Detecting and Quantifying Resource Contention in Concurrent Programs

by

Dirk Willem Venter

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science in the Faculty of Science at Stellenbosch University

Computer Science Division,
Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Dr. Cornelia P. Inggs

2016

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: March 2016

Abstract

Detecting and Quantifying Resource Contention in Concurrent Programs

D.W. Venter

Computer Science Division,
Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MSc Computer Science

2016

Parallel programs, both shared-memory and message-passing programs, typically require the sharing of resources. For example, software resources, such as shared mutual exclusion locks and hardware resources, such as caches and memory. Shared resources can only be used by one thread or process at a time. The competition for limited resources is called resource contention. The result of resource contention is delays while waiting for access to a resource and/or extra computational overhead to resolve the request for a resource. Thus, the performance of the program can be improved by identifying and reducing contention for shared resources. This study investigates the effect of individual types of contention for hardware and software resources in detail and discusses the three tools that were developed to identify and quantify the sources of contention in concurrent programs.

Dedication

To my father, who never understood why it is necessary for computers to be so complicated.

Acknowledgements

I'd like to thank my supervisor, Dr. Cornelia Inggs, for her good advice and assistance with getting my ideas to look good in print. To my proof-readers, thank you for reading my thesis and giving feedback. In particular Mark Chimes, who gave good suggestions and assisted with figures and the appearance of formulas.

Thank you to Stellenbosch University for the use of their computer equipment. There are several challenges when results can be affected by the computer that is used to generate the data. My thesis would not have been the same without it. In fact, doing research became much easier after obtaining access to the server.

To the people ensuring that the lab had a good and working coffee machine, you are life savers.

To Franklin, my guide dog, for walking every step of this journey with me.

And finally, I'd like to thank my friends, family and especially the 'wolf pack' for their constant support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel programs, both shared-memory and message-passing programs, typically require the sharing of resources; for example, hardware resources, such as caches and memory or software resources, such as shared mutual exclusion locks or data that need to be distributed to other processes for them to function correctly. Shared resources can only be used by one thread/process at a time. When a thread of a shared-memory program or a process of a message-passing program requests a shared resource that is busy, execution of the thread/process stalls until the resource becomes available. The competition for limited resources is called resource contention. The result of resource contention is delays while waiting for access to a resource and/or extra computational overhead to resolve each request for a resource. Thus, the performance of the program can be improved by identifying and reducing contention for shared resources.

Each shared hardware resource has particular characteristics that govern how it is shared among running programs. The shared last-level cache has a limited amount of space for data, while the memory controller can only serve requests in the service buffers. The rate at which requests are served also depends on the location of the data in memory and how busy the data channels of the memory are. Instructions can only be executed if the instruction has been decoded, the operands it requires are available, and there are hardware execution units available to do the computation.

Contention for hardware resources results in an increase in work cycles, the sum of memory stall cycles and CPU cycles. Each time a request for data are sent to main memory the executing thread has to wait until the data are retrieved. This occurs for all running threads. When there is memory contention, requests may take longer to be served due to other sources of delay such as row buffer misses, a lack of space in buffers or a lack of available memory bandwidth. This is discussed in detail in Section 2.1.1. Modern Intel CPU cores, as found in the system used to perform the tests, have multiple execution units which can be concurrently active during the same cycle, but this can only happen if the operands required by the instruction are available. If the operands for any instruction are not available, that instruction cannot complete, but the instruction still uses space in the reordering buffer. Longer delays mean that instructions in the instruction stream take longer to be executed and more work cycles from the CPU cores (CPU cycles during which any execution unit of a CPU core is busy doing work or stall cycles during which data is retrieved) are required. Contention for space in the last-level cache increases the total number of requests sent to the main memory, resulting in more stall cycles while data is retrieved.

When a thread requests ownership of a mutex that is owned by another thread, execution of the requesting thread is blocked until the thread currently owning the mutex releases ownership of the mutex. Access to other software resources are restricted in similar ways or under certain conditions. Figure 1.1 illustrates how a higher demand for a shared resource (in this case, a mutual exclusion lock that guards a critical section) results in contention for the resource and thus more idle time. In this figure a solid blue line represents a period where a thread is active and executing instructions outside a critical section, a solid red line represents a period where a thread is active and executing instructions inside a critical section guarded by a mutual exclusion lock, and a gap before or after a solid section represents a period where a thread is inactive. See Section 2.1.2 for a detailed description of software contention.

CPUs continue to increase in computational power. A trend over the past ten years is for the number of cores to increase rather than an increase in clock speed. To make use of the full power of CPUs that have multiple CPU cores, a program must have at least as many threads as the number of cores in the CPU package. However, using multiple threads may cause contention for the shared resources

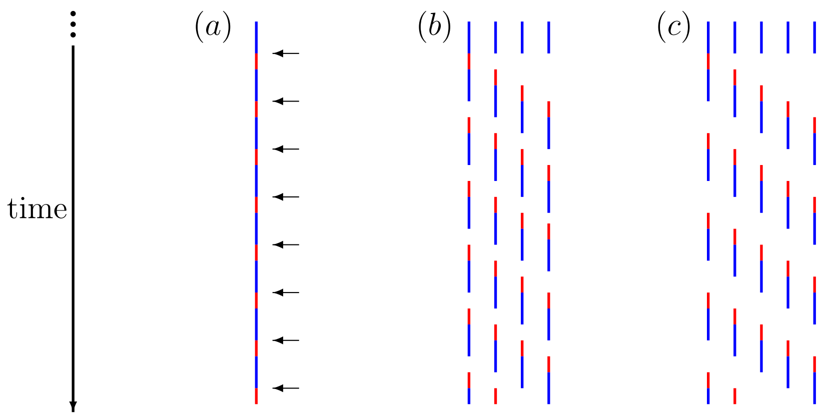

Figure 1.1: Figure (a) represents a thread that alternates between executing outside (blue line) and inside (red line) a particular critical section guarded by a mutual exclusion lock, where the execution outside the critical section is twice as long as the execution inside the critical section. Figure (b) shows that with four threads like the one in (a) accessing the same critical section, every time a thread requests a lock it has to wait (in an idle state) for at least one other thread to release the lock before it can enter the critical section. Figure (c) shows that with five threads like the one in (a) accessing the same critical section, every time a thread requests a lock it has to wait (in an idle state) for at least two other threads (except for thread two at its first request) to release the lock before it can enter the critical section.

and overhead (in the form of idle time and/or extra computation) to resolve this contention. Furthermore, an increase in the total number of concurrent requests for a contended resource causes a higher total amount of overhead among all competing threads.

This study investigates and quantifies the effect of individual types of contention for hardware and software resources in detail. Several tools exist that allow the study of individual sources of contention. Scalasca can be used to find wait states in message-passing programs [22]. Tools such as Intel ParallelZ and Tao provide a profile of the hardware utilisation which can be used to study the demand for hardware resources.

While the study of individual sources of contention is important, it is also important to study multiple sources of contention simultaneously. Contention for specific resources have additional effects on how other resources are used while contention is occuring. When a thread is frequently idle due to software contention, fewer CPU and memory bandwidth resources will be used while the thread is idle. Conversely when there is contention for hardware resources it could take longer to execute code in critical sections and will therefore increase the contention for software resources.

A set of tools was thus created to quantify the effects of contention for shared resources. Using ideas proposed by Chen and Stenstrom, and Geimer et al., we can report the time that the program spent idle and the specific type of contention, the location in source code, and or the resource (where applicable), such as the lock, for which there is contention [5, 9]. Contention on the critical path of execution is distinguished from contention that does not lie on the critical path of execution as proposed by Chen and Stenstrom [5]. A technique proposed by Geimer et al. is used to find wait states in message-passing programs [9]. Using a model proposed by Tudor and Teo, speed-up loss due to memory contention and speed-up loss due to other dependencies are quantified [17].

Each tool have been separated to operate in multiple stages. Each stage consists of a single component for each tool. Resource usage data is gathered in the first stage called the data gathering stage. Components used in the data gathering stage were optimised to use as few resources as possible to avoid affecting the performance of the target program. The next stage, called the analysis stage, takes the output from the first stage as input, generates a profile by analysing the data, and writes all the values to a file. Output from the analysis stage can then be compared or a table can be created using tools that take the output from the components used in the analysis stage as input.

Using the data gathering components does not change the layout in memory of the target program. This is important because it might have changed how resources are used by the target program. Two of the three components, the General Metrics Gathering tool (GMG) and message-passing data gathering tool do not require recompilation of the target program. The GMG tool gathers data by querying hardware registers and operating system data structures. In the message-passing data gathering tool, calls to message-passing functions are substituted with calls to wrapper functions that perform the original message-passing function and record data about the operation. In the mutex data gathering tool, calls to mutex functions are replaced with calls to wrapper functions by including a header and recompiling the target program while linking with the library that contains the mutex wrapper functions.

Chapter 2 provides background about the shared resources. It also contains a section describing the related work. Chapter 3 describes formulas, models, and techniques used during the analysis of resource contention, and the data gathering and analysis tools are described. Chapter 4 describes the testing methodology and contains results. Chapter 5 summarises our findings.

Chapter 2

Background

Programs require several shared hardware resources to run: the main resources are memory and caches that are used to store a program's data and instructions and CPU cores to execute a program's instructions. Each CPU core can do a limited amount of computation per time unit. For this reason work is spread over multiple CPU cores to reduce the time it takes to run a program. A computer system has a limited number of CPU cores and caches and a limited amount of main memory and bandwidth between the respective components. The threads or processes of a running program have to share the hardware resources among the threads of the same program as well as with those of the Operating System and in many cases also other programs.

The threads or processes of a program also share software resources such as data or message channels. Some software resources can only be accessed by one thread or process at a time or can only be accessed once another part of the program completes. When the demand for resources is higher than the available resources, there is contention for these resources.

In this chapter we describe what data can be gathered about resource usage and how we can use it to identify contention for particular resources. The type of resources studied are classified as either hardware or software resources. Descriptions of the hardware resources studied are provided in Section 2.1.1 and descriptions of the software resources studied are provided in Section 2.1.2.

2.1 Contention for Shared Resources

The live threads/processes of a concurrent program, can be doing useful work, waiting for software resources, or contending for hardware resources. A detailed description of how the execution time is classified is provided in Chapter 3. However, it is useful to note here that not all stalls are due to contention. Stalls are already present when the program runs with a single thread on a single core. For example, when an instruction has to be executed, the operands have to be retrieved from the main memory and loaded into the caches and CPU registers. While this is happening all computation for that thread stalls until the data is available, even, for example, if it is the only thread of the only program running on the system. Causes of stalls not due to contention, are, for example, branch mispredictions and pipeline hazards. Therefore, stalls not due to contention are considered part of the "useful" (non contentious) work time of an algorithm.

However, when there is contention for resources among threads it affects the execution of the concurrent program. For example, requests for data take longer to serve when there is contention for memory, the number of requests to main memory increases when there is cache contention, and threads are blocked longer or more frequently when there is contention for software resources, such as synchronisation primitives.

2.1.1 Contention for Hardware Resources

The main hardware resources that are shared among all running programs are: the CPU cores, the last-level cache, and the main memory. Each of them will be described separately in the following subsections.

2.1.1.1 CPU core

All threads of the programs running on a computer require service from a CPU core and all threads should receive fair service from the available cores. To ensure fair service the operating system assigns threads to be run on CPU cores according to a scheduling policy. If there are more threads than CPU cores, it is possible that a thread could be ready to run, but waiting in the ready queue for a CPU core to become available.

The scheduling of program threads on a limited number of CPU cores has been studied widely in the context of Operating Systems and falls outside the scope of this study.

2.1.1.2 Cache

Cache contention occurs when requests for data by one thread or process evicts data of another thread or process from a shared cache earlier than when there is no contention. If the evicted data is required again the request is sent to main memory. Requests for data that is in the cache can be completed immediately, but completion of requests for data in the main memory delay execution until the data has been retrieved.

2.1.1.3 Memory

The use of shared memory by programs can lead to contention. Contention for memory or access to shared data often manifests as stalls in program execution. There are three sources of delay when memory contention occurs. The delays are caused by limited availability of hardware resources.

- The memory controller controls access to main memory. It serves all memory requests. Contention for access to the memory controller causes memory requests to queue and execution to be stalled in the threads where the queueing requests originated. While the thread is stalled it is still active and consuming CPU time.
- Contention for memory bandwidth and load/store buffers cause delays similar to contention for the memory controller. Only requests that have space in the buffer can be processed by the memory controller and considered by the prefetching hardware. Only a limited amount of data can be transferred at a time over the bus between the CPU and the main memory. When the bus is saturated with requests delays occur.
- Flushing of the row buffer of the main memory causes delays in retrieving data from memory. The memory controller in current x86-64 hardware processes requests on a first-come-first-served basis. Each memory bank is divided into

rows. When the row containing the data is already loaded into the row buffer it only requires transmission to the memory controller. However, if the data is in a different row, the current row needs to be written back to memory and another row needs to be loaded. When memory accesses from different threads are mixed it increases the number of times that a new row has to be loaded. A detailed explanation of main memory is beyond the scope of this thesis.

2.1.2 Contention for Software Resources

Concurrent programs share data by making use of shared variables guarded by mutexes or by making use of message-passing. The sharing of data has to be carefully regulated to avoid corruption of data. Delays are introduced when a process has to wait for a message to be sent or received or when a thread has to wait for a mutex lock, that is used to synchronise access to a shared variable, to become available. While this occurs the thread is idle, but not stalled. Very little or no CPU time will be consumed and no stall cycles are caused by idle threads.

2.1.2.1 Contention for synchronisation primitives

In multi-threaded programs that share data, access to the shared data needs to be synchronised and is therefore guarded by, for example, a mutex lock, which has to be acquired before the protected data can be accessed. Other threads attempting to acquire the shared mutex lock will be blocked until the mutex lock is released.

Performance of shared-memory programs can be improved by reducing the time that threads are blocked, waiting for mutex locks. Every delay on the critical path of execution lengthens the total running time proportional to that delay. When a user knows which mutex locks cause contention they can adapt their program to reduce contention by reducing requests for the mutex lock, spreading out the requests for the mutex lock, or reducing the time that a mutex is locked.

2.1.2.2 Wait States in Message-passing Programs

In message-passing programs, execution is delayed when a process has to wait for a communication event to complete before it can continue. Communication among processes of a message-passing program are bound by rules. Blocking communication prevents execution beyond the call of the message-passing function until all participants have completed their part of the communication. This results in delays for all processes other than the last process to reach the communication; for a particular process the delay is equal to the difference between the time that the last communication completes and the time that this process completes.

Geimer et al. described three types of wait states: late senders, late receivers and collective wait states [9]. Wait states are described in more detail in Section 3.1.2.2.

Another form of contention in message-passing programs is contention for bandwidth. When a message-passing program uses the network in ways that create contention for bandwidth (e.g. sending large amounts of data from multiple processes at the same time), communication is delayed.

2.2 Statistical Data Required to Create a Profile of Resource Contention

Statistical data about how hardware and software resources are used is required to estimate the effects of resource contention on program execution.

The Intel CPUs used for this study contain performance monitoring units (PMUs) that record information about hardware usage in the performance registers. The operating system keeps track of the service time, the time each thread of the program spent running on the available CPU core after it was started.

Metrics that show how the hardware was used are derived by combining data from the PMU, and the service time that the program receives over time as read from "/proc". See the Linux manpages for more information. A complete list of all the statistical data that is recorded in Intel performance registers can be found in [10]. Data from the PMU can be retrieved by using Linux perf [15].

In the rest of this section, a description is only provided of the statistical data that is required for this study. A detailed description of how the data is gathered and analysed is given in Section 3.2.

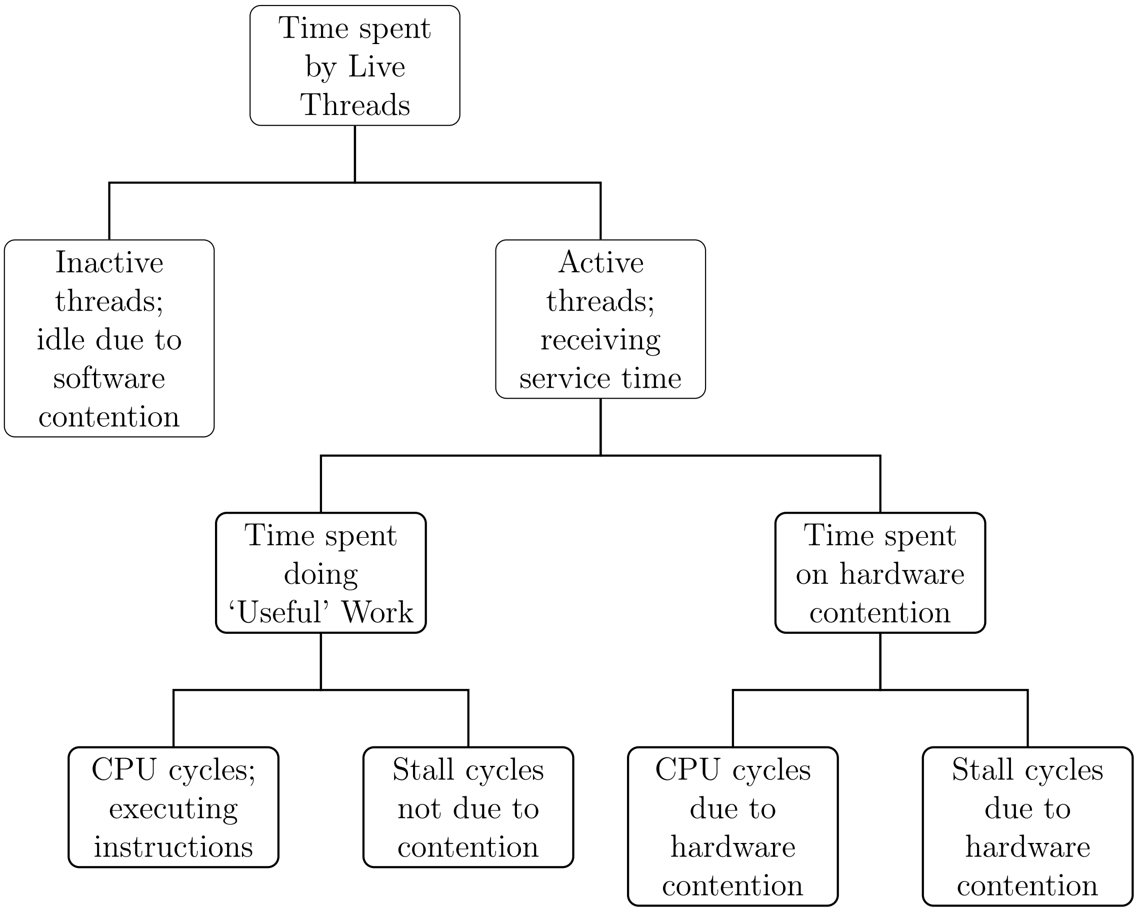
Time spent by Live Threads
Inactive threads; idle due to software contention
Active threads; receiving service time
Time spent doing 'Useful' Work
Time spent on hardware contention
CPU cycles; executing instructions
Stall cycles not due to contention
CPU cycles due to hardware contention
Stall cycles due to hardware contention

Figure 2.1: A classification of execution time in terms of CPU and stall cycles

2.2.1 Run Queue Length and Service Time

All active processes and threads queue for service from the available CPU cores in a system. The operating system keeps track of the threads/processes by recording an entry in the run queue. Entries in the run queue include both running processes (currently assigned to a CPU core) and threads/processes that are waiting for service. The length of the run queue at any given time provides a measure of the number of threads/processes that require service from the available CPU cores.

Threads that require service are called "live threads". As depicted in Figure 2.1, live threads can be either active or inactive. A thread that is live and receiving service from a CPU core is considered active. An active thread spends time either doing useful work or contending for shared hardware resources. The operating system keeps track of the service time each thread receives; A program with two threads may, for example, receive two seconds of service time (one second for each thread) during one real time second.

A thread that is live, but not receiving service, is considered inactive (idle). A thread/process is inactive if it is blocked and waiting for access to a shared software resource; it therefore relinquishes the opportunity to do work on a CPU core. When a thread is not receiving service, the service time does not increase. Note that threads that have terminated early due to load imbalance are not alive any more. If a thread is waiting for more jobs, it is still alive, but inactive so service time will not increase (it is therefore waiting for a software resource). The average number of active threads can thus be calculated by adding the service time each thread in the program received and dividing it by the total number of program threads.

2.2.2 Active CPU Cycles

Each CPU core can process one or more instructions at regular intervals. The frequency at which instructions are processed is called the “clock rate” of the CPU.

Whenever any of the execution units, e.g., arithmetic logic units (ALU’s), of a CPU core processes any instructions in a cycle that cycle is accounted as active in the performance counter registers. The active cycles are a measure of how much work was done.

2.2.3 Execution Time

In an ideal world, the execution time of a parallel program running on n cores would be $\frac{1}{n}$ of the time it runs on one core. This is called ideal speed-up. Ideal speed-up can be defined as $S = \frac{T(1)}{n}$, where S is the ideal speed-up, $T(1)$ is the time it takes to run a program on one CPU core and n is the number of threads.

However, in reality ideal speed-up is rarely attained. For example, it is possible that some parts of the program can only be run sequentially; this is formalised as

Amdahl's law [1]. Additional execution time is also caused by, for example, thread creation, scheduling, and idle time due to load imbalance (which happens when the available work is not distributed evenly among the available CPU cores) and contention.

The real time it takes for a program to run is usually called its critical path time. The more work can be done in parallel, the shorter is the critical path time. Work or idle time that contributes to a longer execution time is said to be on the critical path of execution.

Superlinear speed-up, where the speed-up is larger than ideal speed-up, is also possible. This is achieved when the total amount of work done by the parallel program is less than the total amount of work done by the serial version of the program. This can happen when the order in which tasks are executed has an influence on the number of tasks that need to be executed. For example, if a breadth-first search is executed and the target for the particular search-tree happens to be on the last branch being searched by the serial version, while it is the first branch being searched by one of the threads/processes of the parallel version.

Superlinear speed-up can also happen due to the change in hardware resources available to the parallel version of a program compared to the hardware resources available to the sequential version. For example, if the working set of a sequential algorithm does not fit into the local cache available to the program, but the working set of each process in the distributed version of the algorithm does fit into the local cache available to each process, the total time spent per task (and thus the total amount of work) will be less for the distributed version of the program.

2.2.4 Memory Stall Cycles

Memory stall cycles are the cycles that the CPU spends waiting to read or write data to the main memory. More specifically we count the cycles where the load buffer, store buffer, and reservation station of the CPU are too full to handle requests. When these buffers are full the memory requests queue for service and execution of instructions stall. Memory stall cycles are only one of the indications that stalls occurred. When operands are not available a program uses more CPU cycles to execute the instructions in the reordering buffer. For this reason both memory stall cycles and CPU cycles are combined. This is referred to as work cycles.

2.2.5 Cache Hits/Misses

Caches are random access memory that is typically integrated with the CPU chip and is much faster to access than main memory. Caches provide faster access to data and/or instructions that are used frequently. Thus, when a program has good temporal locality (the same data is used multiple times in a short period) there is a good chance that the data only has to be loaded into the cache the first time it is accessed during that period, or if it has good spatial locality (the data accessed is near data that was accessed earlier), the data might already be in the cache, because when data is accessed, the whole cache line (typically 64 bytes) in which the data is stored is loaded into the cache.

When a program requests data and it is found in the cache it is recorded as a cache hit and it means that it was not necessary to retrieve the data from main memory. Cache misses occur when data that a program requests are not in the cache. The request for the data is passed to a higher level of cache. If a cache miss occurs in all caches the request is sent to main memory and the request is completed.

The hit / miss ratio of a cache is a measure of how well that cache was utilised. A number close to 0 indicates that a large fraction of the requests could not be satisfied by the cache, while a higher number indicates that a higher fraction of the requests could be satisfied by that cache. A low hit / miss ratio can signify either cache contention or a low reuse of data by the particular program.

2.3 Related Work

Various studies investigated contention for resources. Some studies focused on improving the throughput of a computer system, see for example the articles by Weinberg and Snavely, and Wang et al. [21, 20]. Roth, Chandramowlishwaran et al., Tudor and Teo, and Barns et al. investigated the general performance of concurrent programs, such as scalability and resource usage [16, 4, 17, 2]. Examples of studies on specific types of hardware contention are those by Wu and Martonosi, and Xiang et al. who studied cache contention [23, 24] and those by Kim et al., Ebrahimi et al., and Tudor and Teo who studied memory contention [12, 7, 17]. Contention for software resources were studied from various angles. Chen and Stenstrom, Bohme et al., Ebrahimi et al., and Barnes et al. reported on contention that affects the critical path of execution [5, 3, 8, 2], Chen et al., Tzenakis et al., Roth, and Zakkak et al. studied data dependencies in concurrent programs [5, 18, 16, 25], and Geimer et al. and Bohme et al. studied wait states in message-passing programs [9, 3].

2.3.1 Improved utilisation of hardware resources on a system level

Weinberg and Snavely considered how to best utilise the available major resources in a computer given a subset of workloads, without favouring any single program [21]. They considered information about how shared resources are utilised and created a scheduler that improved the throughput of their system by 20 percent. Wang et al. proposed an analytical model that maximises performance relative to various performance objectives. These objectives are: maximum system throughput, fairness, or harmonic weighted speed-up. The model computes the partitioning of the memory bandwidth that will achieve the best results. They investigated providing a guaranteed quality of service. Scalability analysis was also investigated as a guiding metric for their system.

Wu and Martonosi reduces cache contention by partitioning the cache for each program [23]. Their system calculates how much of the cache each application requires and then assigns a number of cache lines based on the demand on the system and the cache access patterns of the running programs. Xiang et al. gathered information about cache contention among programs and used scheduling to reduce contention [24]. For each program they keep track of the cache "footprint", the number of cache lines in use by a program at any time. Based on the footprint and the miss rate they predict how many cache misses will occur. If the prediction suggests contention, tasks are regrouped to avoid the contention. They optimise to reduce the total running time of all workloads, similar to Weinberg and Snavely [21]. Programs that make use of shared data are not considered.

Zhuravlev et al. classify programs based on how they use shared resources [26]. This classification is used to construct a scheduling algorithm that takes contention for the last-level cache, the memory controller, and the memory channel into account in order to improve the quality of service that each program receives. It was found that improving the quality of service that a machine provides yields gains for all running programs, instead of improving the performance of individual programs. The study initially only investigated cache contention, but efforts were extended to include memory resources when tests showed contention for memory resources to be a major factor in the degradation of performance. Contention for shared resources among threads and processes of the same program were not investigated.

Kim et al. created a scheduler that reduces memory contention while still giving fair service to all programs [12]. They classify programs as either latency-sensitive or bandwidth-sensitive and then use this classification to schedule the programs such that memory contention is reduced.

Latency-sensitive programs are favoured, because they generate fewer requests for data and require more computational resources. Each bandwidth-sensitive program is given a fair opportunity for requesting data according to a policy that balances fairness with the number of requests.

As more CPU cores share the same memory controller, the effectiveness of data prefetching is reduced. Ebrahimi et al. studied how to retain the performance gains attained by data prefetching [7]. This is done by counting how many of each thread's prefetch requests are accurate during each time period—i.e., the prefetched data was actually required by the thread—and then using this information during scheduling to favour those threads that could most often prefetch the correct data. This feedback mechanism was integrated into two schedulers, one that takes only memory access patterns into account and another technique that additionally performs source-throttling of contentious programs. Memory non-intensive (compute intensive) programs are favoured similar to work already mentioned in other studies.

2.3.2 Improved Utilisation of Resources at the Program Level

Chandramowlishwaran et al. documented the process of analysing a specific program, called the Fast Multipol Method, and tuning the program for better performance on a single multi-socket node [4]. They classified parts of the code as either compute-bound or memory-bound. Compute-bound code makes heavy use of arithmetic operations and has fewer data accesses in comparison. Memory-bound code makes heavy use of data in the cache or memory and has fewer arithmetic operations. The performance of the memory-bound parts are then improved by reducing contention for the shared cache, shared memory, and inter-socket bandwidth. Contention for the cache is reduced by executing code that uses large parts of the cache on different sockets. The performance of the compute-bound parts are improved by reducing migration among CPU cores, context switches and contention for access to CPU cores. They further reduce contention by running compute-bound threads with memory-bound threads instead of running all the compute-bound code in one stage and the memory-bound code in another stage. A similar categorisation of programs as either latency sensitive or bandwidth sensitive was proposed by Kim et al. [12].

Ding et al. examined loops in a program and reorderd data to improve the performance of a program [6]. They proposed three data reordering schemes: reordering data to reduce cache misses, optimising row buffer hits in such a way that cache misses do not increase, and trading cache hits for fewer row buffer flushes. They obtained the best performance by trading cache hits for better row buffer utilisation.

2.3.3 Reducing Idle Time Due to Dependencies

Tzenakis et al. created a framework that analyses the source code of a program and runs parts of the program as independent tasks [18]. They use static dependency analysis of source code and a custom memory allocator to ensure that the tasks they run do not depend on data that are being used or have not been calculated yet. Zakkak et al. focused on reducing data dependencies in programs [25]. Using static dependency analysis they remove unnecessary run-time checks in the source code. They identify independent sections of the code, that can be run without affecting the rest of the program. When the inputs for those sections of code are ready and there are CPU cores available the task is run.

Ebrahimi et al. proposed techniques that give priority to some threads over others to reduce contention on the critical path. They identify which threads are on the critical path and which threads are currently using shared resources that are also required by others [8]. The techniques estimate lock contention and measure loop progress. Threads that hold a contended lock and loops that have made the least progress towards a barrier are given a higher priority. The priorities of threads that are equally critical are shuffled so that all threads make equal progress.

2.3.4 Profiling of Contention in Concurrent Programs

Liu and Mellor profiles programs running on NUMA systems [13]. Using information about the domain (either local or remote) of every memory access and the latency of remote accesses three metrics are derived: total number of local memory requests, total number of remote requests, and the number of instructions per remote memory request. The last is a measure of how long it takes to serve a remote memory request. These three metrics indicate how effectively a program running on a NUMA machine accesses memory. To improve performance, programs were adapted to ensure that data is mapped to the NUMA node that uses it the most.

Tudor and Teo quantifies speed-up loss due to contention for memory bandwidth or other shared resources [17]. According to their model the lifetime of a program can be divided into useful work, overhead due to memory contention (waiting for memory requests), and inactivity induced by data dependency (waiting at a synchronisation point or waiting for work), so in their model any inefficiencies not due to memory contention or data dependencies are considered part of the normal work. The speed up loss due to other dependencies is defined as the difference between the number of threads, spawned by the parallel program, and the actual average number of active threads (provided that there are at least as many cores as threads available). Speed-up loss due to memory contention is defined as the difference between the actual average number of active threads (active threads include threads executing instructions as well as threads stalled, waiting for memory requests) and the number of threads doing useful work (speed-up). To measure the memory contention, they measure the growth in the number of stall cycles due to memory contention compared to a baseline value on one core, where there is no memory contention among cores.

Roth decomposes the execution time of a running program into working time, distribution overhead, and delays due to contention [16]. Distribution overhead includes the time spent on distributing the work among the available threads, idle time due to load imbalance and idle time due to serial sections of code that can only be executed by one thread (also called insufficient parallelism). Delays due to contention for hardware or software resources include delays due to contention for hardware resources such as memory or caches and delays due to synchronisation (e.g., waiting to acquire a lock held by another thread). Roth focuses on parallel programs with task-level parallelism such as OpenMP, where work is distributed on demand as CPU cores become available. The time spent doing actual work and the time spent on the distribution of work by the parallelization framework are recorded. Idle cycles are measured by counting the number of cycles when a core is idle while other cores are busy doing work. Overhead due to contention for software resources is measured by measuring the time spent on acquiring the resource (e.g., waiting at a barrier). The overhead due to contention for hardware resources are not measured. Since the overhead due to contention for hardware resources is the only unknown factor, they infer it by subtracting the time taken by all the other factors from the total execution time of each thread. Performance is improved by changing the parameters of the paralilisation framework, such as the number of threads used, and the method of distributing tasks. Using these performance metrics it is possible to either automatically adjust the configuration as the program runs or to suggest a static configuration that is passed to the parallelisation framework when the program is started.

Barnes et al. studied the scalability of a given program, given a specific large-scale system configuration; they describe three techniques with increasing complexity and effectiveness in their article [2]. The simplest technique analyses the communication patterns of important sections of code in the program when run on a small number of processors. By using regression with a prediction function they estimate how well the computation will scale when executed on a larger number of processors.

Another technique collects the time it takes to complete each computation and communication among processes. A representative set of processes is identified and regression is used to calculate how well the program will scale. The third technique in addition to the previous technique also takes the global critical path into account.

Geimer et al. defined the idea of "wait states" in message-passing programs [9]. Wait states occur when processes are blocked while waiting for message-passing to complete. Even when processes are not blocked, but the message cannot be passed immediately, there is still overhead. In point-to-point communication there are two types of wait states: late sender and late receiver. In a late sender, the receiving process is blocked until the message can be sent and in a late receiver, the sending process is blocked until the message is received. Collective communication block processes in a way similar to a traditional barrier. All processes involved in the communication are blocked until the communication completes in every process. The effect of these wait states can be quantified. For each place in the program where wait states occur, the time spent waiting is calculated. Bohme et al. further studied the effect of wait states on load imbalance [3]. They found that wait states on the critical path of the program are the cause of load imbalance and result in other parts of the program waiting. They reduce wait states by removing the root causes of wait states.

Chen and Stenstrom describes how they find the time spent contending for locks and barriers on the critical path [5]. Any blocking that occurs as a result of contention causes overhead which can be reduced by reducing the contention, but there is an additional advantage to reducing contention on the critical path. Every moment that threads on the critical path spend blocked lengthens execution time of the program directly. Removing these inefficiencies will shorten the running time of the program. A user is alerted when contention occurs on the critical path. They include information about the specific lock and location in the source code that causes the contention.

Chapter 3

Measuring Resource Contention in Concurrent Programs

Resource contention affects how the program accesses resources it requires and how many resources can be used at any given moment by a program.

3.1 Description of Contention for Hardware and Software Resources

Recall that the live threads of a program can be either inactive due to software contention or active and receiving service. Service time is spent either doing useful work or contending for resources. The useful work includes all the work required by a serial implementation plus all the extra work required by the parallel implementation that would not be needed by a serial version, e.g., the distribution of tasks among the processes/threads, executing requests for locks, and preparing a message before sending it. A parallel implementation executed on a single core would include most, if not all, this extra work, but would include no time spent on contention.

This chapter provides a description of how idle time due to software contention and extra service time due to hardware contention is computed. The overhead due to contention for hardware resources (mostly contention for memory) is computed by using the model proposed by Tudor and Teo to measure the speed-up loss due to memory contention among threads compared to the case where there is no contention [17]. We also calculate the average number of active threads using the method proposed by tudor and Teo. The average number of active threads provides a measure of the speed-up loss due to all contention other than memory contention (idle time). In a shared memory program, this idle time is dominated by contention for synchronisation primitives and in a message passing program this idle time is dominated by synchronisation time, contention for message-passing primitives and message buffers. Our methods for quantifying contention for software resources that provide synchronisation and identifying its impact on the critical path are based on the work by Geimer et al. [9], and Chen and Stenstrom [5].

In our classification of execution time, the overhead of task distribution without any contention is part of the useful work of the program and is thus not measured separately; it is the part of service time that is required by the parallel implementation. If the distribution of tasks does not result in any contention for software or hardware resources, the overhead of distributing the tasks is a function of problem size and does not increase with an increase in the number of cores. A programmer can measure this by simply timing the distribution sections of the program. If, on the other hand, there is contention, the idle time it causes will be measured as part of software contention.

In the model proposed by Tudor and Teo, idle time due to a temporary unbalanced partitioning of tasks, or waiting for a single thread to complete a serial section of code, is seen as due to software dependencies, and is thus measured as part of the total speed-up loss due to software contention. If the work is statically partitioned into independent parts at the start of the execution and the execution times of the partitions are unequal; a thread that completes its tasks will no longer be live. This is not seen as idleness due to software contention and not measured as such, but could be measured separately by timing how long completed threads wait for others to complete. Idleness caused by serial sections of code can also be measured separately by simply timing the serial sections. Any other causes of load imbalance will be measured as part of software contention.

A set of tools was implemented. These tools gather and analyse data about resource contention. The set of tools consists of two phases: a data collection phase and a data analysis phase. The tools need to identify the main sources of idle time due to contention without any knowledge about specific algorithms. The data collection tools therefore use techniques that do not require modification of the target program; such as reading hardware counters and creating wrappers around calls that send/receive messages or require/release locks. The mutex data gathering tool does require recompilation of the target program, but the layout of the program data and instructions will not change if the same compiler is used. The only difference is that the program that gathers data contains calls to the mutex wrapper functions and the original program contains the original Pthread functions. The data collection and analysis phases are separated to minimise the impact on the performance of the program being profiled.

A performance profile of a program running on a specific computer system is created using information from hardware performance counters. In particular, information is read from the PMU about the use of shared resources such as the last-level cache and the main memory. The number of CPU cycles shows how many cycles the CPU core used to do the work. The number of processes / threads in the run queue and the amount of service time that the program receives is regularly read from the operating system and recorded in a trace. Information is gathered for the same workload using m threads, running on a single CPU core with no contention, and running on $n = m$ CPU cores. Increases in the total service time and work cycles often indicate contention for shared resources. This information is used to predict the speed-up loss due to memory contention and all other dependencies. Specific sources of contention for shared software resources were selected for study. Point to point communication in MPI programs, collective communication (barrier-style) operations in MPI programs, and Pthread mutex locks were selected as sources of contention for software resources. A set of tools was created to gather data, analyse it, and calculate the time lost due to contention for the selected resources.

The use of shared memory resources has various performance implications. High traffic through the memory busses, frequent row buffer flushing, and the queueing of requests at the memory controller lead to longer delays in satisfying all requests for data. These three conditions occur when there is a high number of memory requests. Additionally contention for space in the cache leads to an increased number of cache misses and a higher number of memory requests. For this reason memory contention is the primary hardware resource considered for this study.

3.1.1 Contention to access shared memory

All running programs experience stalls in execution, such as pipeline hazards, branch prediction errors, and the retrieval of data from the main memory and caches. The stalls that are not caused by contention are considered constant for the purposes of this study and tests performed by Tudor and Teo confirmed this.

Retrieval of data is fastest for caches close to the CPU core. Each further level of cache has a longer retrieval time. If the data is not found in the last-level cache, the request is sent to the memory controller and the data is fetched from main memory, resulting in an even longer delay.

The threads of a program can be either active, executing instructions or stalled and actively waiting for memory requests, or they can be inactive (idle), because they are waiting at a synchronisation point or waiting for an event. The total number of work cycles, $C(n)$ can be divided into 'useful' work cycles and work cycles due to contention; see Figure 2.1. Let U be the total number of cycles required to execute a program with m threads on $n = 1$ core. U will thus include all the CPU cycles required by the program as well as stall cycles that are not due to contention. Let $M(n)$ denote the extra number of work cycles due to contention among the $n = m$ cores compared to the case where there is no contention for memory ($n = 1$). Note that for a run on a single core, contention for software resources will still occur, since multiple running threads can still compete for the software resources. The only source of contention that is removed is the contention for memory resources, provided that the patterns in which data is placed in the shared cache remain relatively similar to when m CPU cores are used.

Let $A(m, n)$ denote the average number of active threads over the entire execution time T(n) of a program, partitioned into m threads, running on n cores. If $n = m$ and there is no software contention, then $A(m, n)$ will be equal to the number of program threads m, because all m threads will be active for the entire duration of the execution. A description of how $A(m, n)$ can be computed is provided in Section 3.1.2. The execution time of the program on n cores, T(n), can be expressed in terms of the total number of work cycles for the program on n cores, i.e.,

$$T(n) = \frac{U + M(n)}{A(m,n)}$$

and the speed-up can thus be expressed as

$$S(m,n) = \frac{T(1)}{T(n)} = A(m,n)\frac{U}{U + M(n)} \tag{3.1}$$

Let $\omega(m,n)$ denote the ratio of activity due to contention and activity due to useful work. Recall that the number of work cycles increases for each thread that is active during that cycle. Therefore ω can be expressed as the ratio of the total number of work cycles due to contention and the total number of work cycles due to useful work $\omega(m,n) = \frac{M(n)}{U}$. Modern CPU cores contain multiple execution units that may be active concurrently. Only instructions that have been loaded into the reordering buffer can be considered for execution by these execution units. An instruction is only executed when all operands are available. When contention occurs retrieval of operands is delayed and execution units are left waiting for operands. Although service time is also a measure of activity, it does not include the effects of delayed operand retrieval. For this reason using the number of CPU cycles is a better option than using service time alone when investigating memory contention, because the effect of the delayed operand retrieval is expressed in ω. This is the only source of an increase in execution time when memory contention occurs.

Since ω represents the ratio of activity due to contention and activity due to useful work, it can be used to calculate the speed-up loss due to memory contention. Recall that for a program with m threads running on $n = m$ cores, the total service time is the sum of the time spent doing useful work and the time spent on contention. When the total number of work cycles are used to calculate ω, contention for software resources has no effect on ω, since the total number of work cycles only increases when doing work or when stalled due to contention for hardware resources. Thus the speed-up loss due to contention for memory can be separated from the speed-up loss due to software contention. We can define the speed-up loss due to memory contention, $R(m,n)$ as the difference between the average number of active threads $A(m,n)$ and the average number of active threads doing useful work $P(m, n)$.

$$\begin{aligned}\omega(m,n) &= \frac{M(n)}{U} = \frac{(A(m,n) - P(m,n))}{P(m,n)} \\ \omega(m,n) + 1 &= \frac{A(m,n)}{P(m,n)} \\ P(m,n) &= \frac{A(m,n)}{(\omega(m,n)+1)}\end{aligned} \tag{3.2}$$

The speed-up loss due to memory contention is thus:

$$R(m,n) = A(m,n) - P(m,n) = A(m,n) - \frac{A(m,n)}{(\omega(m,n)+1)} \tag{3.3}$$

The speed-up of a program, see Equation 3.1, can also be written in terms of $\omega(m, n)$:

$$\begin{aligned}S(m,n) &= A(m,n)\frac{U}{U+M(n)} = A(m,n)\frac{1}{1+\frac{M(n)}{U}} \\ S(m,n) &= \frac{A(m,n)}{1+\omega(m,n)}\end{aligned} \tag{3.4}$$

Thus speed-up can essentially be seen as the number of active threads doing useful work.

If $M(n)$ is written as $C(n) - C(1)$, i.e., the difference between the total number of work cycles with contention and the total number of 'useful' work cycles, then $\omega(m, n)$ can be written as:

$$\omega(m,n) = \frac{C(n) - C(1)}{C(1)} = \frac{C(n)}{C(1)} - 1 \tag{3.5}$$

The value of $\omega(m, n)$ can thus be calculated for any number of cores n if the values of $C(n)$ and $C(1)$ are known. The next two sections describe how the total number of work cycles $C(n)$ for a program partitioned into m threads can be predicted for any number of n by taking only two measurements on a UMA machine and only three measurements on a NUMA machine.

3.1.1.1 Single socket UMA machines

Each CPU core contains multiple execution units, i.e. arithmetic logic units. All execution units can be active concurrently, provided there are instructions that requested the execution units. Instructions are loaded into the reordering buffer, operands of that instruction are requested, and the instruction is executed when an execution unit is available. When the data is unavailable or there is no execution unit available to execute the instruction, execution of that instruction stalls until it is available. Stalls caused by branch mispredictions or unavailability of execution units are known as "front-end stalls". Stalls while waiting for data to be retrieved are called "back-end stalls". In particular, memory stall cycles are back-end stalls caused by the load buffer, store buffer or reservation station being full. When any execution unit is active during a CPU cycle, the cycle is accounted as active, but when only some instructions can be executed, more cycles are used to do the same amount of work. Furthermore, every instruction that is waiting for resources uses space in the reordering buffer, delaying the loading and execution of instructions that are further along the instruction stream.

For memory-bound programs, the critical path of the execution time of a program is dominated by the response time of memory requests. All the CPU cores in a single CPU socket share the same last-level cache which is connected to a single memory controller that processes memory requests in the order they arrive. The number of requests sent to the memory controller is therefore equal to the number of last-level cache misses. Since the memory requests are filtered by two/three levels of cache, it is assumed that the inter-arrival times of the requests from the different cores are identically distributed. According to the M/M/1 model [11], the response time (in number of CPU cycles) of one memory request that has arrived at a memory controller that services n cores, $C_{\text{req}(n)}$ is a function of the service rate of memory requests, μ, and the arrival rate of memory requests, λ. Thus $C_{\text{req}} = \frac{1}{\mu-\lambda}$. Let $r(n)$ be the total number of cache misses and L be the number of memory requests originating from each of n CPU cores (assuming the requests are evenly distributed among all cores). Then, for a single socket system with n active cores $\lambda = n.L$ and

$$C(n) = r(n).C_{\text{req}} = \frac{r(n)}{\mu - (n.L)} \tag{3.6}$$

Using measured values of $C(n)$ and $r(n)$ for at least two values of the number of cores, n_1 and n_2, the values for μ and L can be calculated by regression through $(n_1, \frac{1}{C(n_1)})$ and $(n_2, \frac{1}{C(n_2)})$, $\ldots$, $(n_i, \frac{1}{C(n_i)})$, provided that $r(n)$ stays constant irre-

Threads	Program	Measured	modelled	Program	Measured	modelled
2	CG	0.005	0.003	EP	-0.008	-0.004
4	CG	0.042	0.010	EP	-0.003	-0.001
8	CG	0.138	0.015	EP	-0.004	-0.001

Table 3.1: Examples of measured ω and modelled ω for the CG benchmark and the EP benchmark. CG is memory-bound and EP is compute-bound.

spective of the number of processors. Using the algorithm for finding the line of best fit shown in Listing 3.1, a line is plotted on the plane with m as x-axis and $\frac{1}{C(n)}$ on the y-axis. The slope of the line of best fit is μ and the y-intercept is λ (L can be computed as $\frac{\lambda}{n}$). We find the line of best fit for values of $C(n)$ measured with $n = 1$ and $n = m$ (the number of cores equal to the number of threads), respectively. Once the values of μ and L have been calculated using measurements for runs on $n = 1$ and $n = m$ cores, Equations 3.6 and 3.5 can be used to calculate $C(n)$ and $\omega(m, n)$, respectively, for values of n other than $n = 1$ and $n = m$ without any further measurements.

When $n = 1$ Equation 3.6 always resolves to zero. The calculation of μ and L are affected by the data from the run where there is no contention and the number of cycles are under-predicted. In turn the number of cycles for $n = 1$ are over-predicted. However, the ratio between the prediction for $n = 1$ and $n = m$ provides an indication of the values of ω. When the number of useful work cycles is known, ω can be used instead of modelling ω. For results shown in this thesis measured values of ω were used, since the aim is not the prediction of performance. Table 3.1 contains examples for the compute-bound EP benchmark and the memory-bound CG benchmark.

3.1.1.2 Multiple sockets on NUMA machines

For a multiple socket NUMA system, when two sockets and two memory nodes are active, there is an additional delay to send the memory requests to a remote node. Let δ be the additional time required to send the memory requests to a remote memory controller, compared to the case when only the local controller is active. The increase in delay depends on the ratio of remote memory accesses to total memory access. If n cores are split such that there are c on the first socket and the other $n - c$ cores on the second, then, on average, $\frac{c}{n}$ of the accesses will go

Listing 3.1: Regression algorithm to find the line of best fit given a set of points. It is used to calculate μ, λ and δ.

```
Function lineBestFit(points)
  sumY = 0
  sumX = 0
  sumX2 = 0
  sumXYMul = 0
  for each p in points {
    sumX = sumX + p[0]
    sumX2 = sumX2 + (p[0] * p[0])
    sumY = sumY + p[1]
    sumXYMul = sumXYMul + (p[0] * p[1])
  }
  n = points.length()
  meanX = sumX / n
  meanY = sumY / n
  slope = (sumXYMul - sumX*meanY) / (sumX2 - sumX*meanX)
  yIntercept = meanY - (slope * meanX)
  return (slope, yIntercept)
end lineBestFit
```

to the first memory controller and $\frac{n-c}{n}$ of the accesses are to the second memory controller. Thus

$$C_{NUMA}(n) = C(c) + r(n).\delta.C(n-c) \tag{3.7}$$

We measure $C(1)$, $C(c)$, and $C(m)$, where there are c cores per socket and $m > c$. The values of μ and L are calculated as before (using the data for $n = 1$ and $n = c$), and δ is computed by regression through the line $(c, C(c))$ and $(m, C(m))$ using the line of best fit algorithm shown in Listing 3.1. The values of μ, L, and δ can then be used in Equations 3.6 and 3.7 to calculate C(n) for values of n other than $1, c$, and the chosen m. The value of $C(n)$ can then be used to calculate $\omega(m, n)$ using Equation 3.5.

3.1.2 Contention for software resources

The computation that a concurrent program performs is distributed among different threads or processes. When data is required by multiple threads, data has to be shared either by using a variable that can be accessed by the required threads or by sending data between processes. Access to shared resources is controlled by synchronisation primitives, such as a mutual exclusion lock (mutex) or a semaphore, to avoid race conditions. When a request to acquire a busy mutex is received, that thread needs to wait until the mutex is released. Other synchronisation primitives can also be used to control how the program executes. The execution of a thread that reaches a barrier is blocked until all the threads have reached the barrier. Some message-passing operations block execution until the operation completes on some or all processes.

The effect of software contention is idleness of threads while waiting for a mutex to be acquired or a message-passing operation to complete. The higher the demand for access to a mutex or bandwidth to pass messages, the longer the delays. As described in Section 2.1.2 service time is the time that the operating system schedules a program to receive service from a CPU and measurements from the performance counters show how that time was spent. Programs only make progress when they receive service from a CPU core and when execution is not stalled. This is depicted in Figure 1.1(a) and (b). Figure 1.1(a) represents a thread that alternates between executing outside and inside a particular critical section, where the execution outside the critical section is twice as long as the execution inside the critical section, while Figure 3.1(b) shows that with four such threads accessing the same critical section, every time a thread requests a lock it has to wait (idle) for at least one other thread to release the lock before it can enter the critical section.

When the number of cores, n, available is equal to the number of threads, m, and there is no contention for software resources, then all the threads will be active for the entire duration of the program's execution and the number of active threads, $A(m,n)$, will be equal to the number of program threads m.

In the absence of both hardware and software contention this will mean an ideal speed-up of $S(m,n) = \frac{T(1)}{T(n)}$. In the presence of software contention, not all threads will be active for the entire duration of the program and the difference $m - A(m,n)$ expresses the speed-up loss due to software contention. This is shown in equation 3.8. When there are fewer cores available than active threads, i.e., $n \leq m$, then some active threads will be executing and the rest will be in the run-queue.

$$D(m) = m - A(m,n) \tag{3.8}$$

The average number of active threads $A(m,n)$ can be computed as follows. Let τ be the total service time that a concurrent program with m threads running on n CPU cores, receives. Let $A(m,n,t)$ denote the number of active threads running on n cores at time t. If ΔT, the time between samples of the service time (say time t_1 and time t_2), is small, the number of active threads does not change substantially between measurements and then $\tau = \sum_{j=1}^{m} \tau_j$, the sum of the service times each thread received during the time interval ΔT (where τ_j is the service time received by thread j). The critical path time, ΔT_{cp}, for the interval is equal to the service time of the thread with the longest service time of all the threads that were active during the interval. The average number of active threads during an interval is $\frac{\tau}{\Delta T_{cp}} = \frac{\sum_{j=1}^{m} \tau_j}{\max\{\tau_j\}}$. The critical path time for the entire program is equal to $T_{cp} = \sum \Delta T_{cp}$. and the average number of active threads during the entire execution of the program will be $A(m,n) = \frac{\sum A(m,n,t)\Delta T_{cp}}{T_{cp}}$. When there are enough cores to execute all threads ($n \geq m$), there is no constraint on the number of threads that can be active at any time. However, when $n \leq m$, only n threads can be active in the interval ΔT.

To determine the average number of active threads of a program without memory contention, we run the program, partitioned into m threads on one core and measure, at regular time intervals, ΔT, the service time for each active thread. As an example of how the average number of active threads is calculated, see Figure 3.1(a), which depicts four threads executing on four cores. In two cases the number of memory stall cycles of thread four are double due to contention. Figure 3.1(b) depicts the first four rounds of a Round Robin (RR) scheduling of the four threads on one core. In this case there is no contention for memory and all the stall cycles will be considered as part of useful work. In (a) the fourth thread is stalled during the first and second RR quanta it receives, because thread three is accessing memory; thread four therefore only accesses memory during the third and fourth time quanta it receives, but in (b) thread four can continue executing instructions during the third and fourth quanta it receives (which is the 12th and 16th quanta in (b)). Assume each time quantum is $x$ms and that we measure the service time of each thread i (for $i = 1, 2, 3, 4$), every eight rounds of the RR schedule (i.e., $\Delta T = 32x$ms and each of the four threads will be scheduled eight times). Then during the first interval $\frac{\tau}{\Delta T_{cp}} = \frac{8x+7x+7x+8x}{8x}$ and during the second interval

$\frac{\tau}{\Delta T_{cp}} = \frac{7x+8x+6x+4x}{8x}$, so $A(m,n) = A(4,1) = \sum \frac{A(4,1,t)\Delta T_{cp}}{T_{cp}} = \frac{30x+25x}{16x} = 3.4375.$

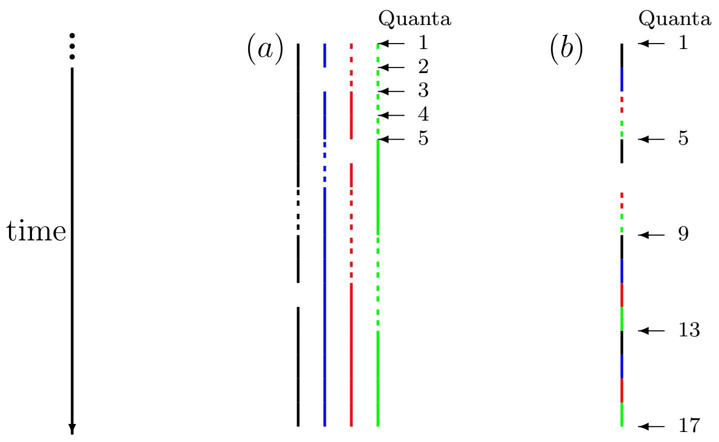

Figure 3.1: Figure (a) represents the execution of four different threads in parallel on four cores; in two cases the memory stalls of thread four is double due to contention. The arrows represent the start/end of a time slice (RR scheduling). Figure (b) represents the execution of the four threads on a single core; the first four rounds of a RR schedule is shown. Threads 1, 2, 3, 4 are scheduled in order and each arrow points to the start of thread 1's time slice.

3.1.2.1 Contention for synchronisation primitives

Contention for shared software resources occurs in multi-threaded programs that share data. Access to shared data needs to be synchronised to avoid race conditions, and is therefore guarded by, for example, a mutual exclusion lock (mutex), which has to be acquired before the protected data can be accessed. Once a mutex has been acquired, any other thread attempting to acquire the mutex will be blocked until it is released.

Performance of shared-memory programs can be improved by reducing the time that threads are blocked waiting for locks. Every delay on the critical path of execution lengthens the total running time proportional to that delay. When a user knows which locks cause contention they can adapt their program to reduce contention by reducing the number of requests for a lock, lengthening the time between requests for a lock, or shortening the time that a lock is held before release.

A tool was created that records information about synchronisation events in a program, analyses the trace of events and reports summary results to a user. The tool is based on work by Chen and Stenstrom [5]. The tool recognises three types of events on a lock: acquiring a requested lock immediately (without any delay), waiting to acquire a lock, and releasing a lock. This technique can be applied to any kind of synchronisation primitive such as semaphores and pthread barriers. As a proof of concept our current implementation supports only mutex locks, but will be extended to support other synchronisation primitives in future work.

3.1.2.2 Wait states in message-passing programs

Sources of resource contention in message-passing programs are wait states. Geimer et al. [9] described three types of wait state: late senders, late receivers and collective wait states. The message-passing library has to wait and keep track of communications until they complete, requiring additional processing time to keep track of the waiting message. Often execution of the waiting process is blocked until the communication completes. When processes are blocked this may cause load imbalance or more wait states since the program does not run evenly over all the available processors.

Geimer et al. define three types of wait state. Point to point late senders, point to point late receivers and collective wait states. When one process sends data to another in point to point fashion, the communication involves a sending process and a receiving process. When the data is sent before the receiving process is ready to receive the data, the sending process cannot continue until the data has been received. This is known as a late receiver. When the receiver has to wait for data to be sent it is called a late sender. Processes that complete their part of the message-passing communication later than the first process, delays the other processes by the amount of time that each process has to wait. All processes that are involved in a collective communication are prevented from continuing execution after the barrier, until all participants complete their part of the communication. This results in idleness of all participants that complete the communication before the last participant. This is known as a collective wait state. To reduce idleness all participants should complete the communication at the same time. Note that functions such as MPI_Barrier and other barrier constructs have a similar effect as collective communication with regard to idleness.

3.2 Implementation

We created three different tools to analyse the performance of concurrent programs. Each tool has a component that gathers data and a component that analyses the gathered data. The General Metrics Gathering Tool (GMG) focuses on gathering and analysing data from the performance monitoring units and service time data of a running program. It can be used to launch and monitor a target program and bind it to specific CPU cores or it can be attached to an already running program. Two data files are created, one that contains all the statistics obtained from the performance monitoring units and another that contains a trace of the service time that each thread of the program received. These files are then used by the analyser component of the GMG.

The second tool, called the Mutex Wrappers, gathers and analyses data about synchronisation events. The data gathering component is created as a wrapper around the functions used to lock and unlock mutexes. The mutex functions are replaced with the wrapper functions at compile time. A trace of mutex acquisition and release events are recorded in a file that is then used by the analysing component.

The third tool gathers data about message-passing events in a program. The data gathering component is a library with wrapper functions around MPI functions. The wrapper functions are loaded by using LD_PRELOAD when the program is launched. A trace file is created for each MPI process. Trace files are called “MPITrace.rank”, where rank is the number returned by MPI_Rank. The analysing component performs analysis on the traces. A standard text format is used for all trace files.

Each trace file for the mutex wrappers and MPI wrappers contains two sections. The first section is the actual trace of events in the program and the second is a table containing additional information. Trace events are written to the file as they occur. The information table is written to the file just before the program exits. The Trace events and table entries are separated by a new line character. A tab character is used as field separator. If a trace contains an index table, the trace data is separated from the index table by “—”. This is followed by a line containing the process rank (or -1 if no rank was assigned) and the starting time offset. Traces from the mutex and MPI wrapper tools record index tables

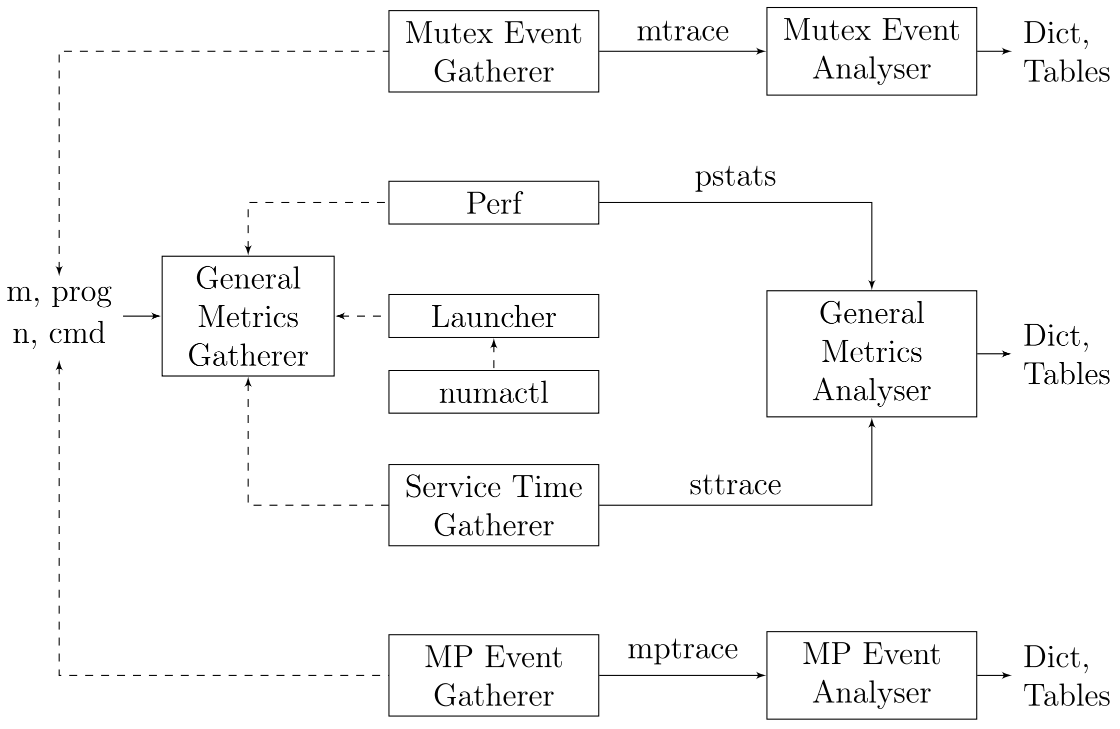

Figure 3.2: Tool Layout

containing static information. Instead of repeating this information unnecessarily, only the index number is given as reference.

A diagram of the tools and their components is depicted in Figure 3.2 and the tools are described separately in Sections 3.2.1.1–3.2.1.3.

3.2.1 Data Gathering Components

3.2.1.1 General Metrics Gathering Tool

The GMG uses a combination of components to gather data. It uses Linux Perf to read the statistics stored in the performance monitoring units (PMU), a component (written in C) to record service time data, and a launcher (written in C) to either launch the target program or attach itself to a running program. In turn the launcher uses the numactl utility to map processes to nodes according to a machine layout file [14]. The GMG uses a shell script to call the components and set up all the environment variables; it takes one of two sets of arguments. The first set of arguments is used to launch a program and consists of:

- The name of the program to monitor.
- The number of threads to use.
- The number of CPU cores to use.
- A program launch command that can have any number of arguments as long as they appear after the first three mandatory arguments.

The second set of arguments is used to gather data from a program that is already running. The arguments consist of: “-p” followed by a process ID. The GMG finds the process with the given process ID and gathers data from that process.

When not gathering data from an already running program, the launcher component launches the target program (using the specified launch command) in its own address space using Numactl to restrict execution to specific processors and sockets. A machine layout file called “systemConfig.txt” contains a list of processor numbers. Processors are assigned to a program in the order that they appear in the systemConfig file.

Gathering service time information After the target program has been launched, the /proc directory is searched for a running program matching the name passed as one of the arguments to the GMG. The service time of each thread that is found is then monitored. The total service time that each thread received is read from the /proc/programPid/task/childThreadPid/stat file. The user and system service times are the 14th and 15th fields in the file, respectively. The sum of these two values is the total service time. This component of the GMG keeps track of the total service time of each thread and records only the difference between the current measurement and the previous measurement. This reduces the size of the trace file compared to when the total service time is recorded. The length of the run queue is also recorded each time that service times are recorded.

A trace of the run queue length and service times each thread receives is written to a file called “stdata.txt”. Service times are sampled at regular intervals. Shorter intervals will yield traces with higher accuracy and larger traces. Smaller traces with lower accuracy can be obtained by increasing the sampling interval. We selected a 100th of a second as a good balance between high accuracy and the size of the trace file. The time scale used by the operating system is also a hundredth of a second. As an example the data for six entries of a service time trace file is shown in Table 3.2. This trace file was generated for a program with two threads. Each entry contains three values, i.e., the length of the run-queue, the total service time received by thread 1 during the previous interval, and the service time received by thread 2 during the previous interval. The service time trace consists of samples. Each sample is a textual list of the service time each thread of the program received in the sample period. Each sample is separated by a new line character.

Runqueue Length	Thread 1 Service time (s)	Thread 2 Service time (s)
2	0.01	0.01
2	0.01	0.01
2	0.01	0.01
2	0.01	0.00
2	0.00	0.01
2	0.01	0.01

Table 3.2: An example of a trace of the runqueue length and the service times received by two threads. Times are given in seconds.

The GMG shell script calls the Linux perf program to collect the performance metrics specified in the shell script, by prepending the launch command with the necessary parameters. These metrics are written to a file called "pdata.txt". An example of the output produced by Linux Perf is shown in Listing 3.2, We use the libpfm4 library and included programs to manually look up the event codes for the metrics we want to measure. The event codes often differ depending on the CPU and other hardware contained by the system running the tests. We gather the following metrics:

- Last-level cache misses.
- Last-level cache references.

- Memory related stalls (stalls in the load and store buffers as well as stalls occurring due to the reservation station being full).
- Total CPU cycles used.

Listing 3.2: Linux Perf output example

```
 Performance counter stats for './sp.C.x':

861,030,921,116 r530ea2 #memory stall cycles
2,022,464,104,458 cycles
5,001,824,565 r53412e #last-level cache misses
218.119667568 seconds time elapsed
```

The list of metrics can be extended by adding any other metrics found in the libpfm4 output.

3.2.1.2 Mutex Wrappers

A tool was created to gather data about mutex events of a program based on work described in Section 2.1.2.1. The component that gathers information about mutex events is a library that wraps the pthread_lock, pthread_trylock, and pthread_unlock functions while writing information to a trace file. Calls to the pthread functions are replaced by calls to the wrapper functions at compile time.

The wrapper has the following functionality:

- Takes a mutex as argument.
- Records a starting timestamp.
- Gets the address where the mutex function was called.
- When acquiring a mutex the wrapper function attempts to acquire the mutex by using "pthread_mutex_trylock". If this is successful no lock is holding the mutex.
- In all other cases the original pthread function is called with the mutex as argument and the return status is recorded.

- Calculates the duration of the original function call by subtracting the starting timestamp from a timestamp taken after the original function exits.
- Updates the trace by calling a function with the starting timestamp, thread ID, calling address, mutex address and event type as parameters.
- Returns the same value that the original function returned.

Using LD_PRELOAD to replace which function is called would offer better flexibility, but since libunwind, the library used to get the address where the function was called uses the pthread functions this causes infinite recursion.

Each time a mutex is acquired or released the tool records an event in a trace file. The trace file contains mutex events and a table with additional information about the event such as the calling address, mutex address, and type of event. Instead of recording this information for every event we store the information in a table containing this information only once. In every trace line we record the index where this information can be found in the table along with a timestamp read at the start of the event and the thread ID in which the event occurred. An extract from a mutex trace is shown in Table 3.3 and an example of the index table is shown in Table 3.4.

Timestamp	Thread ID	Index number
364	6324	0
418	6324	1
434	6324	0
437	6324	1
439	6326	2
455	6326	3
446	6324	0
464	6324	1
448	6327	2
483	6327	3

Table 3.3: An example of a trace containing mutex events. Timestamps are given in microseconds.

Index number	Address	Address	Event Type	Event name
0	0x408717	0x60fd60	C	trylock
1	0x40873b	0x60fd60	D	unlock
2	0x4089e7	0x60fd60	C	trylock
3	0x408a0b	0x60fd60	D	Unlock

Table 3.4: An example index table extracted from a mutex trace. Index numbers are assigned as new events are triggered. The order of the items in the table therefore makes the index number redundant. We chose to include it to improve readability and to ease debugging. Every event is recorded once and the trace size is not affected. The full event name is also included to improve readability and to ease debugging, even though the analyser uses the event type alone to identify events.

3.2.1.3 Message-passing wrappers

Wait states in a message-passing program are described in Section 3.2.1.3. A wrapper library allows us to gather information about message-passing events in a program. The library is implemented as wrapper functions around MPI message-passing functions. Each wrapper function records information about the message-passing operation and performs the operation specified by the original MPI function.

The wrapper functions are contained in the "libmpiwrap.so" library. The functions contained in this library are loaded when the program starts by pre-pending "LD_PRELOAD=libmpiwrap.so" to the program launch command. Initialisation, which includes initialisation of the event table and opening of trace files), is done inside a wrapper of the MPI_Init function. Each wrapper function adds similar functionality to the original function:

- Takes the same parameters as the MPI function it is replacing.
- Records a starting timestamp.
- Gets the address where the message-passing function was called.
- Calls the original MPI function with the "PMPI" prefix instead of "MPI". (The PMPI function calls have identical functionality to the MPI function calls. The PMPI interface is part of the MPI library. It is intended to ease the creation of wrappers for MPI functions.) All the parameters passed to

the wrapper function are passed "as is" to the original function. The return value is stored in a variable.

- Calculates the duration of the original function call by subtracting the starting timestamp from a timestamp taken after the original function exits.
- Updates the trace by calling a function with the starting timestamp, duration, calling address, event type, and optionally the message size as parameters.
- Returns the same value that the original function returned.

Information such as the address where the function was called, and the type of message-passing operation remains constant for each location in the program where message-passing functions are called. The first time that a message-passing function is called, the static information is recorded in an indexed table data structure and only the index to the information is recorded in the trace message. Writing only the index number reduces the size of each trace file. The index table of each process trace is written to the end of the trace file.

The traces are stored in text format. Each process has its own trace file. To prevent unnecessary overhead, the amount of data written to traces is kept to a minimum; such that each message is less than 30 characters in most cases. Timestamps are recorded relative to an offset, which shortens the length of the number. An example extract from an MPI trace file is shown in Table 3.5 and the corresponding index table is shown in Table 3.6.

Timestamp	Duration (μs)	Destination	Index number
585	8		0
945 969	0	1	1
946 008	3	2	1
990 707	0	1	1
990 718	213	2	1

Table 3.5: An extract from a trace for a single MPI process. Each process records events to a unique file.

Index number	Calling Address	type	name
0	0x7fbc85850db4	7	MPI_Bcast
1	0x7fbc85856345	4	MPI_Irecv
2	0x7fbc85858ff7	1	MPI_Send
3	0x7fbc85850d33	6	MPI_Barrier
4	0x7fbc858584cb	8	MPI_Reduce

Table 3.6: An index table generated from an MPI trace. Entries are created as functions are called and index numbers are assigned sequentially. The order of the entries make the index number redundant, but we record it in the table for the sake of readability. We record the MPI function name for the sake of readability. The analyser uses the "type" field to determine the MPI function type.

3.2.2 Analysis of Performance Data

3.2.2.1 Analysis of Global Program Metrics

The analyser reads output from the GMG for each test case, where a test case is a run of a specific program using m threads. The program is run twice using one CPU core and $n = m$ CPU cores. For NUMA programs a run on $n = c$ cores is required. Each run is repeated five times to provide multiple data points. The Perf output, and the trace of service time values and the run queue length is read from two separate files. Data from multiple program runs are aggregated and placed into an intermediary dictionary. This is passed to a function that performs the analysis. The data contained in the dictionary is saved to a file. A component that generates tables and a component that compares different analysis outputs takes the output files of the analysis component as input. See Section 3.2.3 for a description of these components.

The analysis component processes the data for each test case according to the model as described in Section 3.1.1, data about CPU cycle usage, memory stall cycles, cache misses, cache references, a trace of the service time, and a trace of the run-queue length is used to construct a profile.

The GMG analyser calculates additional metrics and stores the following data.

- The measured and modelled average number of active threads as shown in Listings 3.3 and 3.4 respectively.

- Total service time for $n = 1$ and $n = m$.
- The average number of work cycles for $n = 1$ and the average number of work cycles for $n = m$.
- The average number of last-level cache misses for all tests using 1 and m cores.
- The average number of last-level cache hits for all tests using 1 and m cores.
- The minimum and maximum number of CPU cycles, memory stalls, and cache misses for all tests.
- ω, the ratio of activity due to contention and activity due to useful work.
- The speed-up loss due to memory contention, calculated using the formula $\frac{averageNumberOfActiveThreads.\omega}{1+\omega}$.
- The service rate of the memory controller μ
- The arrival rate of memory requests λ when $n = m$ CPU cores are used
- L, the arrival rate of requests at a single core, calculated as $\frac{\lambda}{n}$.
- The modelled number of cycles to be used when $m = n$, given the number of cache misses, μ, λ, and δ (δ is only for NUMA systems).

The algorithm used to calculate the measured average number of active threads and the modelled average number of active threads is shown in Listing 3.3. As explained in Section 2.1.2, the average number of active threads $A(m, n) = \frac{\sum A(m,n,t)\Delta T_{cp}}{T_{cp}}$.

The service rate of memory requests μ and the arrival rate of memory requests λ are calculated using regression on the plane with the number of CPU cores n on the x-axis and $\frac{1}{C(n)}$ on the y-axis. For a UMA system data for C(1) and C(m) is required. For a NUMA program, data for C(1), C(c), and C(m) is required, where $m > c$. δ, the number of additional cycles required to transfer data between nodes in a NUMA system is calculated using the line of best fit algorithm shown in Listing 3.1, δ is calculated by regression on the plane with the number of threads on the x-axis and the number of cycles on the y-axis. Data points for c, and m are used. The slope of

Listing 3.3: Calculation of measured average number of active threads using service time traces

```
Function MeasuredAmn
  sum = 0       #A(m, infinity)
  tcp_tot = 0   #total critical path time
  for each item in data
    s = 0
    tcp = 0
    for each current_thread in item
      s = s + current_thread
      if current_thread > tcp:
        tcp = current_thread
    if tcp > 0:
      sum = sum + s
      tcp_tot = tcp_tot + tcp
  return sum/tcp_tot
end MeasuredAmn
```

the line and y-intercept are returned by the line of best fit algorithm. The standard equation for a line is used to calculate delta: $\delta = (\text{slope}.m) + \text{y-intercept}$.

The modelled number of cycles required to do the work is calculated as

$$C\_NUMA(n) = C(c) + r(n).\delta.C(m - c)$$

. The results dictionary is saved to a "savedict.out" file after all data has been loaded. We make use of a feature in Python called pickling to write the entire data structure to a file. This makes it easier to reuse the data. Loading the service time traces, Perf output and calculating derived metrics takes in the order of 40s when running the script with the "Pypy" interpreter and four minutes with the standard Python interpreter. When reusing the savedict file, the time taken to load and process the data is reduced to a second or less for both Python interpreters.

3.2.2.2 Analysis of mutex Traces

The analysing component of the mutex tool reads the trace file produced by the gathering component of the mutex tool. The tool matches each acquisition of a mutex to a release of a mutex to form a mutex event.

Listing 3.4: Calculation of modelled number of active threads using service time traces

```
Function modelledAmn
  sumAbove = 0  #{Sum of (A(m,n,t) Delta T(n))}
  sumBelow = 0  #{Sum of {Delta T(n)}}
  for each item in data
    s = 0     #sum of service times for time t
    tcp = 0   #critical path time
    for each current in item
      s = s + current
      if current > tcp then tcp = current
      if tcp > 0 then
        sumAbove = sumAbove + s;
        Aminft = s/tcp;
        sumBelow = sumBelow + (s/min(numCores, Aminft));
  return sumAbove/sumBelow
end modelledAmn
```

The tool marks all mutex locks that lie on the critical execution path of the program using Chen and Stenstrom's algorithm listed in Listing 3.5. The algorithm starts by finding and marking the last mutex event where contention occurred. This mutex event is guaranteed to be on the critical path of execution, because it is the last mutex for which there was contention. The list of mutex events is traversed in reverse order. While contention is occuring for the same mutex, all occurances of contention lie on the critical path of execution. When access is uncontended, an earlier mutex for which there was contention is found and counted on the critical path. This continues until all mutex events have been considered.

Generating a report The mutex analysis tool generates a report that summarises the mutex events. The tool reports how much time was spent in sections protected by mutex locks and how many of those mutex locks were on the critical path; this is reported in terms of number of events and time spent.

The mutex analysis tool can filter events by including or excluding events with a duration larger or smaller than a given value. This can be specified in a configuration file called "mutexTool.conf". Other filters can also be enabled by specifying it in the configuration file. Events can be filtered by specific mutex or by the address in the program where the mutex was acquired. The mutex analysis tool generates

Listing 3.5: Marking Algorithm

```
function markLocksOnCriticalPath():
current = last_mutex_with_contention()
stop = first_mutex_with_contention()

while (current != stop) {
 if (current.isContended) {
  current =
   find_previous_owner_of_current_mutex();
 } else {
  current =
   find_previous_mutex_with_contention();
 }
}
end function
```

a report of the mutex events using the filtered lists of mutex events.

3.2.2.3 Analysis of MPI Traces

The analysis component of the MPI tool reads the data generated by the MPI data gathering component and matches the entries in the traces of the processes to communication events. Based on the message type and the order in which the processes involved in the event started communicating, the MPI analyser calculates the time spent by the longest waiting process and the total waiting time for all processes. The algorithm for grouping trace messages into events and then identifying wait states is shown in Listing 3.6.

The analysing tool can ignore events that are below a user-specified time threshold. Separate thresholds are used for point to point and collective communication events. These thresholds are specified in the configuration file called “MPITool.conf”. This file contains all configuration options of the analyser component. The tool can be run multiple times while adjusting the thresholds to find a balance between the number of events that are considered and the severity of the wait states. The analyser can also report how much time is spent waiting based on the location of the event in source code. This section of the report is generated by default, but it can be disabled in the configuration file.

Listing 3.6: Algorithms for grouping MPI trace messages into events

```
global var traces
global var processNumbers

function getEventsList()
  for (processNumber in listOfProcessNumbers) {
    currentItem = trace[processNumber].removeFirstItem()
    items=[] # empty list
    if (currentItem.isP2pSender) {
      match = findP2pReceiver(origin, destination)
      items.append(match)
    else if (currentItem.isP2pReceiver) {
      match = findP2pSender(origin, destination)
      items.append(match)
    } else{ #collective event
      items.append(findCollectiveEvents(current.rank,
         current.type))
    events.append(new event(items))
  }
end function

function findP2pEvent(origin, destination, type):
  targetTrace =traces[destination]
  if (type == '0') targetTypes =['0', '2'] #receivers
  else targetTypes = ['1', '3'] #senders
  for (item in targetTrace) {
    if (item.type in targetTypes) {
      if (item.dest == origin) {
        return targetTrace.remove(item)
      } else continue
    } else continue
  }
end function
```

```

function findP2pSender(origin, destination):
  findP2pEvent(origin, destination, '1') #'1' indicates
      a sender
end function

function findP2pReceiver(origin, destination):
  findP2pEvent(origin, destination, '0') #'0' indicates
      a receiver
end function

function findCollectiveEvent(origin,, type):
  matchingItems =[]
  for (index in processNumbers) {
    if (index == origin) continue
    else {
      for (item in traces[index]) {
          if (item.type ==type) {
            match = traces[index].remove(item)
            matchedItems.append(match)
            break
      }
  }
end function
```

3.2.3 Comparing the Output of Analysis and Generating reports

The analysis component of the GMG, mutex component and MPI component stores data in a standard format. All data is stored in a Python dictionary (hash map). Each entry is stored in the dictionary using the key "programName.valueName.m.n", where programName is the name of the tested program, valueName describes the data stored in that entry (e.g. cache misses), m is the number of threads, and n is the number of cores. Where $m = n$ n is omitted. The mutex component and the MPI message component store a separate dictionary for each filter that is applied. These dictionaries have an additional entry with the key "name", denoting the type of filter that was applied (e.g. "MPI p2p wait states" or "mutex critical path". This allows for the comparison of different program configurations using multiple conditions to filter events. The mutex component and MPI component can produce multiple dictionaries, stored in an ordered list.

When the analyser is run the dictionary or list of dictionaries is written to a file. The comparison tool, report generator and table generator can open specific dictionaries. Each component has a distinct feedback script that understands only the output from the specific component (e.g. the MPI report generator only understands output from the MPI analysing component). Each component focuses on different aspects and the feedback it gives should therefore be different from other components.

The table generator can output a table with selected rows and columns, provided that all data is in the same dictionary. This is used primarily for output from the GMG. The mutex wrapper component and MPI component each generate a report detailing the specific wait states that were found. Examples are shown in Chapter 4.

The comparison component can compare any number of analyser outputs, provided that they compare the same target programs and output from the same data analysis component. The output of the comparison tool is a table. It takes a list of output file names as arguments. Each item in all files with the same key name are compared to the first output. For each value in the output a row is created in the table containing the key name followed by the key value found in the first output given as argument. This value is compared to all outputs containing the same entry. Entries are created for the compared value and the percentage of change compared to the first value. If all outputs given as arguments do not contain the same keys, no table will be created. An example is shown in chapter 4.

All tables can be produced as Latex source or plain text. All mutex and wait state reports are produced in plain text.

Chapter 4

Shared Resource Contention Analysis

In Chapter 3 the techniques used to measure resource usage and quantify the effects of contention for resources, were described. Three tools were developed to gather data during program runs, analyse it using the techniques described, and report the results of the analysis. This chapter presents a representative subset of the data and reports that were generated for a number of benchmark programs and discusses the results.

The machine used for testing, the programs used for testing, and the testing methodology are described in Section 4.1. Section 4.2 discusses the results of analysing contention for hardware resources. In Section 4.3, the effect of software contention in general and specific sources of contention for software resources are discussed. The time lost due to memory contention, wait states, and contention for mutexes are quantified. Although some sources of contention were not directly measured, such as contention for software resources outside the program or cache contention, indicators of possible causes of time loss are identified and discussed.

4.1 Testing Methodology, Test Machine, and Benchmarks

4.1.1 Machine Used For Testing

The main machine used for testing is an Intel Xeon server running Redhat Linux with kernel version 4.2.1. The machine has two nodes, each with an Intel 2640v2 CPU and a total of 256 GB of shared memory divided equally between the nodes. Each node has eight physical CPU cores that have hyperthreading enabled. The 'turbo boost' feature was disabled and each core has a clock speed of 2.0 GHZ. The last-level cache of each CPU is shared between all cores and can hold 20MB of data. It is connected to a single quad-channel memory controller. During testing, each thread or process of the benchmark program being run, was assigned to its own physical CPU core. All tests were performed when the machine was completely idle. Numactl was used to restrict which CPU cores were used to minimise the impact of the Operating System's program scheduler.

4.1.2 Performance Impact of Data Gathering Components

We designed the tools to have minimal impact on the performance of the running program being analysed. Data gathering and analysis are separated to minimise computation and the data gathering components were created with minimal resource use in mind. The service time trace component in the GMG, as well as the trace writer used in the mutex and MPI libraries were written in C and C++; and the standard C / C++ libraries were used whenever possible to read and write files. All programs and wrapper libraries are compiled with the highest optimisations. Furthermore, the chosen third party programs, Linux Perf and Numactl, are widely used and are also well optimised. See Sections 3.2.1.1, 3.2.1.2, and 3.2.1.3 for a detailed description of these components.

Components that generate data have to be active when tests are run. If the resource requirements of these components are high, the results will be affected. Keeping the demand for resources by these components to a minimum was important during design and creation of these components. The largest impact on the performance of the running program being analysed, originated from the code that generates data. No quantifiable reduction in performance was observed when no data files were generated. For the tests we ran there was no performance impact in terms of running time when information about service time or wait states was collected. For example, the CG benchmark running on sixteen processes generates 223400 events over 18.9 seconds (6.6 MB of data) without changing the running time. The components that create traces were improved by avoiding the duplication of information in the traces. Thus, the sizes of the traces were reduced. For MPI traces, the number of bytes written for each line in the trace was reduced by eight bytes to thirty bytes, and by seventeen bytes for trace lines in a mutex event trace. It is thus a total reduction of $2 * 8 = 16$ bytes for each point to point event, $numberOfProcesses * 8$ for each collective event and $2 * 17$ bytes for each mutex event.

4.1.3 Benchmarks Used For Testing

Six benchmarks from the NAS (Numerical Aerodynamic Simulation) Parallel Benchmark suite (NPB) were used. These benchmarks were chosen because they have workloads that are representative of computations done in the field of science. A message-passing variant using MPI and a shared-memory variant using OpenMP were used. The shared memory variants were also chosen because they satisfy the requirements for accurately measuring memory contention. The NPB benchmarks are a collection of programs that perform computation frequently found in computational fluid dynamics (CFD). Refer to the original NPB specification for more information [19]. The EP, CG, FT, and IS benchmarks are small kernels that perform specific computation and the BT, and SP programs emulate real-world problems found in fluid dynamics. Each of the six benchmarks is described briefly. Descriptions are quoted from the NPB specification.

- "EP: An embarrassingly parallel kernel. It provides an estimate of the upper achievable limits for floating point performance, i.e., the performance without significant interprocessor communication."
- "CG: A conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix.

This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication."

- "FT: A 3-D partial differential equation solution using FFTs (fast Fourier transforms). This kernel performs the essence of many spectral codes. It is a rigorous test of long-distance communication performance."
- "IS: A large integer sort. This kernel performs a sorting operation that is important in particle method codes. It tests both integer computation speed and communication performance."
- "BT: A block Tri-diagonal solver."
- "SP: A scalar Penta-diagonal solver"

The NPB benchmarks do not make use of mutexes. To test the mutex tool a benchmark from the Parsec benchmark suite called Dedup, and two versions of a synthetic benchmark we created were used. The dedup program is described by the authors as a program that "compresses a data stream with a combination of global and local compression that is called 'deduplication'. The kernel uses a pipelined programming model to mimic real-world implementations." Two versions of a synthetic benchmark, called Sharequeue, were implemented. Sharequeue is a micro-benchmark that we created to demonstrate contention for mutex locks. Both versions repeatedly remove a value from a shared queue, compute a new value, and insert the new value into the queue. Access to the queue should be guarded to avoid race conditions. The first version has a single mutex protecting both the head and tail of the queue and the second version has two mutexes, one protecting the head of the queue and another protecting the tail of the queue.

4.1.4 Deliberately Induced Contention

To test the impact of memory contention, tests were performed while deliberately causing contention, but keeping all conditions under which the benchmark operates the same. While the way in which multi threaded programs execute is unpredictable due to the non-deterministic order of execution, each test was run five times to get a representative sample of a typical test run. Several techniques can be used to induce contention. Software contention can be induced by either modifying the target program to use the shared resources differently by either increasing the amount of time that resources are held (by inserting additional computation or idleness) or by increasing the number of requests for the resource. Any changes to the data access patterns may have additional effects on performance that are not related to contention. Creating idleness would simulate the effect of how a thread has to wait for access to a mutex, but the duration of the idleness often varies when real software contention occurs. The variations differ from program to program and depend on the data access patterns. For this reason we chose not to induce contention for software resources. To induce hardware contention, the demand for a hardware resource can be increased, for example running multiple threads on the same physical CPU core or by running a program that generates memory requests. Since our aim was to measure memory contention, inducing contention for other shared resources such as shared execution units among CPU cores was not tested.

Memory contention can be induced by a program independent of the affected program. As part of testing memory contention, contention for access to memory was deliberately induced. We used a modified version of the "stream" benchmark for this purpose and called it the MCG (memory contention generator). Values are streamed directly from memory, bypassing the caches using the MMX instruction set found in newer Intel CPU's. The MCG directly streams data to and from memory and no space is used in the last-level cache after initialisation. This means that only memory contention is induced and no cache contention. This creates contention for access to the memory controller, contention for memory bandwidth, and row buffer misses. There is no contention for CPU cores or execution units. The threads for the MCG are bound to their own CPU cores on the same node as the benchmark. When the MCG is used to induce memory contention while a benchmark program is run, the benchmark program never competes for CPU resources with the MCG, as it is assigned to its own group of cores.

The MCG was tested on two machines, an Intel 2400 CPU with 4 GB of main memory and the Xeon server used in the other tests. It was found that the effect of memory contention was less pronounced on the Intel Xeon machine than on the

Intel Ivy Bridge machine. When an unmodified version of the Stream benchmark, a memory-bound benchmark used to test cache contention and streaming of data was run in conjunction with the MCG, performance degraded substantially for the Intel 2400 machine (running time doubled), whereas it only increased by 9.75% on the machine used for testing. This indicates that the effect of memory contention is less pronounced on the Xeon machine. On the Xeon machine, requests were generated at a rate of 5 596 047 memory requests per second on average.

4.1.5 Test Automation

A script is used to automatically run multiple tests in sequence. Each test is run with a set of tools enabled and a specified number of processors and threads. Each test is repeated five times to avoid the effects of performance fluctuations. Unless specified, the running test was the only job running on the machine. The data for each run of each test is placed into its own directory by the script. The analysing components traverse these directories when analysing the data.

The Xeon machine used for testing consists of two nodes, each containing eight physical CPU cores. Selected benchmarks are executed using two, four, and eight threads with all threads bound to its own CPU core on the same node in order to test UMA workloads. NUMA workloads are tested for all benchmarks using nine and sixteen threads using a fill socket first policy with each thread allocated to its own physical CPU core.

After the raw data for the GMG, the mutex tool, and the MPI tool have been processed, the metrics are stored in a python dictionary that has a standard format. All values that may be compared have the same key name (e.g. BT.time.4.4) where the two numbers respectively denote the number of threads and processors. In this way two or more metrics from different tests may be compared. The comparing and table generating tools described in Section 3.2.3 read the output from the analysis components and reports the results.

4.2 Profiling the Contention for Hardware Resources

Using shared hardware resources allows easier sharing of data and each thread or process can use as many resources as it requires leaving the rest free for other threads/processes to use. However when there is a high demand for shared hardware resources, contention for resources results in stall cycles and an increase in the number of work cycles required to do the same amount of work.

4.2.1 Memory-bound vs Compute-bound Programs

Resource contention only occurs when there is competition for the available resources. It is therefore useful to have a metric of the demand for the shared resources. Three metrics can be derived for this purpose. Work rate measures the demand for computing resources and is expressed as the number of cycles per second. The work rate is not a useful metric, because it is affected by the demand for both memory and CPU resources. In essence the work rate only expresses activity. For this reason work rate is not used. Miss rate expresses the demand for memory resources as the number of last-level cache misses (memory requests) per second. While the work rate expresses the rate of activity and miss rate expresses resource demand for only memory resources, the number of work cycles per cache miss (CCM) express the activity in relation to the number of memory requests. Thus, CCM does not take idleness into account, whereas miss rate takes idleness into account. Table 4.1 shows the miss rate and CCM values. These values express how memory-bound a program is.

The CG benchmark has the smallest number of work cycles between each cache miss, indicating a lower ratio of computation to memory requests, where as the EP benchmark has the highest number of work cycles between cache misses. A low miss rate goes along with a higher CCM value for the more compute-bound programs. However, since the miss rate does not account for the burstiness of requests, idle periods or distinguish between misses due to contention and misses not due to contention, miss rate is not as good an indicator of how memory or compute-bound a program is.

Number of cache misses per second					
Threads/ Program	2	4	8	9	16
CG	70 823 000	133 615 000	289 470 000	251 755 000	402 707 000
SP	10 655 000	27 783 000	60 948 000	61 944 000	75 279 000
FT	7 600 000	15 068 000	30 087 000	33 162 000	53 648 000
IS	7 553 000	15 078 000	31 004 000	31 175 000	56 962 000
BT	5 546 000	11 712 000	23 960 000	25 481 000	44 857 000
Dedup	3 571 000	5 247 000	4 979 000	5 848 000	9 327 000
EP	811 000	1 860 000	63 000	74 000	100 000
Average number of cycles between cache misses (CCM)					
Threads/ Program	2	4	8	9	16
CG	79.382	84.133	77.382	97.198	110.121
SP	611.435	491.651	455.503	496.946	740.546
FT	884.061	895.460	905.510	921.821	1 036.826
IS	965.329	969.463	939.175	1034.990	1 020.193
BT	1 193.051	1 133.393	1 107.542	1162.493	1 171.368
Dedup	1 329.100	1 302.380	1 845.095	1484.624	1 195.012
EP	7 332.795	6 406.132	377 493.678	36 1393.616	475 655.092

Table 4.1: The average number of cycles between cache misses (CCM) and cache misses per second (which expresses the number of memory requests generated per second). Both express how memory-bound a program is. CCM excludes idle time and miss rate does not. Lower CCM values and higher miss rate values mean a program is more memory-bound. These values are used to create a profile of memory-boundness.

4.2.2 The Effect that Memory Contention has on CPU cycles

When memory contention occurs, there is an increase in the time that threads have to wait for operands as well as an increase in the number of work cycles. Contention for space in the cache results in an increase in the number of memory requests as data is evicted from the cache and retrieved again. In turn an increase in the number of memory requests will increase the demand for memory resources.

Program	Number of threads, m	Useful Work: the number of work cycles when $n = 1$ ($\times 10^9$)	The number of work cycles when $n = m$ ($\times 10^9$)	The % change in the number of work cycles for $n = m$, from $n = 1$
CG	2	910	1 066	17
CG	4	911	1 131	24
CG	8	913	1 039	14
CG	9	913	1 315	44
CG	16	916	1 492	63
SP	2	2 503	2 802	12
SP	4	2 573	4 058	58
SP	8	2 610	5 235	101
SP	9	2 612	5 464	109
SP	16	2 633	9 934	277
FT	2	222	230	4
FT	4	220	242	10
FT	8	219	249	14
FT	9	219	260	19
FT	16	221	308	39
IS	2	160	168	5
IS	4	160	171	7
IS	8	162	163	1
IS	9	162	184	14
IS	16	164	194	18
BT	2	4 206	4 378	4
BT	4	4 203	4 595	9
BT	8	4 211	4 620	10
BT	9	4 199	4 831	15
BT	16	4 230	5 537	31
Dedup	2	119	126	6
Dedup	4	118	133	12
Dedup	8	124	152	22
Dedup	9	129	160	24
Dedup	16	136	199	46
EP	2	997	1 002	0.50
EP	4	990	998	1.81
EP	8	991	987	-0.40
EP	9	991	988	-0.30
EP	16	992	992	0.00

Table 4.2: This table lists the number of work cycles without contention (where the number of cores, $n = 1$), the number of work cycles with contention (where the number of cores, $n = m$), and the percentage change between the two values.

Program	Number of threads, m	Number of cache misses for $n = 1$ ($\times 10^9$)	Number of cache misses for $n = m$ ($\times 10^9$)	The % change in the number of cache misses between columns three and four
CG	2	13 399 219	13 424 901	0.19
CG	4	13 400 949	13 439 020	0.28
CG	8	13 404 266	13 429 481	0.19
CG	9	13 404 573	13 527 266	0.92
CG	16	13 404 274	13 551 729	1.10
SP	2	4 183 629	4 582 213	10
SP	4	4 698 069	8 253 739	76
SP	8	5 236 527	11 492 933	119
SP	9	5 245 758	10 994 850	110
SP	16	5 316 514	13 415 048	152
BT	2	3 598 653	3 669 591	2
BT	4	3 658 981	4 054 480	11
BT	8	3 708 829	4 171 415	12
BT	9	3 720 587	4 156 028	12
BT	16	3 870 276	4 726 966	22
FT	2	254 204	260 372	2
FT	4	262 855	270 439	3
FT	8	286 076	274 976	-4
FT	9	290 604	281 829	-3
FT	16	302 350	297 129	-2
IS	2	170 493	174 339	2.26
IS	4	171 099	176 141	2.95
IS	8	174 556	174 047	-0.30
IS	9	176 804	177 628	0.47
IS	16	183 050	190 373	4.00
Dedup	2	72 802	94 776	30
Dedup	4	77 277	101 806	32
Dedup	8	79 590	82 651	4
Dedup	9	81 910	108 033	32
Dedup	16	85 520	166 768	95
EP	2	4 654	136 590	2 835
EP	4	5 099	155 863	2 957
EP	8	6 695	2 615	-61
EP	9	6 970	2 735	-61
EP	16	15 576	2 086	-87

Table 4.3: This table lists the number of cache misses without contention (where the number of cores, $n = 1$), the number of cache misses with contention (where the number of cores, $n = m$, and the percentage change between the two values.

Tables 4.2 and 4.3 shows how contention influences the total number of work cycles and cache misses. The third column of each table shows the values without contention (for the number of cores (n) = 1) and the fourth column shows the values with contention (for n = m; the number of threads (m) is shown in the second column). Most notably, the CG benchmark, which is the most memory-bound benchmark, shows an increase in the number of work cycles of between 17% (for $n = 2$) minimum and 24% (for $n = 4$) maximum for UMA tests and 44% (for $n = 9$) and 63% (for $n = 16$) for NUMA tests, while the number of cache misses increases at most by 1.1% for the test with 16 threads. Given that the workload remains constant, the only source of these additional work cycles is memory contention.

In contrast, the programs that are compute-bound such as the EP benchmark show small changes in the number of work cycles, even when there is a drastic change in the number of cache misses between tests on 1 and m CPU cores (changes in the number of cache misses vary from −87% to 2957%). The variation in the number of cache misses does not affect the total service time that the program receives or the total work cycles substantially. This supports the claim by Tudor and Teo that the only source of additional work cycles for a constant workload are cycles spent on memory contention [17]. However, if the total number of cache misses vary, the total number of "useful" cycles may be under-estimated when there are fewer cache misses and over-estimated when there are more cache misses. This should be taken into account when evaluating results, especially for memory-bound programs. An example of this is the results for the SP benchmark where the number of cache misses vary for tests with more than eight threads and cores, compared to tests on a single core.

Cache contention is not explicitly measured in this thesis and we can therefore only say that we suspect cache contention to be an additional source of performance loss based on the changes in the metrics that were observed.

4.2.3 Measuring the effect of Memory Contention

The degree to which programs contend for memory depends on three factors: the rate at which memory requests arrive, the rate at which memory requests are processed, and the number of row buffer misses that occur. Current hardware technology only allows us to get precise information about the number of memory requests, while the change in the number of work cycles provides an indication of when the number of memory requests cannot be processed as fast as they arrive. Currently there is no way to count row buffer misses for the machine that was used.

The method for calculating speed-up loss due to memory contention as proposed by Tudor and Teo, and described in Section 3.1.1, is used to quantify memory contention for seven benchmarks. Two requirements of this method for calculating memory contention are that the number of memory requests remain relatively constant, and that the total amount of work does not fluctuate for runs of the same program using the same number of threads and cores. The number of work cycles and cache misses were measured for different runs of several benchmarks and Table 4.4 shows the variation, expressed as the relative standard deviation percentage, in the number of work cycles and cache misses for the tests. Values closer to 0 indicate less variation. Most notably the number of cache misses vary for the EP and SP benchmarks. Since the EP benchmark is compute-bound this does not have an effect on U, the useful work, but for the SP benchmark which is memory-bound U may be under-estimated.

The tool that implements the model proposed by Tudor and Teo, described in Section 3.1.1, is used to calculate factors of speed-up loss due to memory contention and all other (software) dependencies. Table 4.5 shows the factor of speed-up loss due to contention. Column three shows the factor of speed-up loss calculated using Equation 3.3. Column four shows the factor of speed-up loss calculated using the formula shown in Equation 3.8. In columns five and six the respective speed-up loss factors due to contention for both memory and other dependencies are multiplied with the service time on a single core to calculate how much time was lost. In general, speed-up loss is lower for a lower number of threads. Using a second node increases the contention for both data and memory, because access to memory in a remote node takes longer and as a consequence computation takes longer (requiring mutexes in protected sections to be held longer). The programs with larger speed-up loss factors due to memory contention such as CG and SP are memory-bound as shown in Table 4.1. On the other hand, the compute-bound programs such as EP have a low speed-up loss factor due to memory contention. The Dedup program has the largest speed-up loss factor due to other dependencies compared to the

Program	Number of threads, m	Standard deviation for the number of work cycles C(1)	Standard deviation for the number of work cycles C(m)	Standard deviation for the number of cache misses
BT	2	0.686	0.312	1.088
BT	4	0.282	0.542	1.744
BT	8	0.270	0.336	5.876
BT	9	0.166	0.917	5.539
BT	16	0.134	0.597	9.966
CG	2	0.027	0.040	0.026
CG	4	0.033	0.112	0.003
CG	8	0.039	0.173	0.094
CG	9	0.021	0.154	0.456
CG	16	0.020	0.307	0.547
EP	2	0.926	0.056	33.229
EP	4	0.053	0.027	49.672
EP	8	0.041	0.022	44.080
EP	9	0.059	0.062	43.721
EP	16	0.122	0.026	76.836
FT	2	0.540	0.522	0.594
FT	4	1.036	0.558	1.174
FT	8	0.169	0.316	1.994
FT	9	0.190	0.942	1.568
FT	16	0.281	0.331	0.918
IS	2	0.004	0.217	0.064
IS	4	0.011	0.199	0.022
IS	8	0.028	0.245	0.161
IS	9	0.043	0.051	0.243
IS	16	0.027	0.089	1.961
SP	2	0.200	0.355	1.286
SP	4	0.881	0.175	3.746
SP	8	0.067	0.486	37.398
SP	9	0.156	0.457	35.400
SP	16	0.099	0.500	43.237
Dedup	2	0.896	1.632	1.206
Dedup	4	0.414	2.054	1.377
Dedup	8	0.578	0.391	1.926
Dedup	9	0.819	0.383	13.764
Dedup	16	0.372	0.288	32.207

Table 4.4: The variation, expressed as the relative standard deviation percentage, in the number of work cycles and cache misses for all the benchmarks (Note that less variation is better.)

other programs. More time is spent being idle, as seen in Table 4.8.

4.2.4 Effects of Induced Memory Contention

A program, called the memory contention generator, was used to generate a constant amount of memory requests each second. The MCG reads and writes data directly to and from memory in order to only test contention for memory. This program is fully described in Section 4.1.4. Results for tests with and without the MCG enabled are compared in Tables 4.6 and 4.7.

The contention generator does not share any resources with the benchmarks apart from the resources used to retrieve data from the main memory. For each of the programs, there is a noticeable increase in the total number of work cycles and service time when memory contention is induced. The effect is larger for memory-bound benchmarks such as SP and CG. SP and CG will have to be studied in more detail to clarify why induced memory contention has a larger effect on SP compared to CG. We suspect cache contention among the threads of the SP program, coupled with longer and more frequent stalls to access memory to be a contributing factor, because the number of cache misses increased for SP.

4.3 Profiling the Contention for Software Resources

Recall, from Section 3.1 that a thread or process can either be active and receiving service or idle while waiting for a resource. When it is active, time is either spent on useful work or contention. Table 4.8 shows the service time with no contention, the actual service time on $n = m$ cores, the ideal service time (that is the service time that would have been achieved if ideal speed-up was achieved), and the difference between the actual service time and the ideal service time.

Program	Number of threads	Speed-up loss due to memory contention (ms)	Speed-up loss due to other dependencies (ms)	Time lost due to Memory Contention (s)	Time lost due to other dependencies (s)
BT	2	0.74	0.53	88.89	63.66
BT	4	3.15	2.97	378.61	356.97
BT	8	5.02	11.26	602.88	1 352.27
BT	9	7.59	14.38	911.90	1 727.69
BT	16	23.59	63.96	2 678.31	7 261.76
CG	2	2.73	0.52	83.61	15.93
CG	4	7.02	2.85	215.40	87.45
CG	8	6.93	11.06	213.00	339.94
CG	9	17.72	15.89	544.93	488.65
CG	16	41.01	66.45	1 185.43	1 920.79
EP	2	0.08	0.59	2.53	18.62
EP	4	0.33	2.94	10.40	92.63
EP	8	-0.24	10.43	-7.57	328.82
EP	9	-0.18	12.31	-5.69	389.14
EP	16	-0.02	55.35	-0.59	1 642.10
FT	2	0.70	0.55	4.35	3.42
FT	4	3.34	2.98	20.68	18.45
FT	8	6.96	11.19	42.98	69.09
FT	9	9.02	15.62	55.79	96.62
FT	16	28.06	56.42	163.28	328.31
IS	2	0.94	0.57	3.90	2.37
IS	4	2.25	2.49	9.33	10.33
IS	8	0.66	8.29	2.75	34.56
IS	9	6.92	12.41	28.97	51.95
IS	16	16.26	49.20	64.35	194.72
SP	2	2.00	0.53	145.84	38.65
SP	4	13.07	2.94	968.23	217.80
SP	8	27.86	12.27	2 106.14	927.58
SP	9	29.76	18.03	2 253.68	1 365.39
SP	16	64.36	67.99	4 622.54	4 883.26
Dedup	2	0.95	5.79	7.23	44.05
Dedup	4	3.06	21.95	27.79	199.37
Dedup	8	7.47	59.98	93.11	747.64
Dedup	9	6.37	74.55	86.78	1 015.64
Dedup	16	12.00	136.04	146.50	1 660.78

Table 4.5: Speed-up loss due to memory contention and other dependencies.

Program	Number of threads	Number of cycles without induced memory contention $\times 10^9$	Number of cycles with induced memory contention $\times 10^9$	Change(%)
BT	2	4 213	4 378	+3.910%
BT	4	4 319	4 595	+6.404%
CG	2	915	1 066	+16.488%
CG	4	949	1 131	+19.169%
Dedup	2	123	126	+2.513%
Dedup	4	132	133	+0.642%
EP	2	989	1 002	+1.267%
EP	4	987	998	+1.206%
FT	2	223	230	+3.226%
FT	4	225	242	+7.448%
IS	2	160	168	+5.050%
IS	4	161	171	+6.031%
SP	2	2 533	2 802	+10.625%
SP	4	2 858	4 058	+42.008%

Table 4.6: Comparison of the total number of cycles measured during execution of each benchmark program without induced memory contention and with induced memory contention.

Program	Threads (m)	Service time without memory contention ($n = 1$)	Service time with memory contention ($n = m$)	Change(%)
BT	2	1 270.48	1 323.12	+0.041%
BT	4	1 309.15	1 384.57	+0.058%
CG	2	324.07	379.06	+0.170%
CG	4	339.38	402.23	+0.185%
Dedup	2	70.30	74.20	+0.055%
Dedup	4	84.90	84.06	-0.010%
EP	2	329.07	335.55	+0.020%
EP	4	329.07	335.55	+0.020%
FT	2	65.97	68.47	+0.038%
FT	4	67.25	71.71	+0.066%
IS	2	43.62	46.13	+0.057%
IS	4	44.23	46.66	+0.055%
SP	2	776.41	860.05	+0.108%
SP	4	874.18	1 188.19	+0.359%

Table 4.7: Comparison of the total service time measured during execution of each benchmark program without induced contention and with induced contention. The time measurements are given in seconds.

Program	m	Total service time for m threads on 1 core	Total service time for m threads on m cores	% difference between columns 3 and 4	Ideal running time for $n = m$	Actual running time for $n = m$	% difference between actual and ideal running time
BT	2	1 201.20	1 323.12	10.15	600.60	661.61	10.16
BT	4	1 201.93	1 384.57	15.20	300.48	346.17	15.21
BT	8	1 200.95	1 392.61	15.96	150.12	174.10	16.97
BT	9	1 201.45	1 467.69	22.16	133.50	163.10	22.18
BT	16	1 135.36	1 580.31	39.19	70.96	105.38	49.50
CG	2	306.28	379.06	23.76	153.14	189.56	24.78
CG	4	306.83	402.23	31.09	76.71	100.58	31.12
CG	8	307.36	371.00	20.70	38.42	46.39	20.75
CG	9	307.52	483.41	57.20	34.17	53.73	57.25
CG	16	289.06	504.45	74.52	18.07	33.65	86.27
EP	2	315.66	335.55	6.30	157.83	168.48	6.75
EP	4	315.07	334.90	6.29	78.77	83.78	6.37
EP	8	315.27	331.38	5.11	39.41	41.44	5.15
EP	9	316.11	331.08	4.74	35.12	36.86	4.95
EP	16	296.68	312.90	5.47	18.54	20.96	13.03
FT	2	62.18	68.47	10.13	31.09	34.26	10.19
FT	4	61.91	71.72	15.84	15.48	17.95	15.96
FT	8	61.75	72.95	18.15	7.72	9.14	18.41
FT	9	61.86	76.32	23.38	6.87	8.50	23.65
FT	16	58.19	82.79	42.28	3.64	5.54	52.30
IS	2	41.51	46.13	11.12	20.76	23.08	11.22
IS	4	41.48	46.66	12.48	10.37	11.68	12.65
IS	8	41.69	44.74	7.33	5.21	5.61	7.73
IS	9	41.86	51.10	22.09	4.65	5.70	23.51
IS	16	39.58	49.84	25.93	2.47	3.34	35.11
SP	2	729.26	860.05	17.95	364.59	430.07	17.96
SP	4	740.80	1 188.19	60.39	185.20	297.08	60.41
SP	8	755.97	1 508.30	99.52	94.50	188.57	99.55
SP	9	757.29	1 597.16	110.91	84.14	177.50	110.95
SP	16	718.23	2 672.37	272.08	44.89	178.20	296.98
Dedup	2	76.08	74.20	-2.48	38.04	26.54	-30.23
Dedup	4	90.83	84.06	-7.45	22.71	19.40	-15.55
Dedup	8	124.65	123.89	-0.61	15.59	16.60	7.55
Dedup	9	136.24	145.18	6.57	15.14	18.47	22.04
Dedup	16	122.08	169.50	38.85	7.63	17.88	134.34

Table 4.8: Service time using 1 and $n = m$ cores (where $m =$ the number of threads and $n =$ the number of CPU cores), and the ideal and actual speed-up on $n = m$ cores. The service time values are given in seconds.

4.3.1 Detecting Contention for Mutexes On and Off the Critical Path

Mutexes are used to protect shared data in a shared-memory program. Shared data is only accessed by a thread when it has acquired the mutex protecting the data. A thread can only acquire a mutex when that mutex is not held by another thread. When there is competition for a mutex, it leads to idleness while threads wait to acquire the mutex. When a thread is idle no useful work is being done. A side-effect of this is a reduction in hardware contention since idle threads do not fetch data.

Mutex event summary			
Program duration	14 770		
Number of mutex events	991 471	100%	of all events
Total combined time spent in protected sections; overlapping mutex events on the same thread are counted for every event	153 494	100%	of total time
Total time spent in protected sections; overlaps are only considered once	151 800	100%	of the total duration of all mutex events
Time spent waiting to acquire a mutex	38 502	25.36%	of the duration of all mutex events
The average mutex duration; overlaps are counted once	0.153		

Table 4.9: An example report for the Dedup benchmark using eight threads and eight CPU cores. Times are given in milliseconds.

To quantify the waiting times caused by contention for mutexes, data is gathered about how mutexes are used. Metrics are gathered for each mutex event. This includes the time it took for the mutex to be acquired, how long the mutex was held, and whether the mutex is on the critical path or not. The Mutex tool summarises these metrics in a report. An example of a mutex report is shown in Table 4.9. The report is for the Dedup benchmark using eight threads and eight CPU cores. All times for mutex reports are reported in units of 10^{-6} seconds, except when specified differently. The "Time spent waiting to acquire a mutex" is the time that was spent idle and waiting for a mutex to be acquired. The other time measurements include this idle time and additionally measures the time spent in a protected section. Note that the "Total combined time spent in protected sections" counts the time spent in nested mutex events for every event in the nested section. The "Total time spent in protected sections" only counts time for the most inner mutex of the nested section and time measurements are not overlapped.

4.3.1.1 Contention on the Critical Path

The algorithm shown in Listing 3.5 is used to find contention for mutexes on the critical path. All idleness and extra work cycles that lie on the critical path of execution of a program directly contribute to the total running time. If these inefficiencies are removed, execution time will be reduced proportional to the time spent on these inefficiencies. For this reason there is a larger incentive to remove contention on the critical path compared to contention in general.

Table 4.10 reports the contention for mutexes that lie on the critical path for the Dedup program using eight threads and eight cores. Note that not all contention lies on the critical path (here only items on the critical path are shown, compared to Table 4.9.

4.3.1.2 Reporting Time Lost Due to Mutex Contention

Table 4.5 reports the factor of speed-up loss and time lost due to sources of contention. Specifically, values in column four represent the factor of speed-up loss due to software dependencies and values in column six show the total time lost due to software dependencies. The mutex report contains specific time values that represent how much time of the total duration of the program was spent waiting to acquire mutexes. The total speed-up loss due to software dependencies, the speed-up loss due to contention for mutexes in the program, and the speed-up loss that is not attributed to a specific source are shown in Table 4.11. Data is only gathered for mutexes inside the program and not for external libraries or operating system functions. The time that cannot be accounted to a specific source is shown in the

A mutex report for the Dedup program			
Events on critical path	69 565	7.016%	of selected mutex events
Total Critical path time spent in protected sections; overlapping mutex events on the same thread are counted for every event	15 357	10.01%	of total time
Total Critical path time spent in protected sections; overlaps are only considered once	14 905	9.82%	of the total duration of all mutex events
Time spent waiting to acquire a mutex that is on the critical path	3 234	2.13%	of the duration of all mutex events
The average mutex duration; overlaps are counted once	0.214		

Table 4.10: An example of a mutex report for the Dedup benchmark using eight threads and eight CPU cores. Only items on the critical path of execution are reported. Times are given in milliseconds.

Program	Number of threads, m	Total time lost due to all software dependencies	Total time Lost due to waiting for mutexes	Total time lost due to unaccounted-for dependencies
Dedup	2	44.05	23.26	20.79
Dedup	4	199.37	26.34	173.03
Dedup	8	747.64	38.50	709.14
Dedup	9	1 015.64	443.28	572.36
Dedup	16	1 660.78	96.88	1 563.89

Table 4.11: Source of speed-up loss due to software dependencies. Times are given in units of seconds. Most of the speed-up loss in this program is not due to contention for mutexes inside the program.

last column. For the Dedup benchmark, most of the speed-up loss due to other dependencies is not due to contention for mutexes in the program.

4.3.1.3 Reporting Contention Based on Specific Mutexes

During analysis, mutex events can be filtered based on which mutex was used. The mutex tool can be set to report mutex events based on which individual mutex was used, so that it is clear in which mutexes the program spends most of its time and where the most contention for mutexes occurs. These reports guide a user when improving the efficiency of a program. It indicates which sections of code can benefit from optimisations (the more time was spent inside a specific protected section, the larger the potential for improving the overall performance of the program by improving that code). When a program spends time waiting to acquire a specific mutex, performance can be improved by reducing the demand for that mutex or reducing the time that a mutex remains locked. A report for the Dedup program using eight threads and cores are shown in Table 4.12 and events are separated based on the mutex involved in the event. Only two of the five mutexes in the program are shown for the sake of brevity. The first mutex (memory address 0xa6ccf0) is used in several places in the program, but less time was spent in protected sections, and less time was spent on the critical path, compared to the second mutex (memory address 0x7f2638e6a978). Of the 3.5 seconds spent on the critical path 3.2 seconds was spent waiting to acquire the second mutex.

4.3.1.4 Quantifying Changes in the Amount of Contention for Mutexes

In this section we compare the contention for the mutexes of the two versions of the shared-queue program, described in Section 4.1.3. Using data from mutex analysis the performance improvement can be quantified. Both versions of the benchmark remove and add the same number of work items from the shared-queue, but the “Queue-single” version has a single mutex protecting the shared data queue, while the “Queue-two” version has two mutexes protecting the head and the tail of the shared queue respectively. Thus, in the Queue-single version the same mutex is requested for inserts and removals, while in the Queue-two version inserts and removals are guarded by different locks. However, the total number of mutex requests is the same for both versions. The two benchmarks are identical in all other respects.

Description	Mutex (0xa6ccf0) used at **encoder.c:** Fragment:1171 **queue.c:** terminate:42 dequeue:82 enqueue:112 isTerminated:34 **queue.h:** ringbuffer_ isEmpty:61	Mutex (0x7f2638e6a978) used at **encoder.c:** Deduplicate:512, Deduplicate:489
Number of mutex events (% of all events)	83 209 (8.39%)	369 950 (37.31%)
Total combined time spent in protected sections; overlapping events on same thread counted for every event (% of total time)	24 570 775 (16.01%)	23 796 706 (15.50%)
Total time spent in protected sections; overlaps only considered once (% of duration of all mutex events)	24 569 558 (16.19%)	23 796 706 (15.68%)
Time spent waiting to acquire a mutex (% of duration of all mutex events)	20 431 880 (13.46%)	18 066 678 (11.90%)
The average mutex duration; overlaps counted once	295.275	(64.32%)
Events on critical path (% of selected mutex events)	10 051 (1.014%)	45 592 (4.60%)
Total Critical path time spent in protected sections; overlapping events on same thread counted for every event (% of total time)	491 359 (0.320%)	3 526 654 (2.30%)
Total Critical path time spent in protected sections; overlaps only considered once (% of duration of all mutex events)	491 359 (0.324%)	3 526 654 (2.32%)
Time spent waiting to acquire a mutex that's on the critical path (% of duration of all mutex events)	1 374 (0.001%)	3 232 861 (2.13%)
The average mutex duration; overlaps are counted once	48.887	77.352

Table 4.12: An example of a mutex report for the Dedup benchmark using eight threads and eight CPU cores. All times are given in microseconds. Events are separated based on the mutex involved in the event. There are five mutexes, but the report for only two mutexes are shown here for the sake of brevity.

In the Queue-single benchmark only one operation on the shared queue is possible at any time. The queue-two benchmark allows simultaneous operations on the head and the tail of the queue. In this way contention for mutexes is reduced. Figure 4.1 compares the total time spent waiting for the single mutex of the Queue-single benchmark and the total time spent waiting for the two mutexes of the Queue-two benchmark, using four threads. Figure 4.2 also compares the time spent waiting for the mutexes of the two benchmarks, but counts the total duration of events that lie on the critical path. The waiting time for events in the Queue-two benchmark is mostly shorter than for the Queue-single benchmark and the Queue-two benchmark spent less time waiting on the critical path compared to the Queue-single benchmark. This shows a large reduction in the amount of contention for the Queue-two benchmark.

4.3.2 Detecting Wait States in Message-passing programs

Recall from Section 3.1.2.2 that blocking message-passing operations introduce delays in running processes. Data is shared between the processes of message-passing programs by copying it from the address space of the sending process to the address space of the receiving process. Collective communication operations may have multiple sending processes or receiving processes. The data can only be copied once the sender is ready to transmit the data and the receiver is ready to receive the data. When any of the processes involved in a message-passing operation is not ready, other processes that are ready have to wait. This is known as a wait state.

An example of a general report of message-passing operations for the CG benchmark using sixteen processes is shown in columns two and three of Table 4.13. The majority of events complete faster than 0.005 seconds. When events that complete in less time than 0.005 seconds, are excluded, and the tool distinguishes between late senders, late receivers, and collective wait states, a report as shown in columns four and five of Table 4.13 is generated; this report shows data for the CG benchmark using sixteen processes. Table 4.13 refers to “Late time” and “Delay time”. Late

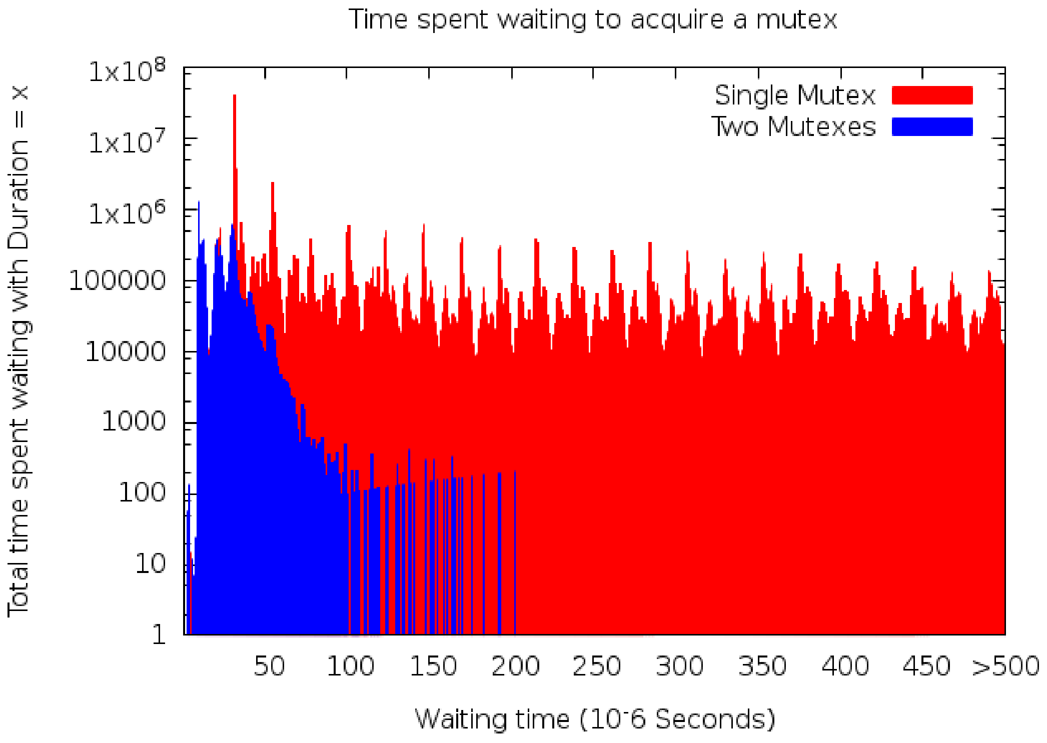

Figure 4.1: The total time spent waiting to acquire mutexes for each benchmark is expressed. The y-axis (scaled to log base ten) indicates the total amount of time spent waiting for wait-events that had a duration of x10^{-6} seconds. The x-axis shows the duration of individual events. E.g. y 10^{-6} seconds was spent waiting in total for all events that had a duration of x 10^{-6} seconds.

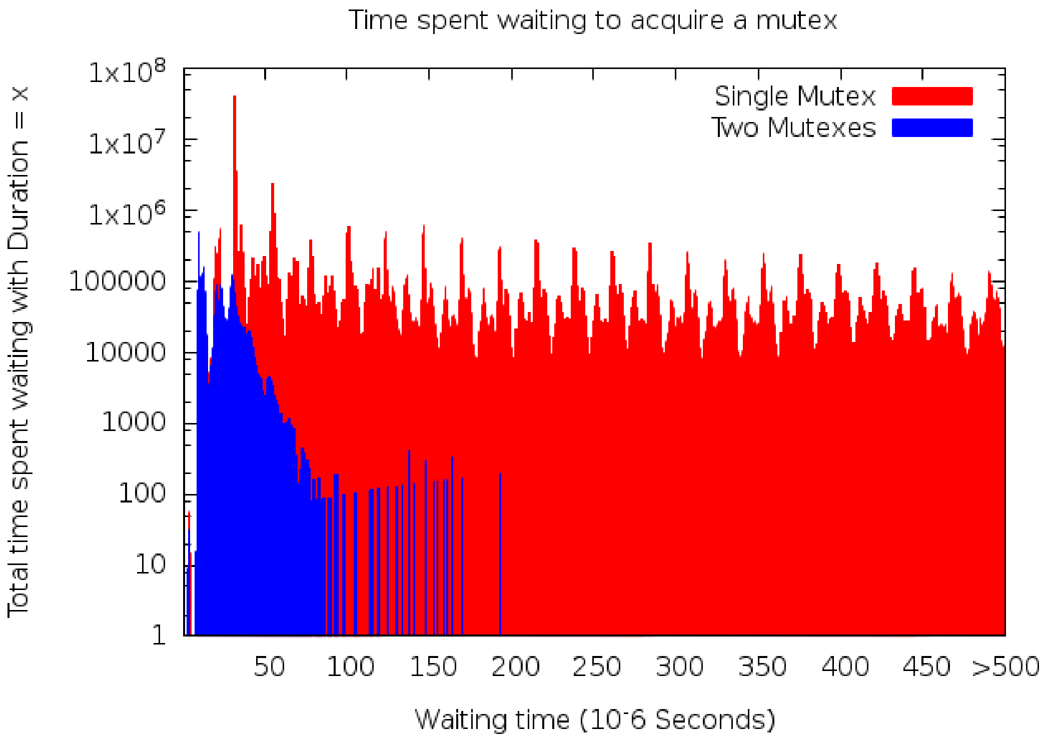

Figure 4.2: This figure is similar to Figure 4.1, but contains only the total waiting times for events that lie on the critical path of execution.

Trace Summary (All times are given in μs)				
	P2P	Collective	Late Sender	Late Receiver
Number of events (for the late senders/receiver events, only events that took longer than the threshold of 0.005s were counted)	223 744	3	603	774
Total duration of events	30 791 622	129	1 009 775	1 106 635
Total duration of events as a % of total running time for all processes	9.97%	0%	0.327%	0.358%
Total running time for each of the 16 processes	19 294 445	19 294 445	19 294 445	19 294 445
Avg duration per event	137.62	43	1 674.59	1 429.76
Min duration	0	31	501	501
Max duration	46 371	60	46 266	46 371
Total time delay during all events	18 116 204	1 610	919 770	997 536
Average time delay per event	80.97	536.67	1 525.32	1 288.81
Minimum total-of-all-delays experienced per process	540 310 (proc 14)	22 (proc 0)	9 334 (proc 5)	9 401 (proc 5)
Maximum total-of-all-delays experienced per process	1 622 606 (proc 8)	148 (proc 8)	151 954 (proc 1)	173 549 (proc 1)
Total late time	18 116 204	870	919 770	997 536
Average late time	80.97	290	1 525.32	1 288.81
Minimum total-of-all-lateness (total-of-all-delays caused) per process	608 854 (proc 9)	7 (proc 8)	7 724 (proc 3)	8 713 (proc 3)
Maximum total-of-all-lateness (total-of-all-delays caused) per process	2 339 081 (proc 14)	133 (proc 0)	181 003 (proc 0)	202 698 (proc 0)

Table 4.13: An example of a report of message-passing operations for the CG benchmark with 16 processes. Most events are point to point events. A total of 30.92 seconds was spent on communication in total by all processes. The fourth and fifth columns report on all the late sender and late receiver events that took longer than the threshold of 0.005s. Process 0 had the most late time, while process 1 was delayed the most for both senders and receivers. 603 late sender events took 0.101 seconds to complete, and 774 late receiver events took 1.1 seconds to complete

time is the total time that events in a process were late and delay time is the total time other processes were idle while waiting for another process.

The time that a process spends idle while waiting for a message-passing operation to complete is not measured separately. However, the threshold of the wait state report generator can be used to exclude events below a given amount of time. Message-passing events can be filtered according to the location in source code where the message-passing operation was used. This is shown in Table 4.14 and Table 4.15. Both examples are for the IS benchmark using sixteen processes. The first example has fewer events compared to the second example and the average duration of an event is shorter.

MPI_Send at is.c:full_verify:515(0x40172b)	
Number of events	61
Duration of all events	571μs
Average duration per event	9μs

Table 4.14: An example of a summary by the location of events in source code of an event from the IS benchmark. Looking at the average and the total time it can be seen that this location does not cause wait states

MPI_Reduce at is.c:main:1066(0x402739)	
Number of events	64
Duration of all events	1 282 406μs
Average duration per event	20 037μs

Table 4.15: An example of a summary by the location of events in source code of an event from the IS benchmark. Looking at the total duration and average duration it can be seen that the total duration of all events at this location was 1.282 seconds. Each event takes on average 0.0200 seconds to complete

4.3.2.1 Comparison of Wait State Reports

To assess the performance impact of wait states in message-passing programs results should be comparable and the increase or reduction in performance should be quantified. To test how effectively changes can be detected, the CG benchmark using sixteen processes was run using two different placement policies of the message-passing processes. The default policy places the first $\frac{n}{2}$ processes on the first node and the rest on the second node, while the Bynode policy alternates the placement of sequential processes between nodes (all processes with even ranks will run on the first node, while processes with odd-numbered ranks will run on the second node). A comparison of these results are shown in Tables 4.16, 4.17, and 4.18.

Metric	Default	By Node	Change(%)
events.total	30791622	26886115	-12.684%
delay.total	18116204	16118553	-11.027%
late.total	18116204	16118553	-11.027%

Table 4.16: Comparison of the P2P events for runs of the CG benchmark using 16 processes. The total amount of work and the total number of events is identical for both reports.

Late Sender Events			
Metric	Default	By Node	Change(%)
events.count	603	88	-85.406%
events.total	1 009 775	729 303	-27.776%
delay.total	919 770	720 053	-21.714%
late.total	919 770	720 053	-21.714%

Table 4.17: A comparison of the late sender events that took longer than 0.005 seconds to complete. The total number of late senders decreased from 603 to 88 for the bynode run compared to the default. There is a reduction of the total waiting time of the bynode run compared to the default of 27.776%.

Late Receiver Events			
Metric	Default	By Node	Change(%)
events.count	774	114	-85.271%
events.total	1 106 635	745 699	-32.616%
delay.total	997 536	730 986	-26.721%
late.total	997 536	730 986	-26.721%

Table 4.18: A comparison of the late receiver events that took longer than 0.005 seconds to complete. The total number of late receivers decreased from 774 to 114 for the bynode run compared to the default. There is a reduction of the total waiting time of the bynode run compared to the default of 32.616%.

4.4 Summary

This chapter presented the results of studying contention in concurrent programs. The machine used for testing, programs used for testing, and testing methodology are described in Section 4.1.

Section 4.2 discusses the results of studying contention for hardware resources. Indicators such as CCM and miss rate are used to estimate the demand for shared resources. The factor of speed-up loss due to contention is calculated. Using the service time of a run of the program with no contention the speed-up loss factors are expressed as time units.

Section 4.3 reports the findings of the analysis of contention for software resources by studying contention for mutexes in shared memory programs and wait states in message-passing programs. Reports of software resource use are generated. Sources of contention or wait states can be identified using data from these reports. The time that was spent waiting to acquire mutexes and communicating data among processes are quantified. For runs of the same program, data can be compared. Some metrics such as the time lost due to contention can be compared directly, while other metrics such as the CCM, miss rate or cache hit/miss ratio compares how effectively resources were used and the rate at which these resources were used. Data about specific sources of software contention can be examined in isolation by filtering the list of resource events and sorting them according to the mutex that was acquired or the address in source code.

The performance of any concurrent program is a complex interaction of the demands for resources and the resources that are available. All of the metrics that we gather and derive reflect this interaction. By studying these metrics the causes of contention can be estimated and the resource usage behaviour of a program can be characterised. Performance can then be improved by focusing on reducing specific causes of contention.

Based on our results we created guidelines for reasoning about performance:

- All programs will be disrupted if workloads evict important data of other threads from the cache. The more important the data that is evicted, the larger the delay, so programs that reuse fewer values are affected less.

- Highly compute bound programs will be disrupted more by interrupting execution or context switching and less by memory contention. See Table 4.1.
- Memory-bound programs are highly affected and work-flow is reduced by contention for memory resources. Interruptions by the scheduler and context switches have a smaller effect on execution progress compared to compute-bound programs. See Tables4.7 and 4.6.
- All software contention and wait states cause idleness, during which no progress is made on useful work.
- Any delays on the critical path directly lengthen execution time.
- Mutex contention delays are caused by competition for the mutex.
- If work is unequal between synchronisation events, the threads with less work will have to wait for the threads with more work to reach the synchronisation point.

When a user knows how each program uses resources and which resources are available they can select a combination of workloads that uses a machine optimally. Based on our results we created guidelines for reasoning about how programs use a machine:

- Compute-bound programs use maximum CPU resources provided by a CPU core.
- Compute-bound programs will interfere with another compute-bound program, resulting in slow-downs for both if they compete for the same CPU core or execution units in a core.
- Memory-bound programs saturate memory bandwidth and buffers
- Memory-bound programs interfere with other memory-bound programs. Compute-bound programs are less affected.
- All programs that use cache will compete, advantage to programs with a smaller cache footprint, fewer reuse of data and compute-bound programs. Cache contention causes more memory traffic.

- All idleness in execution implies no useful work (and very little computation and memory activity) for the idle thread. Idleness might reduce the demand for shared hardware resources.

Chapter 5

Conclusion and Future work

Modern computers rely on shared resources to make them more effective. However, contention for shared resources causes less effective utilisation of resources. The goal of this thesis was to study, detect, and quantify contention for shared resources in concurrent programs. This study provides a detailed discussion about how programs execute on a computer and share resources. Inefficiencies (in the form of an increase in the number of work cycles and idleness) are introduced when contention occurs. Three main sources of contention were selected for study: accessing shared memory, acquiring mutexes, and synchronising the sending and receiving of messages.

The time that a program spent on each source of contention is quantified by using techniques proposed by Tudor and Teo [17], Chen and Stenstrom [5], and Geimer et. al. [9]. Contention that is not measured explicitly, such as contention for software resources and contention for space in the cache, can be detected. When causes of idleness are not explicitly measured, the idleness due to unmeasured data dependencies can be quantified.

Three tools were created to study the three selected sources of contention. A tool based on a model of contention as proposed by Tudor et. al. records data for a running program that include service time, work cycles, and memory accesses. This data is analysed to create a profile that indicate characteristics of how resources are used. Metrics such as the cycles between each cache miss (CCM), and the rate at which cache misses occur, show whether a program is compute-bound or memory-bound. The average number of active threads indicate how much time was spent idle. The factors of speed-up loss due to memory contention and all other dependencies is calculated, and the time loss is quantified. Two tools were developed to study sources of idleness in shared memory and message-passing programs respectively. The mutex tool detects contention for mutexes and the message-passing tool detects wait states in concurrent programs. The time spent waiting to acquire mutexes is calculated based on work by Chen et. al. Analysis can focus on specific mutexes or contention that lies on the critical path of execution. Based on work by Geimer et. al., analysis of the time spent on message-passing is done for point to point communication and collective (barrier-style) message-passing operations. The tool filters wait state events based on various criteria including how long events take to complete and the location in source code where the event was called. The time spent on specific sources of contention is subtracted from the time that the program spent idle. The patterns in which shared resources are used and the time spent on various sources of contention can guide a user in deciding where and when to optimise a concurrent program.

The main contribution of this thesis was to both study the combined effect of contention for multiple shared resources and to study individual sources of contention. An important component of understanding resource contention in a specific program is understanding how the program uses the available resources. Knowing which resources are in demand may indicate probable sources of contention. Metrics such as CCM and miss rate as shown in Table 4.1 express the degree to which a program is CPU / memory-bound. When comparing the service time data, work cycles and cache hits / misses of a program where there is no contention to a run on more than one CPU core, changes in these metrics may also indicate contention. Knowing exactly how much time was lost due to contention provides a clear way to express performance loss. Table 4.5 quantifies how much time was spent contending for memory and the time spent on all other software dependencies in general. The time that individual software resources spent contending can further be isolated as shown in Table 4.11. The time lost due to sources of software dependencies that is not accounted for can be calculated by subtracting the time spent on known sources of contention. Analysis can be refined to only consider specific mutexes or message-passing events, such as shown in Section 4.3.1.3. Additionally mutex events that are on the critical path of execution can be identified. Changes in the performance of a program can be clearly observed as shown in Section 4.3.1.4 and Section 4.3.2.1.

The rest of this chapter discusses possible future work. The effects that resource contention has on the performance of a program changes depending on the types of contention. Contention for software resources causes idleness and consequently less CPU and memory resources are used. When cache contention occurs more memory requests are generated and this increases the contention for memory. These are only two examples of how resource contention can change the demand for other resources. Further studies are required to include the combined effect of contention over time to better understand contention for shared resources in concurrent programs. This implies periodic reading of the performance counters similar to the service time trace. Another implication is that the system should have better awareness of when cache contention is occurring.

The quality of the results that are reported can be improved in several ways. Currently the mutex tool and message-passing tool show the locations in source code where contention occurs for software resources. Reports produced by the tool can be made more useful by extending the awareness of the location in source code where contention occurred to other shared resources. Programs may contain code with different resource requirements. Separating data from different sections would improve the accuracy of reports. As a proof of concept only three sources of contention were considered. Information about cache contention, the root causes of wait states, and contention for more software resources than mutexes and wait states, will extend the effectiveness of the analysis.

Bibliography

[1] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (New York, NY, USA, 1967), AFIPS '67 (Spring), ACM, pp. 483–485.

[2] Barnes, B. J., Rountree, B., Lowenthal, D. K., Reeves, J., de Supinski, B., and Schulz, M. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing* (2008), ACM, pp. 368–377.

[3] Bohme, D., Geimer, M., Wolf, F., and Arnold, L. Identifying the root causes of wait states in large-scale parallel applications. In *Parallel Processing (ICPP), 2010 39th International Conference on* (2010), IEEE, pp. 90–100.

[4] Chandramowlishwaran, A., Madduri, K., and Vuduc, R. Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), IEEE Computer Society, pp. 1–12.

[5] Chen, G., and Stenstrom, P. Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE Computer Society Press, p. 71.

[6] Ding, W., Kandemir, M., Guttman, D., Jog, A., Das, C. R., and Yedlapalli, P. Trading cache hit rate for memory performance. In *Proceedings of the*

23rd international conference on Parallel architectures and compilation (2014), ACM, pp. 357–368.

[7] Ebrahimi, E., Lee, C. J., Mutlu, O., and Patt, Y. N. Prefetch-aware shared resource management for multi-core systems. *ACM SIGARCH Computer Architecture News 39*, 3 (2011), 141–152.

[8] Ebrahimi, E., Miftakhutdinov, R., Fallin, C., Lee, C. J., Joao, J. A., Mutlu, O., and Patt, Y. N. Parallel application memory scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), ACM, pp. 362–373.

[9] Geimer, M., Wolf, F., Wylie, B. J. N., and Mohr, B. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing 35*, 7 (July 2009), 375–388.

[10] Intel 64 and ia-32 architectures software developer manuals. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`. Accessed: 2015-12-06.

[11] Jain, R. *The art of computer systems performance analysis.* John Wiley & Sons, 2008.

[12] Kim, Y., Papamichael, M., Mutlu, O., and Harchol-Balter, M. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on* (2010), IEEE, pp. 65–76.

[13] Liu, X., and Mellor-Crummey, J. A tool to analyze the performance of multithreaded programs on numa architectures. *ACM SIGPLAN Notices 49*, 8 (2014), 259–272.

[14] The numactl tool. `http://man7.org/linux/man-pages/man8/numactl.8.html`. Accessed: 2015-12-06.

[15] The perf wiki. `https://perf.wiki.kernel.org/index.php/Main_Page`. Accessed: 2015-12-06.

[16] Roth, M. *Performance factors in parallel programs.* PhD thesis, Simon Fraser University, 2012.

[17] Tudor, B. M., and Teo, Y. M. A practical approach for performance analysis of shared-memory programs. In *IPDPS* (2011), IEEE, pp. 652–663.

[18] Tzenakis, G., Papatriantafyllou, A., Kesapides, J., Pratikakis, P., Vandierendonck, H., and Nikolopoulos, D. S. Bddt:: block-level dynamic dependence analysis for deterministic task-based parallelism. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 301–302.

[19] Van der Wijngaart, R. F., and Wong, P. Nas parallel benchmarks version 2.4. Tech. rep., NAS technical report, 2002.

[20] Wang, R., Chen, L., and Pinkston, T. M. An analytical performance model for partitioning off-chip memory bandwidth. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (2013), IEEE, pp. 165–176.

[21] Weinberg, J., and Snavely, A. User-guided symbiotic space-sharing of real workloads. In *Proceedings of the 20th annual international conference on Supercomputing* (2006), ACM, pp. 345–352.

[22] Wolf, F., Wylie, B. J., Abrahám, E., Becker, D., Frings, W., Fürlinger, K., Geimer, M., Hermanns, M.-A., Mohr, B., Moore, S., et al. Usage of the scalasca toolset for scalable performance analysis of large-scale parallel applications. In *Tools for High Performance Computing.* Springer, 2008, pp. 157–167.

[23] Wu, C.-J., and Martonosi, M. Adaptive timekeeping replacement: Fine-grained capacity management for shared cmp caches. *ACM Transactions on Architecture and Code Optimization (TACO) 8*, 1 (2011), 3.

[24] Xiang, X., Bao, B., Ding, C., and Shen, K. Cache conscious task regrouping on multicore processors. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on* (2012), IEEE, pp. 603–611.

[25] Zakkak, F. S., Chasapis, D., Pratikakis, P., Bilas, A., and Nikolopoulos, D. S. Inference and declaration of independence: impact on deterministic task par-

allelism. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques* (2012), ACM, pp. 453–454.

[26] Zhuravlev, S., Blagodurov, S., and Fedorova, A. Addressing shared resource contention in multicore processors via scheduling. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 129–142.