Dynamic Parallelism Detection for the HotSpot Java Virtual Machine

Christopher Edward Atkin-Granville

Master of Science
Computer Science
School of Informatics
University of Edinburgh

2013

Abstract

Contemporary computers are highly 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 consider two main approaches to trace collection: exact approaches and probabilistic approaches.

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 parents, Sandra and Ian who have supported me throughout my entire University career.

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

List of Algorithms

Chapter 1

Introduction

In this chapter, some background to the problem of parallelism and parallel expression is presented. Some alternative approaches to dynamic detection are covered (i.e., alternative parallel expression techniques), with a critical analysis of their strengths and merits. Lastly, a brief overview of the outline of this dissertation is presented.

1.1 Background

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.

1.2 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

[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.

hardware (i.e., the abstraction layer that compilers introduce, with a special focus on ISA abstraction) meant that developers could simply urge their users to buy new hardware for a performance improvement.

In a slightly less-than-idea 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 would 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, involved 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 (EHL) languages (sometimes called scripting languages) commonly sacrificed execution speed for programmer ease of use, safely, 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

1.3 Cheating the System

However, these increases cannot occur indefinitely. There exists not only a fundamental lower-bound on the size of an individual transistor (as a result of quantum tunnelling), but also the extent to which contemporary techniques can provide performance improvements. For example, many common 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, 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 more simply through increasing the clockspeed.

1.4 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 are slightly slower when taken individually. 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 (such as ray tracing and scientific computing). 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.

1.5 Parallelist Approaches

One solution to this problem that has been advocated by many theoretical computer scientists and language designers is to switch language paradigm to functional languages such as Haskell and Erlang. Languages and systems of this class do have significant advantages - both theoretical and applied - over more conventional paradigms. Purely functional languages have first-class, referentially-transparent functions which can be easily reasoned about by both the user and the compiler. However, here are two main disadvantages to this approach. Although it is the most optimal from a theoretical perspective (and mainstream switching to a functional language would bring many benefits, not just increased parallelism), it would require rewriting existing programs in a functional language. Moreover, most developers do not have experience with functional languages - and are not, unfortunately, willing to learn. It is not a problem of technology - more one of education.

Moving more towards the practical side of things, there are two main alternatives. The first is to use language-level constructs which are built into the semantics of the language. There has been some work in this area, and they usually focus on extending existing languages with parallel semantics. These additions usually provide language-level constructs for message passing, parallel loop construction, barriers and other such low-level parallelism primitives. Although there are many such research languages (Click-5, Chapel, X10 etc), there exists no commonly used language supporting these features. There are also some hybrid languages, such as Lime which are backwards compatible with existing infrastructures. The advantages of these languages is that because they support parallelism primitives intrinsically, reasoning and inference (both by humans and in an automated fashion) is much easier than it is for other languages. Adding support for parallelism at the language level also, depending on the level of abstraction used, implies tying a language to a particular parallel paradigm (or set of paradigms). If a language implements low-level constructs instead of higher-level constructs (similar to algorithmic skeletons except at the language level), then the user has to implement many commonly used operations manually, leading to bugs and inconsistencies between implementations. The education issue is also present with language-level constructs. Despite these disadvantages, some progress has been made in this area by Microsoft with PLINQ (Parallel Language Integrated Query), which is a declarative extension to CLI languages. PLINQ is successful because it requires little to no changes from standard LINQ - this is the ideal scenario in many cases (although note that PLINQ is not a general-purpose solution).

A more pragmatic approach to language-level constructs is a library-based approach such as POSIX Threads (Pthreads), OpenMP and OpenMPI. Pthreads is an implementation of the POSIX (*Portable Operating System Interface*) standard regarding threads and is therefore compatible with a wide variety of hardware and operating systems (including some non-POSIX compliant systems such as Windows). OpenMP is a library for C, C++ and Fortran for shared-memory programming, although it is commonly used to implement parallel loops (i.e., work sharing). OpenMPI is similar to OpenMP with the exception that it is designed for distributed-memory programming instead. The advantage of a library-based approach is that users can 'mix-and-match' between different libraries - for example, a user can use both OpenMP and OpenMPI within the same program. However, many of the libraries require significant low-level knowledge of parallelism, and are not particularly user friendly.

To illustrate the substantial difficulties of adding parallel semantics to existing programs with sequential semantics, let us consider the challenges programmers face when using parallel features in programming languages, such as threads. Mutual exclusion is, currently, the most widely adopted parallel programming paradigm, as it is the easiest to use (although it does not nessecarily offer the greatest performance; nevertheless it can offer performance greater than that of sequential programs). Some common issues include [Fraser, 2004; Herlihy et al., 1993; Cole, 2013]:

- **Deadlock**: where two or more threads are waiting on each other to finish before continuing
- **Livelock**: similar to deadlock, except that the threads are changing state yet not achieving work
- **Priority inversion**: when a thread with high priority is pre-empted by a thread with a lower priority
- **Race conditions**: the result of two or more threads attempt to access the same resource(s) at the same time, leading to non-deterministic behaviour

There have been some attempts to mitigate these issues (e.g. Liu et al. [2011]), but they have not yet been widely adopted.

Many web-scale companies (Facebook, Google, Yahoo! etc) have large datasets that require (offline) processing and are now using parallel infrastructures such as Hadoop (MapReduce) to support this computation. Such infrastructures are similar to library-based solutions except that they also provide management solutions and other such features. Hadoop, along with its sister projects HDFS, Hive and HBase, provide an entirely framework for programming, executing, distributing and managing parallel applications. An 'all-in-one' solution such as Hadoop is extremely attractive simply because it provides everything one needs to set up distributed/parallel applications. The disadvantage is that, in general, such frameworks are only suited to a single kind of parallel framework. In the case of Hadoop, it can only support MapReduce-based computations. In other words, all computations that are not based around a MapReduce model are incompatible with Hadoop. Additionally, such frameworks typically require a large number of resources in order to be effective (Hadoop may actually *reduce* performance of computations if the number of processors is small), although that disadvantage is not inherent to MapReduce-style computations.

Perhaps the 'holy grail' of parallelism is the idea of an auto-parallelising compiler. In other words, a compiler that can infer enough information about the semantics of the program in order to apply transformations that convert a program with sequential semantics into one with parallel semantics. This approach sounds extremely attractive, as it would not require (ideally) any effort on the behalf of the programmer; despite this there are significant disadvantages. Firstly, given the scope and context of contemporary common languages (C, C++, Objective-C, C#, Java), applying compile-time transformations to add highly context-sensitive parallel semantics is likely in intractable problem. The reason for this is that, with current compiler technology, it is not possible to infer enough information regarding the semantics and syntax of the languages to reason about them. To illustrate this point, even when additional parallel semantics are added to existing languages (with sequential semantics), loops may still not be easily parallelisable because of pointer aliasing For example, imagine a function that returns an array of values which cannot be determined at compile-time. If that array is then used to index another array (say in a vector addition), reasoning about the parallel semantics of that vector addition are intractable at compile-time.

The last kind of parallelism is hardware-supported parallelism through constructs such as transactional memory (*TM*). TM solutions have the advantage of being easily programmable (to the extent that they are as easy as marking a method as *synchronized*, similar to how Java's monitors operate i.e., coarse-grained parallelism) whilst retaining the performance of fine-grained parallelism. The disadvantage of this approach is that high-performance TM architectures are reliant on hardware support (although software-based approaches do exist, they typically exhibit significantly lower performance characteristics than a similar hardware-based approach). Additionally, simply adding TM into a program does not add parallel semantics to a program with sequential semantics - constructs that allow e.g. threads to be created still need to be added. TM can rather be seen as of a way to make parallel programming easier, not a complete solution.

Transactional memory is, in effect, an optimistic memory model in that multiple threads attempt several transactions at the same time, and any conflicting writes are *rolled fine*. The final operation to be performed in a transaction is a *commit* operation - where the changes are recognised permanently in global state. However, unlike database management systems which are concerned with the ACID properties

[Garcia-Molina et al., 2002, p. 14]:

- **Atomicity**: the 'all-or-nothing' execution of transactions
- **Consistency**: no transaction can move global state from a consistent state to an inconsistent state
- **Isolation**: the fact that each transaction must appear to be executed as if no other transaction is executing at the same time
- **Durability**: the condition that the effect on global state of the transaction must never be lost once the transaction is complete

However, transactional memory when used in the context of computer systems (TM can be implemented in both hardware and software, but as the semantics are the same we shall consider only transactional memory 'in the large'), we are principally concerned with only atomicity and isolation, as we assume that changes to memory do not need to be durable (memory operations are transient) [Marshall, 2005].

Another way of utilising hardware support in parallelism is to make use of low-level constructs such as compare-and-swap, test-and-set (and test-and-test-and-set) and so on. These mechanisms are very simple, but they allow programmers to implement more complicated and sophisticated parallelism constructs using them. For example, the test-and-test-and-set operation can be used to implement locks in the following way:

```
boolean function test-and-set(boolean v) {
    < boolean initial = v;
      v = true;
      return initial;
    >  }
// usage of test-and-set
lock_t lock = false;
co [int i = 1 to n] {
    while (something) {
        // test lock before test-and-set(lock)
        // via short-circuit evaluation
        // this is to avoid trashing the cache
        while (lock || test-and-set(lock)) ;

        critical-section();
        lock = false;
        non-critical-section();
    }
}
```

Listing 1.1: Pseudocode for a test-and-set atomic operation and its use

Note that any code within $<$ and $>$ occurs atomically, and the `co` notation refers to n threads executing the block in parallel.

However, such approaches are equivalent to using a low-level language such as assembly or C to write a modern application - they are simply too low-level for programmers and are highly error prone.

It is interesting to note that languages with theoretical, rather than pragmatic, roots display excellent characteristics for automatic parallelisation. For example, functional languages display characteristics such as referential transparency. Referential transparency is the property that a function is composed entirely of *pure functions* - functions that do not modify global state. The advantage of this, along with the other properties of functional languages, is that it allows the compiler to reason about the program. Functions that display referential transparency lend themselves to easy automatic parallelisation.

Indeed, this idea of higher-level languages displaying good parallelisation characteristics extends to languages other than just functional languages. Many declarative languages (where the programmer describes *what* he/she wants to happen, in a sense without expressing control flow) lend themselves to automatic parallelisation, because they allow the compiler (or interpreter) to reason about the language. Declarative languages are semantically a ‘purer expression’ of computation in the sense that they do not contain implementation details, which allows this reasoning by the compiler.

1.6 Language Semantics and Compiler Reasoning

Referential transparency is advantageous for parallelisation because it limits shared state. The reason that the techniques listed above such as semaphores, mutexes and so on is to limit concurrent access to shared state variables. As referential transparency specifically disallows shared state, the advantages are clear. To give a concrete example of this, consider the following code, noting that function `f` is declared as referentially transparent:

```
int transparent function f(int x, int y) {
    return x * y;
}

results = int array[10000][10000]

for int i = 0 to 10000 {
    for int j = 0 to 10000 {
        results[i][j] = f(i, j)
    }
}
```

Listing 1.2: Example of a referentially transparent function

Because f has been declared as transparent, the compiler can perform aggressive optimisations - e.g. for p processors, computing $\frac{10000}{p}$ iterations of the outer loop on each processor. It must be emphasised that the specific runtime configuration is not important - it is up to the compiler or runtime to determine the appropriate execution strategy. In theory, the compiler or runtime could even execute the loops on a more specialised device, such as a GPU.

So far in this chapter we have justified the use of declarative languages over imperative/procedural languages for parallelisation because they allow the compiler to reason about the semantics of the language [Peyton-Jones, 2011]. But what does it mean for a compiler to 'reason' about a language? To answer this, let us consider some examples.

Consider the following SQL statement:

```
SELECT
    username, firstname, lastname
FROM
    Users
WHERE
    id = 3
```

Listing 1.3: Sample SQL query

As we know, SQL is a declarative language. The user has *no* control over the way that the SQL compiler/optimiser/interpreter/runtime executes the query, and this is precisely the power that declarative languages offer. Research has shown that optimistic optimistic compilers for declarative languages can, in some cases, outperform hand-tuned C code [Anderson et al., 2013; Mainland et al., 2013].

Declarative languages have already successfully been applied to several different area of parallel computing [Orchard et al., 2010; Grossman et al., 2011; Holk et al., 2011].

1.7 Outline

This dissertation is split into several chapters.

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 major results are considered.

2.1 Parallelising Compilers

Programs and/or runtime systems are attractive because they require no access to the source code. In 2011, Yang et al. [2011] introduced one of the main advances on 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. 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

Ketterlin and Clauss

Dong et al.

2.2 Parallelism Detection

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.

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

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 displays shows that a language (and associated platform) has a certain level of 'maturity', that it is now ready for large software projects (or at

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

least not totally unprepared for one).

Graal is an attempt to bring this approach to the Java language. 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 for their .NET platform. The new Windows Runtime include deep metadata integration into the platform (which is the basis for, amongst other things, the Common Object System in .NET languages[2]).

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).

[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

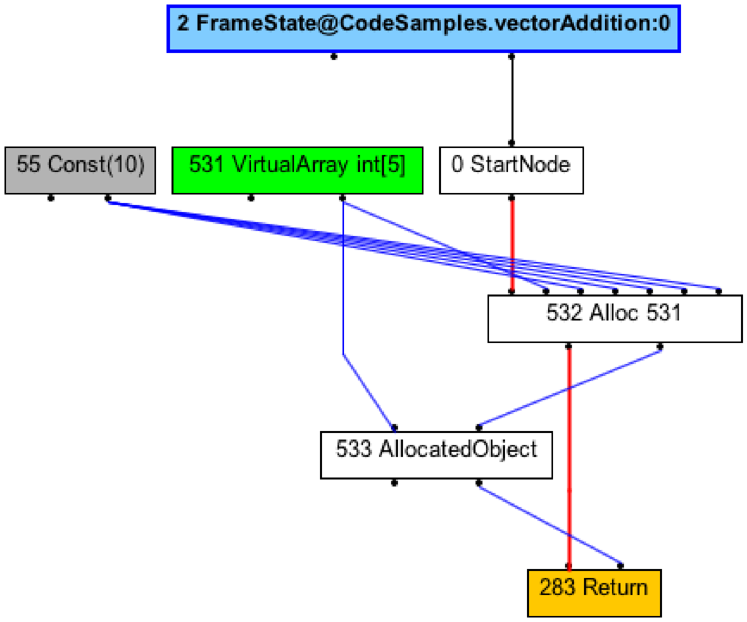

Figure 3.1: Graal HIR created for a vector addition using two array literals

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 the figure 3.1, 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 represented on the y-axis of the graph, flowing down the page.

Figure 3.1 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 actual memory 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[3].

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 via `com.oracle.graal.phases.PhasePlan.disablePhase()`. 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.

[3] One such tool is the 'Ideal Graph Visualiser', which is included in the Graal repository.

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.

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.

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

Chapter 4

Instrumentation

Flexible and efficient instrumentation is the foundation of this project. Here, possible approaches to this instrumentation are discussed, along with critical analyses.

4.1 Graal

To write.

4.2 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.

4.2.1 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

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

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 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 *Locomotion* would not need to download and install Graal.

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.

4.2.2 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 are 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, 2013] 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]. I also have some experience of ASM through the *Compiling Techniques* coursework [Franke, 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.

4.2.3 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.

4.3 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 4.1: 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 4.2: AOP-based advice equivalent to listing 4.1

Although that from a software engineering perspective this is clearly a superior solution (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**: like the other models that we have seen in section 1.5, widespread adoption of new programming construct 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.

4.3.1 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.

4.3.1.1 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 loose 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.

4.4 Hybrid Models

4.4.1 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.

- 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 overheads, 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.

4.4.1.1 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 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 TurboDiSl 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.

4.4.2 TEAMS

An alternative to DiSL is the TEAMS project [Rahnavard and Cook, 2013].

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

4.4.3 Bytecode Instrumentation & Graal Transformations

Chapter 5

The Runtime Library

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

Additionally, tools for generating sample dependency patterns are also included.

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`.

5.1 Trace Collection

One of the main functionalities of the runtime library is to enable the collection of traces. This entails logging all appropriate memory operations (i.e., the ones matching conditions presented in **??**), with some additional semantic information which is also required.

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

```java
public static <T> void
arrayLookup(T[] array, int index, int iterator, int id);

public static <T> void
arrayStore(T[] array, int index, T value, int iterator, int id);
```

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 iterator`: the value of the iteration variable
- `T value`: the value being written to the array at the index
- `int id`: a 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).

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 it 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.

5.2.1 Exact Approaches

Exact approaches provide an accurate deterministic response to the

5.2.2 Probabilistic Approaches

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.

5.2.3 Bloom Filters

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

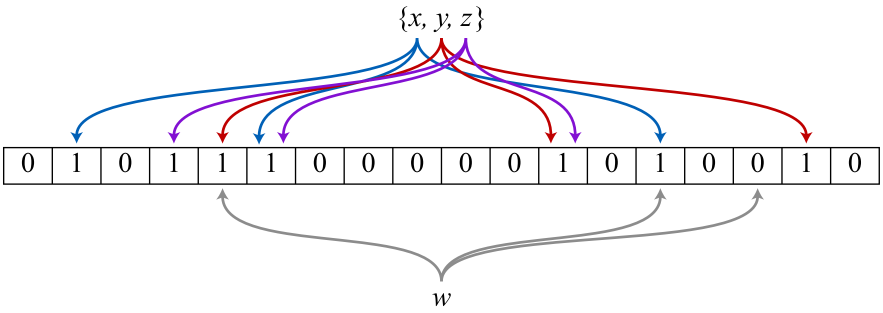

Figure 5.1: Bloom filter operation with $m = 18$ and $k = 3$

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:

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

The error rate is defined as:

$$\sigma = 0.5^k$$

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).

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

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

5.3 Dependency Analysis Algorithms

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

The problem statement for dependency analysis is as follows:

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}, $\mathtt{kind}(\alpha_i) \neq \mathtt{read} \wedge \mathtt{kind}(\alpha_{i-c}) \neq \mathtt{read}$.

There are several kinds of inter-iteration dependency [Ibbett, 2009; Stallings, 2013, p. 526]:

- **Write-after-write** or *output dependency*: given two instructions 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:

  ```
  R3 = R1 + R2
  R3 = R3 + R4
  ```

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

- **Write-after-read** or *antidependency*: given two instructions 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:

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

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

- **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:

  ```
  R1 = R2 + R3
  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 also a theoretical dependency that can be detected by the framework, read-after-read, but since this does not cause two iterations to be dependent, they are not considered by the framework.

5.3.1 Offline Algorithms

An offline algorithm means that any processing 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 $T_{offline} = O(lio)$.

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.

Algorithm 1 Offline 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
```

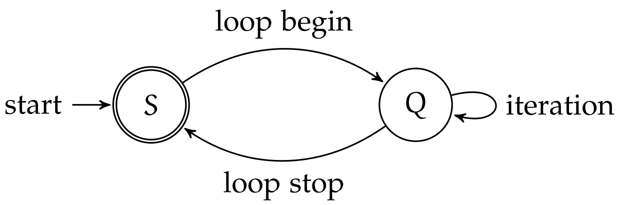

Figure 5.2: Finite state machine for the online algorithm

5.3.2 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 5.2 represents the finite state machine for the algorithm.

Algorithm 2 Online dependency algorithm

```
something
```

5.4 Implementation Details

Chapter 6

Methodology

6.1 Experimental Setup

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

- Intel Core 2 Duo P8800 @ 2.66GHz
- 8GB 1067MHz DDR3
- Nvidia GeForce 9600M GT 256MB
- Disks:
  - Samsung SSD 830 Series 128GB via SATA-2
  - Western Digital Scorpio Blue 1TB via SATA-2

The software configuration is as follows:

- OS X 10.8.4 (build 12E55)
- Java 7 update 25

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

6.2 Repeats

In order to improve the results, each experiment was repeat three times. The average of the three repeats was used in plotting charts etc.

6.3 Benchmarks

6.3.1 Validation and Basic Testing

The first 'item of business' is to prove that the implementation of the instrumentation is correct. We ensure this by comparing the results of algorithms with pre-determined results to the output of the instrumentation.

Since there are several kinds of dependency (see figure 5.3), these benchmarks need to take this into account.

6.3.2 Parametric Benchmarks

In addition to basic verification, a parametric benchmark has been created. This benchmark – called *FractionalDependent* – allows a user to specify

6.3.3 Graph Processing Algorithms

6.3.4 Java Grande

Java Grande [Smith et al., 2001; Bull et al., 2001] is a platform-independent benchmark for Java Virtual Machines and their associated compilers. Indeed, it is aimed at measuring the performance of the *virtual machine*, rather than the Java language.

The authors cite a *grande application* as one that "uses large amounts of processing, I/O, network bandwidth or memory". The benchmarks that are included in the Grande suite are:

- **euler** solves a set of equations using a fourth-order Runge-Kutta method
- **moldyn** compute molecular; it is a Java port of a program originally written in Fortran, for this reason it does not use object-oriented programming techniques.
- **montecarlo** is a financial simulation based on Monte-Carlo methods
- **raytracer** computes a scene containing 64 spheres
- **search** solves a game of Connect4

6.3.5 N-Body Simulation

N-Body simulations [Trenti and Hut, 2008] are computational simulations of real-world physical systems. They simulate a number (N) of particles (although in this context a particle does not need to be very small as in particle physics), acting under some forces (usually gravity).

The N-Body problem was chosen because it is known to be computable in parallel [Warren and Salmon, 1993; Nyland and Prins, 2007]. The program used for these benchmarks is available from Princeton University[1], although it has been slightly modified by fellow Master of Science student Ranjeet Singh. In the modified version, computation of forces has been vectorised, rather than by calling a method on the `Vector` class. The benchmark will focus on a this vectorisation.

6.4 Measurement Methodology

6.4.1 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.

6.4.2 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.

[1] `http://introcs.cs.princeton.edu/java/34nbody/Universe.java.html`

Chapter 7

Results

In this chapter, the results of using Locomotion on the benchmarks presented in section 6.3 are presented, along with a critical analysis of the results.

7.1 Basic Testing

The first testing that was

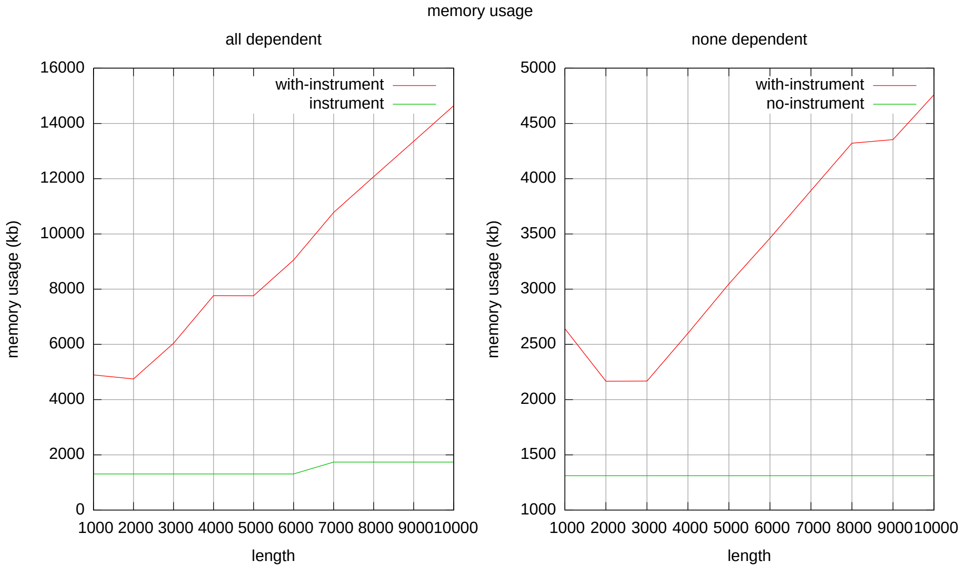

Figure 7.1: Memory usage

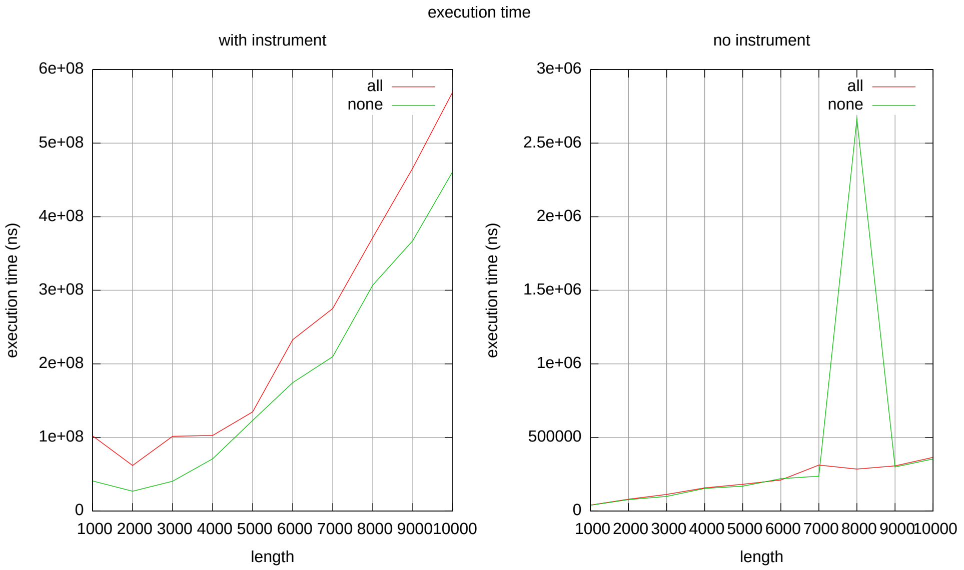

Figure 7.2: Execution time

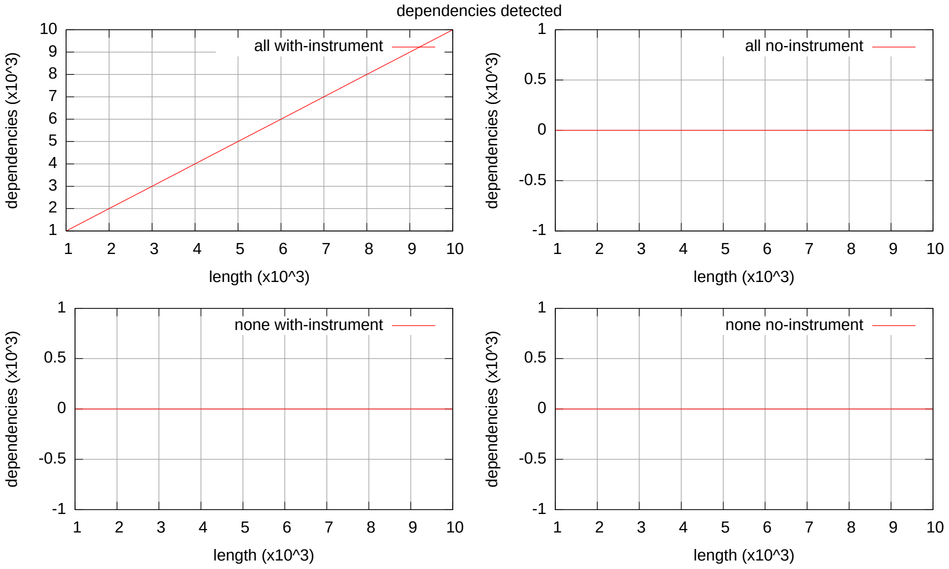

Figure 7.3: Dependencies

7.2 Parametric

7.3 Graph Processing

7.4 Java Grande

7.5 Mandelbrot

7.6 Overhead

7.6.1 Execution Time

7.6.2 Memory Usage

7.7 Analysis

Chapter 8

Conclusion

8.0.1 Concluding Remarks

8.0.2 Unsolved Problems

8.0.3 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.

The framework presented is only currently capable of detecting parallelism at the level of loops (i.e., loop-level parallelism).

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 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.

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]). In order to properly perform recompilation (or not), a cost model would need to be developed.

Bibliography

Keir Fraser. Technical Report. Technical Report 579, University of Cambridge, Cambridge, 2004. URL http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

M Herlihy, J Eliot, and B Moss. Transactional Memory: Architectural Support For Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, volume 21, pages 289–300. IEEE Comput. Soc. Press, 1993. ISBN 0-8186-3810-9. doi: 10.1109/ISCA.1993.698569. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=698569.

Murray Cole. Parallel Programming Languages and Systems, 2013. URL http://www.inf.ed.ac.uk/teaching/courses/ppls/pplsslides.pdf.

Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*, page 327, New York, New York, USA, October 2011. ACM Press. ISBN 9781450309776. doi: 10.1145/2043556.2043587. URL http://dl.acm.org/citation.cfm?id=2043556.2043587.

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, first edition, 2002. ISBN 0-13-031995-3.

Casey Marshall. Software Transactional Memory. Technical report, University of California, Santa Cruz, Santa Cruz, 2005. URL http://www.cs.uic.edu/~ajayk/STM.pdf.

Simon Peyton-Jones. The Future is Parallel, and the Future of Parallel is Declarative. In *YOW*, Melbourne, 2011. Presented at YOW 2011. URL http://yow.eventer.com/yow-2011-1004/the-future-is-parallel-and-the-future-of-parallel-is-declarative-by-simon-peyton-jones-1055

Todd A Anderson, Leaf Petersen, Hai Lui, and Neal Glew. Measuring the Haskell Gap. 2013. URL http://www.leafpetersen.com/leaf/publications/hs2013/haskell-gap.pdf.

Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton-Jones. Exploiting Vector Instructions with Generalized Stream Fusion. 2013. URL http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/haskell-beats-C.pdf.

Dominic A. Orchard, Max Bolingbroke, and Alan Mycroft. Ypnos. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming - DAMP '10*, page 15, New York, New York, USA, January 2010. ACM Press. ISBN 9781605588599. doi: 10.

1145/1708046.1708053. URL http://dl.acm.org/citation.cfm?id=1708046.1708053http://portal.acm.org/citation.cfm?doid=1708046.1708053.

M Grossman, A Simion Sbîrlea, Z Budimlić, and V Sarkar. CnC-CUDA: Declarative programming for GPUs. *Lecture Notes in Computer Science including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*, 6548 LNCS:230–245, 2011.

E Holk, WE Byrd, Nilesh Mahajan, and Jeremiah Willcock. Declarative Parallel Programming for GPUs. In Deschere Boss, Erik H. D'hollander, Gerhard R. Joubert, David Padua, Frans Peters, and Mark Sawyer, editors, *Parallel Computing*, volume 22, pages 297–304, Bloomington, 2011. IOS Press. URL http://osl.iu.edu/publications/prints/2011/2011-parco-holk-harlan.pdf.

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.

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.

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. *Car-*

cinogenesis, 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, 2013. 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.

Björn Franke. Compiling Techniques coursework, 2013. URL http://www.inf.ed.ac.uk/teaching/courses/ct/Coursework/Coursework_2013.pdf.

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.

Gholamali Rahnavard and Jonathan Cook. An Extensible AOP Framework for Runtime Monitoring. In *11th International Workshop on Dynamic Analysis*, 2013. URL `http://web.eecs.umich.edu/~nsatish/woda-2013/Rahnavard.pdf`.

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`.

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.

L A Smith, J M Bull, and J Obdrizalek. A Parallel Java Grande Benchmark Suite. *ACMIEEE SC 2001 Conference SC01*, pages 8–8, 2001. doi: 10.1109/SC.2001.10025.

J M Bull, L A Smith, L Pottage, and R Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande - JGI '01*, volume 15, pages 97–105, New York, New York, USA, 2001. ACM Press. ISBN 1581133596. doi: 10.1145/376656.376823. URL `http://portal.acm.org/citation.cfm?doid=376656.376823`.

M Trenti and P Hut. Gravitational N-body Simulations. *Scholarpedia*, 3:13, 2008.

Michael S Warren and John K Salmon. A parallel Hashed Oct-Tree N-Body Algorithm. In *Proceedings of Supercomputing*, 1993.

Lars Nyland and Jan Prins. Fast N-Body Simulation with CUDA. *Simulation*, 3:677–696, 2007.

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/`.

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.