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.

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

[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.

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

1.2 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.

1.3 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 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.

1.4 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.

As 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.

However, 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 the 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 3). 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 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 techniques 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 techniques is also presented.

1.6 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 3 describes the previous work both the areas of dynamic parallelism detection and parallelising compilers/runtime systems. Chapter 4 is an outline of the Graal compiler infrastucture, the main tool used in the project. Chapter 5 provides an overview of the possible approaches to instrumenting Java bytecode where appropriate. Chapter 6 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 7 describes the experimental design, configuration and other parameters. Chapter **??** presents the findings and a critical analysis of the work. The last chapter, chapter 9

draws final conclusions about the work, and suggests possible areas of future work in this area.