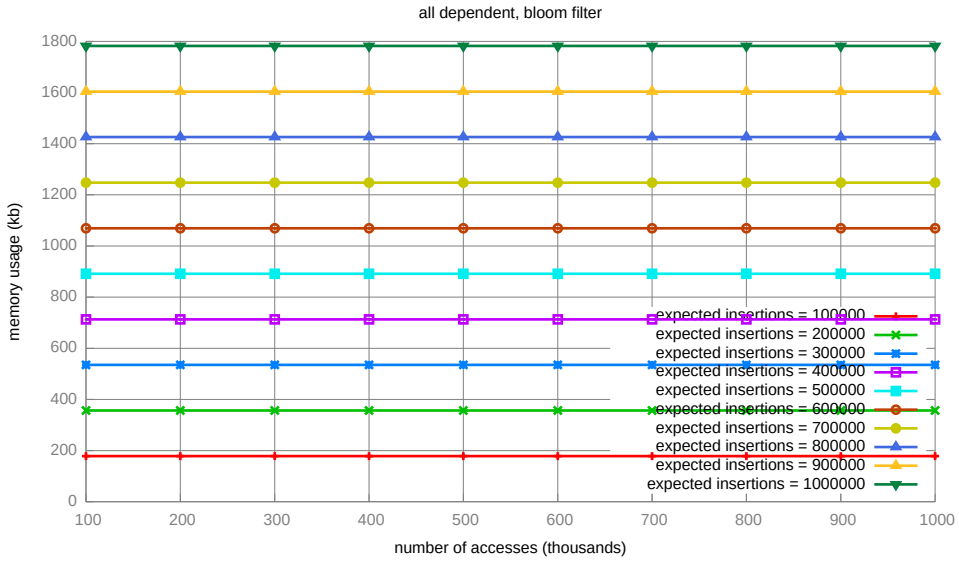

Figure C.4: Precision results for 80% dependent

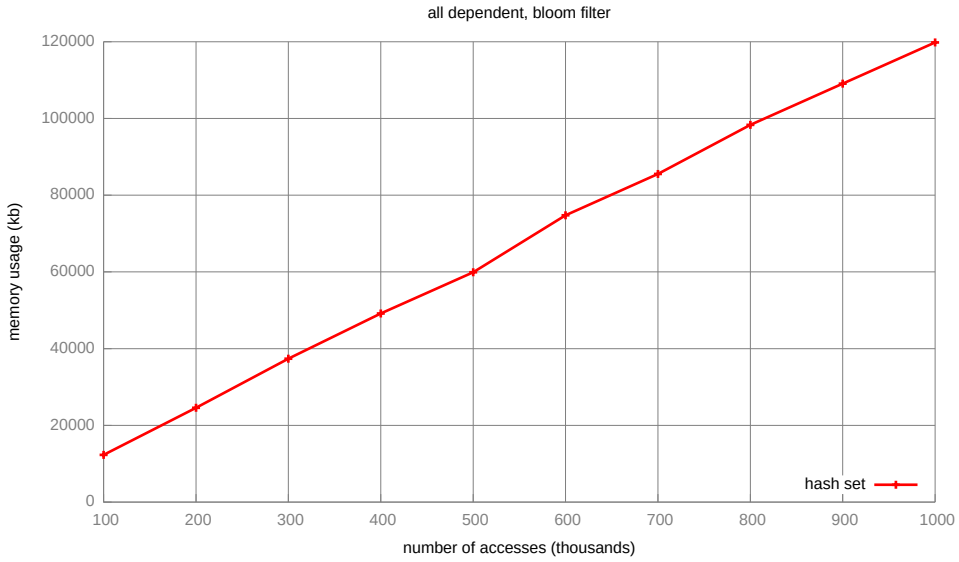

Figure C.5: Precision results for 20% dependent

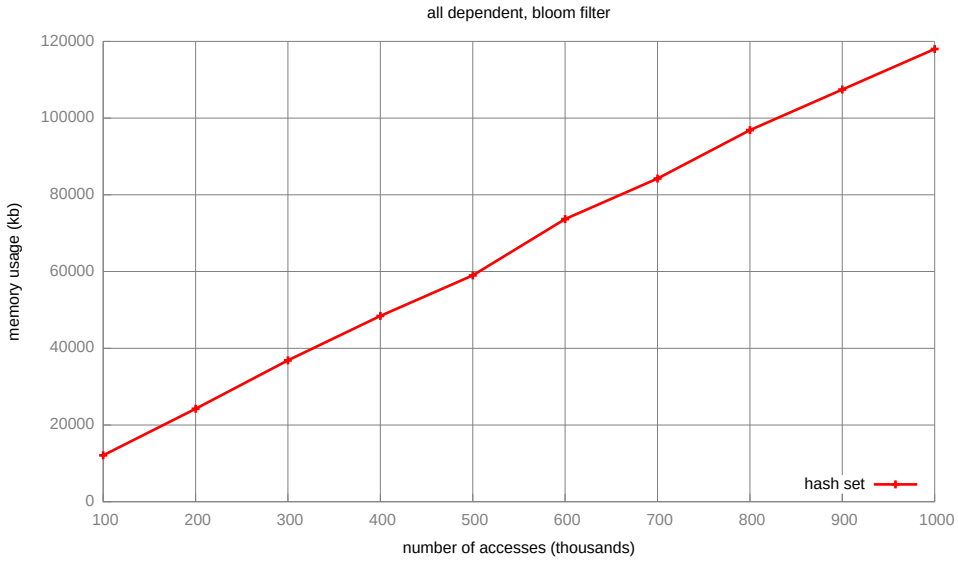

Figure C.6: Precision results for 40% dependent

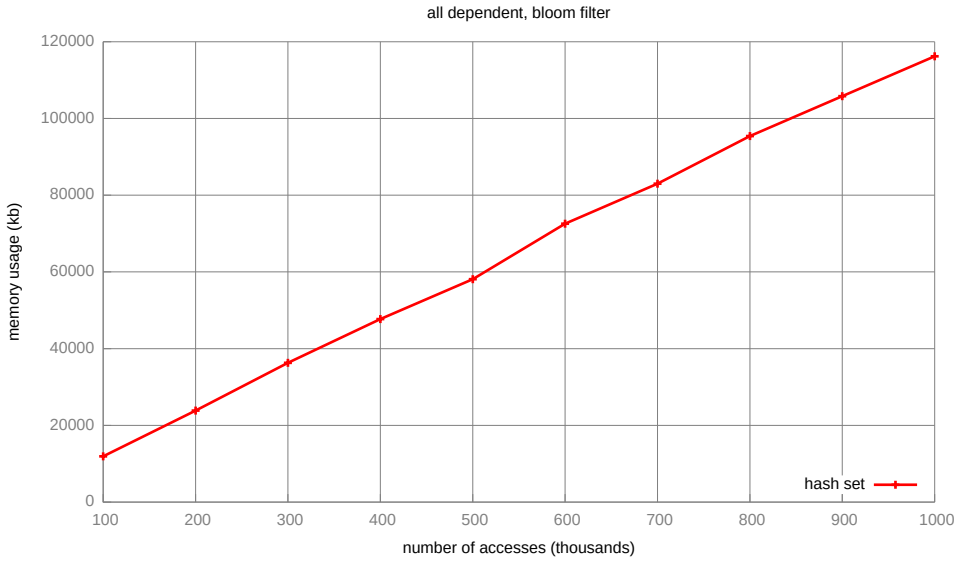

Figure C.7: Precision results for 60% dependent

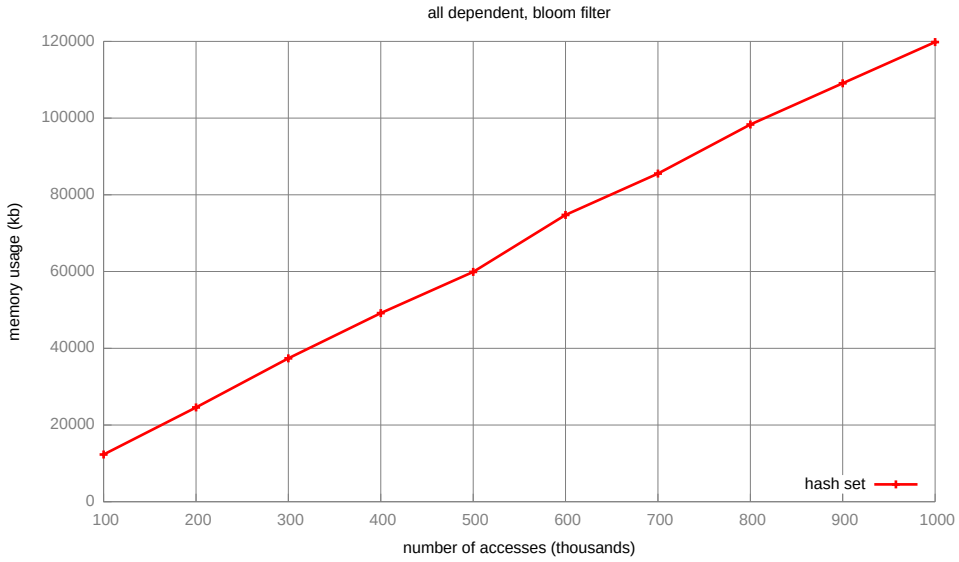

Figure C.8: Precision results for 80% dependent

Bibliography

Roslyn Project. URL https://www.microsoft.com/en-us/download/details.aspx?id=27744.

Arnold Schwaighofer. *Tail Call Optimization in the Java HotSpot VM Diplom-Ingenieur*. Masterstudium informatik thesis, Johannes Kepler University Linz, 2009. URL http://www.ssw.uni-linz.ac.at/Research/Papers/Schwaighofer09Master/schwaighofer09master.pdf.

Wikipedia. Hash table diagram, 2013a. URL https://en.wikipedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg.

Wikipedia. Bloom filter diagram, 2013b. URL https://en.wikipedia.org/wiki/File:Bloom_filter.svg.

B.A. Nayfeh and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997. ISSN 00189162. doi: 10.1109/2.612253. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=612253.

John Backus. The History of FORTRAN I, II and III. *IEEE Annals of the History of Computing*, 1(1):21–37, January 1979. ISSN 1058-6180. doi: 10.1109/MAHC.1979.10013. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4392880.

Jing Yang, Kevin Skadron, Mary Lou Soffa, and Kamin Whitehouse. Feasibility of Dynamic Binary Parallelization. *Usenix*, 2011. URL http://www.cs.virginia.edu/~skadron/Papers/yang_hotpar11.pdf.

Cheng Wang, Youfeng Wu, Edson Borin, Shiliang Hu, Wei Liu, Dave Sager, Tin-fook Ngai, and Jesse Fang. Dynamic Parallelization of Single-Threaded Binary Programs using Speculative Slicing. 2009.

M. Weiser. Reconstructing sequential behavior from parallel behavior projections. *Information processing letters*, 17(3):129–135. ISSN 0020-0190. URL http://cat.inist.fr/?aModele=afficheN&cpsidt=9606727.

Alain Ketterlin and Philippe Clauss. Transparent Parallelization of Binary Code.

Guoxing Dong, Kai Chen, Erzhou Zhu, Yichao Zhang, Zhengwei Qi, and Haibing Guan. A Translation Framework for Virtual Execution Environment on CPU / GPU Architecture. doi: 10.1109/PAAP.2010.53.

Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization. *ACM SIGPLAN Notices*, 44(6):177, May 2009. ISSN 03621340. doi: 10.1145/1543135.1542496. URL http://dl.acm.org/citation.cfm?id=1543135.1542496.

Javabeats.com. Introduction to Java Agents, 2012. URL http://www.javabeat.net/2012/06/introduction-to-java-agents/.

Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002. doi: 10.1.1.117.5769. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.5769.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Table_of_Contents. *Carcinogenesis*, 33(8):NP–NP, August 2012. ISSN 0143-3334. doi: 10.1093/carcin/bgs084. URL http://www.carcin.oxfordjournals.org/cgi/doi/10.1093/carcin/bgs084.

Jason McDonald. Design Patterns. Technical report, DZone, 2008. URL http://cs.franklin.edu/~whittakt/COMP311/rc008-designpatterns_online.pdf.

Apache Foundation. Apache Bytecode Engineering Library, 2013a. URL http://commons.apache.org/proper/commons-bcel/.

James Strachan and The Groovy Project. org.codehaus.groovy.classgen.asm, 2013. URL http://groovy.codehaus.org/gapi/org/codehaus/groovy/classgen/asm/package-summary.html.

Shigeru Chiba. Javassist - A Reflection-Based Programming Wizard for Java. In *Proceedings of the OOPSLA workshop on Reflective Programming in C and*, 1998. URL http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/oopsla98/proc/chiba.pdf.

Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es): 154–es, December 1996. ISSN 03600300. doi: 10.1145/242224.242420. URL http://portal.acm.org/citation.cfm?doid=242224.242420.

C Constantinides, T Skotiniotis, and M Stoerzer. AOP considered harmful. Technical report, Concordia University, 2004. URL http://pp.info.uni-karlsruhe.de/uploads/publikationen/constantinides04eiwas.pdf.

Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968. ISSN 00010782. doi: 10.1145/362929.362947. URL http://portal.acm.org/citation.cfm?doid=362929.362947.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An Overview of AspectJ. *Main*, 2072:327–353, 2001. ISSN 03029743. doi: 10.1007/3-540-45337-7_18. URL http://www.cs.ubc.ca/~kdvolder/binaries/aspectj-overview.pdf.

Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege De Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: The AspectBench Compiler for AspectJ. In *Generative Programming and Component Engineering 4th International Conference GPCE*, volume 3676, pages 10–16, 2005. ISBN 3540291385. doi: 10.1007/11561347_2.

Bruno Harbulot and John R Gurd. A join point for loops in AspectJ. In *Computer*, pages 63–74, 2005. ISBN 159593300X. doi: 10.1145/1119655.1119666.

Kung Chen and Chin-hung Chien. Extending the Field Access Pointcuts of AspectJ to Arrays. *Journal of Software Engineering Studies*, 2(2):2–11, 2007. URL http://www.geocities.ws/m8809301/pub/JSESv2n2_KungChen_970214.pdf.

Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. DiSL. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development - AOSD '12*, page 239, New York, New York, USA, March 2012. ACM Press. ISBN 9781450310925. doi: 10.1145/2162049.2162077. URL http://dl.acm.org/citation.cfm?id=2162049.2162077.

Carlo A. Furia and Sebastian Nanz, editors. *Objects, Models, Components, Patterns*, volume 7304 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-30560-3. doi: 10.1007/978-3-642-30561-0. URL http://www.springerlink.com/index/10.1007/978-3-642-30561-0.

S M Blackburn, R Garner, C Hoffman, A M Khan, K S McKinley, R Bentzur, A Diwan, D Feinberg, D Frampton, S Z Guyer, M Hirzel, A Hosking, M Jump, H Lee, J E B Moss, A Phansalkar, D Stefanović, T VanDrunen, D Von Dincklage, and B Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *ACM Sigplan Notices*, 41:169–190, 2006. ISSN 03621340. doi: 10.1145/1167515.1167488.

Oracle and OpenJDK. Graal Project, 2012. URL http://openjdk.java.net/projects/graal/.

Tim Lindholm, Alex Buckley, Gilad Bracha, and Frank Yellin. The Java Virtual Machine Specification Java SE 7 Edition. Technical report, Oracle, Inc, Redwood City, 2013. URL http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf.

Alfred V. Aho, Monica S. Lam, R Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson, second edition, 2007. ISBN 0-321-49169-6.

Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco, first edition, 1997. ISBN 1-55860-320-4.

Neal Glew and Jen Palsberg. Type-Safe Method Inlining. In Boris Magnusson, editor, *ECOOP 2002 Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 525–544. Springer Berlin Heidelberg, Berlin, Heidelberg, May 2002. ISBN 978-3-540-43759-8. doi: 10.1007/3-540-47993-7. URL http://link.springer.com/10.1007/3-540-47993-7.

Chang Peng. Loop Prediction, 2010. URL https://wikis.oracle.com/display/HotSpotInternals/LoopPredication.

Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. *ACM Sigplan Notices*, 27:32–43, 1992. ISSN 03621340. doi: 10.1145/143103.143114.

Javassist Project. Javasssist documentation, 2013.

Sahni Sartaj. *Data Structures, Algorithms and Applications in C++*. McGraw-Hill, first edition, 1998. ISBN 0-07-109219-6.

Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970. ISSN 00010782. doi: 10.1145/362686.362692. URL http://dl.acm.org/citation.cfm?id=362686.362692.

J.Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979. ISSN 00220000. doi:

10.1016/0022-0000(79)90044-8. URL http://linkinghub.elsevier.com/retrieve/pii/0022000079900448.

S Joshua Swamidass and Pierre Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling*, 47(3):952–64, 2007. ISSN 1549-9596. doi: 10.1021/ci600526a. URL http://www.ncbi.nlm.nih.gov/pubmed/17444629.

Apache Foundation. Cassandra Architecture Overview, 2013b. URL http://wiki.apache.org/cassandra/ArchitectureOverview.

Austin Appleby. SMHasher and MurmurHash, 2013. URL http://code.google.com/p/smhasher/.

Adam Kirsch and Michael Mitzenmacher. Less Hashing , Same Performance : Building a Better Bloom Filter. *Building*, 33:456–467, 2006. ISSN 10429832. doi: 10.1002/rsa.

Randy Allen and Ken Kennedy. *Optimising Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, San Francisco, first edition, 2000. ISBN 1-55860-286-0.

Ronald Ibbett. High Performance Computer Architectures, 2009. URL http://homepages.inf.ed.ac.uk/cgi/rni/comp-arch.pl?Paru/depend.html, Paru/depend-f.html,Paru/menu.html.

William Stallings. *Computer Organisation and Architecture: Design for Performance*. Pearson, Harlow, nineth edition, 2013. ISBN 0-273-76919-7.

Donald E. Knuth. *The Art of Computer Programming Volume 2: Seminumerical Algorithms*. Addison Wesley, third edition, 1997. ISBN 0-201-89683-2.

Oracle. Autoboxing, 2010. URL http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html.

Jin-Soo Kim and Yarsun Hsu. Memory system behavior of Java programs. *ACM SIGMETRICS Performance Evaluation Review*, 28(1):264–274, June 2000. ISSN 01635999. doi: 10.1145/345063.339422. URL http://dl.acm.org/citation.cfm?id=345063.339422.

Kazunori Ogata, Dai Mikurube, Kiyokuni Kawachiya, Scott Trent, and Tamiya Onodera. A study of Java's non-Java memory. *ACM SIGPLAN Notices*, 45 (10):191, October 2010. ISSN 03621340. doi: 10.1145/1932682.1869477. URL http://dl.acm.org/citation.cfm?id=1932682.1869477.

Oracle Inc. Runtime (Java Platform SE 7), 2013. URL http://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html.

Jonathan Ellis. Java Agent for Memory Measurements, 2011. URL https://github.com/jbellis/jamm/.

Apache Commons. EnumeratedDistribution¡T¿ documentation, 2013. URL http://commons.apache.org/proper/commons-math/javadocs/api-3.2/org/apache/commons/math3/distribution/EnumeratedDistribution.html.

Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing Dynamic Binary Instrumentation Overhead. *WBIA Workshop at ASPLOS*, 2006.

Oracle Corporation. JSR-133: Java Memory Model and Thread Specification. 2004. URL http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf.

Klaus Mueller. GPU Programming: CUDA Threads. Technical report, Stony Brook University, New York, New York, USA, 2009. URL http://www.cs.sunysb.edu/~mueller/teaching/cse591_GPU/threads.pdf.

C Nvidia. NVIDIA CUDA C Programming Guide. *Changes*, page 173, 2011.