THE UNIVERSITY *of* EDINBURGH

Ensuring Performance and Correctness for Legacy Parallel Programs

Andrew J. McPherson

Doctor of Philosophy

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2015

Abstract

Modern computers are based on manycore architectures, with multiple processors on a single silicon chip. In this environment programmers are required to make use of parallelism to fully exploit the available cores. This can either be within a single chip, normally using shared-memory programming or at a larger scale on a cluster of chips, normally using message-passing.

Legacy programs written using either paradigm face issues when run on modern manycore architectures. In message-passing the problem is performance related, with clusters based on manycores introducing necessarily tiered topologies that unaware programs may not fully exploit. In shared-memory it is a correctness problem, with modern systems employing more relaxed memory consistency models, on which legacy programs were not designed to operate. Solutions to this correctness problem exist, but introduce a performance problem as they are necessarily conservative. This thesis focuses on addressing these problems, largely through compile-time analysis and transformation.

The first technique proposed is a method for statically determining the communication graph of an MPI program. This is then used to optimise process placement in a cluster of CMPs. Using the 64-process versions of the NAS parallel benchmarks, we see an average of 28% (7%) improvement in communication localisation over *by-rank* scheduling for 8-core (12-core) CMP-based clusters, representing the maximum possible improvement.

Secondly, we move into the shared-memory paradigm, identifying and proving necessary conditions for a read to be an acquire. This can be used to improve solutions in several application areas, two of which we then explore.

We apply our acquire signatures to the problem of fence placement for legacy well-synchronised programs. We find that applying our signatures, we can reduce the number of fences placed by an average of 62%, leading to a speedup of up to 2.64x over an existing practical technique.

Finally, we develop a dynamic synchronisation detection tool known as *SyncDetect*. This proof of concept tool leverages our acquire signatures to more accurately detect ad hoc synchronisations in running programs and provides the programmer with a report of their locations in the source code. The tool aims to assist programmers with the notoriously difficult problem of parallel debugging and in manually porting legacy programs to more modern (relaxed) memory consistency models.

Lay Summary of Thesis

To perform computations that would be infeasible on a single processor, programmers turned to parallelism, where multiple processors cooperate to perform larger computations. There is a therefore a large body of legacy programs written for parallel computers. However, on modern systems these programs may not achieve their full potential performance, or even operate correctly.

Technological advances have lead to the development of Chip Multiprocessors (CMPs) where multiple processors are placed on a single silicon chip. This change has lead to parallel computers that are constructed using different configurations to older machines. Additionally these new CMPs have different (more relaxed) rules about how each processor interacts with the other processors. This is largely due to performance reasons. These changes mean that legacy parallel programs will face performance and correctness issues when run on modern systems.

Our focus is on addressing these issues, largely through (semi-) automatic methods. Such methods are attractive as they reduce the required effort and knowledge of the programmer bringing the legacy program to the modern system. We implement most of our techniques in the compiler, a tool that transforms the high level program written by the programmer into low level code that the computer can execute.

In this thesis we propose novel techniques that analyse or transform programs to ensure correctness and performance on modern systems. Our results comprise analyses with improved coverage and techniques that achieve correctness with less performance degradation than existing techniques.

Acknowledgements

I am indebted to Dr. Vijay Nagarajan for his constant guidance while serving as my supervisor throughout this process. I would also like to thank my other supervisor Prof. Marcelo Cintra and also Dr. Susmit Sarkar, with whom I had the pleasure of collaborating. During my time in IF-1.05 (aka New Texas), I had the great fortune to enjoy the company of many fun and industrious people. Their friendship, advice, and patience proved crucial to my success.

George Stefanakis, Luís Fabrício Wanderley Góes, Karthik Thucanakkenpalayam Sundararajan, Vasileios Porpodas, Nikolas Ioannou, Lito Kriara, and Kiran Chandramohan understood the struggle and kept me on track throughout our time together. From my research group, Cheng-Chieh Huang, Bharghava Rajaram, Arpit Joshi, and Marco Elver offered innumerable fruitful discussions and were always willing to challenge my ideas. Their unique perspectives and insights significantly improved my understanding and the quality of my work. I also had the pleasure of sharing New Texas with a plethora of other great people during my studies, including Murali Emani, Stanislav Manilov, Ursula Challita, Praveen Tammana, and Rui Li amongst many.

While pursuing the PhD I had the opportunity to enjoy industrial internships with Intel Labs in Braunschweig, Germany and IBM Research in Haifa, Israel. These experiences were formative and greatly influenced my research and future career path. Therefore I would also like to thank Dr. Matthias Gries and Sergey Novikov with whom I collaborated at the respective institutions.

I must also thank my parents Cath and Mike. They have always been supportive of my efforts and I would not have reached this point without their commitment to my education from my earliest years.

Finally, I would like to thank Dr. Björn Franke (Edinburgh) and Dr. Alastair Donaldson (Imperial) for serving as my viva committee and offering such constructive feedback. I have left for pastures new, but despite all the challenges faced, I will always look fondly on the years I spent in New Texas. With hindsight I can say that it was worth it in the end.

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. Some of the material in this thesis has been published in the following papers:

- *Fence Placement for Legacy Data-Race-Free Programs via Synchronization Read Detection*. Andrew J. McPherson, Vijay Nagarajan, Susmit Sarkar, Marcelo Cintra. Principles and Practices of Parallel Programming (PPoPP'15), San Francisco, California, February 2015. (*Extended Abstract*)
- *Static Approximation of MPI Communication Graphs for Optimised Process Placement*. Andrew J. McPherson, Vijay Nagarajan, and Marcelo Cintra. Languages and Compilers for Parallel Computing (LCPC'14), Hillsboro, Oregon, September, 2014.

(*Andrew J. McPherson*)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Modern computers are based on manycore architectures to take advantage of the large number of transistors now available on each microchip and the limitations of uniprocessor performance, due to power and heat. In this environment, programmers are required to make use of parallelism to fully exploit the available cores. This can either be inside a single Chip Multiprocessor (CMP), or for larger scale programs requiring more cores than are available on a single CMP, multiple CMPs can be used in clusters of various topologies and design.

There are two major paradigms used to write parallel programs, shared-memory and message-passing. Shared-memory programming is traditionally used at a small scale, e.g. a single node, where executing elements (in this case threads) all have relatively efficient access to a common local memory (RAM) and share an address space. This common address space enables efficient communication as only pointers need to be passed between threads. Shared-memory programming can however be susceptible to error as threads can potentially interfere with one another if poorly programmed.

The other major paradigm is message-passing. This has been traditionally used at larger scales, e.g. multiple interconnected nodes. Here the executing elements (generally processes) can be completely independent and indeed may be running on entirely independent systems with private memory and storage. In this paradigm communication occurs through explicit calls to a message-passing library which facilitates the transfer of the specified data. The de facto standard for this paradigm is Message Passing Interface (MPI), which provides a wide range of communication and synchronisation primitives.

With the advent of the ubiquitous manycore CMP, even a handheld computer (e.g. a mobile phone) can have a single CMP with 8 or more cores. In this environment, both paradigms (with the support of the underlying hardware and system software) must now operate efficiently and correctly on manycore based systems. There are a number of opportunities created by this move to CMPs, that these paradigms can exploit. For example, message-passing systems can use threads rather than processes when a shared-memory environment (e.g. the communicating processes are in a single CMP) is available. This can enable more efficient communication. However manycore CMPs can also present performance challenges for legacy or topology agnostic programs that were not written with modern CMPs in mind.

Tiered Topology – To better convey the organisation of a modern parallel computer, we must look at the historical context. Historically, parallel computers were multiprocessors where discrete Central Processing Units (CPUs) would be connected together in various configurations. Now that CMPs are ubiquitous, these instead are used to construct parallel computers. Despite the large (and growing) number of cores per CMP there is always a desire to handle larger (and growing) computational problems, larger than a single CMP can support. Therefore, as with CPUs in multiprocessors before them, CMPs are used collectively, connected together in various configurations. Figure 1.1 shows example configurations of both an historical multiprocessing parallel computer and a modern manycore based parallel computer to better illustrate the differences. It is worth noting that even with a flat interconnect (as in our examples), the use of CMPs necessarily creates a tiered topology where not all cores are equidistant, with those on the same CMP taking advantage of internal interconnects. This inherent complexity means that legacy parallel programs written without knowledge of this topology can face performance degradation if they unnecessarily make use of higher latency or lower bandwidth channels within the system. This also presents an opportunity for tools that improve the spatial scheduling of existing programs by better mapping them to these new topologies.

Memory Consistency Model – One critical factor in the design of a shared-memory parallel architecture like a CMP is the memory consistency model. A memory consistency model precisely defines how accesses to memory (reads and writes) executed in different cores by different threads interact. Programmers generally write programs expecting Sequential Consistency (SC) [CTMT07, Hil98, LP01, SNM+12]. SC is where the operations of each core (thread) are exposed to other cores (threads) in the order that they exist in the source program (program order). For preserving the appearance

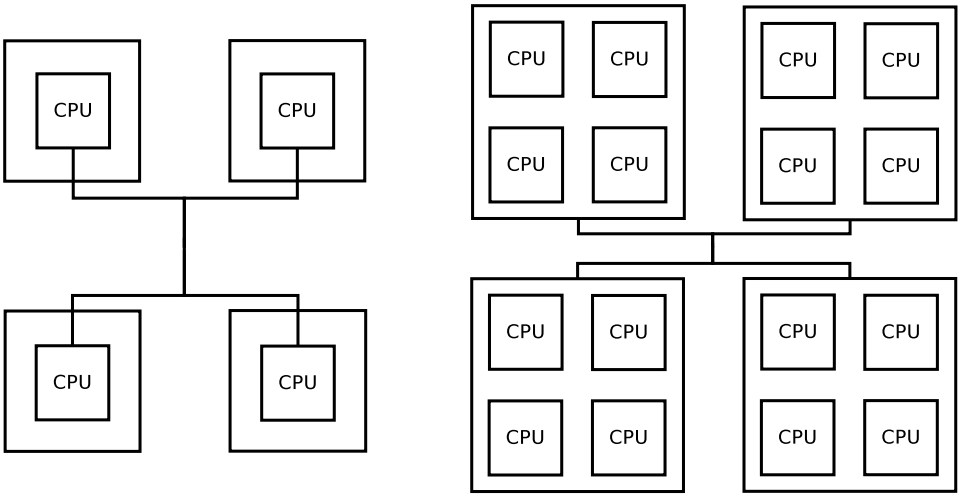

Figure 1.1: Examples of an historical multiprocessing computer (left) and a modern manycore based cluster (right). Note that the use of CMPs in the manycore system (right) necessarily creates a tiered topology, even with a flat interconnect.

of SC we need only be concerned about memory accesses, as other operations are not exposed.

However, while programmers have written assuming SC as it is convenient and simple to reason about, the trend in architectures is towards increasingly relaxed memory consistency models. Some early multiprocessors (e.g. MIPS) did support SC [Yea96] but more modern architectures do not [Int09, SSA+11]. These architectures have hardware memory models where some (or all) the orderings of reads ands writes are not enforced by the hardware. This relaxation is largely for performance reasons as more and more cores are added to each chip. This relaxation already exists to varying degrees in commercial architectures, with x86 chips not enforcing write to read orderings [Int09], and POWER being even more relaxed [SSA+11].

This mismatch between programs written assuming SC and architectures supporting more relaxed models requires specific intervention to prevent unintended behaviour. To better illustrate this unintended behaviour we introduce a simple synchronisation example as Figure 1.2. In this example both shared variables flag1 and flag2 are initialised as 0. Under SC this code ensures that either one or neither thread can enter the critical section. On a more relaxed modern architecture however, the ordering of reads to be after writes (to different addresses in program order) is not enforced. Therefore the at most one thread guarantee is lost and both threads may enter the critical section, potentially simultaneously. If the reads are executed before the writes, both threads will enter the critical section, thus violating mutual exclusion.

T1		T2	
flag2 = 1;	w	flag1 = 1;	w
	$\downarrow$		$\downarrow$
if (flag1 == 0)	r	if (flag2 == 0)	r
{		{	
// Critical Section		// Critical Section	
}		}	

Figure 1.2: Simple example of the importance of memory consistency models in shared-memory programming. Initially, flag1 and flag2 are set to 0. Under SC, this code will ensure that only one or neither thread will enter the critical section. Crucially, under SC the threads cannot both enter the critical section. If the $w \rightarrow r$ ordering is relaxed, then this guarantee is lost.

To correctly run a program like that in our example on a more relaxed architecture, fences must be introduced to prevent the compiler and hardware from reordering the memory accesses. This can either be done manually by an expert programmer or automatically by the compiler [LP01]. The issue with automatic intervention is that it is necessarily conservative and precision is difficult without a detailed understanding of the programmers intention. Conversely such automatic solutions are attractive in that they do not require an expert programmer and avoid the potential for such a programmer to introduce additional errors.

1.2 Problems

While both the paradigms outlined (message-passing and shared-memory) enable parallel computing, the differences inherent in their design are significant. Accordingly, while both have problems in communication and synchronisation for existing (legacy) programs on modern systems, the nature of the problems and the remedies available are paradigm specific.

In MPI, where all communication is performed through library calls, the correctness issue is one for library developers. However, a performance issue exists where programs written, unaware of the topology of the cluster of CMPs used, may have heavily communicating processes forced to use higher latency or lower bandwidth

channels within the cluster. This negative impact on communication can significantly increase the start-to-end runtime of the program.

In a shared-memory environment, the problem is ensuring correctness without needlessly sacrificing performance. The relaxed memory models used in modern architectures (as outlined above) create a correctness issue for (legacy) code that was written assuming SC. An ad hoc synchronisation, that is one written using a sequence of reads and writes, that assumes SC, will not have expected behaviour on a more relaxed architecture unless the required orderings are explicitly enforced. Explicitly enforcing every ordering would solve this correctness issue, but have a seriously detrimental performance impact and remove the benefits of hardware implementing a relaxed memory model. Therefore there is a need to determine the minimal number of orderings that need enforcement, to solve the correctness issue with minimal loss of performance.

1.3 Approach

To address the problems present for existing code in both paradigms on modern architectures, we present static (compile-time) solutions. Static solutions have significant advantages, as compared to dynamic or profile-guided approaches. In particular, profile-guided analysis requires additional work on the part of the programmer and the use of potentially scarce or unavailable resources. Additionally, where correctness is a concern, the results of even multiple executions may not reveal all potential behaviours. Dynamic (runtime) transformation approaches can also introduce additional overhead, which must be overcome before providing an improvement over the baseline.

Static analyses do have limitations, in regard to reliance on alias analysis and the unavailability of program input. Program input can be particularly useful in applications such as debugging. Therefore we also present a proof of concept dynamic approach to the shared-memory correctness issue outlined above. This dynamic approach aims to identify and report ad hoc synchronisations to the programmer, to assist in debugging and porting legacy code to a relaxed architecture.

1.4 Contributions

In this thesis we make the following contributions. Firstly, a method for statically approximating the MPI communication graph, then used to optimise process placement in a CMP-based cluster. We then move into the shared-memory paradigm, identify-

ing and proving necessary conditions for a read to be an acquire. An application of these conditions is then developed, optimising fence placement through an improved approximation of delay-set analysis. Finally we present a tool for dynamic detection and reporting of ad hoc synchronisation leveraging the necessary conditions earlier identified. Before moving on to a background discussion, we first outline the novelty of each of the techniques proposed in this thesis.

1.4.1 Static Approximation of MPI Communication Graphs

Inefficient communication can be a significant bottleneck in parallel programs. When an MPI program is run on a cluster of CMPs not all pairs of processes can communicate with equal bandwidth and latency. Programs written without knowledge of the topology of the cluster and the number of cores per CMP may not make efficient use of the system. Our aim is to colocate heavily communicating processes to the same CMP, though our work is applicable to other cost models or objective functions. Previous work has shown that MPI communication is generally statically determined by the programmer, implicit information that is currently ignored by the compiler. This static determination by the programmer allows compile-time analysis to effectively attempt to determine the communication graph and take action to intelligently place processes.

In Chapter 3 we present a purely static approach to determining the point-to-point communication graph of an MPI program. We propose a fully context and flow sensitive, interprocedural analysis framework for analysing MPI programs. This framework leverages a new data structure for maintaining partially evaluated variable representations for on-demand process sensitive evaluation. We use this framework to determine optimised process placement on a CMP-based cluster. Our analysis is the first to statically resolve and characterise the full point-to-point communication graph. In all but one case this only requires specifying of the number of processes.

1.4.2 Acquire Detection and Fence Placement for Legacy DRF Programs

In shared-memory parallel programming, being able to identify ad hoc synchronisations has a number of applications. These range from debugging (notoriously hard for parallel programs) to (as we will show) improving fence placement. In a data race free (DRF) program, synchronisation is annotated to allow the system to ensure no data races are introduced during compilation or execution. Recently programming mod-

els have been moving towards such DRF variants (e.g. C11 [BOS+11, BA08] and Java [MPA05]). However, there exists a large body of legacy code which has no annotations. These legacy programs would be DRF if only the annotations were added. As described earlier, ensuring correct operation of a program written assuming SC on a relaxed architecture requires explicitly enforcing orderings with fences. Where there are no annotations, the techniques used require program analysis. The seminal work in this area is delay-set analysis [SS88] which detects critical cycles between threads and the writes that would conflict. Our realisation is that programmers are not seeking to achieve SC, but data race freedom. For well-synchronised programs this only requires providing SC behaviour for synchronisation accesses.

In Chapter 4 we examine the nature of ad hoc synchronisation in a shared-memory environment. From this we determine the conditions a read must meet to be an acquire in a data race free (DRF) program. We then, for the first time, prove these are the necessary conditions. This is a significant contribution as it allows us to improve upon existing solutions in a number of application areas.

To demonstrate an application of this work we take the conditions determined and use them to improve delay-set analysis for well-synchronised (legacy DRF) programs. This improvement, is through using our signatures (conditions) to prune the number of required orderings determined by delay-set analysis (or its conservative approximation). This enables fence minimisation algorithms to place fewer fences, leading to improved performance.

1.4.3 Signature-based Dynamic Detection of Ad Hoc Synchronisation

Relatively precise identification of otherwise unmarked acquires and releases has two other important applications. The first is in the assistance of debugging of parallel programs, a notoriously difficult task. The second relates to the development of new language memory consistency models, e.g. C11, that require synchronisation to be explicitly marked. There exists a large body of legacy code that lacks such annotations. Tools that assist in identifying synchronisation can significantly aid the programmer in migrating programs to these new models.

In Chapter 5 we introduce SyncDetect, a proof of concept tool for dynamically detecting and reporting ad hoc synchronisations. Built on Intel's Pin framework, it is generally applicable and requires no program modification. It leverages the signatures

proven in Chapter 4 to identify reads that may be acquires and offers detection of (the more common) control acquires. We are also able to leverage the precision provided by dynamic analysis to additionally report the associated releases.

Chapter 2

Background

2.1 Architectures

2.1.1 Practical Need for Parallel Architectures

A significant class of computational problems (the majority of practical tasks) can to a greater or lesser extent be parallelised, that is they are not inherently sequential. Additionally, there is always a desire to attack larger and larger computational problems that are always infeasible for all practical purposes (in terms of running time), on a single CPU. These two factors create a climate in which using multiple CPUs in concert, to produce better absolute performance than a single CPU can offer, is a common use case.

Historically, one can define two major types of parallel computer. First is the cluster (multicomputer), constructed of multiple computers loosely connected over a network. Absent any additional abstraction, such a machine would use message-passing to operate in parallel. Second is the multiprocessor, where multiple CPUs are connected on a single bus using the same address space and therefore sharing memory.

2.1.2 Early Multiprocessors

Early multiprocessors were constructed by interconnecting discrete processors on a shared bus with a shared address space. Shared-memory programming is the natural paradigm here, although both major programming paradigms, the other being message-passing, are possible in this environment [LM92].

While many topologies and interconnect network designs are possible, ranging from a single shared bus to a full crossbar interconnect [BYA89], the distribution of

memory is arguably a more significant discriminant. These machines can be categorised into Uniform Memory Access (aka Symmetric Multiprocessing (SMP)), and Non Uniform Memory Access (NUMA) [HP11]. In an SMP machine the shared-memory is centrally located, with all processors having essentially equal access to it. In a NUMA machine, the shared-memory is partitioned. In most cases, each partition is co-located with one of the processors creating a notion of local and remote memory, which can be exploited for performance reasons.

Significant historical examples include the Stanford Dash [LLG+92], which was the first to have a scalable cache coherency protocol. This allows shared-memory programs to make full use of the caches, rather than face the time penalty of communicating with memory on every access to shared data.

2.1.3 Current Manycores

Current manycore systems are Chip Multiprocessors (CMPs), that is several interconnected CPUs co-located on a single silicon microchip. These manycore systems are now ubiquitous, present at all scales of general computing from the embedded (mobile phones) to supercomputers. This means that now even the most basic computer, outside of specialist embedded domains, supports parallelism. Current commercial architectures, such as Intel's Haswell processors [Int14] and those based on ARM's architectures [ARM14], support shared-memory programming. Whether it will be possible to maintain efficient cache coherent shared-memory as more and more cores are added to a CMP, is not yet known but seems unlikely. Therefore, future manycores may need to rely on Network on Chip (NoC) communication, where relatively distant cores on a CMP use message-passing to communicate. In such a future, shared-memory programming may be possible for regions of the CMP but not as a whole. One such example of a possible future architecture is Intel's Single-chip Cloud Computer (SCC) [HDH+10]. This can be operated as a cluster on chip with each core running its own operating system, with communication through on chip message-passing.

As discussed above, the demand for computational power has always outstripped the supply provided by a single CPU. As with previous multiprocessors and clusters, the way to achieve increased performance is to construct machines connecting multiple units of the basic component. In the modern world this means connecting CMPs. At a small scale this could be a shared-memory machine where two or more CMPs share memory and an address space. At a larger scale this is as a cluster with many CMPs

connected by a network and able to use message passing.

Before moving on to discuss programming models, it is worth noting that in recent years significant work has been done towards using Graphics Processing Units (GPUs) for general purpose computing. These architectures have proved effective at large scale computing [CBM+08], but are largely beyond the scope of this thesis and are therefore not discussed further.

2.2 Programming models

2.2.1 Message-passing

Message-passing is a parallel programming paradigm in which communication takes place through explicit means. The de facto standard in this paradigm is Message Passing Interface (MPI) [mF]. Message-passing is prevalent as it allows for the writing of portable code, where the program is not tied to specific features of the architecture. Indeed a message-passing program can be supported on both shared-memory and more distributed machines. To run, the program only requires that some implementation of the message-passing library is provided for the target system.

To understand the organisation of MPI programs, it is first necessary to introduce two central concepts, the *communicator* and the *rank*. The *communicator* represents a set of processes that can communicate with each other. An MPI program normally starts by using MPI_COMM_WORLD, a communicator that encapsulates all the processes. Programmers can then however create communicators for subsets of those processes. The communicator is important as it enables simple use of global communication operations, e.g. reductions and broadcasts. In MPI these operations are run within a specific communicator, with all processes in that communicator involved.

To identify individual processes for communication, particularly on a non-global basis, MPI assigns each process a *rank* for each communicator of which it is a part. This is analogous to a thread ID that a shared-memory programmer might use to determine behaviour and organise communication. However, in MPI, these assignments are made by the library providing the MPI implementation. A process requests its rank through a library call. For example *MPI_Comm_rank*(MPI_COMM_WORLD, *&my_rank);*, would store the processes rank for the MPI_COMM_WORLD communicator in the *my_rank* variable.

MPI programs are traditionally written in a Single Program Multiple Data (SPMD)

style, where each process executes the same program, with process specific behaviours engendered through control statements. This, combined with the fact that communication takes place through explicit library calls, is significant as it determines how the programs can be analysed. The explicit nature of the communication means that (in contrast to shared-memory programming), identification of communication statements is, absent function pointers, trivial. Understanding the nature of the communication however, with regard to which processes are involved and the volume of data transferred, is not necessarily so simple.

One significant advantage MPI (or another message-passing implementation) offers is the separation between the programmer intention and the implementation of communication and synchronisation. This allows the programmer to use MPI library calls to communicate and synchronise, without considering the underlying implementation. This also means library developers can develop the most efficient implementations of communication primitives for their target systems. The outcome of this is portable code that (in theory) can make use of the most efficient communication algorithms available on any of the target systems on which it is run.

The portability of MPI programs, while an extremely valuable feature, with a program able to (without alteration) be compiled and run correctly on any system for which an MPI library has been implemented, does also introduce some issues. In particular, because the organisation of the program is decoupled from the organisation of the system, a legacy or system agnostic program may not achieve its optimal performance. Assuming a relatively homogenous system, in terms of computing resources, the key aspect in terms of relative performance is how the communication graph of the program maps onto the system. We define the communication graph as one where processes are represented as vertices, with undirected weighted edges representing the total volume of communication (in bytes) between processes.

If the mapping of this graph onto the system is poor, as without deliberate intervention it may be, the start-to-end runtime can be negatively affected. In more detail, if heavily communicating processes are required to make use of higher latency or lower bandwidth channels of communication within the system, the start-to-end runtime of the program may not be the minimum possible on the system. This is caused by processes being forced to wait for the communication to occur (blocking communication). In fact, even in nonblocking communication waits may be introduced if the receiving process reaches an *MPI_Wait* call, where future statements are dependent on the receipt of data.

In this thesis we seek to address the spatial scheduling issue in message-passing programs created by the running of legacy or system agnostic message-passing programs on modern manycore clusters. We exploit the SPMD and explicit communication nature of MPI programs to statically determine the communication graph. This work is presented as Chapter 3.

2.2.2 Shared-memory

Shared-memory programming is a parallel programming paradigm that allows threads to access memory using a common address space. Given the requirement of a system providing such an address space, shared-memory programming is normally and historically used at a smaller scale, a single multiprocessor, in modern terms within a Single CMP. While shared-memory programming does require a system that supports the paradigm, on such a system it does provide more flexibility than message-passing. Specifically, the programmer is not limited to the primitives provided by the message-passing library (e.g. MPI) and is free to construct their own synchronisations. This flexibility can however be the source of errors, either through poor programming, or as we will discuss further, a mismatch between the programmed for memory consistency model and that provided by the system.

Significantly, communication in shared-memory programming can merely be the passing of pointers between threads, rather than (as in message-passing) transmitting the entire data required by the receiving process. There are other costs associated with cache coherence (or manual flushing if the caches are not coherent), but these are minor considering that the passing of a potentially unbounded volume of data is possible via a single pointer.

This ability to work on a single set of data not copied or fragmented is advantageous but requires correct synchronisation to avoid potentially silent errors. In shared-memory programming, correctness (with regard to program behaviour versus programmer expectation) depends on a correct understanding of the memory consistency model. Exclusively using library synchronisations does to some extent absolve the programmer of this responsibility but also removes much of the flexibility of shared-memory programming. In more detail, when using library synchronisations, the programmer can assume that the library developer has correctly met the requirements of the memory model, by placing appropriate fences. When the programmer implements their own synchronisations, they cannot rely on that assumption. As we will see in Sec-

tion 2.3, when the programmer makes use of ad hoc synchronisations, if the assumed memory consistency does not match that of the target system, errors may occur.

Recognising that shared-memory systems no longer support SC, development in languages that support shared-memory programming is trending towards data-race-free (DRF) based models. In these models, such as C11 [BOS+11, BA08] and the Java Memory Model [MPA05], the programmer is required to explicitly annotate synchronisation. This means that the programmer's expectations are explicitly defined and therefore the compiler is able to place the fences necessary to effectively strengthen the target system's memory consistency model for the annotated accesses. As these language models are relatively recent there exists a large body of legacy code that is well-synchronised and would meet these models, but lacks the annotations [XPZ+10]. We refer to such programs as Legacy DRF and target these in much of our shared-memory work as presented in this thesis.

In this thesis we seek to address the correctness issues introduced by running well-synchronised shared-memory programs written with the expectation of SC on more relaxed architectures, while minimising the performance issues introduced by current solutions.

To identify acquires, we identify and prove two signatures. At least one of these signatures must be fulfilled for a read to be an acquire. We then use these signatures to improve existing fence placement techniques by reducing the set of accesses that must be considered synchronisations. Finally we use these signatures to power a dynamic synchronisation detection tool, where they serve to minimise false positives.

2.3 Memory Consistency Models

2.3.1 Foundations

The memory consistency model is a crucial element in the design of a shared-memory multiprocessor. It defines how the operations of one CPU will appear to another CPU. This is critical, as performance reasons dictate that operations are not actually executed in the (program) order specified by the programmer. This is also true of a uniprocessor, where the programmers instructions are reordered by the compiler and the hardware as well. This is done to make the best use of the resources and minimise stalls, where elements of the processor are forced to wait. In a uniprocessor, as long as control and data dependencies are respected, this reordering is safe.

	$w \rightarrow w$	$w \rightarrow r$	$r \rightarrow w$	$r \rightarrow r$
Sequential Consistency	✓	✓	✓	✓
Total Store Order	✓	✘	✓	✓
Fully Relaxed	✘	✘	✘	✘

Figure 2.1: Overview of different memory consistency models, by the orderings of accesses to different memory locations that are enforced. Ticks indicate that an ordering is enforced by that model.

Where these reorderings, and (logically) simultaneous accesses by multiple CPUs to the same location, become problematic is in a shared-memory multiprocessor. Here, without a well-defined consistency model, adhered to by the programmer and the system, a read may return a stale value. In short, memory can be inconsistent.

Strict consistency, as provided by most uniprocessors, where any read to a location is guaranteed to return the result of the last write and a global order exists, is all but impossible to provide on a multiprocessor as memory accesses are not instantaneous. Therefore real world multiprocessors implement some weaker form of consistency, either the still relatively strong Sequential Consistency or a relaxed consistency model. We will now outline the details of common consistency models, beginning with Sequential Consistency. We will compare consistency models based on the memory orderings that they enforce. There are other details in specific implementations such as early access to data but considering the orderings enforced is sufficient for our purposes. Before going into the various models, we first present an overview of the orderings that each model enforces as Figure 2.1.

2.3.2 Sequential Consistency

Sequential Consistency (SC) as introduced by Lamport [Lam79] is the strictest form of memory consistency that is practical on a shared-memory multiprocessor. However, modern machines do not provide it as enforcing SC carries a severe performance penalty. Some earlier multiprocessors (e.g. the SGI Origin2000 [LL97] based on the MIPS R10000 [Yea96]) did provide SC, but these are no longer current.

SC is defined by the fact that the operations of a CPU must appear to other CPUs to have been executed in program order and that all executions are equivalent to all the operations having been performed in some linear sequence. It entails that all memory accesses (reads and writes) must appear to have been executed in program order, but

T1	T2
flag[0] = true;	flag[1] = true;
turn = 1;	turn = 0;
// Point A	// Point A
while (flag[1] && turn == 1){}	while (flag[0] && turn == 0){}
// Critical Section	// Critical Section
// Point B	// Point B
flag[0] = false;	flag[1] = false;

Figure 2.2: Peterson's Algorithm [Pet81]. This provides mutual exclusion under SC. Under more relaxed consistency models, fence(s) are required to prevent incorrect behaviour and the violation of mutual exclusion. In particular, under a model like TSO where $w \rightarrow r$ orderings are not enforced, a fence is required in each thread at Point A. Fences at Points B prevent accesses in the critical section from being executed outside the critical section.

operations that are local to a CPU and therefore not exposed to other CPUs may still be reordered.

While SC is no longer practical to provide, its intuitive model of operations appearing to happen in program order mean that it is still popular among programmers [CTMT07, Hil98, LP01, SNM+12]. There is also significant existing (legacy) code that expects SC and will behave incorrectly if executed on a system with a more relaxed memory consistency model. For example the classic Peterson's algorithm [Pet81] for mutual exclusion, presented as Figure 2.2, will not guarantee mutual exclusion unless SC is provided, as it relies on the program order of independent operations in a single thread being maintained.

If we examine Peterson's algorithm we see that the write to flag[0] in thread T1 has no dependency with the loop condition guarding the critical section. The same is true of the write to flag[1] in thread T2. Therefore, if the memory model is relaxed the writes to flags could occur at a later point in the execution. For example if a thread made the read in the while loop condition before writing to its own flag (in contravention of program order), both threads could see the while loop condition return false simultaneously. This could lead to a breach in mutual exclusion (both threads in the critical section simultaneously).

2.3.3 Total Store Order

Total Store Order (TSO) is a more relaxed model than SC. It relaxes the ordering of reads after writes ($w \rightarrow r$). This means that reads to a location A that are ordered (program order) after a write to location B may be reordered and executed before the write to location B. Accesses to the same location will not be reordered due to the dependency that would exist. In this consistency model all other orderings ($w \rightarrow w$, $r \rightarrow w$, and $r \rightarrow r$) are still respected and enforced. TSO is the same memory model in terms of orderings as that provided by Intel's x86 processors [Int09] and is therefore a significant model to consider, given the widespread adoption of this architecture in desktop and HPC environments.

With regard to Peterson's algorithm, as presented in Figure 2.2, the relaxation of the $w \rightarrow r$ memory ordering means that a memory fence is required at Point A. This prevents violation of mutual exclusion that could occur if the writes and reads to the flag variables were reordered. Fences will also be used at Point B to ensure that accesses in the critical section are not reordered outside the critical section.

On Intel machines, the fence (mfence) used to enforce $w \rightarrow r$ memory orderings has a significantly negative impact on performance [AKNP14, DMT13]. This means that when SC semantics are required on an x86 machine, minimising the number of fences placed (and executed) is crucial to ensure that as little performance is lost as possible.

2.3.4 Fully Relaxed

A fully relaxed memory model is an even more relaxed model, where none of the four orderings are enforced. The additional advantage of relaxing the $r \rightarrow w$ and $r \rightarrow r$ orderings is that it allows read latency to be hidden [GGH92]. Relaxed models are currently in commercial use, for example a relaxed model is currently implemented by the POWER processors from IBM [SSA$^+$11] amongst others.

The simplest fully relaxed model is known as Weak Ordering (WO) [AH90a]. Here memory operations must be regarded as data or synchronisation. If one of the operations in a potential ordering is a synchronisation operation then it will not be reordered. In this model efficiency demands minimising the set of memory operations considered synchronisation, while correctness demands not mislabelling synchronisation accesses as data accesses.

A more complex classification is introduced by the Release Consistency (RC)

Processor	2x Intel Xeon E5645 (2.40 GHz, 6 cores)
Memory	24GB RAM
Interconnect	Gigabit Ethernet

Table 2.1: The component configuration of a single node on the Eddie cluster.

model [GLL+90], which additionally provides *nsync* for asynchronous operations and separates synchronisation operations into *acquire* and *release*. Here, a programmer must use special acquire (read) and release (write) operations when a stricter consistency model is required.

2.4 Architecture Examples

To place the programming models and memory consistency models in a better context, we will now present short outlines of a modern High Performance Computing (HPC) architecture and a modern workstation architecture. We will focus on real world examples, to illustrate that the problems described and addressed in this thesis are present in current systems and not merely intellectual curiosities.

2.4.1 HPC Architecture

Modern HPC architectures vary dramatically, with some rather exotic configurations used for specific problem domains. For our purposes it is sufficient to consider a more orthodox design. We consider an Ethernet linked cluster, specifically "Eddie", the cluster run by the Edinburgh Compute and Data Facilities at the University of Edinburgh [ECD]. At the time of writing the cluster contains 156 nodes connected by Gigabit Ethernet. Each node is a IBM dx360M3 iDataPlex server, with two 6 core processors on each server. A more detailed component configuration for each node is presented as Table 2.1.

In this cluster of CMPs environment, each node of 12 cores (2 CMPs) shares access to 24GB of node local memory. Programs running on a single node can therefore take advantage of shared-memory programming. However, when using multiple nodes, for larger scale computation, message-passing is used as the node local memory is not immediately accessible to processes on other nodes. Without considering over-scheduling of the system, 1,872 MPI processes could cooperatively participate in a

Processor	Intel Xeon E5-2620 v3 (2.40GHz, 6 cores)
Memory	16GB RAM
Disk	1TB 7200rpm SATA

Table 2.2: A potential component configuration details of an HP Z840 Workstation.

single computation. In this system, the work presented in this thesis with regard to spatial scheduling of MPI programs is most relevant.

We should also note however, that with each node being a shared-memory domain our work in the shared-memory paradigm is also applicable for programs that use shared-memory programming on a single node. This could either be independently or as part of a larger mixed mode parallel program. Here shared-memory programming is used to run one MPI process per node (with message-passing communication between nodes) with each process launching multiple threads and communicating using shared-memory within the node.

2.4.2 Workstation Architecture

As we are most interested in performance, we will consider a high performance workstation. Specifically, we examine the Z840 workstation from Hewlett-Packard [Hew15]. One potential configuration of this workstation is presented as Table 2.2.

We see that the workstation makes use of a single CMP, in this case with 6 cores (CPUs). In fact other configurations of this workstation allow up to 18 cores in a single CMP (using an Intel E5-2699 v3) [Hew15]. This workstation provides a single shared-memory domain, with Intel's implementation of the TSO memory consistency model. Parallel programs written for this workstation can therefore make use of the shared-memory programming paradigm, though message-passing through MPI is also supported. With regard to this workstation, the work presented in this thesis that is most relevant is that focused on improving the performance of legacy shared-memory programs on non-SC architectures, while still maintaining correctness.

Chapter 3

Static Approximation of MPI Communication Graphs

3.1 Introduction

In this chapter we look at the message-passing paradigm and seek to address the problems faced by legacy message-passing programs when executed on modern CMP-based clusters. To do this we focus on the static analysis of point-to-point communication to better determine process placement within a CMP-based cluster.

Message Passing Interface (MPI) is the de facto standard for programming large scale parallel programs. Paradigm-aware static analysis can inform optimisations including process placement and communication/computation overlap [DPS07, DPSC09], and debugging [XLW+09]. Fortunately, message-passing lends itself effectively to static analysis, due to the explicit nature of the communication. This is in contrast to shared-memory or shared address space programming models, where communication can be difficult to detect.

Previous work in MPI static analysis produced several techniques for characterising communication [Bro09, SPS99, SKH06]. Common to these techniques is the matching of send and receive statements, which while potentially enabling interprocess dataflow analyses, can limit their ability to discover all communications. More importantly, the techniques are limited in their context sensitivity, from being limited to a single procedure [Bro09, SPS99], to only offering partial context sensitivity [SKH06]. Therefore, the existing techniques do not provide viable tools applicable to determining the full communication graph.

In comparison to static approaches, profiling can be effective [CCH+06], but is

more intrusive to workflow. As Zhai et al. [ZSH+09] note, existing tools such as KOJAK [MW03], VAMPIR [NAW+96], and TAU [SSM06] involve expensive trace collection, though lightweight alternatives e.g. mpiP [VM01] do exist. While our static analysis is able to operate on a single workstation, profiling a large program can require access to the target machine, a potentially scarce and expensive resource. A profiling approach therefore compares unfavourably to a static approach that achieves similar results, given the cost and inconvenience of repeated executions on the target machine. *The main question we address in this chapter is whether a static analysis can provide comparable insight into the MPI communication graph, without requiring the program to be executed.*

Tools for understanding MPI communication have several applications. For example, one can consider the running of an MPI program on a cluster of Chip Multiprocessors (CMP). Here, there exists a spatial scheduling problem in the assignment of processes to processor cores. In MPI, each process is assigned a *rank*, used to determine its behaviour and spatial scheduling. For example, OpenMPI [GFB+04] supports two schedules, **by-rank** – where processes fill every CMP slot before moving onto the next CMP, and **round-robin** – where a process is allocated on each CMP in a round-robin fashion. Without intervention, there is no guarantee that the communication is conducive to either schedule. This may lead to pairs of heavily communicating processes scheduled on different nodes. Communication between nodes, using Ethernet or even Infiniband, can be subject to latencies significantly larger than in intra-node communication. This inefficient scheduling can cause significant performance degradation [ASLK06, MJ11, ZZCZ09]. Prior analysis allows intelligent placement to alleviate this issue.

If one assumes a deterministic MPI program (at least in terms of communication), we can define the communication graph function for a program. The communication graph function is a function that maps the program input to an undirected graph with weighted edges, where vertices represent MPI processes. Here edge weights are the total number of bytes communicated between each pair of processes during the execution of the program. Our analysis does generate directed information, but this is not required for our purposes. To better illustrate this we present Figure 3.1, which shows an example output from such a function for an MPI program with 12 processes (A).

From this definition of the communication graph function, it is clear that for different inputs, different graphs can be produced. However, we find static analysis can still be effective due to two observations. Firstly, for a significant class of MPI programs,

the communication pattern is found to be broadly input independent and therefore amenable to static analysis [Bro09, CGS10, FY02, PSK+08]. Secondly, as we discover through our experiments, the edge weightings are often directly parameterised by the input size. Therefore the edge weightings are fixed in relative terms. Both these observations stem from the practical circumstances where the programmer statically designs the work distribution and communication that the algorithm will perform, without precise knowledge of the input.

In cases where the first observation proves false, and the communication graph is highly dependent on the input, static analysis can still prove valuable. From the analysis we develop, it is trivial to determine whether or not the communication graph has input dependencies, something not possible from a dynamic analysis without repeated experiments.

Returning to Figure 3.1, we see that triples of the processes communicate heavily and that a default schedule such as round-robin (B) would lead to all communication being inter-CMP in this 4 core per node system. The other default schedule by-rank (C) has 2 triples communicating between CMPs. In contrast, the intelligent schedule (D) is able to localise all but 1 of the heavily communicating triples. Therefore in this example the intelligent schedule is shown to be the best solution and an improvement to communication localisation over either default.

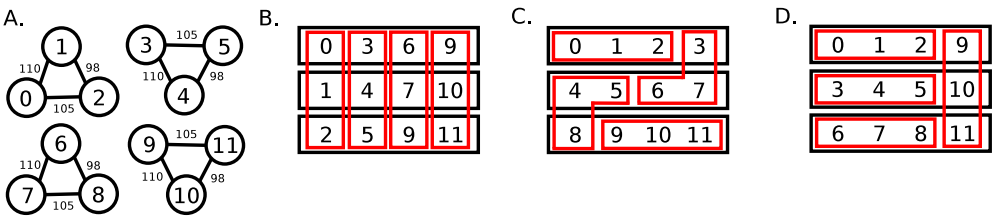

Figure 3.1: A simplified communication graph for a 12 process program (A), where triples of processes communicate heavily (see edge weights). Also shown are three possible spatial schedules; Round Robin (B), by-rank (C), and intelligent placement (D). Note that Round Robin scheduling leads to all significant communication taking place between CMPs, with intelligent placement localising communication from all but one of the triples on a 4 core per node system. Additionally the other default schedule, by-rank similarly splits 2 of the triples across multiple nodes. Assuming a cost model of intra-CMP communication being cheaper than inter-CMP communication, Intelligent Placement is the best solution.

In this chapter, we propose a fully context and flow sensitive, interprocedural analysis framework for the static analysis of MPI programs. Our framework is essentially a forward traversal examining variable definitions; but to avoid per-process evaluation, we propose a data-structure to maintain context and flow sensitive partially evaluated definitions. This allows process sensitive, on-demand evaluation at required points. Our analysis is best-effort, prioritising discovering communications over soundness; for instance we assume global variables are only modified by compile-time visible functions.

We instantiate our framework to determine an approximation of the point-to-point communication graph of an MPI program. Applying this to programs from the NAS Parallel Benchmark Suite [BBB+91], we are able to resolve and understand 100% of the relevant MPI call sites, i.e. we are able to determine the sending processes, destinations, and volumes for all contexts in which the calls are found. In all but one case, this only requires specifying the number of processes.

To demonstrate an application of our analysis, the graph is used to optimise spatial scheduling. An approximation is permissible here, as spatial scheduling does not impact correctness in MPI programs. We use the extracted graph and a partitioning algorithm to determine process placement on a CMP-based cluster. Using the 64 process versions of the benchmarks, we see an average of 28% (7%) improvement in communication localisation over *by-rank* scheduling for 8-core (12-core) CMP-based clusters, representing the maximum possible improvement.

The main contributions of this technique are:

- A novel framework for the interprocedural, fully context and flow sensitive, best-effort analysis of MPI programs.
- A new data structure for maintaining partially evaluated, context and flow sensitive variable representations for on-demand process sensitive evaluation.
- An instantiation of the framework, determining optimised process placement for MPI programs running on CMP-based clusters.

3.2 Our Approach

In this section we explain the key elements of our approach in terms of design decisions, data structures, and present an overall analysis algorithm. To motivate our approach we examine a sample MPI program, presented as Listing 3.1.

3.2.1 General Principles

The basic aim of a static approach to approximating the point-to-point communication graph is to understand *MPI_Send* calls (as in line 22 of our example in Listing 3.1), or similar, e.g. *MPI_Isend*. There are four elements to this, the **source** - which processes make the call, the **destination** - to which processes do they send data, the **send count** and the **datatype** - from which the volume of bytes transmitted can be calculated.

```
1  #include <mpi.h>
2  int my_rank, comm_size, indata, outdata;
3  MPI_Status stat;
4
5  int main (int argc, char **argv) {
6    MPI_Init (&argc, &argv);
7    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
8    MPI_Comm_size (MPI_COMM_WORLD, &comm_size);
9    indata = comm_size + 4;
10   if (my_rank < 5)
11     communicate ();
12   if (my_rank < 6)
13     indata = indata + my_rank;
14   if (my_rank > 7)
15     communicate ();
16   MPI_Finalize ();
17   return 0;
18 }
19
20 void communicate () {
21   if (my_rank % 2 == 0 && my_rank < comm_size - 1)
22     MPI_Send (&indata, 1, MPI_INT, my_rank + 1, 0,
23         MPI_COMM_WORLD);
24   else
25     MPI_Recv (&outdata, 1, MPI_INT, MPI_ANY_SOURCE,
26         0, MPI_COMM_WORLD, &stat);
27   indata = 0;
28 }
```

Listing 3.1: Example of a simple MPI program

As we can see from line 10, the call to *communicate*, which contains the *MPI_Send* can be conditional. On this basis we can say that an interprocedural approach is essential, as an intraprocedural approach fails to capture the fact that any process with a *rank* greater than 4 would not make the first call to *communicate* and therefore not reach the *MPI_Send* in this instance.

Accepting the need for full context sensitivity, there are two basic approaches that could be employed. One could use some form of interprocedural constant propagation [GT93], within a full interprocedural dataflow analysis [HMCCR93], to determine the relevant parameter values (*destination*, *send count* and *datatype*). However, such an approach is not without issue. Significantly, the SPMD nature of MPI programs means the path through the program may be process sensitive (as seen in our example). Therefore, a constant propagation approach would require complete evaluation of the program for each intended process to determine the processes communicating (*source*) at each call site. Also, even with flow sensitivity [CH95], such a rigorous approach may not be enough to provide an approximation of the communication graph due to its strictness.

The alternative basic approach is a static slicing, based on a partial data flow analysis [GS94], that identifies the *MPI_Send* and then evaluates at the program point before the call, for each of the contexts in which the call is found. While such a technique is possible and requires potentially less computation than the previous approach [DGS95], it suffers from the same weaknesses, with regard to strictness and full reevaluation to determine the *source*.

Due to these issues, we choose to follow a composite approach based largely on a forward traversal to establish interprocedural context without backtracking. This traversal walks through the Control Flow Graph (CFG) of a function, descending into a child function when discovered. This is analogous to an ad-hoc forward traversal of the Super CFG [ALSU06], but with cloned procedures. This is an expensive analysis, but allows us to achieve context and flow sensitivity and exploit the SPMD nature of MPI programs. To avoid full reevaluation, we do not treat process sensitive values as constants and instead leave them partially evaluated in a data structure introduced in Section 3.2.3. Therefore, we progress in a process insensitive manner, only performing process sensitive evaluation for function calls and MPI statements, using our data structure to perform on-demand slicing. To allow characterisation of the maximum number of communications, we make the approach best-effort, applying the assumption that global variables are only modified by functions visible to the compiler. While

this renders our evaluations strictly unsound, this is required to characterise even the minimal amount of communications.

3.2.2 Context, Flow, and Process Sensitivity

Focusing on the *MPI_Send* in our example, we see that establishing definitions with our approach requires understanding two elements; which processes enter the parent *communicate* function (context sensitivity) and of those processes, which reach the call (flow sensitivity). Due to the SPMD semantics, process sensitivity (which processes reach a certain program point), is derived from the context and flow sensitivities. These are handled using two related techniques.

To understand which processes call the parent function and therefore potentially make the *MPI_Send*, we introduce the *live vector*, a boolean vector to track which processes are live in each function as we perform the serial walk. The length of the vector is the number of processes for which we are compiling, initialised at the main function as all true. Requiring the number of processes to be defined entails compiling for a specific scale of problem. However we do not believe this is a significant imposition, given the typical workflow of scientific and high performance computing. Notably, this requirement also applies to profiling, where a new run is needed for each change in the number of processes.

The live vector is a simplification of the context of the call for each process. This allows for, at a subsequent assignment or call, evaluation using the live vector and flow information, rather than repeated reevaluations within the context of the entire program. When a call is found, we generate a live vector for that function before descending into it. This *child live vector* is generated from the live vector of the parent function of the call and is logically a subset of those processes that executed the parent function. The evaluation of which processes are live in the child live vector uses the flow sensitivity technique, described next.

Within a function, which processes make a call depends on the relevant conditions. We examine the CFG in a Static Single Assignment form where the only back edges are loop backs, all other edges make forward progress. A relevant condition is defined as one meeting three requirements. Firstly, the basic block containing the condition is not post-dominated by the block containing the call. Secondly, there are no blocks between the condition block and the call block that post-dominate the condition block. Thirdly, there exists a path of forward edges between the condition block and the call

block.

The evaluation of relevant conditions is done with regard to their position in the CFG and the paths that exist between them. This ensures that calls subject to interdependent conditions, as seen in line 21 of our example, can be evaluated correctly. The definitions for the condition and its outcome can be process sensitive, so the evaluation of the relevant conditions must be performed separately for each process. The method by which this and the evaluation of MPI arguments is achieved is introduced in the next section.

3.2.3 On-demand Evaluation

To evaluate the conditions and the arguments of the *MPI_Send* as detailed above, we implement a tree-based representation to hold the partially evaluated variables as our approach requires. Our representation provides the ability to perform on-demand static slicing, sensitive to a particular process, without repeated analysis of the program. In fact, since only a fraction of the variables influence the communication graph, most will not need evaluation.

To allow efficient access to the representations of each variable we maintain global and local hash tables of pointers to the most recently defined node in the representation. This split between global variables and local variables allows us to perform what is essentially garbage collection to remove representations that are no longer required.

For each assignment or ϕ-node encountered, a new node of our representation is created, or if a definition for the variable already exists, its node is modified. These nodes are stored in either the global or the local hash tables allowing efficient lookup and discarding of out of scope definitions that are unreferenced by any in scope.

Each node is of one of eight types, representing all the cases that arise. These are detailed in Table 3.1. The node type used is defined by the node types of the operands of the defining statement and whether a definition already exists. ϕ-nodes are treated as multiple definitions to a variable, resulting in a **many** node.

To better convey the operation of this data structure we present Figure 3.2, which shows the state of *indata* by the end of the program described in Listing 3.1 (line 16). By the end of the program, *indata* has been defined multiple times, but not all definitions apply to all processes. For this example, we assume the program has been compiled for 12 processes.

The first definition (line 10), is to add *comm_size* to the constant 4. While *comm_size*

Type	Description
Array	Handles array definitions, see Section 3.2.4.
Builtin	Required for built in functions (e.g. square root), contains an operator and pointer to the node upon which it is to be applied.
Constant	Represents a constant.
Expression	Represents an arithmetic expression and contains an operator and pointers to nodes upon which to apply it.
Iterator	Identical to Constant, but specially controlled for loop operations.
Many	Handles repeated definitions to the same variable, allowing context, flow, and process sensitive resolution.
SPMD	Definitions generated by operations with process sensitive results, e.g. a call to *MPI_Comm_rank*.
Unknown	Unresolvable definitions.

Table 3.1: Descriptions of node types used in our representations of partially evaluated variables. Each node in the representation is exactly one of these types.

is an SPMD value, because it is the same for all processes this expression can be reduced to a **constant** (marked (0) in Figure 3.2). Then after descending into *communicate* for the first time, *indata* is redefined in line 27. Since *indata* has already been defined, as well as creating a new **constant** definition (marked (1)), a **many** (marked (2)), copying the live vector of the new definition is also created, as the new definition does not apply to all processes. Definition (2) is now the current definition stored in the hashtable. Were *indata* to be evaluated at this point, processes with a rank of less than 5 would take the right branch (to the newer definition) and evaluate *indata* as 0, whereas all others would use the previous definition.

Upon returning to the parent function, *indata* is redefined again (line 13). This time as its previous definition plus the rank of the process. Since the components are not both of type **constant**, an **expression** is created (marked (4)). This **expression** will combine the evaluation of the child **many** (marked (2)) with the rank for *MPI_COMM_WORLD* for the particular process (an **SPMD** marked (3)). Again because this variable has been defined before, a **many** (marked (5)) is created, linking the old and new definitions. Note that we do not need to copy the old definition, merely including it in the new definition with appropriate pointers is sufficient. Note also that this new definition is subject to a condition, the details of which are also

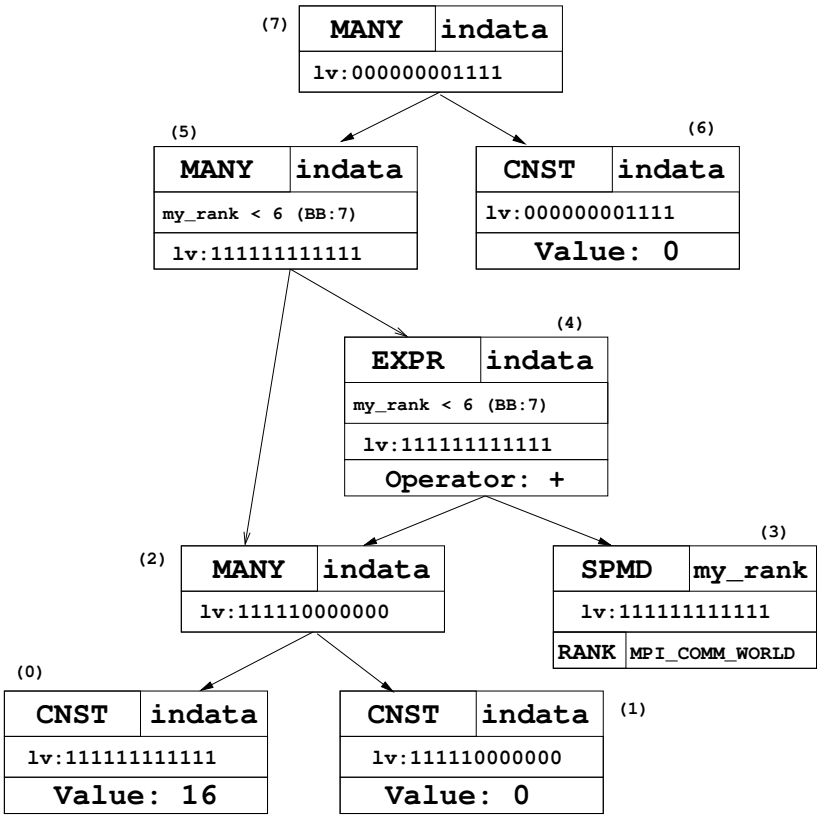

Figure 3.2: The representation of *indata* at line 16 in Listing 3.1. In this figure **lv** represents live vector. We can see that after being redefined several times multiple nodes have been created and organised such that *indata* resolves to different values depending on the rank of the process.

associated with both the **expression** and the **many**. The association of conditional information allows for differentiation between multiple definitions where the live vector is the same, i.e., the difference is intraprocedural. Finally, the program descends again into *communicate*, creating another definition (marked (6)) and **many** (marked (7)).

3.2.4 Special Cases

There are a few special cases that merit further explanation:

Arrays - Viewing elements as individual variables, there is a complication where the index of an array lookup or definition is process sensitive. Operating on the assumption that only a small fraction of elements will actually be required, efficiency demands avoiding process sensitive evaluation unless necessary. Therefore, an array is given a single entry in the hash table (type **array**), that maintains a storage order vector of definitions to that array. A lookup with an index that is process sensitive returns an

array with a pointer to this vector, its length at the time of lookup, and the unevaluated index. Evaluating an element then requires evaluating the index and progressing back through the vector from the length at time of lookup, comparing (and potentially evaluating) indices until a match is found. If the matched node doesn't evaluate for this process, then the process continues. This ensures that the latest usable definition is found first and elides the issue of definitions applying to different elements for different processes.

Loops - Again we take a best-effort approach, assuming that every loop executes at least once, unless previous forward jumps prove this assumption false. At the end of analysing a basic block, the successor edges are checked and if one is a back edge (i.e. the block is a loop latch or unconditional loop), then the relevant conditions are resolved without respect to a specific process. This determines whether the conditions have been met or whether we should loop. This means that when an iterator cannot be resolved as the same for all processes, the contents of the loop will have been seen to execute once, with further iterations left unknown. These loops are marked so that calls inside them are known to be subject to some unknown multiplier. To handle more complex loops with additional exits, the exits are marked during an initial scan and evaluated as they are reached.

The choice to only resolve loops with a process insensitive number of iterations does potentially limit the power of the analysis. However, it is in keeping with our decision to analyse serially. Parallelising for the analysis of basic blocks and functions inside a loop would complicate the analysis to the point where it would be equivalent to analysing the program for each process individually. As we see in Section 3.4, this decision does not have a negative impact on our results with the programs tested.

Parameters - Both pass-by-value and pass-by-reference parameters are handled. In the case of pass-by-value, a copy of the relevant definition is created to prevent modifications affecting the existing definition. Lookups and evaluations for parameter definitions are handled in the same manner as those for global or function local definitions.

3.2.5 Overall Algorithm

Combining the elements described, we produce an algorithm for the analysis of MPI programs, presented as Listings 3.2, 3.3, 3.4 and 3.5. To examine the application of the algorithm to an MPI program we also present a simple MPI program as Listing 3.6.

```
1  global_defs = hash_map<var, node>();
2
3  walker (function, live_vector, param_defs)
4  {
5    local_defs = hash_map<var, node>();
6    for basic_block in function
7      for statement in basic_block
8        if is_assignment (statement)
9          record_assignment (statement, live_vector, local_defs)
10       else if is_call (statement)
11          child_live_vector = eval_conds (statement,
12            live_vector, local_defs)
13         if is_mpi (statement)
14           eval_mpi_call (statement, child_live_vector,
15             local_defs)
16         else if has_visible_body (statement)
17           child_param_defs =
18             generate_param_defs (statement, local_defs)
19           walker (statement, child_live_vector,
20             child_param_defs)
21     if is_loop_back_or_exit (basic_block)
22       // Adjust basic block if loop back or exit
23       basic_block = check_loop_conditions (basic_block,
24         local_definitions)
25 }
```

Listing 3.2: Algorithm for process and context sensitive traversal

We now apply our analysis to the example program (Listing 3.6) to demonstrate the process. For the purposes of the example, we consider the program to be compiled for 4 processes. Initially, *walker* is called on the *main* function of the example, with a live vector of all true (e.g. 1111).

Analysing the first statement (a call to MPI_Init), we see it is a call, with no conditions and therefore all processes live in this function (all of them) would execute this function. As it is an MPI function, we call *eval_mpi_call*. As it is not one of the functions where we take action, nothing is done. We could use analysis of MPI_Init to perform a debugging action (i.e. checking that no communication occurs before this

```
1  record_assignment (statement, live_vector, local_defs)
2  {
3    lhs = get_lhs (statement)
4    rhs = get_rhs_terms (statement)
5    new_def = build_node (rhs)
6    old_def  = get_def (local_defs, global_defs)
7    if old_def is not null
8      new_def = build_many (new_def, old_def, live_vector)
9    if is_global(lhs)
10     global_defs[lhs] = new_def
11   else
12     local_defs[lhs] = new_def
13 }
```

Listing 3.3: Algorithm for storing a definition

call for each function, but in our work we only considered correct MPI programs.

Next, we reach the call to MPI_Comm_rank, where again there are no conditions and all processes are live in this function, so *eval_mpi_call* records a node to the global_defs hash table, mapping the variable my_rank to an SPMD node that when evaluated returns the processes rank. The same process is then followed for MPI_Comm_size, however here a node mapping comm_size to a constant node with value 4 is stored in global_defs.

For the assignment to indata, the analysis uses *record_assignment* (Listing 3.3). The two right hand side terms are retrieved and since both are constant (comm_size is found in the global_defs and 4 is immediate), no expression node is needed as a new constant node (value 8) can be created and stored in global_defs.

We enter the loop, by storing a node for i with value 0 in the local_defs. As we pass through the condition statements no action is taken, as we only return to evaluate these if necessary. We reach the call to communicate and after checking the pre-generated basic block information, detect that the two conditions are relevant to the call. In *eval_conds*, we copy the live vector and then for each live process (all of them) we evaluate the set of conditions, setting false the relevant bit in the child live vector if the process sensitive evaluation of the conditions shows that this process would not reach this function call. Therefore in this example, *eval_conds* returns a vector of the form 1100.

```
1  eval_mpi_call (statement, child_live_vector, local_defs)
2  {
3    if is_comm_rank (statement)
4      lhs = get_lhs (statement)
5      new_stmt = build_stmt (lhs, MPI_RANK)
6      record_assignment (new_stmt, child_live_vector)
7    else if is_comm_size (statement)
8      lhs = get_lhs (statement)
9      new_stmt = build_stmt (lhs, NUM_PROCS)
10     record_assignment (new_stmt, child_live_vector)
11   else if point_to_point_communication (statement)
12     for live_process in child_live_vector
13       src = live_process
14       dest = evaluate (get_dest_param (statement))
15       datatype = evaluate (get_datatype_param (statement))
16       num_elements = evaluate (get_num_elem_param (statement))
17       volume = sizeof(datatype) * num_elements
18       record_graph_edge (src, dest, volume)
19 }
```

Listing 3.4: Algorithm for evaluating MPI statements and recording additional graph edge weights when determined

The analysis then descends into communicate, calling *walker* on communicate, passing the created child live vector and the value of i (i.e. 0). Similarly to the call to communicate, we pass by the conditions and reach the MPI_Send call where again similarly to the call to communicate, the conditions are evaluated. As only process 0 is live in the live vector and the conditions evaluate to indicate the call is made, when *eval_mpi_call* iterates through the live processes a graph edge is only recorded as having been emitted by process 0. No action is taken on MPI_Recv as we do not perform send and receive matching.

Having reached the end of communicate, the analysis then returns from the child call to *walker* and continues analysing main. The analysis then reaches the loop latch condition which is process insensitive and not met, so the basic block is adjusted backwards so that we simulate the action of the loop executions. At this point we also reach the i++ (i.e. i = i +1) statement in the code and therefore i is iterated.

```
1  eval_conds (statement, live_vector, local_defs)
2  {
3    child_live_vector = live_vector
4    for live_process in child_live_vector
5      bb = get_basic_block (statement)
6      condBBs = get_cond_BBs (bb)
7      curBB = condBB[0]
8      for condBB in condBBs
9        cond_stmt = last_stmt (condBB)
10       outcome = evaluate (cond_stmt)
11       if outcome
12         curBB = follow_pos_edge (condBB)
13       else
14         curBB = follow_neg_edge (condBB)
15       if no_path_exists(curBB, bb)
16         child_live_vector[live_process] = 0
17         break
18   return (child_live_vector)
19 }
```

Listing 3.5: Algorithm for evaluating conditions at a function call

Therefore we reach the call to communicate again however, evaluating the same conditions this time results in a child live vector of 0110. We again call *walker*, this time passing this new child live vector, and the new value for i (i.e. 1). The execution path then continues as described above, until we reach the loop latch where i is increased to 4 before the latch is tested. Here the loop exit edge is followed and the final statements of the main function are evaluated. After the analysis is complete the edge weights output by *record_graph_edge* in *eval_mpi_call* are collated and the final graph returned.

3.2.6 Scalability

Scaling the number of processes results in a worst case $O(n)$ growth in the number of evaluations. This is due to the worst case being where all evaluations are process sensitive, with the number of evaluations increasing in line with the number of processes. A caveat to this is if the length of the execution path of the target program changes with

```
1  #include <mpi.h>
2  int my_rank, comm_size, indata, outdata;
3  MPI_Status stat;
4
5  int main (int argc, char **argv)
6  {
7    MPI_Init (&argc, &argv);
8    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
9    MPI_Comm_size (MPI_COMM_WORLD, &comm_size);
10   indata = comm_size + 4;
11
12   for(int i = 0; i < 4; i++)
13   {
14     if (my_rank == i || my_rank == i + 1)
15       communicate (i)
16   }
17
18   MPI_Finalize ();
19   return 0;
20 }
21
22 void communicate ()
23 {
24   if (my_rank == i)
25     MPI_Send (&indata, 1, MPI_INT, i + 1)
26   else
27     MPI_Recv (&outdata, 1, MPI_INT, MPI_ANY_SOURCE,
28       0, MPI_COMM_WORLD, &stat)
29 }
```

Listing 3.6: SImple MPI program

the number of processes. Specifically, if the length of the execution path is broadly determined by the number of processes then the scalability would be program specific and unquantifiable in a general sense. However, in such a situation one would often expect to see better scalability than the stated worst case, as a fixed problem size is divided between more processes, reducing the length of the execution path.

To improve upon the worst case, process sensitive and insensitive evaluation results are stored for each node of the data structure. This includes all nodes evaluated in the process of evaluating the requested node. These results are then attached to the relevant nodes. This means that reevaluation simply returns the stored result. While storage of these results requires additional memory, it prevents reevaluation of potentially deep and complex trees. Since we find only a fraction of nodes need evaluating and that those that are evaluated are evaluated multiple times, this does not pose a great memory capacity issue. As we will show in Section 3.4.4, we achieve far better than the worst case for all the benchmarks.

3.2.7 Limitations

There are a few limitations to the technique, some are fundamental to the static analysis of MPI, others particular to our design.

Pointers - The use of pointers in a statically unprovable way, with particular reference to function pointers, can lead the analysis to miss or misinterpret certain definitions. Specifically, assignments to an unresolvable pointer cannot be associated with the correct variable, and assignments that use unresolvable pointers will be incomplete. For function pointers the issue is more severe. If the pointer cannot be resolved, then all definitions contained in that function (and any functions called by that function) may be missed. This can mean either old definitions appearing current or for variables with no prior definitions, unknowns being returned if they are evaluated. Again we prioritise detective communications over soundness, neglecting the potential impact of statically unresolved pointer usage.

Recursive Functions - We take no account of recursive functions, which could lead to non-termination of the algorithm. Subject to the previous caveat, recursiveness can be determined by an analysis of the call graph or as the algorithm runs. The simple solution would be to not pursue placement if recursion is detected, but it is perhaps possible to allow some limited forms.

Incomplete Communication Graphs - If the complete communication graph cannot be resolved, it could produce performance degradation if placement or other optimisations are pursued. However, as we see in Section 3.4.2, certain forms of incompleteness can be successfully overcome. Automatically dealing with incompleteness in the general case remains an open problem.

3.3 Graph Partitioning

Generating an optimised spatial schedule for a communication graph is considered as a graph partitioning problem. To this end we apply the k-way variant of the Kernighan-Lin algorithm [KL70]. It aims to assign vertices (processes) to buckets (CMPs) in such a manner as to minimise the total weight of non-local edges. As the algorithm is hill climbing, it is applied to 1,000 random starting positions, and the naive schedules, to avoid only reaching a local maxima.

For this proof of concept we enforce placement by compiling in a lookup table, mapping the default *by-rank* schedule to the optimised schedule. This allows processes to behave as their assigned *rank* in the optimised schedule, with the destinations of their communications unmasked as required. A more general solution, free of masking correctness issues, would be to include the optimised schedule in a program header, to be acted upon by a modified MPI library.

3.4 Results

The primary goal of our experiments is to evaluate the efficacy of our framework in understanding communication in MPI programs. To this end, we evaluate our coverage – in terms of the percentage of sends we are able to fully understand. Next we investigate the improvements in communication localisation that are available from better process placement, guided by our analysis. This is followed by an evaluation of the performance improvements available from improved process placement. Finally, we explore the scalability of the technique.

We implemented our framework in GCC 4.7.0 [gcc], to leverage the new interprocedural analysis framework, particularly Link Time Optimisation. Experiments were performed using the 64 process versions of the NAS Parallel Benchmarks 3.3 [BBB+91], compiling for the *Class A* problem size. The NAS programs are a collection of parallel applications designed for the evaluation of supercomputers. They were chosen for our evaluation as they cover a broad range of scientific computing tasks and are full applications rather than simply kernels. We tested all the NAS programs that use point-to-point communication (BT, CG, IS, LU, MG and SP).

	Profiling		Analysis	
Benchmark	No. Call Sites	No. Bytes	No. Call Sites Correct	No. Bytes
BT	12	8906903040	12	$58007040 + n(44244480)$
CG	10	1492271104	10	1492271104
IS	1	252	1	252
LU	12	3411115904	12	$41035904 + n(13480320)$
MG[1]	12	315818496	12	$104700416 + n(52779520)$
SP	12	13819352064	12	$48190464 + n(34427904)$

Table 3.2: Coverage results and comparison with profiling for NAS Class A benchmarks using 64 MPI processes. As we can see, with the exception of MG, each *MPI_(I)Send* call site is being automatically and correctly evaluated in all contexts for all processes.

3.4.1 Coverage Results

We quantify coverage by two metrics: the number of *MPI_(I)Send* call sites that we can *correctly* understand, and the the total *number of bytes* communicated. An *MPI_(I)Send* is said to be understood correctly if we can identify the calling process, the destination process, and the volume of data communicated *in all the circumstances under which the call is encountered* – as seen in Listing 3.1, the same call site can be encountered in multiple contexts. In addition to this, each of the sends can repeat an arbitrary number of times, necessitating that the analysis resolves relevant loop iterators. To quantify this, we measure the total number of bytes communicated.

The coverage our analysis provides is shown in Table 3.2, with profiling results for comparison. With the exception of MG, each *MPI_(I)Send* call site is being automatically and correctly evaluated in all contexts for all processes. This means that our analysis is correctly identifying the calling processes, the destination and the volume of data for every *MPI_(I)Send* call site.

In CG and IS the number of bytes communicated also matches the profile run. For these programs, the relevant loops could be statically resolved by our framework. However, in BT, LU, MG and SP an unknown multiplier n exists. This occurs when the iteration count of a loop containing send calls cannot be statically determined; in the case of the four benchmarks affected, the iteration count is input dependent. So while we understand each call site in all cases, we do not know how many times the call sites inside the loop are reached. As will be seen in the following section, this has no impact on the schedule, and hence the communication localisation.

In contrast, simple analysis of MG fails to determine the point-to-point commu-

[1] Requires partial input specification, see Section 3.4.1

nication graph. Our analysis correctly determines the sending processes (**source**) and the **datatype**, for each call site. However, the **destination**, **send count**, and number of iterations are input dependent. In the case of MG, the **destination** and **send count** depend on four input variables (nx,ny,nz,and lt). If these variables, which determine the problem scale, are specified, then our analysis is able to correctly evaluate each call site. With programs such as MG where the input is partially specified, one could specify the whole input (including the number of iterations), but this is not necessary.

The case of MG highlights the issue of input dependency and how it can blunt the blind application of static analysis. For programs where the communication pattern is input dependent, analyses of the form proposed in this work will never be able to successfully operate in an automatic manner. However, by supplying input characteristics (as would be required for profiling), it is possible to determine the same communication graph that profiling tools such as mpiP [VM01] observe. Crucially, unlike profiling, this is without requiring execution of the program. Additionally, the structure of our representation makes it trivial to identify input variables upon which the communication graph is dependent. Therefore an interactive version of this analysis may be the ideal solution for the general case. For the following sections, we will assume that the four required input variables have been specified for MG, with results as shown in Table 3.2.

3.4.2 Communication Localisation

In this section, we evaluate the communication localised by applying the partitioning algorithm to the communication graph generated by our analysis. We compare our localisation with four other policies. *Round-robin* and *by-rank*, the two default scheduling policies; *random* which shows the arithmetic mean of 10,000 random partitionings; and *profiling* in which the same partitioning algorithm is applied to the communication graph generated by profiling.

As described in the previous section, four of the programs (BT, LU, MG and SP) have an unknown multiplier in the approximation extracted by analysis. To see the impact of this, communication graphs for each of these benchmarks were generated using values of n from 0 to 1,000. Partitioning these graphs yielded the same (benchmark specific) spatial schedules for all non-negative values of n. Therefore we can say that the optimal spatial schedules for these programs are insensitive to n (the only difference in coverage between profiling and analysis).

Figure 3.3 shows partitioning results for the NAS benchmarks on 8-core per node machines. Figure 3.4 shows the same for 12-core per node machines. One can see from these results that of the naive partitioning options *by-rank* is the most consistently effective at localising communication, better than *round-robin* as has previously been used as a baseline [CCH+06]. In fact we see that *random* is more effective than *round-robin* for these programs. Confirming our coverage results from the previous section, and our assertion of the null impact of the unknown multipliers, we see that our *analysis* localisation results match the *profiling* localisation results for each of the programs tested, as the same schedules are generated.

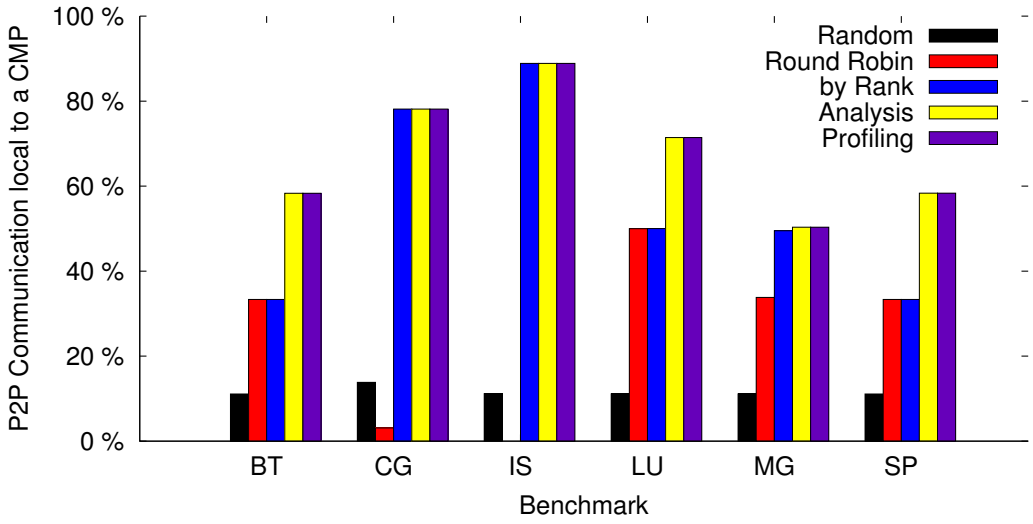

Figure 3.3: Percentage of point-to-point communication localised to an 8-core per node CMP. We can see that in all cases we match the localisation provided by profiling. In 4 out of the 6 benchmarks we see an improvement over *by-rank*, on average an improvement of 28%.

At 8-core per node we see improvement in 4 out of the 6 benchmarks. On average [2] we see 28% improvement over *by-rank*. We also see that *round-robin* performs equivalently to *by-rank* in 3 cases (BT, LU and SP), in the others it performs worse. For 12-core per node systems we see improvement in 5 out of the 6 benchmarks. On average we see 7% improvement over *by-rank*. Again *round-robin* significantly under performs other strategies. In fact in 4 cases it fails to localise any communication.

As Figure 3.3 and Figure 3.4 show, it is not always possible to improve upon the best naive scheduling (*by-rank*). This occurs when the program is written with this

[2] Geometric mean is used for all normalised results.

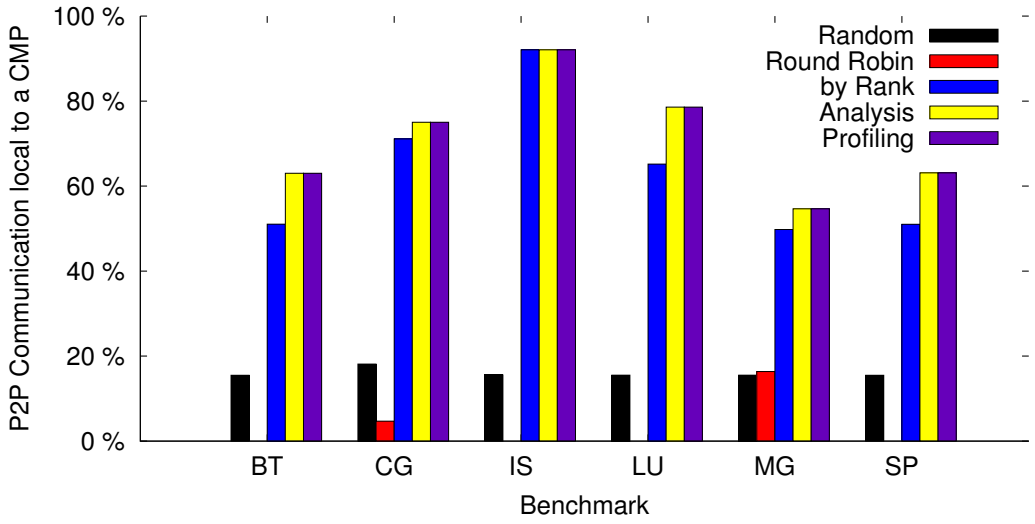

Figure 3.4: Percentage of point-to-point communication localised to a 12-code per node CMP. We can see that in all cases we match the localisation provided by profiling. In 5 out of the 6 benchmarks we see an improvement over *by-rank*, on average an improvement of 7%.

schedule in mind and the underlying parallel algorithm being implemented is conducive to it. However as the results show, analysis of the communication graph and intelligent scheduling can increase the localisation of communication.

3.4.3 Performance Results

To evaluate the performance benefits available through intelligent spatial scheduling, we perform a number of performance experiments on a gigabit Ethernet linked shared use cluster which has both 8-core and 12-core nodes available. We use the *by-rank*, *round-robin* and *analysis* schedules to compare the schedule determined by our technique with the naive alternatives. As the schedules determined by *profiling* match those determined by *analysis*, *profiling* is not shown separately. To mitigate the impact of noise, particularly due to other workloads on the shared interconnect, for each benchmark the 3 potential schedules are executed repeatedly in a consecutive manner for a period of 20 hours. Though on this system [ECD], one can specify the scale of node required, specifying individual nodes is not possible. To control for the impact of the selection of nodes provided by the system, each experiment was repeated on 10 selections of nodes, with arithmetic means from all 200 hours of experimentation, with extreme outliers removed, taken for comparison. The reasoning behind the repetitions

is that not all pairs of nodes possess the same internode latency, as a graph partitioning based placement assumes. This step is to eliminate any bias produced by a particularly unbalanced selection of nodes.

The results for experiments on 8-core per node machines are shown in Figure 3.5. Figure 3.6 shows the same for 12-core per node machines. For 8-core per node machines there is an average speedup of 1.01x over *round-robin* but no average improvement over *by-rank*. Our best result at this scale is SP, achieving a speedup of 1.03x (1.06x) over *by-rank* (*round-robin*). For 12-core per node machines the average speedup is 1.04x over *round-robin* and 1.02x over *by-rank*. Our best result at this scale is CG, achieving a speedup of 1.08x (1.18x) over *by-rank* (*round-robin*). Achieving significant average speedup over *by-rank* on 8-core per node machines is challenging, as in 2 of the benchmarks the schedules are the same and at this scale there are no spare slots in the system. Additionally, for MG the improvement in communication localisation at 8-core per node is only 1%. As Figure 3.5 shows, taking advantage of this localisation actually resulted in reduced performance.

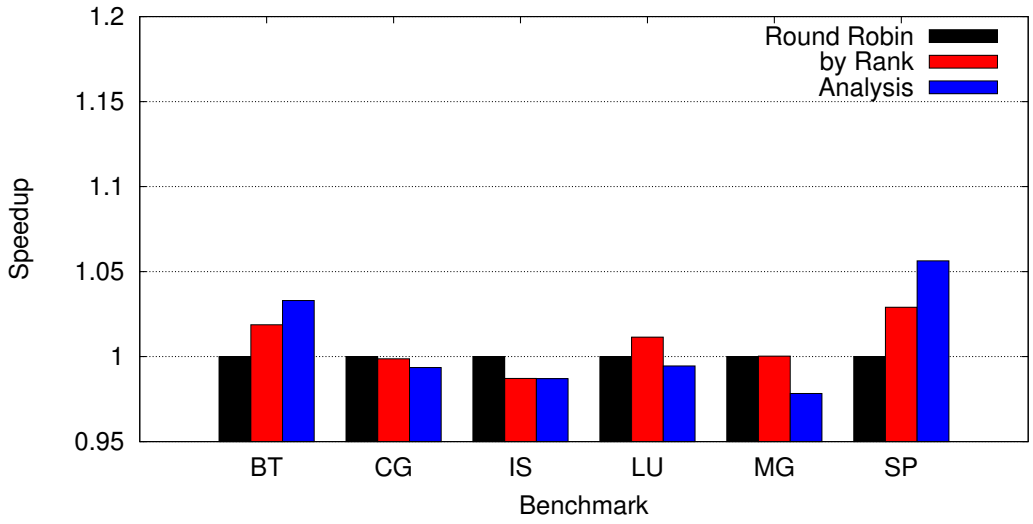

Figure 3.5: Normalised speedup for 8-core per node machines for *round robin*, *by-rank* and *analysis*. The best result at this scale is SP, achieving a speedup of 1.03x (1.06x) over *by-rank* (*round-robin*). On average there is no speedup over *by-rank*, and only 1.01x over *round-robin*.

Our speedups at 12-core per node broadly correlate with those found by previous work [CCH+06], that used profiling to inform process placement. By comparison we can now state that we can achieve optimised placement using our static analysis. In

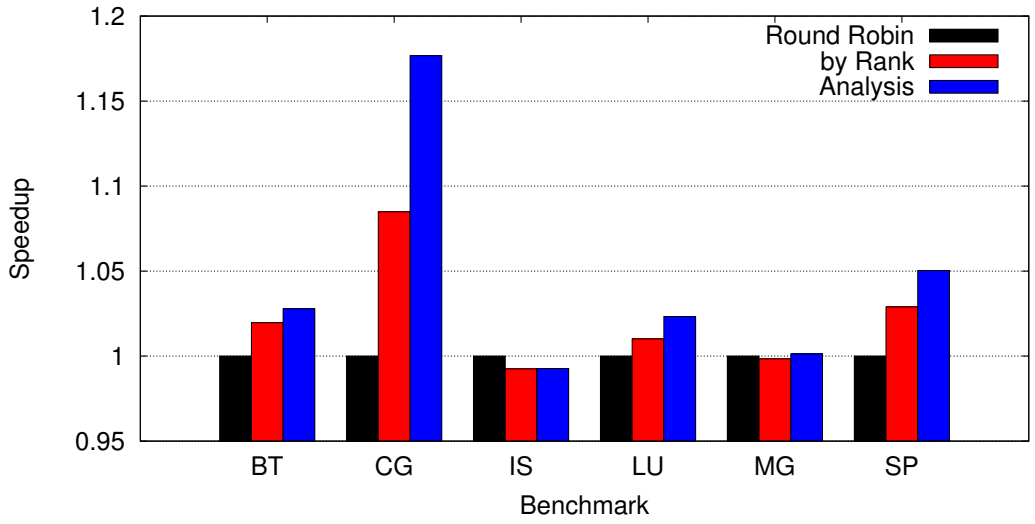

Figure 3.6: Normalised speedup for 12-core per node machines for *round robin*, *by-rank* and *analysis*. The best result at this scale is CG, achieving a speedup of 1.08x (1.18x) over *by-rank* (*round-robin*). On average the speedup is 1.02x over *by-rank* and 1.04x over *round-robin*.

addition, the results presented in their work [CCH+06] used only 16 or 32 processes, with greater speedups available at 16 processes.

The lack of a tight correlation between the improvement in communication localisation and the speedup observed can be attributed to factors not explored in an analysis of point-to-point communication. Analysis of the NAS Benchmarks has also shown that of those tested, only IS would be communication-bound, with the others spending only a small fraction of time communicating [WMADC99].

The slowdown seen in IS in both 8-core and 12-core results is explicable through the use of a *MPI_Alltoallv* call for a majority of directed communication. The analysis could extend to this primitive, but in this case it would not improve the results, as the communication through this primitive is input dependent. The *by-rank* and *analysis* schedules are identical in IS, but due to the aforementioned input dependence, exploiting speedup due to scheduling depends on the input data for a particular run. Therefore a profiling approach could only produce better results if tuned for a specific input.

3.4.4 Scalability Results

To confirm our assertions in Section 3.2.6, we compiled the benchmarks for different numbers of processes. Figure 3.7 presents the results by comparing the total number of

nodes of the data structure evaluated during each compilation. Note that a reevaluation returning a stored result, as described in Section 3.2.6 still adds 1 to total count.

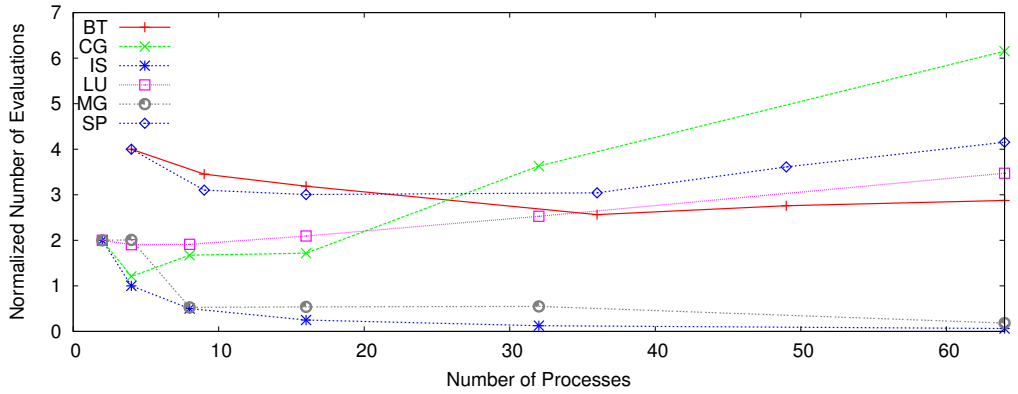

Figure 3.7: Normalised total number of evaluations at each usable number of processes. BT and SP are normalised to 4 processes as they only support square numbers. Note that we achieve significantly better than the $O(n)$ worst case. In IS and MG we can also see the impact of reduced work per process as the number of processes is scaled.

As Figure 3.7 shows, we achieve notably better than the $O(n)$ worst case. This demonstrates the effectiveness of the optimisations described in Section 3.2.6. With particular reference to IS and MG, we can also see the impact of the reduction in work per process, manifesting as a reduction in the number of evaluations, as the process specific program simplifies. Overall the scalability results are positive for all programs, with significant improvement over the worst case.

3.5 Conclusions

In this chapter we proposed a novel framework for the interprocedural, fully context and flow sensitive, best-effort analysis of MPI programs. This framework leverages a new data structure for maintaining partially evaluated, context sensitive variable representations for on-demand process sensitive evaluation. We instantiated this framework to provide a static method for determining optimal process placement for MPI programs running on CMP-based clusters.

Our analysis is able to resolve and understand 100% of the relevant MPI call sites across the benchmarks considered. In all but one case, this only requires specifying the number of processes. Using the 64 process versions of the benchmarks we see

an average of 28% (7%) improvement in communication localisation over *by-rank* scheduling for 8-core (12-core) CMP-based clusters, which represents the maximum possible improvement.

With this pragmatic technique we are able to determine the communication behaviour of an MPI program at compile-time. This allows for the intelligent placement of MPI processes to cores within a CMP-based clusters. Within the context of this thesis, we have addressed the issue of mapping legacy message-passing programs to CMP-based clusters and the performance issues this can introduce.

Chapter 4

Acquire Detection and Fence Placement for Legacy DRF Programs

4.1 Introduction

We now turn to the shared-memory paradigm and examine the problem of ensuring that legacy shared-memory programs, specifically legacy data race free programs, written assuming Sequential Consistency (SC) operate correctly on modern architectures. In this chapter we focus on the identification of acquires and the application of this identification to fence placement. In the following chapter we will show another application of this identification with our *SyncDetect* tool.

4.1.1 The Problem

A memory consistency model is at the heart of shared memory concurrency, and specifies the value that each read in the program can return. Sequential consistency (SC) [Lam79] in which each read returns the last value written to that location in a global order found by interleaving the actions of each thread, is arguably the most intuitive of memory models [CTMT07, Hil98, LP01, SNM+12].

Unfortunately, as is now well-known, modern hardware does not provide SC to the programmer. Instead, different hardware architectures produce different varieties of relaxed consistency behaviour [AG95]. An agnostic compiler could also perform optimisations that could violate SC.

The primary means by which the compiler can provide support is to insert appropriate fences to enforce sufficient orderings to restore SC. Each processor architecture

provides different fences to enforce various types of orderings. The challenge is to insert sufficient fences to restore SC, while at the same time not inserting too many. Fences are expensive, since they limit many of the optimisation opportunities available to hardware because of the relaxed memory consistency. Indeed, placing fences between every pair of accesses would guarantee SC, but would be far too expensive.

The starting point of understanding the required placement of fences is the seminal *Delay-set analysis* of Shasha and Snir [SS88]. They observed that to ensure SC, it is not necessary to order all pairs of accesses. Only conflicting pairs of accesses (the delay sets) that can potentially lead to SC violations need to be ordered – where conflicting accesses are two accesses to the same address, at least one of which is a write. The memory orderings produced by Delay-set analysis are then subject to *fence minimisation* [LP01], which seeks to minimise the number of fences required to enforce the above memory orderings.

One major issue that limits the practicality of Delay-set analysis is its reliance on alias analysis which is notoriously imprecise for programs that make heavy use of pointers. In addition to this, scalability is also an issue for large programs. To overcome the scalability issue, approximations of Delay-set analysis using escape analysis have been developed, notably by the the Pensieve project [FLM03, SFW+05]. More recently, attempts have also been made to address the scalability issue without resorting to escape analysis [AKNP14] – although recursion and dynamic thread creation continues to limit applicability. For either approach however, the imprecision issue remains unresolved, even with state-of-the-art alias analysis. This causes Delay-set analysis to produce a large number of superfluous orderings for real-world programs [AG95, LNG10, SNM+12].

4.1.2 Our Approach

We take a fresh look at fence placement. Our point of departure is that we do not seek to enforce SC for the general case. Instead, we insert sufficient fences to ensure that those memory accesses that are race free[1] in the SC world continue to be race free in the relaxed world. To put it succinctly, we guarantee SC behaviour only for race free accesses.

Our approach is based on the realisation that SC (which strongly orders all ac-

[1] A memory access is said to be race free if in all legal SC executions, it is ordered with its conflicting accesses in each execution, via the ordering chain introduced in section 3 (following Gharachroloo [Gha95])

cesses) is not an end in itself to programmers; rather it is enough for programmers to have SC semantics only for synchronisation accesses (where synchronisation accesses are those accesses that are used to guard other data accesses from racing). Therefore, it suffices if we identify such synchronisation accesses and provide SC semantics for only those accesses.

In other words, we consider all behaviours exhibited by the original program, as executed under SC semantics, to be the only behaviours intended by the programmer. In order to run the program correctly on modern (more relaxed) architectures, we need to ensure that only those behaviours seen under SC are seen under the more relaxed consistency model. Limiting the behaviours to those seen under SC is done by placing fences to limit the reordering of memory operations.

In order to understand this better, let us consider the two examples shown in Figures 4.1(a) and 4.1(b). In the producer-consumer example shown in Figure 4.1(a), the programmer synchronises using the flag variable, to ensure that the read b_2 returns the value produced by a_1 (and not the old value). In this example, accesses a_2 and b_1 are synchronisation accesses. Therefore, providing SC semantics to these accesses ensures that b_2 reads the correct value. The second example, shown in Figure 4.1(b), is a piece of code similar to that found in a relaxation solver [CM69, FS00], in which the four accesses involved are unsynchronised accesses (by design). Here, it is permissible for the accesses in either thread to be reordered, e.g. for the read of x in P2 to return a stale value (occurring before a_1 in P1) while b_1 reads the value written by a_2. In other words, they are data races, albeit benign in this case. Therefore, providing SC semantics to such unsynchronised accesses is not required.

(a)

P1	P2
$a_1 : data = 1;$	
$a_2 : flag = 1;$	b_1 :while$(flag == 0);$
	$b_2 : x = data;$

(b)

P1	P2
$a_1 : x = C_1;$	
$a_2 : y = C_2;$	$b_1 : local_2 = y;$
	$b_2 : local_1 = x;$

Figure 4.1: Examples of well-synchronised (a), and not well-synchronised (b) programs. Note that in example (a) SC semantics are required to ensure correct operation on a relaxed architecture. In example (b) no such semantics are required as the code is unsynchronised by design.

Although we do not promise SC in general, it is important to note that our approach guarantees SC for well-synchronised programs i.e. legacy data-race-free programs[2]. Figure 4.1(a) is an example of a well-synchronised program, whereas Figure 4.1(b) is not.

Our approach is similar in spirit to DRF (data-race-free) programming models, which form the basis of recent concurrent programming language models, such as the C11 concurrency model [BOS+11, BA08] and the Java Memory Model specification [MPA05]. This is a programming model which gives semantics to only DRF programs: programs in which synchronisation operations are correctly labelled and the program is well-synchronised using those operations. In return for this discipline the system (hardware + compiler) guarantees SC. However, legacy programs lack the distinction between data and synchronisation. Our approach automatically discovers synchronisation operations for such legacy programs.

4.1.3 Our Solution

We look for ways to conservatively identify synchronisation operations. If we can be relatively precise, we can prune unnecessary orderings found by more traditional approaches. The existing fence minimisation techniques can then be applied on the pruned orderings to achieve improved performance. An alternative application would be to use this identification to provide minimal annotations to make the program DRF, such that a compliant compiler and the hardware will prevent incorrect reordering.

We have identified two signatures, at least one of which must be fulfilled for a read to be a synchronisation, i.e. an acquire operation:

- **Control acquire**: a read feeds its value to a predicate tested for in a branch in its forward slice.
- **Address acquire**: a read provides the address value for a subsequent data access that the read (acquire) protects.

We formally prove that at least one of these must hold for a read to be an acquire. The second signature (address acquire) is less prevalent, and in particular is observed to appear along with the first signature (control acquire) in all cases in our experiments. We do not improve the identification of releases and, as in Pensieve, conservatively consider every shared write (escaping write) to be a release.

[2]More formally, these refer to a class of programs whose behaviour is characterised by values returned by only those reads that are race free under SC.

To evaluate the significance of our contribution, we next design and implement practical algorithms for identifying the acquires. Our simpler first algorithm (**Fast**) detects only control acquires, and does not do interprocedural flow analysis (which is expensive). This does mean that the algorithm theoretically does not detect all acquiring reads. In particular, it does not detect cases where the acquiring read and the branch (both of which intuitively form the acquire) are split across two functions[3]. We believe this will only rarely if ever be violated. In all our experiments we never see such a split, though contrived programs can be constructed.

Fast will also not detect address acquires. Again, in all our experiments, we have never seen an address acquire which is not also a control acquire. However, for completeness, we also develop a conservative variant of our algorithm (**Safe**). This variant detects address acquires in addition to control acquires.

We implemented our analysis in LLVM and applied it to the SPLASH-2 benchmark suite and a set of lock-free programs. Our experimental results show that on average, Fast reduces the number of orderings considered by 66% on average. Applying a fence minimisation technique, this translates to an average of 62% fewer fences on x86-TSO and up to 2.64x speedup over an existing practical technique. Safe on average reduces orderings considered by 32%, fences placed by 27% and produces a speedup of up to 1.54x.

The contributions of this chapter are:

1. We improve fence insertion for legacy programs by discovering synchronisation read operations.

2. We prove that for all the necessary orderings (*essential orderings*) involving a synchronisation read, the read has to satisfy at least one of two specific signatures: (a) that there is a conditional branch whose condition depends on the value returned by the read in the forward slice of the read. (b) that a read provides the address for a subsequent access that would otherwise be unknown.

3. We propose two practical algorithms: **Fast** that detects only control acquires and **Safe** that detects both address and control acquires. Both algorithms work in the presence of pointers.

4. We implement our algorithm within LLVM, and observe an average of 62% fewer fences and up to 2.64x speedup over an existing practical technique with

[3]Note that the data accesses which the acquire protects are subject to no such assumption, and can be located in a separate function.

the simpler algorithm, and an average of 27% fewer fences and up to 1.54x speedup with the conservative algorithm.

4.2 Our Approach

4.2.1 Fence Placement: Background

The starting point of understanding the required placement of fences is Shasha and Snir's Delay-set analysis. Its key insight is that not all pairs of memory accesses need to be ordered to ensure SC. Only pairs of memory accesses that conflict with accesses from other threads, potentially leading to (minimal) SC violations known as *critical cycles* need to be ordered. Identifying such critical cycles however, presents a scalability issue on real-world programs (with pointers, recursion etc.), as it relies on heavyweight interprocedural static analysis. To overcome this, practical tools such as Pensieve [FLM03, SFW$^+$05], conservatively approximate Delay-set analysis.

This conservative approximation is attained by such tools in a two step process. Firstly, a conservative thread-escape analysis is performed on each access in a function, to determine a set of potentially escaping accesses, E. Secondly, for $u, v \in E$, if analysis of the control flow graph shows that v can occur after u, then an ordering, $u \rightarrow v$, is recorded.

While this does generate a correct set of orderings, it produces a large number of false positives due to the thread-escape analysis being necessarily conservative. In practice this means that all references to memory that cannot be proven to be restricted to the local function, must be marked as potentially escaping.

Once a set of orderings has been identified, these orderings are fed as input to a fence minimisation algorithm. Such an algorithm will attempt to determine where to minimally place fences to ensure that all the orderings are enforced. It may also distinguish between types of orderings, to minimise the cost of enforcement. This can be achieved by using different types of fences or compiler directives, depending on the memory consistency model of the target architecture. For example, x86-TSO only requires orderings of the type $w \rightarrow r$ to be enforced with full memory fences, as other orderings are enforced by the hardware. These other orderings however, still have to be preserved during the compilation (optimisation) process.

4.2.2 Fence Placement for DRF Programs

Now let us consider fence placement for a DRF program. Recall that in a DRF program, synchronisation is achieved using special memory operations – a write known as a *release* and a read known as an *acquire* – such that there are no races amongst data operations. This implies that given such a well-synchronised program without data races, enforcing the orderings defined in Table 4.1 is sufficient to ensure correctness [Adv93].

$r/w \rightarrow w_{rel}$	All reads and writes before the release (in program order) should be ordered before the release
$r_{acq} \rightarrow r/w$	All reads and writes after the acquire (in program order) should be ordered after the acquire
$w_{rel}/r_{acq} \rightarrow w_{rel}/r_{acq}$	All synchronisation operations should be ordered among themselves[4]

Table 4.1: Sufficient orderings for correctness in a DRF program. Given a well-synchronised program without data races, if these orderings are enforced then this is sufficient to ensure intended behaviour.

In more detail, the first rule requires that all accesses to shared data must be performed before a release. Similarly, the second rule requires that all accesses to shared data must be performed only after an acquire. These two, combined with the third rule, ordering all acquires and releases, ensures correctness.

With precise information as to which of the reads (writes) are acquires (releases), determining the minimal set of required orderings is trivial. Specifically, orderings that do not conform to one of the definitions in Table 4.1, could be safely ignored. The set of required orderings could then be fed as input to a fence minimisation algorithm.

4.2.3 Identifying Acquires for Legacy DRF

There exists however, a large body of (legacy) code which is correctly synchronised, but the distinction between a read (r) and an acquiring read (r_{acq}), and a write (w) and a release (w_{rel}) is not made explicit by the programmer [XPZ+10]. We call such programs Legacy DRF.

One way to perform fence placement for such programs is to treat it like a general multithreaded program, i.e. use Delay-set analysis (or its conservative approximation) followed by fence minimisation techniques. Our key insight is that we can do better if we can conservatively identify synchronisation operations. In this chapter, we focus on detecting acquires.

[4]Weaker models which relax some of these requirements, such as RC_{PC} [AG95] in hardware and C11 [BOS+11, BA08] at the language level also exist.

We prove that for a read to be an acquire it must match at least one of two signatures. The first is that there exists a branch whose predicate is data dependent on the read, in the forward slice of that read. The second is that the read provides the address value for a subsequent data access that the read protects. Any read that fails to satisfy at least one of these signatures cannot be an acquire.

Intuitively, an acquire is a read which determines if shared data can be accessed. This necessarily involves either checking the value read and acting upon it (the first signature), or providing the address of data, which would otherwise be inaccessible (the second signature). A formal proof of these assertions can be found in Section 4.3.

By applying the two signatures to every read which may be thread-escaping, we determine a subset that includes every potential acquire. False positives can however still remain, either through the imprecision of the escape analysis or through conditional accesses on shared data that are already protected by synchronisation. Despite these limitations, as our results show, we still significantly reduce the number of acquires marked.

Having identified a conservative subset of the shared reads as potential acquires, we are able prune the orderings. Starting from the set of orderings given by Delay-set analysis (or its approximation that uses escape analysis), we prune all those orderings that do not adhere to one of the definitions in Table 4.1. Despite not identifying a subset of the shared writes and therefore having to consider all shared writes as releases, we are still able to prune a number of potentially expensive orderings.

Specifically, any ordering of the form $r_1 \rightarrow r_2$ requires at least r_1 to be an acquire to avoid being pruned, i.e. it must be of the form $r_{acq} \rightarrow r$. Similarly, any ordering of the form $w_1 \rightarrow r_2$ requires r_2 to be an acquire to avoid being pruned, i.e. it must be of the form $w \rightarrow r_{acq}$.

To demonstrate an application of these signatures this reduced number of orderings is provided as (an improved) input to a fence minimisation algorithm, resulting in a much reduced number of fences.

4.2.4 An Example

To illustrate the impact of pruning orderings, we now demonstrate the application of Delay-set analysis to a section of legacy DRF code and the fences that this would require. Then, using the acquire signatures and applying the pruning rules defined above, we determine the reduced set of fences required to enforce the remaining orderings.

Legacy DRF Code		Delay-set Fence Placement		Pruned Orderings Fence Placement	
P1	P2	P1	P2	P1	P2
$a_1 : x =$	$b_1 : *p1 =$	$a_1 : x =$	$b_1 : *p1 =$	$a_1 : x =$	$b_1 : *p1 =$
		—$(F1)$—	—$(F3)$—		
$a_2 := y$	$b_2 := *p2$	$a_2 := y$	$b_2 := *p2$	$a_2 := y$	$b_2 := *p2$
		—$(F2)$—		—$(F2)$—	
$a_3 : flag = 1$	$b_3 :$ while$(flag! = 1);$	$a_3 : flag = 1$	$b_3 :$ while$(flag! = 1);$	$a_3 : flag = 1$	$b_3 :$ while$(flag! = 1);$
			—$(F4)$—		—$(F4)$—
	$b_4 : y =$		$b_4 : y =$		$b_4 : y =$
			—$(F5)$—		
	$b_5 := x$		$b_5 := x$		$b_5 := x$

Figure 4.2: An Example of (full) fence placement on legacy DRF code for Delay-set and pruned orderings. By identifying that a_2, b_2, and b_5 are not acquires we are able to avoid placing $F1$, $F3$ and $F5$ as shown in Pruned Orderings Fence Placement.

In Figure 4.2, we present a section of legacy DRF code which contains a busy-waiting synchronisation. For the purposes of this example we assume that alias analysis has determined that $*p1$ and $*p2$ may potentially alias with both x and y, but not $flag$. If one were to apply Delay-set analysis, the following orderings would be determined to avoid the following critical cycles:

- $a_1 \rightarrow a_3, b_3 \rightarrow b_5$: to avoid $(a_1, a_3, b_3, b_5, a_1)$.
- $a_2 \rightarrow a_3, b_3 \rightarrow b_4$: to avoid $(a_2, a_3, b_3, b_4, a_2)$.
- $a_1 \rightarrow a_2, b_4 \rightarrow b_5$: to avoid $(a_1, a_2, b_4, b_5, a_1)$.
- $a_1 \rightarrow a_2, b_1 \rightarrow b_2$: to avoid $(a_1, a_2, b_1, b_2, a_1)$.

In the final cycle our assumption regarding $*p1$ and $*p2$ potentially aliasing with x and y but not $flag$ comes into play.

Using these orderings as input to a fence minimisation algorithm, 5 (full) fences are required to be placed to enforce the orderings. Placement of these fences is shown as *Delay-set Fence Placement* in Figure 4.2.

Pruning the orderings by applying the signatures defined in Section 4.2.3, we find that only the following remain:

- $a_1 \rightarrow a_3, b_3 \rightarrow b_5$: to avoid $(a_1, a_3, b_3, b_5, a_1)$.
- $a_2 \rightarrow a_3, b_3 \rightarrow b_4$: to avoid $(a_2, a_3, b_3, b_4, a_2)$.

Of the orderings which have been pruned: $a_1 \rightarrow a_2$, $b_1 \rightarrow b_2$ and $b_4 \rightarrow b_5$ are not required as none of a_2, b_2 or b_5 are acquires. Using this reduced set of orderings as input to the same fence minimisation algorithm, only 2 (full) fences are required to be placed. These fences are shown as *Pruned Orderings Fence Placement* in Figure 4.2.

$F1, F3$ and $F5$ are no longer required and have been removed. However, $F2$ and $F4$ are still required. Together they prevent $(a_1, a_3, b_3, b_5, a_1)$ and $(a_2, a_3, b_3, b_4, a_2)$, with $F2$ enforcing $a_1 \rightarrow a_3$ and $a_2 \rightarrow a_3$, and $F4$ enforcing $b_3 \rightarrow b_4$ and $b_3 \rightarrow b_5$.

In summary, we expect our signatures to considerably reduce the number of orderings that need to be enforced. With reference to our example, there are three major benefits.

- Acquire detection allows us to avoid enforcing many orderings that are not necessary (e.g., data $\rightarrow$ data orderings such as $a_1 \rightarrow a_2$ and $b_4 \rightarrow b_5$), since the program is well-synchronised.
- The inherent imprecision of Delay-set analysis (or its approximation) in the presence of pointers results in the enforcement of orderings which are not necessary. Acquire detection allows us to prune some of these orderings (e.g., $b_1 \rightarrow b_2$).
- This reduction in the number of orderings, allows a fence minimisation algorithm to place fewer fences, (in this case, not placing $F1, F3$ and $F5$).

4.3 Correctness of Acquire Signatures

In this section we formally prove[5] the basis of our assertions above, that is, a synchronisation read (acquire) matches (at least) one of two signatures. One is that in its forward slice, there must be a conditional dependent on the value returned by the read. The other is that the acquire reads a value determining the address of a subsequent access.

4.3.1 Language

For concreteness, we define our programming language to be a simple multi-threaded "while" language with pointers. Expressions e are pure, defined as making no shared-memory loads or stores, though local variables (marked with an r are allowed. State-

[5]The proof presented was developed by Susmit Sarkar, Vijay Nagarajan, and the author. It was then formalised by Susmit Sarkar.

ments then can dereference pointers, load from and store to shared-memory locations, either explicitly or via pointers. The language is presented in Figure 4.3.

$$
\begin{array}{llll}
\text{Shared locations} & x; & & \text{Local variables} \quad r \\
\text{Expressions} & e & ::= & \&x \mid r \mid e + e \mid \ldots \\
\text{Statements} & s & ::= & x := e \mid r := x \\
 & & & \mid r := *e \mid *e := e \\
 & & & \mid \mathsf{skip} \mid \mathsf{if}\,(e)\,\mathsf{then}\,s\,\mathsf{else}\,s \\
 & & & \mid \mathsf{while}\,(e)\,\mathsf{do}\,s \\
 & & & \mid s;s \mid s||s \mid \ldots
\end{array}
$$

Figure 4.3: The programming language for proofs. This tiny language is sufficient to deliver all the needed results.

This tiny language captures all the essential features needed for our results. Note that in comparison to a full-scale language such as C, key simplifications are that all shared-memory loads and stores from a single thread are explicitly sequenced, and that function calls and returns are ignored. We also ignore read-modify-writes, but these can easily be added to the proof below, by considering them to be a read followed by a write to the same location.

4.3.2 Intended Behaviour

Given a program in the above language, we assume that there is some intended marking of accesses (shared-memory loads and stores) into data and synchronisation accesses. Data accesses are programmer-intended accesses; more formally, the behaviour intended by the programmer is defined by the values read by these operations. The rest of the accesses are assumed to be synchronisation accesses; these are assumed to be written only to make sure there are no races on the data accesses. Following standard practice, we call synchronisation reads *acquire* reads and synchronisation writes *release* writes.

4.3.3 Behaviour under SC

A *sequentially consistent execution* is an execution trace (a linear order of read and write actions) which is a free interleaving of thread-wise actions, such that actions belonging to any thread appear in the execution trace in the order they occur in that

thread, and each memory read reads the value of the last write to that location in the trace. Note that in general, a single access in the program might lead to one or more actions in the trace (due to loops), or none (in case of a conditional). There is a straightforward way of associating each action in the trace to at most one program access, and we associate the corresponding kind (data or synchronisation) of program access to the actions. Of course, because there might be several possible interleavings, a program has a set of allowed sequentially consistent executions. For each such execution, we intuitively consider the results of the execution to be the values returned by the data reads. We formally consider the intended behaviour of the program to be the set of data read actions of any possible sequentially consistent execution.

4.3.4 Behaviour under relaxed consistency

A program actually executes not on a sequentially consistent machine but on a machine with relaxed consistency. We follow the approach of Adve and Hill [AH90b] (the approach of Gharachorloo [GLL+90] is very similar), and define that a program is correct iff it has no more behaviour in a relaxed consistency setting than in the sequentially consistent world.

We define happens-before following Gharachorloo [Gha95] by first defining conflict order and program order. Define *conflict order* $\xrightarrow{con}$ to be an order relation between conflicting actions in an execution (the order says one happens before the other), where two actions conflict if they are to the same address and at least one is a write. In particular, a write is conflict-ordered before a read if the read reads from that write. Also, there is an obvious *program order* relation $\xrightarrow{po}$ between actions from the same thread.

Given two actions u and v, u happens-before v (written $u \xrightarrow{hb} v$) in that execution if either $u \xrightarrow{po} v$ or $u \xrightarrow{po} w_1 \xrightarrow{con} r_1 \xrightarrow{po} w_2 \xrightarrow{con} r_2 \ldots w_n \xrightarrow{con} r_n \xrightarrow{po} v$. We consider only executions in which each synchronisation read reads from the last write to that location in happens-before. The behaviour of a program is determined by the data reads (value and location) of all such executions.

4.3.5 Well synchronised programs

We call a program (legacy) data-race-free if in all executions (where synchronisation reads read from the last write in happens-before as above), all conflicting data actions are ordered by $\xrightarrow{hb}$. It has been proved [AH90b, GLL+90] that data-race-free programs have no more behaviour in this sense than sequentially consistent behaviour of the

same program. However, since legacy programs do not have explicit markings of data and synchronisation, and to avoid confusion with the standard data-race-free notion, we equivalently call legacy data-race-free programs well-synchronised.

4.3.6 Ordering edges: Essential and Non-essential

We call a program order edge *essential* if ignoring that edge allows a data read to read a value not possible under SC, and all other program order edges *non-essential*. Thus enforcing all essential program order edges is sufficient to preserve SC behaviour for the data reads.

We now prove a happens-before characterisation of essential edges. Specifically, we prove that an edge in a well-synchronised program, i.e. (legacy) data-race-free program, is essential iff ignoring that edge in happens-before defined as above allows an execution with a data race.

Lemma 1 *For a program which is data-race-free for a certain mapping, and $U \rightarrow V$ a program order edge, the edge is essential iff deleting $U \rightarrow V$ from happens-before allows an execution with a data race involving a read and write.*

Proof Both directions follow easily from unfolding the definitions.

For one direction, ignoring an essential edge allows a data read to read a value not possible under SC. That data read and the write it reads from must be in a data race, since if they are ordered via happens-before, then the read is still possible under SC.

In the other direction, suppose deleting $U \rightarrow V$ from happens-before allows an execution with a data race between a read and a write. Consider that read. Since the program is well-synchronised (that is, no data races before removing that edge), the read could not have read from that write. ■

Intuitively, if we disregard an essential ordering edge, the program is no longer data-race-free, and thus the DRF guarantees of [AH90b] and [GLL$^+$90] do not apply. In that case (disregarding essential orderings), there will be data reads observable that are not possible in sequentially consistent executions. This happens-before characterisation is easier to prove with, as we can now analyse the shapes of happens-before.

4.3.7 Informal explanation

We are now in a position to give the formal proof of our main result, Theorem 1. Before that, to orient the reader, we give the main idea of the proof informally.

The key insight is that if there is an essential ordering involving an acquire, then the acquire must have been guarding a data access; only then will relaxing the above ordering result in a data race (and thus, by Lemma 1, non-SC behaviour for the data reads). We illustrate 3 different ways in which an acquire can guard data. The formal proof will essentially say that these are the only cases to consider, which allows us to safely deduce the acquire signatures.

The first way in which an acquire can guard data is illustrated via the classic Producer-Consumer or MP (Figure 4.4). Here the data access (of x) is guarded by control-dependency, that is, control only flows to it if the (acquire) read of $flag$ reads 1.

MP	
P1	P2
$a_1 : x :=$	
$a_3 : flag := 1$	b_3 : while $(r_1 != 1)\{r_1 := flag\}$
	$b_5 : r = x$

Figure 4.4: The MP example. A classic producer-consumer synchronisation where the data access of x is guarded by a control-dependency.

The second way is when the value read by the acquire is used to calculate the address touched by the data access (that is, it only reads from the location if the acquire read a certain value). This could happen in the example in Figure 4.5, an example adapted from Gharachorloo. Here y (analogous to $flag$ above) stores the address of z initially, and the second read on the second thread reads from x only if the prior read reads x (otherwise it reads from z).

MP with Pointers	
Initially $z = 0, y = \&z, x = 0$	
P1	P2
$a_1 : x =$	
$a_3 : y = \&x$	$b_3 : r = y;$
	$b_5 : r_1 = *r$

Figure 4.5: The MP example with pointer arithmetic.

The third possible way is to have some form of mutual exclusion, in which the data access is in a critical region. In this case (seen in the Dekker's example in Figure 4.6),

the data access is prevented from performing in an execution where the synchronisation read reads the wrong value.

Dekker	
P1	P2
$a_1 : x := 1$	$b_1 : y := 1$
$a_2 : \text{if}(y == 0)\{$	$b_2 : \text{if}(x == 0)\{$
$a_3 :$ touch $z\}$	$b_3 :$ touch $z\}$

Figure 4.6: The Dekker Example.

4.3.8 Formal proofs

Given a program, and if we knew the marking into data and synchronisation, we call two accesses *potentially racing* if they are on different threads, at least one of them is a data write, and they are either statically to the same location, or at least one of them is is to a statically unknown location (this can happen if it is to a location derived from a value read before on the same thread).

Lemma 2 *For two potentially racing accesses U and V in the program, and any legal execution X according to the relaxed consistency model, at least one of the following must happen:*

1. *U and V correspond to two actions which form a data race in X;*
2. *U and V correspond to actions u and v respectively in X that are ordered $u \xrightarrow{po} w_1 \xrightarrow{con} r_1 \xrightarrow{po} w_2 \xrightarrow{con} r_2 \ldots w_n \xrightarrow{con} r_n \xrightarrow{po} v$ in X;*
3. *U and V correspond to actions u and v respectively in X that are to different locations (this can only happen for statically unknown locations);*
4. *at least one of U and V do not correspond to any actions in X;*

Proof Immediate from the definitions of data races and happens-before. ■

Lemma 2 intuitively says that for static program accesses that potentially race, in any execution either there is an actual race, or there is a proper happens-before ordering such as in Figure 4.4 between the actions corresponding to the race, or one or the other access is to a different locations (such as in Figure 4.5) or absent altogether (such as in Figure 4.6).

Lemma 3 *For all essential orderings which are of the following form:*

1. *$R \rightarrow A$, where R is an acquire and A is a subsequent access; or*
2. *$W \rightarrow R$, where W is a write and R is a subsequent acquire,*

the value read from the acquire must feed into:

- *Either a conditional which guards a subsequent access;*
- *Or an address computation which determines the location of a subsequent access.*

Proof Given the essential ordering edge in the premise of the theorem. It can be of two types: R $\rightarrow$ A, or W $\rightarrow$ R. Consider disregarding this ordering edge in happens-before. Since the ordering edge is essential, by Lemma 1 there is a data race in some execution. Call that execution X, and consider the two data accesses U and V involved in the race. Since they correspond to racing actions in an execution, they must be *potentially racing* accesses. Consider the execution Y with the ordering edge present, and otherwise is the same as X, except that because reads may read different values, some actions may not occur or occur with different values in Y than in X. Apply Lemma 2 to the legal execution Y. Then one of the four cases must apply.

Case 1: In Y, U and V correspond to two actions u and v which form a data race. Since the program is assumed data-race-free, and Y is a legal execution, this case cannot occur.

Case 2: In Y, U and V correspond to actions u and v respectively in X that are ordered $u \xrightarrow{po} w_1 \xrightarrow{con} r_1 \xrightarrow{po} w_2 \xrightarrow{con} r_2 \ldots w_n \xrightarrow{con} r_n \xrightarrow{po} v$ in X. The ordering edge in question must occur in this chain. Since there is no W $\rightarrow$ R ordering edge in this chain, the essential ordering edge we are dealing with must be of the form R $\rightarrow$ A. We now see where the action corresponding to R occurs in this chain. It cannot be the first step ($u \xrightarrow{po} w_1$), since u is a data access. It can be r_n in the last step ($r_n \xrightarrow{po} v$), or r_i in an intermediate thread ($r_i \xrightarrow{po} w_{i+1}$). In each case, R reads the value of a synchronisation write in this execution Y. Furthermore, v or w_{i+1} respectively is the access A in question. Consider now a different execution where R does not read the value of the same synchronisation write. Then it must be the case that either A does not occur, or A exists but accesses a different location, since otherwise the ordering chain does not exist and the program has a race. Thus either R feeds into a conditional guarding A or is used to calculate the address touched by A, as required.

Case 3: U and V correspond to actions u and v respectively in Y that are to different locations.

Since U and V correspond to racing actions u' and v' in X, at least one of the pairs (u,u') and (v,v') must be to different locations. Without loss of generality, let u and u' be to different locations. Then U must be to a statically unknown location, that is in fact different in X and Y. Since X differs from Y in that the essential ordering edge (either R $\rightarrow$ A or W $\rightarrow$ R) is not required, in either case the calculation of the location for U must be derived from the value returned by R.

Case 4: At least one of U and V do not correspond to any actions in Y.

Without loss of generality, let there be no actions corresponding to U in Y. Since U corresponds to an action u in X, U must be guarded by a conditional that is true in X but not in Y. Since X differs from Y in that the essential ordering edge (either R $\rightarrow$ A or W $\rightarrow$ R) is not required, in either case this conditional must be derived from the value returned by R.

■

Theorem 1 *For all essential orderings involving an acquire R, the value read from the acquire must feed into:*

- *Either a conditional which guards a subsequent access;*
- *Or an address computation which determines the location of a subsequent access*

Proof The possible orderings involving an acquire R are:

Case 1: $R_1 \rightarrow R$, where R_1 should also be an acquire (since data $\rightarrow$ acquire ordering is not essential). Proof is from Lemma 3 (treating R_1 as the acquire, first form applies).

Case 2: $W \rightarrow R$, where W is a write. Proof is from Lemma 3, second form applies.

Case 3: $R \rightarrow A$, where A is any access. Proof is from Lemma 3, first form applies.

■

4.4 Implementation

In this section we present two algorithms for identifying synchronisation reads, as used in our implementation. The first algorithm (**Fast**) only identifies acquires that meet our control signature, while the second (**Safe**) is conservative, as it additionally identifies acquires that only match our address signature.

While conservatism demands application of the address signature, in practice we find that only the control signature is required. In all the experiments we perform (see Section 4.5) we find no acquires that only meet the address signature. To reinforce this point we performed an empirical study of 9 common synchronisation primitives, the results of which are presented as Table 4.2. It is worth noting that these primitives represent common patterns used in synchronisation, indeed some underpin programs we examine later in Section 4.5. As we can see, acquires that match the control signature are far more prevalent. While there are acquires that meet the address signature, all of those also meet the control signature.

	Acquires			
	Addr	Ctrl	Pure Addr	Source
Chase Lev WSQ	✓	✓	✘	[CL05]
Cilk-5 WSQ	✘	✓	✘	[FLR98]
CLH Lock	✓	✓	✘	[Cra94]
Dekker	✘	✓	✘	[Dij65]
Lamport	✘	✓	✘	[Lam87]
MCS Lock	✓	✓	✘	[MCS91]
Michael Scott LFQ	✓	✓	✘	[MS96]
Peterson	✘	✓	✘	[Pet81]
Szymanski	✘	✓	✘	[Szy88]

Table 4.2: Breakdown of the types of acquires found in common synchronisation kernels. Notably, no acquires are found to only meet the address signature. That is all acquires found to meet the address signature also meet the control signature.

We make one simplifying assumption in our implementations, this is that the synchronising reads occur in the same function as the condition to which they lead. While an interprocedural algorithm would be a necessary step to achieving soundness, such a guarantee would also require access to all libraries/functions used, at compile time. We believe that this assumption is reasonable, since it is extremely rare for these two operations, which intuitively form the synchronisation, to be split across two functions (although it is possible to construct a contrived example). Indeed in none of the implementations of the primitives examined (implementations for CLH Lock and MCS Lock from [DGT13], all others from [AKNP14]), nor the real programs examined in Section 4.5 is this separation found.

Both of the algorithms depend on an intraprocedural static slicer that performs the actual identification of the synchronising reads, this is presented in Section 4.4.1. All the algorithms operate on infinite register load-store intermediate representations. We will now examine each algorithm in detail, before finally outlining the generation of orderings and the fence minimisation algorithm to which we input them. We assume that the set of escaping loads and stores has previously been identified, using a thread-escape analysis as in Pensieve.

4.4.1 Identifying Control Acquires

The algorithm for identifying escaping reads that match our control signature (**Fast**) is presented as Listing 4.1. To determine reads that meet our control signature we must determine which reads have branches (conditions) in their forward slice. To determine this efficiently, the algorithm in fact focuses on each conditional branch and examines the reads in its backwards slice. For each conditional branch in a function we retrieve the instructions that define the branch operands (lines 8 and 9). Then we initiate the backwards slicer to populate *sync_reads* with escaping loads from the backwards slice of the conditional branch, line 11.

```
1  sync_reads = ∅
2  seen = ∅
3
4  for cond_branch in function
5  {
6     work_list = ∅
7
8     for operand in cond_branch
9        work_list.insert(get_def(operand));
10
11    slicer(&work_list, &seen, &sync_reads);
12 }
```

Listing 4.1: Algorithm Fast, for matching the control signature.

Backwards Slicing - The algorithm for backwards slicing and populating *sync_reads* is presented as Listing 4.2. This algorithm performs a conservative intraprocedural backwards slice from the initial contents of *work_list*. Every load found while process-

ing the *work_list* is compared against the results of the prior escape analysis (line 14), and if escaping, added to *sync_reads* (line 15).

To ensure conservatism, whenever a load is found, alias analysis is used to find all stores in the function that potentially wrote the value being read (line 17). These stores are added to the *work_list* to be processed later. For instructions that are not a load, each operand is processed and the defining instructions of those operands are added to the *work_list* (lines 22 and 23).

To avoid becoming trapped in cycles and to improve efficiency, both of the signature matching algorithms maintain sets of previously examined instructions, *seen*. The slicing algorithm is responsible for populating (line 10) and checking against (line 7) these sets. Once the *work_list* has been exhausted, the algorithm terminates.

4.4.2 Identifying Both Control and Address Acquires

As we previously stated, the algorithm presented in the previous sections provides sufficient coverage for all the real programs we have seen. It is however possible that an acquire only meets the address signature. To contend with this eventuality we develop a conservative variant of our algorithm (**Safe**), presented as Listing 4.3. This variant identifies escaping reads that meet either or both of the signatures identified.

As with the algorithm for the control signature, we use a backwards slice. In addition to conditional branches, the slicing is performed from every instruction that is either a dereference or an address calculation. This ensures that any escaping reads that contribute to a value used as an address are added to *sync_reads*. In the case of a dereference, the slicer is applied to the operand of the instruction, i.e., the address (line 16). In the case of an address calculation (for example a *GetElementPtr* instruction in LLVM IR), the offset is sliced (line 13). As is to be expected, these two cases often overlap with an address calculation in the backwards slice and therefore subordinate to a dereference. Here again, the use of the *seen* set prevents reiteration.

4.4.3 Generating Pruned Orderings

Whichever algorithm has been used to populate *sync_reads*, the next step is the generation of orderings. Ordering generation is done in line with Pensieve, generating an ordering for every pair of variables in the set of potentially escaping loads and stores, if there exists a path between them. Within a basic block the order of statements gives a directed linear sequence of accesses. Whether there exists a path between basic blocks

```
1  slicer (*work_list, *seen, *sync_reads)
2  {
3     while (!work_list->empty())
4     {
5        inst = work_list->first();
6        work_list->remove(inst);
7        if (seen->count(inst))
8           continue;
9
10       seen->insert(inst);
11
12       if (inst.is_load())
13       {
14          if (escaping_reads.count(inst))
15             sync_reads->insert(inst);
16
17          for store in potential_writers(inst)
18              work_list->insert(store);
19       }
20       else
21       {
22          for operand in inst
23             work_list->insert(get_def(operand));
24       }
25    }
26 }
```

Listing 4.2: Algorithm for backwards slicing and the registration of escaping reads contained in the slice.

is determined prior to this process with an examination of the CFG, to create a lookup table of reachability. This can then be queried during ordering generation.

The addition that we make to ordering generation is to prune $w \rightarrow r$ and $r \rightarrow r$ orderings which do conform to $w \rightarrow r_{acq}$ and $r_{acq} \rightarrow r$ respectively. The pruning is achieved by querying orderings of the form $w \rightarrow r$ and $r \rightarrow r$ for previously identified synchronising reads.

```
1  sync_reads = ∅
2  seen = ∅
3
4  for inst in function
5  {
6     if (inst.is_address_calculation() or
7        inst.is_dereference() or
8        inst.is_cond_branch())
9     {
10       work_list = ∅
11
12       if (inst.is_address_calculation())
13          work_list.insert(get_def(
14             inst.offset()));
15       else
16          work_list.insert(get_def(
17             inst.operand()));
18
19       slicer(&work_list, &seen, &sync_reads);
20    }
21 }
```

Listing 4.3: Algorithm Safe, that identifies escaping reads that match either signature.

4.4.4 Fence Minimisation

Given the set of orderings to enforce, a fence minimisation algorithm is used to place as few fences as possible, while still enforcing all required orderings. To place fences, we use the locally-optimised fence placement algorithm described in Fang et al. [FLM03]. The only alteration we make to this algorithm is to not automatically place a fence at the beginning of each function, such a fence is only placed if the function contains synchronising reads. The rationale for placing this fence is to enforce interprocedural orderings, under x86-TSO if the function contains no synchronising reads then no interprocedural $w \rightarrow r$ orderings can terminate within the function and the absence of a full fence does not affect correctness.

When determining full fence placement we need only consider orderings that the hardware will not enforce. Our technique is generally applicable, but in our experi-

ments we target x86-TSO and therefore we only consider orderings of the form $w \to r$, as the other orderings are enforced automatically by hardware. However, to prevent incorrect reordering by the compiler, we place compiler directives to enforce orderings of any other form. Specifically, these directives take the form of empty memory-clobbering assembly instructions which have no presence in the final binary but prevent reordering of memory related statements around them. The same minimisation algorithm is used here, with the decision as to whether to place a full fence or a compiler directive determined by whether the set of orderings that would be enforced contains one of the form $w \to r$.

4.5 Results

We implemented our algorithms and a locally-optimised fence minimisation algorithm based on Fang et al. [FLM03], in LLVM 3.4.1. The programs were all compiled using the *O*2 optimisations.

Using a set of lock-free programs and the SPLASH-2 [WOT+95] benchmarks, we compare both the **Fast** (control acquires only) and **Safe** (control and address acquires) variants of our approach with an implementation of Pensieve[6] using locally-optimised fence minimisation (as described in Fang et al. [FLM03]). To establish a performance baseline we also compare against a (minimal) manual fence placement. The lock-free programs are introduced in Table 4.3.

It is worth noting that the programs considered are well-synchronised because they employ user-defined synchronisations which require fences to operate correctly on relaxed architectures. While the lock-free programs use user-defined synchronisation exclusively, the SPLASH-2 programs make use of both user-defined synchronisation (in programs such as FMM [TNGT08a] and Volrend [NMT10]), and also employ library calls to locks and barriers.

Our results are organised as follows. Firstly, we examine how many reads marked as potentially thread-escaping that our algorithms mark as an acquire, giving us a measure of the effectiveness of our technique. Secondly, we compare and breakdown by type the number of orderings generated by the naive and both variants of our approach. Thirdly, we present the reductions in the number of full memory fences placed for an x86-TSO machine, where only orderings of the form $w \to r$ require such enforcement.

[6] We use the term Pensieve throughout this section to refer to the version presented in Fang et al. [FLM03] with locally-optimised fence minimisation, rather than the later Sura et al. [SFW+05].

Canneal	A kernel that seeks to minimise routing cost for chip design using cache-aware simulated annealing. This program was drawn from the PARSEC suite [BKSL08], and was run with the Simlarge input set.
Matrix	A parallel implementation of matrix multiplication, that takes in two matrices and outputs both potential matrix products. To allow 64 threads to compete for work, it is built on top of a lock-free queue as described by Michael & Scott [MS96]. It was applied to two square matrices both of dimension 1,024.
SpanningTree	An implementation of a parallel spanning tree algorithm, built on top of a work-stealing queue as described by Bader et al. [BC05]. It was applied to a graph of 10,000 nodes, each of degree 1,000.

Table 4.3: Descriptions of the lock-free programs used in our experiments.

Finally, we present the performance improvements achieved over Pensieve. For the performance experiments, we used an Intel i3-2100 running Linux 3.2.0-67 (Ubuntu 12.04.4). All the programs were run using 64 threads.

4.5.1 Synchronisation Read Detection

Applying the algorithms as defined in Section 4.4, we are able to mark a subset of the potentially escaping reads as acquires. The percentage of these reads that are marked acquires by each variant of our approach is presented as Figure 4.7.

As we can see, the Fast form of our analysis is able to greatly reduce the number of reads which must be treated as acquires. In the best case (*Water-NSquared*), only 7% are potentially acquires. On average[7] we see 18% of the reads marked as acquires. Even in the worst case our analysis is able to significantly reduce the number of reads that must be treated as acquires. We see this in *Raytrace*, with 33% marked as acquires.

Using the Safe variant, we are still able to reduce the number of reads marked as acquires in all cases. On average we see 60% marked as acquires. In the best case (*Water-Spatial*), only 39% need be marked.

[7]Geometric mean is used for all normalised results.

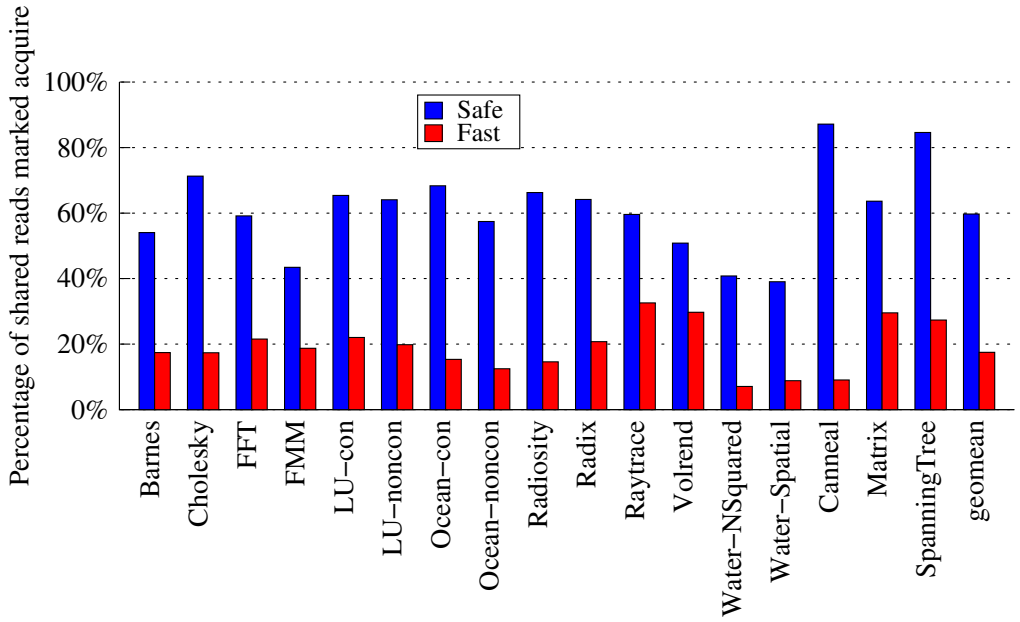

Figure 4.7: Static percentage of potentially thread-escaping reads that our analysis marks as an acquire. The Fast form of our analysis marks on average only 18% of the shared reads as acquires. The Safe form of our analysis marks on average only 60% of acquires.

4.5.2 Ordering Pruning

Using the acquire detection results, we are able to prune the orderings considered by the fence placement algorithm. As detailed in Section 4.2.3, identifying acquires allows pruning of those $w \rightarrow r$ and $r \rightarrow r$ orderings that do not conform to the rules in Table 4.1. Figure 4.8 presents the results of this pruning.

As Figure 4.8 shows, our Fast approach significantly reduces the number of $w \rightarrow r$ and $r \rightarrow r$ orderings required to be considered for fence placement. This result holds across all the programs tested, with an average of 34% orderings remaining after application of our approach. As $r \rightarrow r$ orderings form the majority of orderings in all but two of the programs, reducing them has the largest overall impact on the number of orderings considered. $w \rightarrow r$ orderings are also pruned significantly, though as they often form only a small percentage of overall orderings, the impact of this on the total number of orderings is smaller. As we do not identify a specific subset of writes as releases, $r \rightarrow w$ and $w \rightarrow w$ orderings are unaffected by the pruning process. With $w \rightarrow r$ and $r \rightarrow r$ orderings forming the majority of the orderings, the correlation between the

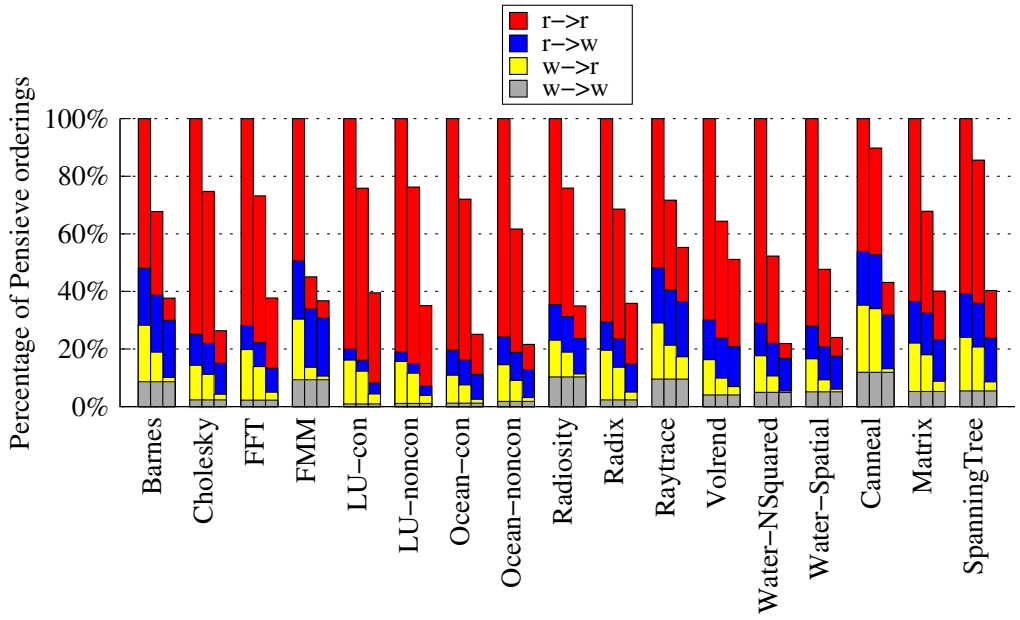

Figure 4.8: A breakdown of orderings by type for Pensieve (left), Safe (centre), and Fast (right). We see how our signatures have pruned $w \to r$ and $r \to r$ orderings. With the Fast approach only 34% of orderings remain. With the Safe approach 68% of the orderings remain.

percentage of reads marked as acquires (Figure 4.7) and the percentage of orderings that survive pruning is not unexpected.

Examining the results for the Safe variant, we see that reductions in $w \to r$ and $r \to r$ are still achieved. Specifically, only 68% orderings remain on average.

4.5.3 Fence Placement

In placing fences, we consider the requirements of an x86-TSO hardware model. Here, only $w \to r$ orderings require enforcement by a full memory fence. Other orderings are automatically enforced by the hardware and are enforced during the compilation process with empty memory-clobbering assembly instructions, that have no presence in the final program. As Figure 4.8 showed, our pruning was very effective at reducing the number of $w \to r$ orderings.

Applying the fence minimisation algorithm to the pruned sets of orderings for both variants of our approach and Pensieve for comparison, we determine the percentage of full fences that are still placed when using pruned orderings. This is shown as Figure 4.9.

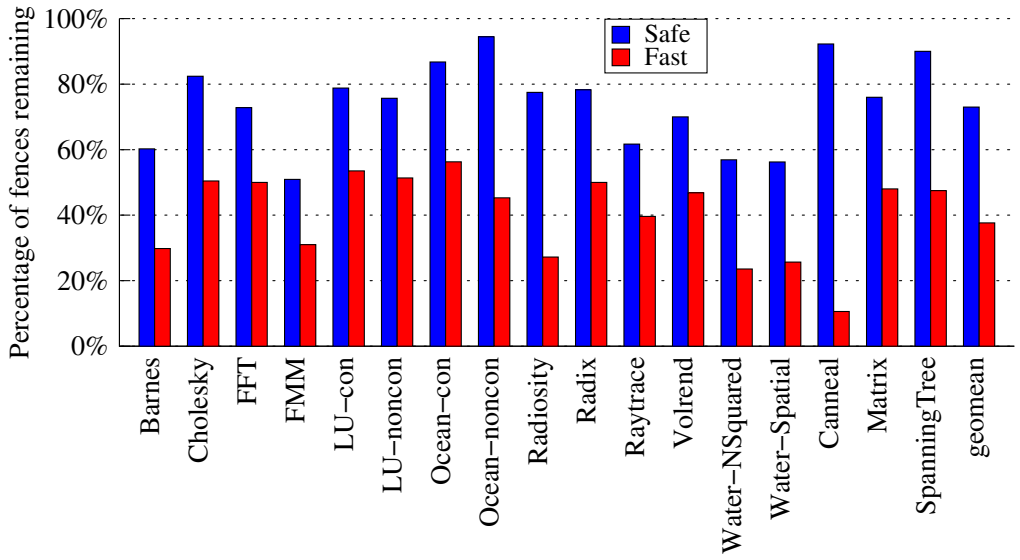

Figure 4.9: Static percentage of full fences that remain on x86-TSO after using pruned orderings. We see that by using the Fast approach only 38% of Pensieve's fences are required. With the Safe approach 73% of the fences placed for Pensieve remain.

As Figure 4.9 shows, the impact of pruning orderings is significant in reducing the static number of fences that the algorithm places to enforce $w \to r$ orderings. As we can see, the percentage of fences placed is quite strongly correlated with the percentage of reads marked as acquires (Figure 4.7). For the Fast algorithm we see on average 38% of Pensieve's fences required, with *Canneal* receiving a 89% reduction in the number of fences placed. For the Safe variant, on average 73% of Pensieve's fences are required.

4.5.4 Performance Improvements

To examine the impact of reducing the number of fences, we executed the programs having applied Pensieve, both variants of our approach and normalise these against manual fence placement. Each of the experiments was repeated 100 times and averages taken. The results of these experiments are presented as Figure 4.10.

As we can see, in all cases the fences placed using either variant of our approach results in a performance improvement over using a naive set of orderings. On average we see that Pensieve is 1.94x slower than the baseline, with our Fast approach being only 1.44x slower than the baseline. The Safe approach is 1.69x slower than the baseline. In other words, on average, our Fast approach results in a 30% speedup over

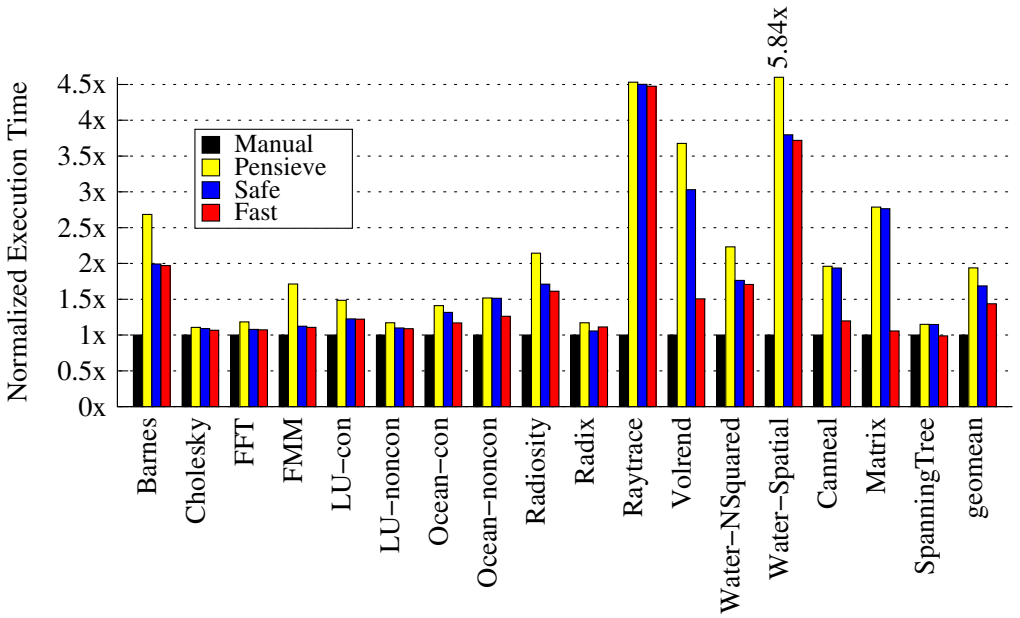

Figure 4.10: Execution time with fences placed using Pensieve, Safe, and Fast, normalised against manual fence placement. On average our Fast approach results in a 30% speedup over Pensieve. The Safe approach results in a 14% speedup on average.

Pensieve, while the Safe approach results in executions 14% faster than Pensieve. In the best case (*Matrix*) we achieve a 90% improvement over Pensieve using Fast. For the Safe approach, the best case (*Water-Spatial*) is 42% faster than Pensieve.

Examining the performance results for individual programs, we see that the speedups achieved over the naive are not strongly correlated with the changes in static fence placement. This is due to specific fences being reached more than others during the execution of the program. This is best highlighted by the case of *Raytrace*, where significant reductions in the number of static fences is not reflected in performance improvement. When looking at the results for Safe, we see that in some cases it is closer to Pensieve (e.g., *Ocean-noncon*) and in others (e.g., *Water-Spatial*) closer to Fast. To which result Safe is most similar depends on the propensity of the use of escaping reads as addresses in heavily executed code regions. In one program (*Radix*), we see Safe outperforming the simple algorithm. This is likely due to the short running time and small number of fences placed, making the result susceptible to noise. This also accounts for why Fast achieves a 1% improvement over the baseline for *SpanningTree*.

In terms of performance comparison with the manual baseline, we see that there is still some improvement possible. There are two reasons for this discrepancy. First is the difficult orthogonal problem of optimal fence minimisation given a set of orderings

to enforce. In extremis this may even require profiling to determine the fence insertion points that have the minimal impact on performance. Secondly, while our signatures significantly prune the number of shared reads considered as acquires, some false positives still remain.

4.6 Conclusions

Relaxed hardware memory consistency models are used to ensure performance in multicore computers. A large body of legacy code assumes SC. Placing sufficient but minimal fences is challenging. The starting point of understanding the required placement is Delay-set analysis. However, in practice approximations are applied, resulting in many superfluous orderings.

With Delay-set analysis too hard in the general case and with languages converging to DRF based memory models, we for the first time attack the problem of Delay-set analysis for legacy DRF programs. We prove that a read of shared data must match at least one of two signatures to be an acquire. We determine that this enables the pruning of a large number of orderings, reducing the set that need be considered for fence placement.

Developing both simple (control acquires) and conservative (control and address acquires) algorithms, we implement them in LLVM and demonstrate the significance of our contribution. Applying our control acquire detection on a set of lock-free programs and to SPLASH-2, we reduce the average number of orderings considered by 66%. Using a fence minimisation technique, this translates to an average of 62% fewer fences on x86-TSO and up to 2.64x speedup over an existing practical technique.

In this chapter we addressed the correctness problem present when legacy shared-memory programs are run on modern CMPs with relaxed memory models. Crucially, our identification of acquires allows us to do this with a lesser negative impact on performance than current general solutions. In the next chapter we will examine another application of our acquire signatures, namely to the dynamic detection of synchronisation, as a basis for debugging and race detection.

Chapter 5

Signature-based Dynamic Detection of Ad Hoc Synchronisation

5.1 Introduction

We now look at another application of our signatures, using them to dynamically detect synchronisation. As we will show, dynamic analysis in combination with our signatures and simple heuristics allows for the precise detection of synchronisation with limited false positives. This is important as it shows that our signatures can be used to significantly improve the results produced by data race detection and debugging tools. Alternatively, these techniques could be applied to modernise legacy code, marking acquires and releases, such that the program has proper semantics under DRF models.

In shared memory programming, synchronisation is key to ensuring that accesses to shared data do not conflict. There are a wide variety of synchronisation constructs, with different behaviours and for different purposes. While locks and synchronisation libraries have widespread adoption, ad hoc synchronisations are also found, even in large commercial applications [XPZ+10]. These ad hoc synchronisations, while if well written will be correct on multiprocessors that support Sequential Consistency (SC), will fail to properly synchronise on modern architectures where only more relaxed memory consistency models are found.

While, as we showed in the previous chapter, detecting synchronisation can be used to improve fence placement, here we instead focus on its application to the problems of debugging and data race detection [TNGT08b, TNGT09a]. From a debugging perspective, accurately identifying synchronisation can significantly aid programmers in understanding the operation of the program and expedite the debugging process. With

T1		T2	
1		1	data = 1024;
2		2	flag = 1;
3	while (flag != 1);	3	
4	local = data;	4	

Figure 5.1

regard to race detectors, being precise with synchronisation detection is essential. If a synchronisation is missed (a false negative) during synchronisation detection, then a race detector may report additional races (false positives) during race detection. Correspondingly, and far more importantly, false positives in synchronisation detection may result in false negatives in race detection, i.e. races present in the program may go unreported. Looking at a basic synchronisation, presented as Figure 5.1, we can see that if the read and write to $flag$ are not identified as synchronisation then a race detector will see two races, one for accesses to $data$ and another for accesses to $flag$. If however, the usage of $flag$ is correctly identified as a synchronisation then a race detector can report that the program is free of races (assuming SC).

In contrast to the previous chapter, the dynamic approach presented here is attractive due to its precision. Due to the imprecision of alias analysis and the conservative nature of the thread-escape analysis, a scalable static approach like that presented in the previous chapter produces many false positives. While the two approaches cannot be directly compared due to differences in infrastructure and synchronisation implementation, it is worth noting that our **Fast** static approach from the previous chapter identifies 86 false positive acquires when applied to FFT from SPLASH-2, even when using only our control acquire signature.

To improve upon existing solutions, we exploit the work presented in the previous chapter. There we identified two signatures which are necessary conditions for a read to be an acquire. We briefly restate them here:

- **Control acquire**: a read feeds its value to a predicate tested for in a branch in its forward slice.
- **Address acquire**: a read provides the address value for a subsequent data access that the read (acquire).

For our purposes in this chapter, the signatures are useful as they permit us to

exclude from consideration as synchronisation, reads of shared data that do not meet one of these signatures from the set of potential acquires. Therefore we should achieve a lower false positive rate than otherwise possible. We also note that as we showed in the previous chapter, in real programs testing for the control signature is sufficient to detect all acquires. Therefore in this chapter we will focus only on detecting these control acquires and not seek to find acquires that only meet the address signature. As was shown in the previous chapter, in practice reads that meet the address signature are also found to meet the control signature.

In this chapter we present *SyncDetect* a proof of concept tool to dynamically detect ad hoc synchronisation. It leverages the control acquire signature described above and is built on top of Intel's Pin framework [LCM+05] for dynamic binary instrumentation. It is generally applicable to shared-memory parallel programs and requires no program modification. The only imposition on the programmer is the inclusion of debugging symbols, without which it is impossible to provide informative output. We examine the effectiveness of our tool by applying it to a set of synchronisation kernels and to a program from the SPLASH-2 [WOT+95] suite.

The main contributions of this chapter are:

- A method for detecting shared reads and matching with the control acquire signature through taint tracking.
- A tool for dynamically detecting and reporting acquires and releases.

5.2 Our Approach

In this section we outline the details of our SyncDetect tool in terms of data structures, design decisions and the overall instrumentation algorithm. We begin by giving an overall description of the general components of our approach.

5.2.1 General Principles

We implement our tool using Intel's Pin framework [LCM+05]. This allows us to dynamically instrument running programs and avoid the need for simulation or additional hardware support as is commonly used in replay based race detectors [MCT08, NWT+07]. To be able to make useful reports to the user, we also require that the program being examined was compiled with debugging symbols. Without these sym-

bols we would be unable to match the reads detected at runtime with their source code counterparts.

There are three major structures in our implementation, providing the ability to track accesses and test reads for being an acquire. The first is a table shared amongst all threads mapping memory addresses to whether or not they have been accessed by more than one thread. The second is a table maintaining information about which instruction from which thread last wrote to an address. The third is a table, of which each thread has its own instantiation, maintaining a mapping of registers to reads which taint the register contents. This final structure is essential for testing if a read matches the control acquire signature. We will now discuss these three structures and their operation in detail, before going on to detail the logic that makes use of them to report acquires and releases to the user.

5.2.2 Shared Access Detection

To allow for precise determination of which memory addresses are accessed by more than one thread and are therefore considered shared, we maintain a global table mapping memory addresses to thread IDs. For every instruction that reads or writes memory this table is checked. If the memory address has never before been accessed then the thread ID is stored. if the address has previously been accessed by a different thread then a negative value is stored to indicate that this address is shared memory. This entry is then no longer modified.

At the end of the execution of the target program reads that meet the control acquire signature are tested for having been accesses to shared memory. The decision to delay testing for being a shared access is to allow for the detection of acquires for which there is no preceding write, i.e. cold reads. Additionally, certain synchronisations may be uncontested in the execution seen, with one thread acquiring, releasing and then reacquiring. By delaying the testing we cover cases where another thread later does acquire using the same synchronisation variable. We discuss the issue of a completely uncontested synchronisation, where no other threads ever make use of the synchronisation variable, in our discussion of limitations in Section 5.3.

5.2.3 Last Writer Tracking

To be able to infer releases, we track writes to memory by the address, thread and instruction that wrote. To perform this tracking we maintain a hashtable using the

memory address written as the key. This address maps to information about the last writer, specifically the instruction pointer and the thread ID. A representation of this last writer table is presented as Table 5.1. This reference to the instruction, with support provided by Pin, allows us to identify the source location of the access. For each instruction that reads memory this table is checked and if there was a previous write to the address read, then this information is associated with the read. This means that if the read subsequently meets the control acquire signature the potential release can also be identified.

Last Writer		
Memory Addr	Inst Ptr	Thread ID
0x7f83f2d00130	0x400dd7	0
0x7f83f2d00220	0x400e22	2
0x7f83f2c04ea0	0x400e83	1
...	...	...
...	...	...
...	...	...

Table 5.1: A representation of the main data structure, mapping memory addresses to the thread number and instruction pointer of the last writing instruction.

5.2.4 Acquire Detection

As shown in Chapter 4, in practice the control acquire signature is sufficient to detect acquires in real programs. To implement the control acquire signature in a dynamic binary instrumentation environment, we need to examine whether the value loaded by a read operation (on shared data) is used in the determination of a conditional branch decision. To do this, we implement register taint tracking to maintain information on which registers contain contents tainted by information loaded from memory.

As register information is thread specific in x86, we implement register taint tracking on a thread by thread basis, maintaining a mapping of registers to reads that taint them. These tables are kept up to date through instrumentation of each instruction. When an instruction is executed the current taint of any registers it reads is copied to an taint buffer, along with new taint information created for any shared memory locations read by the instruction. Then for every register written by the instruction, the taint information in the taint buffer is copied to taint mapping for that register. After

this, the taint buffer is cleared for the next instruction. The pseudocode for the actions taken on a register read is shown in Listing 5.1, with the actions taken on a register write shown in Listing 5.2.

As we can see in Listing 5.1, when an instruction is a branch decision (as determined using Pin's API), then the reads that taint that decision are copied into the set of potential acquires to be examined in our final analysis. Additionally, we can see in Listing 5.2 where on the final register write of an instruction, the taint buffer is cleared. Finally, we have an untaint option in Listing 5.2 that also clears the taint buffer, meaning that taint is not propagated further. Currently this is only set when the instruction being examined is an *xor* instruction which uses the same register as both sources. This prevents taint information being propagated when in reality it would not be. Discussions about how the taint tracking mechanism could be improved for future releases are found in Section 5.3.4.

```
1 onRegRead(Reg r, bool branchDec, unsigned tid)
2 {
3     taintBuffer[tid].insert (regFile[tid][r]);
4
5     if (branchDec)
6         potentialAcquires.insert(taintBuffer[tid].begin(),
7                                  taintBuffer[tid].end());
8 }
```

Listing 5.1: Pseudocode for actions taken for each register read during a run of SyncDetect.

```
1  onRegWrite()
2  {
3      if (untaint)
4          taintBuffer[tid].clear();
5
6      regFile[tid][w] = taintBuffer[tid];
7
8      if (final)
9          taintBuffer[tid].clear();
10
11 }
```

Listing 5.2: Pseudocode for actions taken for each register written during a run of SyncDetect

5.2.5 Detecting Synchronisations

Combining the global shared memory tracking with the last writer table, as detailed in Section 5.2.2 and Section 5.2.3, and the thread specific register files for taint tracking, as detailed in Section 5.2.4 we are now able to construct the full analysis. To summarise the actions taken on a read from memory or write to memory, we present pseudocode for those operations as Listing 5.3 and Listing 5.4 respectively. As we can see from Listing 5.3, when memory is read, the address read and the source location are recorded and if the read has read a memory location previously written then the details of the write are associated with the read. The read is also loaded into the taint buffer of the thread, for use in testing against the control signature. The shared memory table is also updated as necessary. As we can see from Listing 5.4, when memory is written to the details of the write are recorded and (as with reads) the shared memory table updated as necessary.

```
1  onRead(int *addr, int *instPtr, int tid)
2  {
3      reader *read = new reader ();
4      read->r = addr;
5      read->loc = getSourceLocation (instPtr);
6
7      if (isShared.count (addr) && isShared[addr] != tid)
8          isShared[addr] = -1; //Shared Memory
9      else
10         isShared[addr] = tid;
11
12     if (!lastWriter.count (addr))
13         read->wLoc = lastWriter[addr].loc;
14     else
15         read->wLine = 0; //No writer
16
17     lastWriter[addr].readers.insert (read);
18     taintBuffer[tid].insert (read);
```

```
19 }
```

Listing 5.3: Actions taken on a read.

```
1  onWrite(int *addr, int *instPtr, int threadID)
2  {
3      writer *write = new writer ();
4      write->w = addr;
5      write->loc = getSourceLocation (instPtr);
6
7      if (isShared.count (addr) && isShared[addr] != tid)
8          isShared[addr] = -1; //Shared Memory
9      else
10         isShared[addr] = tid;
11
12     lastWriter[addr] = write;
13 }
```

Listing 5.4: Actions taken on a write

For each instruction, we handle memory accessing instructions by modifying the last writer table and storing reads in the taint buffer using the operations defined in listing 5.3 and listing 5.4 in Section 5.2.3 respectively. Then for each register read or written by the instruction we update the taint tracking and potential acquire set using the operations defined in listing 5.1 and listing 5.2 in Section 5.2.4. The pseudocode for the actions taken on each instruction is presented as Listing 5.5.

```
1  onInstruction()
2  {
3      if (readsMemory (ins))
4          onRead (addr, ins, tid);
5      if (writesMemory (ins))
6          onWrite (addr, ins, tid);
7
8      for (regR in registersRead(ins))
9      {
10         branchDec = isBranchDecision(ins);
11         onRegRead(reg, branchDec, tid);
12     }
13
```

```
14      for (regW in registersWritten(ins))
15      {
16          untaint = isXorSameSrcDest(ins);
17          final = regW == registersWritten(ins).last();
18          onRegWrite(reg, final, untaint, ins, tid);
19      }
20  }
```

Listing 5.5: Pseudocode for actions taken for each instruction during a run of SyncDetect.

To better explain the operation of SyncDetect and the interactions of the components, we finally present an overview diagram as Figure 5.2. In this figure we can see the shared last writer infrastructure and the thread specific register taint tracking and taint buffers and how these components interact. Also note the shared memory tracking and the set of potential acquires, which at the end of the execution is processed to produce the final report provided to the user.

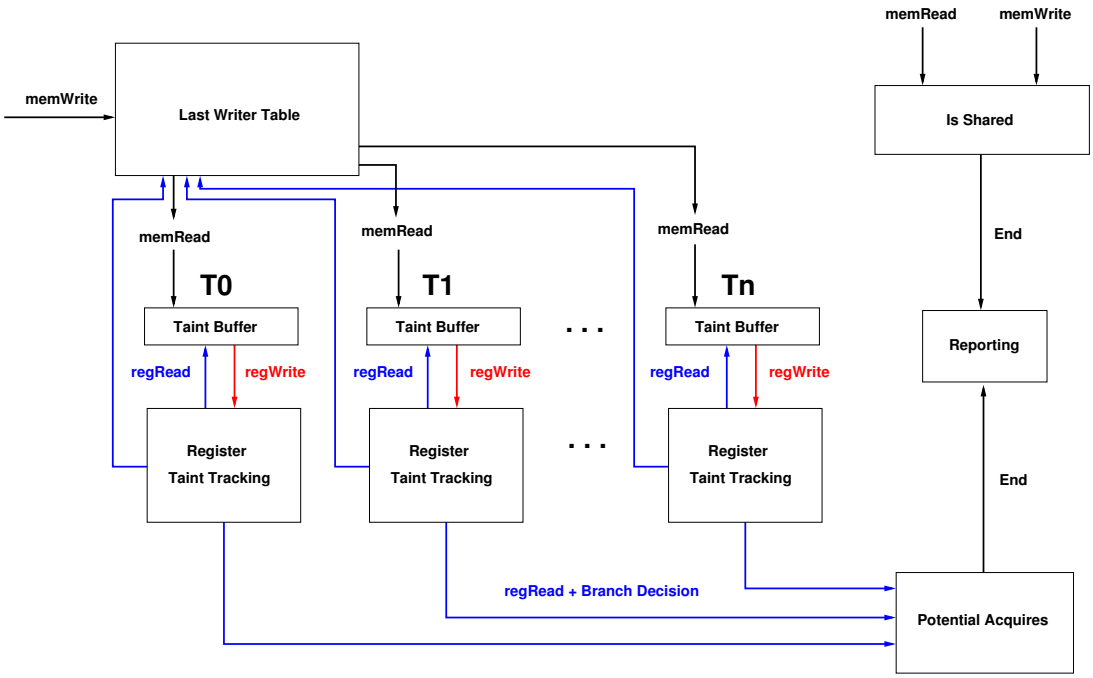

Figure 5.2: A high level overview of the operations of SyncDetect.

At the end of the execution, the potential acquire information and the table of shared memory addresses can be used to determine and report acquire and release pairs. Each read that the taint tracking has determined to match the control signature and is therefore in the set of potential acquires is sought out (by the address read) in the

shared memory table. Those that are found and therefore are shared reads are reported with their inferred releases, after duplicates have been pruned. Before discussing limitations of our current implementation, we first outline a heuristic that we apply to reduce the number of false positives.

5.2.6 Distance Limit

One method that can be employed to reduce the number of false positives recorded is to limit the number the instructions that can be executed between the load (read) and the branch computation for the read to be considered an acquire. This heuristic is based on the observation that in real world examples, acquires are normally found to be tested immediately, with the acquiring read often found within a conditional statement at source level.

An additional benefit this heuristic brings is to limit the negative impact of the over-propagation of taint information. As discussed in Section 5.3.4, the register taint tracking is currently somewhat rudimentary, not making much use of instruction specific information to prevent the propagation of taint. In early experiments we found that this could lead to situations where a taint persists for many instructions after the initial read before erroneously appearing to be used in a branching decision.

To implement this heuristic thread specific instruction counters are maintained and on a read the current value recorded. If the taint left by the read in the register file is later present in the registers used during a branching decision then the instruction count at this point is also recorded, unless this second number has already been set. That is, only the first time the taint information from a read is used in a branching decision is considered so we are recording the minimum distance seen. Finally, in the post-processing phase the difference between these numbers for each potential acquire is tested against a threshold and the read discarded if the threshold is exceeded.

A default value of 10 instructions is currently in use but this can be modified by the user by passing the following parameter at runtime $--$**distance=**$<$**UINT**$>$.

5.3 Limitations

Before presenting our results we examine the limitations of our current implementation and discuss relevant works that could inspire improvements for future releases. We focus on the issues introduced by non-deterministic program behaviour and mitigation

strategies, though also consider the issue of benign data races and the improvements that state-of-the-art taint tracking would also bring.

5.3.1 Uncontested Synchronisation

As detailed in Section 5.2.2 we determine a memory location to be shared if over the course of the execution it is accessed by more than one thread. To be considered an acquire by our tool a read that meets the control acquire signature must be a read to one of these shared memory locations. The rationale behind this is to reduce false positives as if the variable is local to a thread it, by definition, cannot be a shared memory synchronisation variable. There is however the potential for an acquire to be missed if in the execution observed, no other thread makes an access to the synchronisation variable, but in other executions the variable is used for synchronisation. We do not believe this to be a common case, however for completeness we also offer the $--$**local** knob to also report potential acquires that are not found to be shared data.

5.3.2 Non-determinism

Non-determinism poses significant issues to the dynamic analysis of shared-memory parallel programs. Specifically, possible behaviours of the program (i.e. the synchronisations) may not be seen under dynamic analysis. The two major causes of this are timing issues and conditional synchronisation.

It is worth noting that the instrumentation performed by an analysis tool can also introduce non-uniform timing dilation, where some threads are slowed more than others. This can lead to the analysis tool seeing an interleaving of accesses that, while still legal, is not that seen under normal conditions. Therefore a potentially buggy synchronisation that occurs in most real executions may be missed under analysis. Several approaches attempt to mitigate this, either through the introduction of randomised delays [BKMN10] or using hardware-based replay [MCT08, RDB99]. More complex non-deterministic programs can pose additional issues for dynamic analysis techniques if they make use of conditional synchronisation. By this we mean synchronisation that only occurs with specific inputs.

5.3.3 Benign Data Races

If we wish to support programmers attempting to migrate (by annotation) legacy code that does have so called benign data races to modern DRF language models, e.g. C11 [BOS+11, BA08] or the Java Memory Model [MPA05], we must also provide information about them. Currently, we can go partway and offer the ability to additionally report on shared accesses that are not seen as acquires or releases through use of the $--$**data** flag at runtime.

5.3.4 Taint Tracking

Our current taint tracking mechanism does not take architectural details into account, other than the fact that an *xor* instruction that uses the same register or address for both inputs results in the taint being lost. This does mean that our system is relatively general and could be ported to architectures other than x86 with relative ease, providing suitable alternative to Intel's Pin framework is available. The downside of this is that we do not currently make use of any insights into the instruction set architecture to limit taint propagation in our model of the register file, when the real registers would no longer be tainted. More complex and architecturally specific implementations of taint tracking exist [ZJS+11], and adopting such a method would be an improvement for future releases.

Additionally, coverage may also be improved by implementing taint tracking for memory addresses. Specifically, tracking which memory accesses taint which addresses. In conjunction with the register taint tracking, this would handle cases where a read of shared data is stored to a local memory location before being tested in a branching decision. However, as we have seen during the development of heuristics (see Section 5.2.6) real world examples are found to be read and used in branch conditions with minimal intervening instructions. Therefore developing this enhancement is not a priority at this time.

5.4 Results

Our results are organised as follows. We first present results from some basic synchronisations before applying our tool to some synchronisation kernels. Finally we present a case study, applying our tool to FFT from the SPLASH-2 [WOT+95] benchmark suite. For our experiments the synchronisation macros found in FFT are implemented

T1		T2	
1		1	data = 1024;
2		2	flag = 1;
3	while (flag != 1);	3	
4	local = data;	4	

Figure 5.3: Simple blocking synchronisation between two threads.

using user space spin locks.

The programs were all compiled using GCC 4.8.1 [gcc] with debugging symbols enabled and no optimisations. Currently, the DWARF debugging information provided by GCC allows us to access the file and line location in source from which an instruction originates. Column information is currently not provided and therefore SyncDetect does not attempt to report it as output. As output we present file and line numbers where acquires and inferred releases can be found. Given this setup, all of our results are based on the number of lines in the source program where acquires and releases are found.

5.4.1 Blocking Synchronisation

To initially test our system, we consider a basic ad hoc blocking synchronisation in which a flag is used to prevent one thread from accessing shared data until another has completed its accesses. The source code for this application is presented as Figure 5.3.

Using SyncDetect with all the default options, we find that acquire and release pair formed by the accesses to flag (line 3 in T1 and line 2 in T2 in Figure 5.3) are correctly identified and reported. Additionally, the accesses to the data variable are not reported as they are not synchronisation.

5.4.2 Non-blocking Synchronisation

Similarly to the previous test, we now consider a basic ad hoc non-blocking synchronisation. It is presented as Figure 5.4. Here the synchronisation read (in T1 line 4) is non-blocking and if the flag variable has not been set by T2 (line 2) by the time the read executes then thread 1 will not read the shared data. For the purposes of our experiment we send thread 1 to sleep (T1 line 3) to ensure that the synchronisation occurs during the examined execution.

T1		T2	
1		1	data = 1024;
2		2	flag = 1;
3	// Sleep	3	
4	if (flag != 1)	4	
5	{	5	
6	local = data;	6	
7	}	7	

Figure 5.4: Simple non-blocking synchronisation between two threads. A call to sleep in thread 1 is used to ensure that we see the synchronisation occur in our experiment.

Again using SyncDetect with all the default options, we find that acquire and release pair formed by the accesses to flag (line 3 in T1 and line 2 in T2 in Figure 5.4) are correctly identified and reported. Again, the accesses to the data variable are not reported as SyncDetect correctly disregards them.

5.4.3 Synchronisation Kernels

We now consider implementations of some synchronisation kernels, which represent a mixture of blocking and non-blocking synchronisations. The first is an implementation of the well known Dekker's algorithm [Dij65]. The second is an implementation of a lock-free queue (LFQ) as described by Michael and Scott [MS96]. The third is a implementation of Peterson's algorithm [Pet81]. The implementations of all these programs are drawn from or adapted from [Lin13]. Each program implements the synchronisation algorithm and some simple array computations in the critical sections. The results from applying our tool to each of these programs using the default configuration are presented as Table 5.2.

As we can see SyncDetect correctly categorises the vast majority of lines for both acquires and releases with an average of only 5.46% lines with reads and 6.86% of lines with writes being miscategorised. Looking in more detail we see that one acquire in LFQ was not reported (a false negative). It was in fact detected by SyncDetect but the distance between the reading instruction and the branch decision was larger than the default threshold. To examine this in more detail we present Figure 5.5 and Figure 5.6 showing the reporting of false negative and false positive acquires respectively as the threshold is varied.

			Dekker	LFQ	Peterson
Acquires	True	Positive	6	7	2
		Negative	26	48	26
	False	Positive	2	1	2
		Negative	0	1	0
Releases	True	Positive	10	4	6
		Negative	37	50	33
	False	Positive	4	2	4
		Negative	0	0	0

Table 5.2: A breakdown of the results of acquire detection and release inference from applying our SyncDetect tool to three programs with ad hoc synchronisation. Note that the numbers reported are based on lines in the source programs.

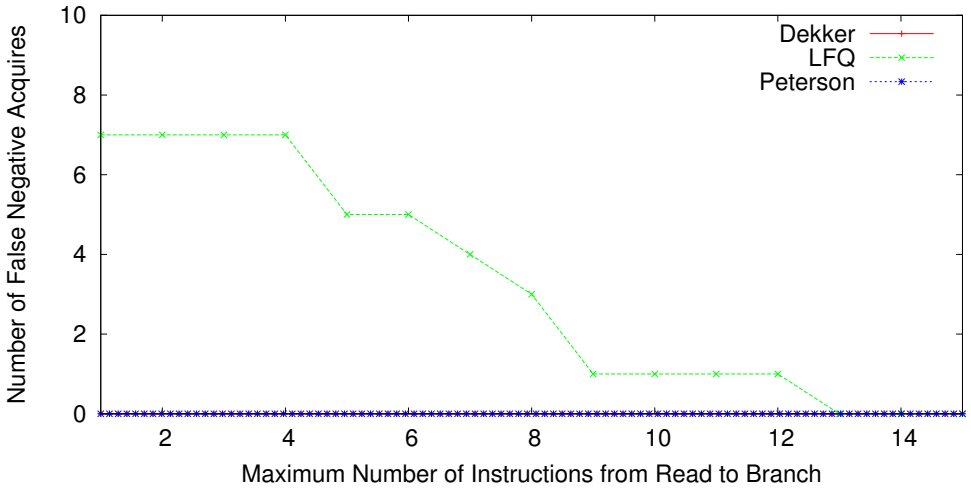

Figure 5.5: The decrease in false negative acquires seen in the synchronisation kernels as the distance threshold is increased. Note that we see no false negatives reported for Dekker or Peterson at any positive threshold value.

Looking first at Figure 5.5, we see that both Dekker and Peterson do not return false negatives at any value greater than 1 as the read is used immediately in a branch decision. Conversely, LFQ requires a larger threshold to avoid missing acquires. This is due to the structure of the synchronisation where multiple acquires are made before being tested together in a branch decision, i.e. there are multiple instructions between the acquiring read and the branch decision. To illustrate this more clearly, we present Figure 5.7. It shows an extract of source code from LFQ and the corresponding assembly level instructions. As we can see, due to the structure of the synchronisation

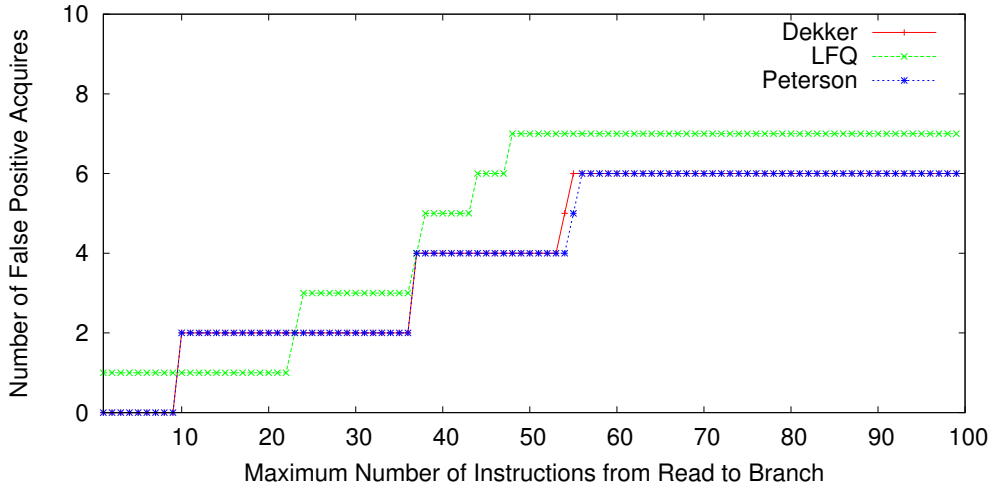

Figure 5.6: The increase in false positive acquires seen in the synchronisation kernels as the distance threshold is increased.

and the referencing of memory, multiple instructions can be present between acquiring reads and their corresponding condition.

Turning to look at the false positives, presented in Figure 5.6, we see the expected result, where the number of false positives increases as the maximum allowable distance is increased. From these figures we can see that how to exactly tune the distance heuristic depends on the target of the tool. If we are targeting synchronisation detection then using a larger maximum distance would minimise false positives. Conversely, if we target race detection then keeping false positives to a minimum by using a smaller maximum distance is preferable, if we prefer additional races being reported over races being missed.

5.4.4 FFT

FFT is a program drawn from the SPLASH-2 [WOT+95] shared-memory benchmark suite. As stated in [WOT+95], this program implements the 1 dimensional version of the radix$-n$ six-step Fast Fourier Transform (FFT) algorithm described in [Bai90].

Like the other programs from the SPLASH-2 suite, FFT is written using synchronisation macros for which the users are left to provide the implementation. For our experiments we implement each of the required macros on top of user space spin locks, which are inlined into the source code of FFT. With this program we show the application of SyncDetect to a larger program with blocking user defined synchronisations. The acquire detection results for FFT are presented as Table 5.3. The corresponding

tail = queue–>tail	movq queue(%rip), %rax
	movq 8(%rax), %rax
	movq %rax, -16(%rbp)

next = tail–>next	movq -16(%rbp), %rax
	movq (%rax), %rax
	movq %rax, -8(%rbp)

if (tail == queue–>tail)	movq queue(%rip), %rax
	movq 8(%rax), %rax
	cmpq -16(%rbp), %rax
	jne .L2

Figure 5.7: Source and assembly level instructions from LFQ. Note that there are multiple instructions before the branch decision.

results for inferred releases are presented as Table 5.4.

Acquires			
Positive		Negative	
True	False	True	False
58	30	271	0

Table 5.3: A breakdown of the acquire detection results on FFT from the SPLASH-2 suite. Note that numbers reported are lines in the source code.

As we can see SyncDetect is able to discount the majority of static source code level accesses as not being synchronisation related. Using the default settings of the tool, we find that only 9.12% of reads are misclassified and 19.56% of writes. Looking at the results in more detail we see 2 false negative releases. These occur as for each of these the release is only executed once and is the final release performed on the synchronisation variable. As our tool infers releases based on acquire detection, we do not detect those releases.

As with the synchronisation kernels, it is worth noting here that the distance heuristic is found to be significantly beneficial. We find that setting the distance limit to 1 instruction, all actual acquires are still detected with the number of false positives falling to 20. This would mean only 6.08% of reads being misclassified, and this in

Releases			
Positive		Negative	
True	False	True	False
48	51	223	2

Table 5.4: A breakdown of the inferred release results on FFT from the SPLASH-2 suite. Note that numbers reported are lines in the source code. The two false negative releases are writes never read from and therefore could not be inferred from an acquire.

conjunction with our results from the previous section indicates that this heuristic can be used rather aggressively in programs with blocking synchronisation (e.g. locks and barriers). We now examine the effectiveness of the distance limit heuristic in more detail. The results of these experiments is presented as Figure 5.8.

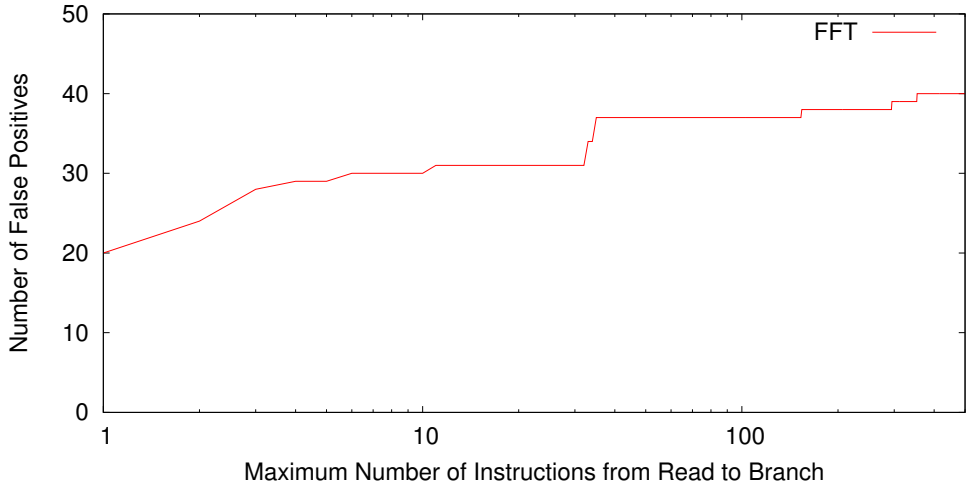

Figure 5.8: A visualisation of the effectiveness of the distance heuristic on FFT. Shown are the number of false positive acquires reported at each potential threshold. There are no false negatives found in our investigation of FFT, so no true positives are missed at any point shown. Note the log scale on the *x* axis.

As we can see, the greater the distance allowed, the greater the number of false positives are reported. False positives occur when reads of shared data are involved in control flow. In FFT these are all well-synchronised accesses protected by locks and barriers, but our analysis alone is unable to separate them as they meet the control acquire signature. Crucially, as our results in Sections 5.4.3 and 5.4.4 show only a small number of false positives are seen, due to the imposition of this heuristic. As discussed in Section 5.2.6 improvements to taint tracking may render this heuristic

redundant in future releases, but it may always be useful as a sanity check, using a larger default value.

5.5 Conclusions

Ad hoc synchronisations are considered dangerous as they can introduce bugs, cause correctness issues with more relaxed hardware memory consistency models and additionally need marking for modern language memory models. Static approaches to these problems exist but due to a lack of precision they introduce a number of false positives. We pursue a dynamic approach, which while losing conservatism, gains the precision not possible in static solutions and also is able to infer the related release. This precision is critical in applications such as data race detection where minimising the number of false positives in synchronisation detection is the key, as those false positives may lead to false negatives in race detection We presented *SyncDetect*, a proof of concept tool for detecting dynamic synchronisation based on recently proven acquire signatures. Built on top of Intel's Pin infrastructure, it dynamically instruments shared-memory parallel programs and reports ad hoc synchronisations.

In this chapter we demonstrated another application of our acquire signatures. Specifically, we developed a prototype dynamic analysis tool that exploits the control acquire signature to precisely identify ad hoc synchronisations, with few false positives. This is significant as reducing the false positives in synchronisation detection means a race detection tool would report fewer false negatives.

Chapter 6

Related Work

6.1 Analysis of Message-Passing Programs

6.1.1 Static Analysis of MPI Programs

A number of techniques have been proposed to statically analyse MPI programs, but they have limitations that prevent them being directly applied to the problem of mapping legacy message-passing programs on to modern systems as described in this thesis. Noted by multiple sources are the SPMD semantics of MPI [Bro09, KSH10, SKH06]. The effect of these semantics can be formally described as certain segments of the program only being executed by certain subsets of the processes running the program. The SPMD semantics are important as they largely define the methods that can be, and are, used to perform communication analysis.

Extending the standard Control Flow Graph (CFG) to take account of these semantics, MPI-CFG [SPS99] and later MPI-ICFG [KSH10, SKH06] annotate CFGs with process sensitive traversals and communication edges between matched send and receive statements. Collective operations are also expressed as communication edges, with their additional complexity noted. Backward slicing is performed on the pure CFG to simplify expressions that are dependent on the process *rank*, but do not directly reference it in the call parameter. *The lack of full context sensitivity prevents these works being applied in practice to the problem described in this thesis*. However, they do influence the work presented in this thesis in highlighting the need to enable a form of backward slicing to determine process sensitive values. Additionally, they illustrate the need to pursue an interprocedural approach to achieve coverage in real programs.

Again extending the CFG, Bronevetsky [Bro09] introduces the parallel CFG (pCFG). It represents the different paths taken by each process through a CFG by creating multiple states for each CFG node as determined by conditional statements. Progress is made by each component until they reach a communication statement, where they block until they can be matched to a corresponding statement. This representation allows communication to be modelled between sets, providing a scalable view of the communication characteristics. The matching process used is complex, and limited to modelling communication between sets across Cartesian topologies, e.g. across the transpose of a matrix. The mechanics of comparison are intricate, with many rewrite and composition rules being needed to prove set equality. Due to the proof requirements they define, wildcards cannot be handled [Bro09]. The tuples of the pCFG are most directly comparable with the data structure proposed in this thesis, but as detailed in Chapter 3 we dispense with the abstract nature of sets, and with matching, achieving the data representation by different means. *Most importantly, pCFG is only intraprocedural and therefore ineffective with real programs*.

There are further analyses less directly antecedent to the work presented in this thesis that are still significant. The high-level grammar based approach of Shao et al. [SJM06] introduces a technique for describing the communication as a short sequence. The resulting sequences are elegant, but lack specifics as to which processes would be involved in which branches of conditional communications. While their analysis framework may be impractical as a guide to transformation, this is largely due to its motivation as a guide to introducing network instructions [SJM06]. This *Compiled Communication* [KYL03, YMG03], is also pursued by Yuan et al. [YMG03], analysing *logical* communication, to define which data structures must be transferred. Then physical communication, the actual data transfer, is considered to determine what network instructions to insert. *Their analysis is however quite restrictive, requiring the programmer to specify data distribution*.

6.1.2 Profiling and Dynamic Analysis of MPI Programs

Profiling and dynamic analysis techniques have also been applied to MPI programs [MW03, NAW+96, SSM06, VM01]. Targeting the same optimisation as the work presented in this thesis, MPIPP [CCH+06] uses the communication graph, extracted from a profiling run, to optimise process placement. *This profiling approach, while effective, would compare unfavourably to a static approach that achieves similar coverage, given the*

cost and inconvenience of repeated executions on potentially scarce resources.

Recognising the burden of profiling, FACT [ZSH+09] seeks to understand communication patterns by only profiling the execution of a program slice. This slice having been determined by static analysis. *While reducing the cost of a profile run, the authors of FACT note that the slicing may alter the communication pattern in non-deterministic applications*.

Dynamic approaches include Adaptive MPI [HLK03, HZKK06], which provides a runtime system, capable of automatic communication/computation overlap, load balancing, and process migration. These techniques allow it to take advantage of communication phases in the program. *Given the cost of migration and need for a runtime system, the methods described are required to overcome further overhead to achieve better speedup.* For programs that lack distinct temporal phases of communication, this may not be possible.

6.1.3 Process Placement

The use of static information to determine better process placement for parallel programs is well known [SH86]. Several works recognise the ability of better process placement to improve performance and resource usage on modern parallel architectures [DRR08, JM10, MJ11, ZSH+09, ZZCZ09]. There are several applicable approaches, the choice of which is largely determined by how much knowledge about the target machine is available and the acceptable algorithmic complexity of the mapping algorithm.

Two notable approaches (as used in [CCH+06]) are graph partitioning and graph mapping. In graph partitioning, the vertices (processes) of the communication graph are assigned to buckets (CMPs), in such a manner as to minimise the total weight of non-local edges. In graph mapping, a resource graph of the target machine is also taken. The communication graph is then superimposed onto this in such a manner as to minimise the total latencies to which communication is subjected. Graph mapping is shown by [CCH+06] to outperform graph partitioning, however it requires a fuller description of the target machine and the ability to schedule at will on the nodes, something not found in all shared access machines.

6.2 Shared-Memory Correctness

6.2.1 Programmer-centric memory models

Adve and Hill [AH90b] and Gharachorloo [GLL+90] were the first to propose programmer centric memory consistency models, where the system enforces SC as long as the programmer writes data-race-free (DRF) programs and provides information about synchronisation operations. Indeed Adve's DRF based models [Adv93] and Gharachorloo's PL based models [Gha95] are the precursors to the memory consistency models adopted by languages such as C [BA08] and Java [MPA05]. The main difference between the above works and that we present in Chapter 4 is that, while they assume programmer-annotated synchronisation labels, we assume unlabelled DRF programs.

6.2.2 Delay-set analysis

Shasha and Snir [SS88] were the first to consider the problem of computing the minimum number of memory orderings (delays) to ensure that a concurrent shared memory program satisfies SC. In the shared-memory work presented in this thesis, we focus on how the above orderings can be pruned if the shared memory program is a DRF (but unlabelled) program. To put it succinctly, we do Delay-set analysis for unlabelled DRF programs.

A more recent work [AKNP14] attempts to address the scalability issues inherent in Delay-set analysis by examining an over-approximation of the critical cycles. It is however limited in failing to handle recursion and dynamic thread creation, the latter of which is common in the programs examined in our evaluation. Specifically, this tool does not handle *pthread_create* calls in loops that could not be statically unrolled. We note however, that our signatures would be equally applicable to [AKNP14] and our choice to build on top of Pensieve is due to its lack of the limitations described above.

6.2.3 Fence minimisation

There have been a number of works [FLM03, KSY05, WSP+02] which focus on computing the minimal number of fences for satisfying the orderings given by Delay-set analysis. These works are orthogonal to our work, as these can very well be applied for satisfying the pruned orderings given by our analysis in Chapter 4.

6.2.4 Synchronisation detection

Our work on shared-memory is related to prior work [TNGT08a, TNGT09b, XPZ+10] on busy-wait synchronisation detection. Tian et al. [TNGT08a, TNGT09b] proposed a dynamic analysis technique for identifying user-defined busy-wait synchronisations. Since the above work uses dynamic analysis, they suffer from false negatives – in other words, some synchronisations can be missed. Subsequently, Xiong et al. [XPZ+10] showed how synchronisations can be identified using static analysis, so that there can be no false negatives. Our work differs from the above in one important aspect. The above analysis is only applicable for busy-wait synchronisation; thus it will miss identifying acquires used in non-blocking algorithms such as those used in our evaluation. It is worth noting that missing such acquires leads to correctness issues in our context which explains why the above detectors cannot be used in the context of our work. Indeed, one of the nice side-effects of our work is that to the best of our knowledge, ours is the first general acquire detector.

6.2.5 Hardware based memory ordering

There have been a number of recent works [BMW09, GGH91, GFV99, LNG10, SNM+12] that have proposed techniques for efficiently enforcing memory ordering. In contrast with the above works each of which involve hardware support, we do not use any hardware support in this thesis. Furthermore, each of the above works are orthogonal to us, in that, they can very well be used to efficiently enforce the pruned orderings given by our work in Chapter 4.

6.2.6 SC-preserving compilers

Ahn et al. [AQN+09] proposed the Bulk compiler which together with Bulk hardware (which enforces hardware SC at chunk level) guarantees SC at the language level. In other words, the Bulk compiler preserves SC by ensuring that it does not reorder memory operations across chunks. More recently, Marino et al. [MSM+11] proposed the SC-preserving compiler which together with SC hardware (which enforces SC at the hardware level) guarantees SC at the language level. Their main result is that it is possible for the compiler to preserve SC without significant slowdown ($<5\%$ on average across a suite of parallel programs). On the other hand, they assume that the hardware cannot reorder operations, i.e. they assume that the hardware enforces SC.

In contrast, our work considers the problem of how to enforce SC on hardware that could reorder memory operations. Of course, to preserve SC at the language level we would need a compiler that preserves SC (i.e. the above works). Recall that in our implementation we ensure that the compiler cannot reorder shared memory operations by inserting an empty memory-clobbering assembly instruction between such operations, which LLVM interprets as a compiler fence. It is worth noting that this corresponds to the naive-SC variant [MSM+11]. We could have very well used the SC-preserving compiler proposed (with all optimisations), which could potentially translate into better performance. In this respect, our work in shared-memory orthogonal to the above works.

6.2.7 Dynamic Scheduling

There have several works focused on manipulating thread schedules to discover concurrency bugs. These involve randomly delaying the execution of threads in combination with various heuristics [FNU03] to ensure schedules where concurrency bugs in the program are forced to manifest [EFG+03, Sen07]. More recent work in this area has produced similar methods that are able to quantify the probability that their analysis has failed to discover a concurrency bug [BKMN10]. These techniques appear to be immediately compatible with our SyncDetect tool and will inform future releases.

6.2.8 Dynamic Race Detectors

Significant work has also been done in the field of race detection, using both online and offline methods. Online methods include Eraser [SBN+97], FastTrack [FF09] and RaceTrack [YRC05]. It should be noted that RaceTrack like some previous methods targeting managed lanuages [CDB01, Nis04] exploits the existence of a JIT compiler to perform its analysis. Online methods, particularly instrumentation techniques like Eraser are similar in operation to our SyncDetect and may very well inform future releases. The other methods listed above that perform their analysis during JIT compilation are more closely related to the static work we presented in Chapter 4.

So-called offline methods have also been developed. These generally build on record and replay technology to store an execution and then perform analysis during some form of replay. These replay systems, including DeLorean [MCT08] and iDNA [BCdJ+06] use hardware support to record the interleaving of memory interactions, which can later be replayed deterministically. Race detectors that use such

record and replay techniques include those by Narayanasamy et al. [NWT+07]. Other methods that do offline analysis such as the earlier RaceFrontier [CM91] and RecPlay [RDB99] are limited, because as noted by Narayanasamy et al. they do not record non-deterministic interactions

Our work is relevant to these tools as it can be used to improve the classification of apparent data races. As Narayanasamy et al. [NWT+07] note, user defined synchronisation, while determined as benign by their tool, is difficult to definitively identify. As iDNA, the replay technology they build on doesn't provide support, their tool incorrectly identifies a race between two correctly formed user defined synchronisation operations.

Chapter 7

Conclusions and Future Work

7.1 Summary of Contributions

Achieving high performance and correctness with parallel programming is notoriously challenging. Unfortunately, even existing (legacy) programs can fail to achieve both of these due to changes in the architectures of modern computers. In message-passing the problem is a performance one, with the message-passing library providing communication guarantees. In shared-memory it is a correctness issue, which existing automated solutions solve but only by introducing performance issues.

Changes in machine topology, with the ubiquity of CMPs, requires legacy message-passing programs to be actively spatially scheduled to ensure efficient operation as they were written without the expectation of such an inherently tiered topology.

Memory consistency models also pose issues to shared-memory programs. As modern CMPs only support relaxed memory models, programs written assuming SC need support (e.g. memory fences) to ensure correct operation on the more relaxed machine.

- Firstly, in Chapter 3, we presented a method for statically determining the point-to-point communication graph of an MPI program. The immediate application of this analysis framework, as demonstrated in this thesis, is to improve spatial scheduling. This allows existing (legacy) MPI programs not written with knowledge of modern cluster topologies to make efficient use of them. We use the framework to colocate heavily communicating processes, but other cost models are immediately applicable.

- Secondly, in Chapter 4, we determined and proved necessary conditions for a

read to be an acquire in a well-synchronised (legacy DRF) program. The two signatures developed have the potential to improve solutions in a number of application areas, as demonstrated in the this thesis.

- Thirdly, also in Chapter 4, we showed an application of our previous contribution in improving automatic fence placement. The use of our acquire signatures to prune the orderings determined by Delay-set analysis (or its conservative approximation) for legacy DRF programs allows us to significantly reduce the orderings considered by fence minimisation. This allows fewer fences to be placed allowing for increased performance while still maintaining correctness.

- Finally, in Chapter 5, we present SyncDetect, a proof-of-concept tool for dynamically detecting ad hoc synchronisation. This demonstrates another application of our acquire signatures, in assisting programmers in debugging and manually porting code to modern language memory consistency models.

7.2 Future Work

While the contributions presented in this thesis are free-standing, there are extensions and further applications for both the message-passing and shared-memory works. Firstly, there are a number of improvements that can be made to the MPI analysis, both to increase the scope and to enable other applications. Secondly, our acquire signatures can be used to inform other tools and the tools presented in this thesis themselves improved.

7.2.1 Static Approximation of MPI Communication Graphs

Extensions to this work include further instantiations of the framework including optimisations targeting communication/computation overlap. Development of a general solution to automatically dealing with incompleteness would also be a significant result. Additionally, refinement of the current analysis is possible, e.g. adding the handling of non-uniform collective operations.These operations can heavily contribute to a communication graph and their inclusion would extend the scope to further classes of programs.

7.2.2 Shared-memory Correctness and Performance

Having identified, proven and shown the significance of our acquire signatures as they apply to Delay-set analysis, future work in this area would see them applied to see them used in automatic annotation or other analysis tools. We did also conduct some investigations into whether provable signatures for conservative identification of releases were possible. After this work we are reasonably convinced that no such provable signatures are possible, however heuristic-based approaches may be reasonable for semi-automatic (user-guided) applications.

Extensions to our *SyncDetect* tool for the dynamic detection of synchronisation also include the further application of heuristics. As the current approach cannot be conservative, it may be possible to further improve the results with heuristics or some of the techniques discussed both in Chapter 5 and Chapter 6.

From an engineering standpoint it would be beneficial to implement more architecturally specific register taint tracking. Additionally, it would interesting to examine any additional benefit that could be brought with a hybrid tool, combining the conservatism of static analysis with the precision of a dynamic approach. Given the lack of pure address acquires seen in practice, implementing address acquire detection is not a development priority for future releases of this tool.

Bibliography

[Adv93] Sarita Vikram Adve. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, University of Wisconsin at Madison, Madison, WI, USA, 1993.

[AG95] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[AH90a] Sarita Adve and Mark D. Hill. Weak ordering - a new definition. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, 1990.

[AH90b] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *ISCA*, pages 2–14, 1990.

[AKNP14] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV*, pages 508–524, 2014.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*, pages 906–908. Pearson, 2006.

[AQN+09] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S.Midkiff, and David Wong. BulkCompiler: High-performance sequential consistency through cooperative compiler and hardware support. *IEEE Micro*, pages 133–144, 2009.

[ARM14] ARM Limited. *ARM®v7-M Architecture Reference Manual (Issue E.b)*. 2014.

[ASLK06] T. Agarwal, Amit Sharma, A. Laxmikant, and Laxmikant V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *IPDPS*, page 145, 2006.

[BA08] Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, pages 68–78, 2008.

[Bai90] David H. Bailey. FFTs in external or hierarchical memory. *J. Supercomput.*, 4(1):23–35, March 1990.

[BBB+91] David H. Bailey, Eric Barszcz, John T. Barton, D. S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, T. A. Lasinski, Robert Schreiber, Horst D. Simon, V. Venkatakrishnan, and Sisira Weeratunga. The NAS parallel benchmarks. *IJHPCA*, 5(3):63–73, 1991.

[BC05] David A. Bader and Guojing Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *J. Parallel Distrib. Comput.*, 65(9):994–1006, 2005.

[BCdJ+06] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, pages 154–163, New York, NY, USA, 2006. ACM.

[BKMN10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, pages 167–178, 2010.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*, pages 72–81, 2008.

[BMW09] Colin Blundell, Milo M. K. Martin, and Thomas F. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *ISCA*, pages 233–244, 2009.

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.

[Bro09] Greg Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *CGO*, pages 1–12, 2009.

[BYA89] Laxmi N. Bhuyan, Qing Yang, and Dharma P. Agrawal. Performance of multiprocessor interconnection networks. *Computer*, 22(2):25–37, February 1989.

[CBM+08] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, October 2008.

[CCH+06] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *ICS*, pages 353–360, 2006.

[CDB01] Mark Christiaens and Koen De Bosschere. TRaDe, a topological approach to on-the-fly race detection in java programs. In *JVM*, page 15, Berkeley, CA, USA, 2001. USENIX Association.

[CGS10] Franck Cappello, Amina Guermouche, and Marc Snir. On communication determinism in parallel HPC applications. In *ICCCN*, pages 1–8, 2010.

[CH95] Paul R. Carini and Michael Hind. Flow-sensitive interprocedural constant propagation. In *PLDI*, pages 23–31, 1995.

[CL05] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA*, pages 21–28, New York, NY, USA, 2005. ACM.

[CM69] Daniel Chazan and Willard Miranker. Chaotic relaxation. *Linear algebra and its applications*, 2(2):199–222, 1969.

[CM91] Jong-Deok Choi and Sang Lyul Min. Race frontier: Reproducing data races in parallel-program debugging. In *PPoPP*, pages 145–154, New York, NY, USA, 1991. ACM.

[Cra94] Travis Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report 93-02-02, University of Washington, Seattle, Washington, 1994.

[CTMT07] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. *SIGARCH Comput. Archit. News*, 35(2):278–289, 2007.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *POPL*, pages 37–48, 1995.

[DGT13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48, 2013.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[DMT13] Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. Weefence: toward making fences free in tso. In *ISCA*, pages 213–224, 2013.

[DPS07] Anthony Danalis, Lori L. Pollock, and D. Martin Swany. Automatic MPI application transformation with ASPhALT. In *IPDPS*, pages 1–8, 2007.

[DPSC09] Anthony Danalis, Lori L. Pollock, D. Martin Swany, and John Cavazos. MPI-aware compiler optimizations for improving communication-computation overlap. In *ICS*, pages 316–325, 2009.

[DRR08] Jörg Dümmler, Thomas Rauber, and Gudula Rünger. Mapping algorithms for multiprocessor tasks on multi-core clusters. In *ICPP*, pages 141–148, 2008.

[ECD] ECDF: The Edinburgh Compute and Data Facility. Eddie Linux Compute Cluster. *http://www.ecdf.ed.ac.uk*.

[EFG+03] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *CCPE*, 15(3-5):485–499, 2003.

[FF09] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133, New York, NY, USA, 2009. ACM.

[FLM03] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS*, pages 285–294, 2003.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[FNU03] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *IPDPS*, page 286.2. IEEE, 2003.

[FS00] Andreas Frommer and Daniel B Szyld. On asynchronous iterations. *Journal of computational and applied mathematics*, 123(1):201–216, 2000.

[FY02] Ahmad Faraj and Xin Yuan. Communication characteristics in the NAS parallel benchmarks. In *IASTED PDCS*, pages 724–729, 2002.

[gcc] GCC: GNU compiler collection. *http://gcc.gnu.org*.

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *PVM/MPI*, pages 97–104, 2004.

[GFV99] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is sc + ilp=rc? In *ISCA*, pages 162–171, 1999.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. Two techniques to enhance the performance of memory consistency models. In *ICPP (1)*, pages 355–364, 1991.

[GGH92] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *ISCA*, pages 22–33, New York, NY, USA, 1992. ACM.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA*, pages 15–26, 1990.

[GS94] Rajiv Gupta and Mary Lou Soffa. A framework for partial data flow analysis. In *ICSM*, pages 4–13, 1994.

[GT93] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementations. In *PLDI*, pages 90–99, 1993.

[HDH+10] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *ISSCC*, pages 108–109, Feb 2010.

[Hew15] Hewlett-Packard Corporation. HP Z840 Workstation QuickSpecs Version 6. February 2015.

[Hil98] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *Computer*, 31(8):28–34, 1998.

[HLK03] Chao Huang, Orion Sky Lawlor, and Laxmikant V. Kalé. Adaptive MPI. In *LCPC*, pages 306–322, 2003.

[HMCCR93] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and René G. Rodríguez. FIAT: A framework for interprocedural analysis and transfomation. In *LCPC*, pages 522–545, 1993.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*, chapter 5, pages 346–348. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[HZKK06] Chao Huang, Gengbin Zheng, Laxmikant V. Kalé, and Sameer Kumar. Performance evaluation of adaptive MPI. In *PPOPP*, pages 12–21, 2006.

[Int09] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. December 2009.

[Int14] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. September 2014.

[JM10] Emmanuel Jeannot and Guillaume Mercier. Near-optimal placement of MPI processes on hierarchical NUMA architectures. In *Euro-Par (2)*, pages 199–210, 2010.

[KL70] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[KSH10] Barbara Kreaseck, Michelle Mills Strout, and Paul Hovland. Depth analysis of MPI programs. In *AMP*, 2010.

[KSY05] Amir Kamil, Jimmy Su, and Katherine Yelick. Making sequential consistency practical in titanium. In *SC*, page 15, Washington, DC, USA, 2005. IEEE.

[KYL03] Amit Karwande, Xin Yuan, and David K. Lowenthal. CC-MPI: a compiled communication capable MPI prototype for ethernet switched clusters. In *PPOPP*, pages 95–106, 2003.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, January 1987.

[LCM+05] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[Lin13] Changhui Lin. *Imposing Minimal Memory Ordering on Multiprocessors*. PhD thesis, University of California, Riverside, 2013.

[LL97] James Laudon and Daniel Lenoski. The sgi origin: A ccnuma highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 241–251, New York, NY, USA, 1997. ACM.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, March 1992.

[LM92] T.J. LeBlanc and E.P. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on*, pages 254–263, Dec 1992.

[LNG10] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. Efficient sequential consistency using conditional fences. In *PACT*, pages 295–306. ACM, 2010.

[LP01] Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput.*, 50(8):824–833, 2001.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[MCT08] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently. In *ISCA*, pages 289–300, June 2008.

[mF] MPI Forum. MPI standard 2.0. *http://www.mcs.anl.gov/mpi/*.

[MJ11] Guillaume Mercier and Emmanuel Jeannot. Improving MPI applications performance on multicore clusters with rank reordering. In *EuroMPI*, pages 39–49, 2011.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL*, pages 378–391, New York, NY, USA, 2005. ACM.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[MSM+11] Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A case for an SC-preserving compiler. In *PLDI*, pages 199–210, 2011.

[MW03] Bernd Mohr and Felix Wolf. KOJAK - a tool set for automatic performance analysis of parallel programs. In *Euro-Par*, pages 1301–1304, 2003.

[NAW+96] W. E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12:69–80, 1996.

[Nis04] Hiroyasu Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *VM*, volume 3, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[NMT10] Adrian Nistor, Darko Marinov, and Josep Torrellas. Instantcheck: Checking the determinism of parallel programs using on-the-fly incremental hashing. In *MICRO*, pages 251–262, 2010.

[NWT+07] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, pages 22–31, New York, NY, USA, 2007. ACM.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[PSK+08] Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. Using MPI communication patterns to guide source code transformations. In *ICCS*, pages 253–260, 2008.

[RDB99] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[Sen07] Koushik Sen. Effective random testing of concurrent programs. In *ASE*, pages 323–332. ACM, 2007.

[SFW+05] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent java programs. In *PPoPP*, pages 2–13, New York, NY, USA, 2005. ACM.

[SH86] Vivek Sarkar and John L. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *CC*, pages 17–26, 1986.

[SJM06] Shuyi Shao, Alex K. Jones, and Rami G. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *IPDPS*, 2006.

[SKH06] Michelle Mills Strout, Barbara Kreaseck, and Paul D. Hovland. Data-flow analysis for MPI programs. In *ICPP*, pages 175–184, 2006.

[SNM+12] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd D. Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. In *ISCA*, pages 524–535, 2012.

[SPS99] Dale R. Shires, Lori L. Pollock, and Sara Sprenkle. Program flow graph construction for static analysis of MPI programs. In *PDPTA*, pages 1847–1853, 1999.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *PLDI*, pages 175–186, New York, NY, USA, 2011. ACM.

[SSM06] Shende Sameer S and Allen D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[Szy88] B. K. Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In *ICS*, pages 621–626, New York, NY, USA, 1988. ACM.

[TNGT08a] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*, pages 143–154, 2008.

[TNGT08b] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*, pages 143–154, 2008.

[TNGT09a] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Automated dynamic detection of busy-wait synchronizations. *Softw., Pract. Exper.*, 39(11):947–972, 2009.

[TNGT09b] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Automated dynamic detection of busy-wait synchronizations. *Softw., Pract. Exper.*, 39(11):947–972, 2009.

[VM01] Jeffrey S. Vetter and Michael O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *PPOPP*, pages 123–132, 2001.

[WMADC99] Frederick C. Wong, Richard P. Martin, Remzi H. Arpaci-Dusseau, and David E. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *SC*, 1999.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[WSP+02] Chi-Leung Wong, Zehra Sura, David A. Padua, Xing Fang, Jaejin Lee, and Samuel P. Midkiff. The pensieve project: A compiler infrastructure for memory models. In *ISPAN*, pages 239–244, 2002.

[XLW+09] Ruini Xue, Xuezheng Liu, Ming Wu, Zhenyu Guo, Wenguang Chen, Weimin Zheng, Zheng Zhang, and Geoffrey M. Voelker. MPIWiz: subgroup reproducible replay of MPI applications. In *PPOPP*, pages 251–260, 2009.

[XPZ+10] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *OSDI*, pages 163–176, 2010.

[Yea96] K.C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, Apr 1996.

[YMG03] Xin Yuan, Rami G. Melhem, and Rajiv Gupta. Algorithms for supporting compiled communication. *IEEE Trans. Parallel Distrib. Syst.*, 14(2):107–118, 2003.

[YRC05] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, October 2005.

[ZJS+11] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, February 2011.

[ZSH+09] Jidong Zhai, Tianwei Sheng, Jiangzhou He, Wenguang Chen, and Weimin Zheng. FACT: fast communication trace collection for parallel applications through program slicing. In *SC*, 2009.

[ZZCZ09] Jin Zhang, Jidong Zhai, Wenguang Chen, and Weimin Zheng. Process mapping for MPI collective communications. In *Euro-Par*, pages 81–92, 2009.