High Performance Thread Scheduling on Shared Memory Multiprocessors

A dissertation submitted to the Faculty of Science, University of Malta, in fulfillment of the degree of Master of Science.

by
Kurt Debattista
February 2001
Abstract

In this thesis we are concerned with the efficient scheduling of threads on both shared memory multiprocessors and uniprocessor machines. We attempt to efficiently schedule large numbers of threads while reducing overheads commonly associated with thread management. For performance’s sake we attempt to avoid any kernel intervention and in the case of shared memory multiprocessors, we attempt to achieve a high degree of speedup within a single application.

We experiment with new and already established scheduling strategies for both uniprocessors and SMPs. We present three distinct uniprocessor schedulers with varying characteristics, one of which is distinguished by very fast context switching, another that provides fast inter-thread communication and a third which highlights the benefits of cache affinity scheduling even on uniprocessors. We implement a series of shared run queue SMP schedulers by decreasing the lock granularity for each subsequent scheduler and present a scheduler which is lock free and in particular non-blocking. Results for shared run queue schedulers demonstrate that although finer grain locks and in particular lock free algorithms are suitable for inter-thread communication and for synchronisation among various internal scheduler data structures, the scheduler’s general performance does not necessarily benefit. Moreover, due the high amount of contention for the run queue, we show how shared run queue schedulers are not suitable for fine grain thread scheduling. On the other hand, per processor run queue schedulers are usually criticised for their weak load balancing. We therefore introduce a series of new migration policies, enabling per processor run queue schedulers to perform better than their shared run queue counterparts. We present two schedulers which use distinct migration policies, the first of which migrates threads in groups we call batches across a shared pool of batches. The second is a completely wait free scheduler since it complements wait free thread synchronisation techniques with a series of wait free migration algorithms.
Acknowledgments

I would like to express my deepest gratitude to my tutor Dr. Kevin Vella for his personal help and patience, and his outright dedication to the project and research group. I would also like to thank all the members of the Software Systems Research Group, in particular Joe Cordina who also worked on SMP thread scheduling.

I am also grateful to all the members of the Department of Computer Science and Artificial Intelligence for having given me the opportunity of pursuing postgraduate studies, while also serving as a Teaching Assistant.

Finally, I would like to thank all my friends, my girlfriend Karen and my parents who encouraged and supported me throughout the entire project.
Contents

1 Introduction .......................................................... 1
  1.1 Symmetric multiprocessors ........................................ 1
  1.2 Multithreading .................................................... 1
  1.3 Multithreading in modern operating systems ..................... 2
  1.4 Motivation ........................................................ 2
  1.5 Thread scheduling ............................................... 2
    1.5.1 Uniprocessor thread scheduling ............................ 2
    1.5.2 SMP thread scheduling .................................... 3
  1.6 Overview ........................................................ 4

2 SMP synchronisation and scheduling overview ...................... 6
  2.1 SMPs .............................................................. 6
    2.1.1 Multiprocessor hardware .................................... 6
    2.1.2 Memory models .............................................. 7
    2.1.3 Cache Consistency .......................................... 8
    2.1.4 Pentium III Xeon .......................................... 10
  2.2 Synchronisation ................................................ 11
    2.2.1 Synchronisation support .................................... 11
    2.2.2 Spin locks ................................................ 13
    2.2.3 Non-blocking and wait free synchronisation ............... 14
  2.3 Scheduling ...................................................... 15
    2.3.1 Processes .................................................. 15
    2.3.2 Scheduling policies ....................................... 16
    2.3.3 Threads .................................................... 20
    2.3.4 Multithreading in modern operating systems ............. 20
    2.3.5 High performance multithreading ......................... 21
  2.4 Locality and load balancing .................................... 23
  2.5 User-level thread issues ....................................... 24
    2.5.1 Co-existence with existing platforms ...................... 24
    2.5.2 Performance issues ....................................... 24
    2.5.3 Language implementation .................................. 25
    2.5.4 Communication ............................................ 25
  2.6 Thread management .............................................. 26
    2.6.1 Interaction between kernel and user threads ............. 26
    2.6.2 Context Switching ........................................ 28
    2.6.3 Active context switching .................................. 28
    2.6.4 Inline context switching ................................. 28
2.7 Scheduling Strategies ................................. 29
2.7.1 Shared run queues ................................. 29
2.7.2 Per processor run queues ......................... 30
2.8 Summary ............................................. 32

3 Uniprocessor scheduling ............................... 34
3.1 MESH ................................................. 34
3.2 Scheduler details ..................................... 34
  3.2.1 Underlying architecture ....................... 34
  3.2.2 The thread descriptor ......................... 36
  3.2.3 General functions specification .............. 36
  3.2.4 The smash API thread functions .............. 37
  3.2.5 The smash API CSP functions ................. 38
3.3 Circular run queue smash implementation .......... 40
  3.3.1 Introduction .................................... 40
  3.3.2 Internal scheduler functions ................. 40
  3.3.3 Thread API functions ......................... 43
  3.3.4 Results ........................................ 43
  3.3.5 Analysis of results ............................ 51
3.4 Scheduling for faster execution and inter-thread communication .......... 51
  3.4.1 Introduction .................................... 51
  3.4.2 Internal scheduler functions ................. 52
  3.4.3 Thread API functions ......................... 54
  3.4.4 Results ........................................ 54
  3.4.5 Analysis of results ............................ 57
3.5 Uniprocessor batching ................................ 57
  3.5.1 Introduction .................................... 57
  3.5.2 Internal scheduler functions ................. 57
  3.5.3 Thread API functions ......................... 59
  3.5.4 Results ........................................ 59
  3.5.5 Analysis of results ............................ 63
3.6 Conclusion .......................................... 67

4 Shared run queue SMP scheduling ..................... 69
4.1 The shared run queue ................................. 69
4.2 Scheduler details .................................... 69
  4.2.1 Underlying architecture ....................... 71
  4.2.2 A case of identity ............................. 71
  4.2.3 Idling around .................................. 72
  4.2.4 Context switching .............................. 73
  4.2.5 Scheduler initialisation ....................... 75
  4.2.6 Scheduler shutdown ............................ 76
4.3 Traditional shared run queue SMP smash ............ 76
  4.3.1 Introduction .................................... 76
  4.3.2 Internal scheduler functions .................. 79
  4.3.3 Thread API functions ......................... 79
  4.3.4 Results ........................................ 80
  4.3.5 Analysis of results ............................ 86
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 Thread scheduling using fine grain locks</td>
<td>86</td>
</tr>
<tr>
<td>4.4.1 Introduction</td>
<td>86</td>
</tr>
<tr>
<td>4.4.2 Adapting Michael and Scott’s dual lock algorithm</td>
<td>87</td>
</tr>
<tr>
<td>4.4.3 Internal scheduler functions</td>
<td>89</td>
</tr>
<tr>
<td>4.4.4 Thread API functions</td>
<td>90</td>
</tr>
<tr>
<td>4.4.5 Results</td>
<td>90</td>
</tr>
<tr>
<td>4.4.6 Analysis of results</td>
<td>93</td>
</tr>
<tr>
<td>4.5 Non-blocking scheduling</td>
<td>93</td>
</tr>
<tr>
<td>4.5.1 Introduction</td>
<td>94</td>
</tr>
<tr>
<td>4.5.2 Internal scheduler functions</td>
<td>95</td>
</tr>
<tr>
<td>4.5.3 Thread API functions</td>
<td>96</td>
</tr>
<tr>
<td>4.5.4 Results</td>
<td>97</td>
</tr>
<tr>
<td>4.5.5 Analysis of results</td>
<td>98</td>
</tr>
<tr>
<td>4.6 Conclusion</td>
<td>98</td>
</tr>
<tr>
<td>5 Per processor run queue SMP scheduling</td>
<td>102</td>
</tr>
<tr>
<td>5.1 Per processor run queues</td>
<td>102</td>
</tr>
<tr>
<td>5.1.1 The shared run queue bottleneck</td>
<td>102</td>
</tr>
<tr>
<td>5.1.2 The principle of locality</td>
<td>103</td>
</tr>
<tr>
<td>5.1.3 Load balancing</td>
<td>103</td>
</tr>
<tr>
<td>5.2 Simple per processor run queue scheduling</td>
<td>103</td>
</tr>
<tr>
<td>5.2.1 Introduction</td>
<td>103</td>
</tr>
<tr>
<td>5.2.2 Internal scheduler functions</td>
<td>106</td>
</tr>
<tr>
<td>5.2.3 Thread API functions</td>
<td>106</td>
</tr>
<tr>
<td>5.2.4 Results</td>
<td>107</td>
</tr>
<tr>
<td>5.2.5 Analysis of results</td>
<td>111</td>
</tr>
<tr>
<td>5.3 SMP batching</td>
<td>111</td>
</tr>
<tr>
<td>5.3.1 Introduction</td>
<td>111</td>
</tr>
<tr>
<td>5.3.2 Internal scheduler functions</td>
<td>113</td>
</tr>
<tr>
<td>5.3.3 Thread API functions</td>
<td>113</td>
</tr>
<tr>
<td>5.3.4 Results</td>
<td>114</td>
</tr>
<tr>
<td>5.3.5 Analysis of results</td>
<td>115</td>
</tr>
<tr>
<td>5.4 Wait free scheduling</td>
<td>115</td>
</tr>
<tr>
<td>5.4.1 Introduction</td>
<td>115</td>
</tr>
<tr>
<td>5.4.2 Wait free migration</td>
<td>117</td>
</tr>
<tr>
<td>5.4.3 Internal scheduler functions</td>
<td>121</td>
</tr>
<tr>
<td>5.4.4 Thread API functions</td>
<td>123</td>
</tr>
<tr>
<td>5.4.5 Results</td>
<td>123</td>
</tr>
<tr>
<td>5.4.6 Varying the batch size</td>
<td>126</td>
</tr>
<tr>
<td>5.4.7 Scalability</td>
<td>128</td>
</tr>
<tr>
<td>5.4.8 Analysis of results</td>
<td>131</td>
</tr>
<tr>
<td>5.5 Conclusion</td>
<td>131</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>134</td>
</tr>
<tr>
<td>6.1 Discussion</td>
<td>134</td>
</tr>
<tr>
<td>6.1.1 Uniprocessor scheduling</td>
<td>134</td>
</tr>
<tr>
<td>6.1.2 SMP scheduling</td>
<td>135</td>
</tr>
<tr>
<td>6.2 Summary of major contributions</td>
<td>137</td>
</tr>
</tbody>
</table>
6.3 Future directions ................................................................. 138
   6.3.1 Extending the scheduler API ........................................... 138
   6.3.2 Object-affinity scheduling through batches ....................... 138
   6.3.3 Scheduler extensions ................................................... 139
   6.3.4 Changing scheduler requirements .................................... 139
   6.3.5 Thread scheduling on other systems ............................... 140
6.4 Closing note ................................................................. 140
# List of Figures

2.1 UMA. ................................................................. 7  
2.2 NUMA. .............................................................. 7  
2.3 Hybrid NUMA. ..................................................... 8  
2.4 NORMA. ............................................................. 10  
2.5 Key for forthcoming figures. .............................. 17  
2.6 Kernel threads bound to a process. .................. 17  
2.7 A simple user thread scheduler. ...................... 18  
2.8 Threads blocked on a communication construct. ........ 18  
2.9 Spin lock access. ............................................... 19  
2.10 Traditional UNIX processes and their relation with the kernel. .......... 19  
2.11 User and kernel thread interaction. ................. 21  
2.12 Shared run queue. .............................................. 29  
2.13 Per processor run queues. ............................... 31  
2.14 A quad processor implementation of SMP batching. ........ 33  
3.1 Underlying architecture. .............................................. 35  
3.2 Circular run queue scheduling. ......................... 41  
3.3 Insert thread onto circular run queue. .............. 41  
3.4 Remove thread from circular run queue. ........... 42  
3.5 Circular run queue yield. ..................................... 42  
3.6 Performance degradation due to cache misses and page faults. ......... 43  
3.7 Dummy head run queue scheduling. .................... 53  
3.8 Dummy head scheduler thread insert. ............... 54  
3.9 Dummy head scheduler thread remove. ............. 54  
3.10 Uniprocessor batch scheduling. ....................... 58  
3.11 Uniprocessor batch scheduling thread insert process. ........... 60  
3.12 Results from Algorithm 3.10, for an array size of 50,000 words. ...... 64  
3.13 Results from Algorithm 3.10, for an array size of 300,000 words. ...... 65  
4.1 Load balancing properties. ..................................... 70  
4.2 Kernel threads bound to a processor. ............... 71  
4.3 Global lock scheduling. ....................................... 77  
4.4 Traditional shared run queue SMP thread scheduling. ............. 78  
4.5 Traditional scheduler, speedup results. ............. 83  
4.6 Dual lock shared run queue scheduling. ............ 88  
4.7 Inserting a thread onto the dual lock shared run queue. ............. 89  
4.8 Inserting a thread onto the dual lock shared run queue. ............. 89  
4.9 Dual lock scheduler, speedup results. .............. 92
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.10</td>
<td>Non-blocking scheduling</td>
<td>94</td>
</tr>
<tr>
<td>4.11</td>
<td>Non-blocking scheduler, speedup results</td>
<td>99</td>
</tr>
<tr>
<td>4.12</td>
<td>Speedup for various schedulers</td>
<td>100</td>
</tr>
<tr>
<td>5.1</td>
<td>Possible gross load imbalances when using per processor run queues</td>
<td>104</td>
</tr>
<tr>
<td>5.2</td>
<td>Simple per processor scheduling</td>
<td>105</td>
</tr>
<tr>
<td>5.3</td>
<td>PP scheduler, speedup results</td>
<td>110</td>
</tr>
<tr>
<td>5.4</td>
<td>SMP batch scheduling</td>
<td>112</td>
</tr>
<tr>
<td>5.5</td>
<td>SMP batching scheduler, speedup results</td>
<td>116</td>
</tr>
<tr>
<td>5.6</td>
<td>Wait free scheduling</td>
<td>118</td>
</tr>
<tr>
<td>5.7</td>
<td>Sender initiated migration</td>
<td>119</td>
</tr>
<tr>
<td>5.8</td>
<td>Receiver initiated migration</td>
<td>122</td>
</tr>
<tr>
<td>5.9</td>
<td>Wait free scheduler, speedup results</td>
<td>125</td>
</tr>
<tr>
<td>5.10</td>
<td>Performance comparison from various schedulers</td>
<td>127</td>
</tr>
<tr>
<td>5.11</td>
<td>Comparison of different batch sizes for wait free scheduling</td>
<td>128</td>
</tr>
<tr>
<td>5.12</td>
<td>Comparison of different batch sizes for wait free scheduling</td>
<td>129</td>
</tr>
<tr>
<td>5.13</td>
<td>Scaling the wait free scheduler</td>
<td>130</td>
</tr>
<tr>
<td>5.14</td>
<td>Comparison of speedup amongst different schedulers</td>
<td>132</td>
</tr>
<tr>
<td>5.15</td>
<td>Comparison of speedup amongst different schedulers. Scaled in and out</td>
<td>133</td>
</tr>
</tbody>
</table>
List of Algorithms

2.1 Attempting to lock a resource. ............................................. 11
2.2 Compare and swap pseudo code. ........................................... 12
2.3 Acquire spin lock. ......................................................... 13
2.4 Release spin lock. ......................................................... 13
2.5 Acquire spin lock, using test and test and set. ....................... 13
2.6 General outline of an operation on a non-blocking data structure. .... 15
3.1 Thread initialisation benchmark. ........................................... 45
3.2 Cache friendly thread initialisation benchmark. ....................... 46
3.3 Thread insert benchmark. .................................................. 47
3.4 Cache friendly thread insert benchmark. ................................ 47
3.5 Thread execution benchmark. ............................................. 48
3.6 Cache friendly thread execution benchmark. .......................... 49
3.7 Context switch benchmark. ............................................... 50
3.8 Calculating an element in matrix multiply. ............................ 56
3.9 Matrix multiplication benchmark. ....................................... 56
3.10 Cache affinity benchmark. ............................................... 66
4.1 Context switch race condition for kernel thread 0. ...................... 73
4.2 Solution for context switch race condition by switching to kernel thread stack. 74
4.3 Change stack to scheduler stack. ....................................... 74
4.4 Solution by switching to thread stack for race condition .............. 75
4.5 Speedup benchmark. ......................................................... 82
4.6 Channel benchmark. ......................................................... 85
4.7 Wait free thread migration for idle processors. ....................... 95
4.8 Wait free thread migration for idle processors using a sleeper counter. 96
4.9 Wait free barrier algorithm. ............................................. 96
4.10 Wait free barrier algorithm at thread termination. .................... 97
5.1 Speedup benchmark for per processor scheduler. ...................... 108
5.2 Sender initiated migration. ............................................... 117
5.3 Receiver initiated migration. ............................................ 120
Chapter 1

Introduction

High performance computing usually falls under the domain of expensive high end systems. However, low cost alternatives to these systems exist in the form of symmetric multiprocessors (SMP) and networks of workstations (NOW), which offer nearly comparable processing power for computationally intensive tasks, at a fraction of the cost of the price of the aforementioned systems. NOWs which are composed of common desktop workstations and communicate by means of network interface cards, are scalable systems composed of readily available “off the shelf” products that can be used as distributed super computers. SMPs, the architecture which we will focus on mainly in this thesis, are tightly coupled, shared memory, symmetric multiprocessor machines [88] which when used correctly can provide a notable increase in computational power.

1.1 Symmetric multiprocessors

SMPs provide uniform access for a number of processors to shared memory by means of a high speed bus (although most systems provide a level of caching per processor). Traditional operating system processes, due to memory management protection, do not take full advantage of the SMP architecture. Consequently, multithreading, a new style of programming has emerged. Modern operating systems offer the advantage of creating processes which can share memory with other processes and thus a single application can be run on more than one processor concurrently.

1.2 Multithreading

Multithreaded applications are usually considered to be either fine grained or coarse grained. Coarse grained applications usually consist of few threads of execution which communicate with each other infrequently. Coarse grained multithreading attempts to attain a high degree of parallelism by minimising on the cost of thread management. On the other hand fine grained applications usually consist of very large numbers of frequently communicating threads. Due to the large amount of threads, the cost of the thread scheduling code might surpass the application’s own code. Therefore, efficient thread management is particularly significant since it enables tens of thousands of threads to be scheduled without incurring high overheads. The philosophy of fine grained multithreading is particularly suitable for applications where the number of processors is known only at run-time, in real-time operating systems (RTOSs) when the scheduler needs to meet specific deadlines, for high
performance scientific calculations and for achieving a high quality of service in server side programming.

1.3 Multithreading in modern operating systems

Multithreading support in modern operating systems can be broadly divided into two categories; kernel-level threads and user-level threads. Kernel-level threads, also appropriately referred to as light weight processes (LWP), fulfill the task of providing processes with shared memory. Kernel-level threads are expensive to manage since the kernel is involved for every LWP operation. Another approach is to implement the threads of control as user-level threads which require no operating system involvement and thus incur low overheads, in the order of a few hundreds of nanoseconds. The kernel thread and user thread combination provides a powerful programming environment. The user-level thread scheduler multiplexes kernel-level threads on top of user-level threads creating a miniature kernel for the threads to operate. User-level threads in particular are useful for fine grained applications.

1.4 Motivation

The purpose of this project is two fold:

1. We propose to tackle the challenge of efficiently scheduling large numbers of user-level threads on both uniprocessors and SMPs, while reducing overheads to well below a microsecond. We would like to create a system where an application programmer can program at a very fine thread granularity without worrying too much about the overheads associated with thread management.

2. By implementing already existing thread scheduling strategies and techniques, and introducing new ones, we conduct a survey of the possible implementation of fine grain user-level thread schedulers. By experimenting with various synchronisation techniques and different run queue configurations we seek to reduce contention on the local bus, reduce kernel involvement and exploit the cache, while trying to decrease the thread management overhead and at the same time improving throughput.

1.5 Thread scheduling

To achieve the above mentioned goal we have created a user-level thread scheduler test bed, which we choose to call smash. We modify an existing uniprocessor thread scheduler, MESH [18] for our our base uniprocessor scheduler, from which we implement a number of uniprocessor and SMP thread schedulers. Our attention will be focused on priority less, non-preemptive thread schedulers. The definition is important since both priority based scheduling and preemption create new dimensions that we do not cater for.

1.5.1 Uniprocessor thread scheduling

A number of uniprocessor thread schedulers which manage to schedule large numbers of threads without incurring the high overhead associated with fine grain multithreading have already been implemented [18, 80, 107, 110]. We adopt similar strategies and present new
ones to implement three distinct uniprocessor schedulers. The main distinguishing factor among our uniprocessor schedulers is the design of the run queue.

The first of our uniprocessor schedulers is similar to MESH. This version uses a circular run queue for fast context switch times. The second of our scheduler configurations uses a novel run queue configuration, which we adapt from SMP based concurrent queue implementations [103]. This particular run queue uses a dummy head which improves enqueue and dequeue performance, directly affecting certain applications, in particular CSP applications. Our final uniprocessor scheduler is an implementation of a scheduling strategy already implemented by Vella [107] for the KRoC scheduler [110]. This particular strategy groups common threads together into batches and as a direct consequence exploits the cache, to improve performance.

1.5.2 SMP thread scheduling

The situation for SMP fine grain thread scheduling is different from that for uniprocessors. Many previous SMP thread scheduler implementations for fine grain applications have not been entirely successful. The overhead associated with synchronisation on SMPs, tends to be prohibitive for fine grain thread scheduling.

The SMP scheduling strategies could be broadly divided into two. On one side are the more commonly used shared run queue schedulers, which make load balancing their strongest point. At the other end there are the per processor run queue configurations, that naturally provide cache-conscious scheduling and reduced run queue contention. We implement various schedulers for both shared run queue and per processor run queue scheduling strategies.

Shared run queue SMP scheduling

Until now efforts at converting fine grain uniprocessor schedulers have been mainly directed at using shared run queue schedulers. SMP versions of the KRoC scheduler [107] and the MESH scheduler [25] both make use of a shared run queue as the scheduling strategy. Results from both these schedulers show that due to contention on the shared run queue, this approach might not be the ideal solution for fine grain applications.

We investigate the situation by implementing a variety of shared run queue schedulers, where we attempt to reduce contention for shared resources, by employing a variety of synchronisation techniques. We implement five distinct shared run queue thread schedulers. The first of our thread scheduling techniques uses a global lock for all shared resources. The more traditional scheduling strategy, whereby the internal scheduler structures (including the shared run queue) are protected by a single lock and all communications structure have their own locks, is embodied in our second scheduler. The same approach was used for the implementations of SMP KRoC and SMP MESH. For our third scheduler we attempt to reduce contention on shared resources even further by employing even finer grain locks, such that each individual scheduler structure has its own lock. Our fourth scheduler takes the fine grain locking philosophy further, by adopting Michael and Scott’s [77] dual lock concurrent queue as the shared run queue. The dual lock feature allows for concurrent enqueue and dequeue routines, improving performance in CSP applications. The final shared run queue scheduler is entirely lock free and in particular non-blocking. We use Michael and Scott’s [77] non-blocking concurrent queue algorithm to implement the shared run queue and Vella’s wait free CSP channels [108] for inter-thread communication. We also introduce a wait free
barrier construct and wait free internal scheduler structures.

**Per processor run queue SMP scheduling**

The per processor run queue strategy was more commonly associated with distributed systems [35]. It was believed that since migration on shared memory machines is very fast, thread scheduling using a shared run queue would be ideal. This situation is indeed true for coarse grain applications, however for fine grain applications the contention for the run queue results in most of the processor time wasted attempting to acquire the shared resource [68]. Moreover, the continuous migration of threads across processors goes against the principle of locality [3]. The main advantages of per processor run queue schedulers are the reduced contention on the run queues, since each processor services its own run queue, and a natural adherence to the principle of locality in terms of cache affinity. The main disadvantage is the possibility of gross load imbalances.

We implement three per processor run queue schedulers. Our first per processor run queue scheduler highlights the advantages of the per processor run queue strategy in terms of reduced contention and processor affinity scheduling. This scheduler implementation does not attempt to migrate threads across processors, demonstrating the possible load imbalances that per processor run queue schedulers are subject to. Our final two schedulers attempt to maintain the advantages of the per processor run queue schedulers and eliminate possible imbalances by means of migration strategies. The first of the two is an implementation of Vella’s [107] description of an SMP batching scheduler. Vella describes how extending his uniprocessor batching strategy, threads could be migrated across processors in large groups called batches. A shared batch queue is responsible for the batch migration. Our final scheduler employs wait free scheduling techniques developed for our non-blocking scheduler and new wait free migration algorithms to pass threads (or batches) across processors. The per processor nature of the scheduler adheres to the principle of locality and being composed of solely wait free structures, naturally eliminates most forms of contention to any of the shared resources. Moreover, the migration policy helps maintain strong load balancing.

1.6 Overview

The entire document is divided into 6 chapters. We have briefly described our motivation and goals in this chapter. The next chapter introduces multiprocessor machines, surveys synchronisation techniques on multiprocessors and describes various schedulable entities, in particular user-level threads. We discuss methods of improving performance for user-level threads directly and through interaction with the operating system. The concepts of locality and load balancing are introduced and contrasted. Finally we finish the chapter by discussing possible scheduling techniques. The subsequent three chapters discuss the smash scheduler implementations. Each chapter represents one of the scheduler categories introduced above. Chapter 3 introduces the uniprocessor schedulers and specifies the API for all the smash schedulers. The first of the SMP scheduler chapters is dedicated to the shared run queue implementations, however it also introduces the underlying general SMP structure. Chapter 4 is also responsible for discussing inter-thread communication. Chapter 5 is solely dedicated to per processor schedulers and techniques for migrating threads across the individual processors’ run queues. In most cases, each scheduler has a section entirely dedicated to it, where we describe the motivation behind the scheduler
and introduce the scheduler’s structure. We also discuss the advantages and disadvantages by means of results and compare and contrast schedulers amongst each other. The final chapter concludes the document by discussing results of all schedulers and highlighting the individual merits of each one.
Chapter 2

SMP synchronisation and scheduling overview

In this chapter we discuss issues pertaining to thread scheduling on SMPs. We begin by introducing SMPs, then we discuss scheduling policies and various synchronisation methods. We finally discuss the various scheduling strategies that led us to choosing the various smash versions.

2.1 SMPs

Multiprocessor system architecture has been widely discussed in the literature and a detailed description is beyond the scope of this document (see [1, 28, 51]). However, we begin this section by introducing multiprocessing hardware to give the reader a better description of the various types of parallel computers. We then discuss where SMPs fit into the global multiprocessor picture and describe the problems relevant to caching on SMPs. Finally we conclude the section with a brief look at a specific SMP architecture, the Pentium III Xeon, which was used to test and implement our SMP schedulers.

2.1.1 Multiprocessor hardware

Flynn [38] describes a general classification of parallel processing architecture. He describes parallel computer architectures in terms of instructions acting on data. All computer systems broadly fall into one of the following classifications:

**Single Instruction Single Data (SISD)** SISD refers to the traditional Von Neumann architecture. One series of instructions acts on a single item of data.

**Single Instruction Multiple Data (SIMD)** SIMD refers to multiple instances of the same instruction working on different data. Classical examples of SIMD are array computers [44].

**Multiple Instruction Single Data (MISD)** In MISD the same data is manipulated by more than one program. Some researchers consider a real MISD computer to not have been created yet. Others consider pipelined vector processors to be existing examples [1] of MISD computers.
Multiple Instruction Multiple Data (MIMD) MIMD involves different instructions running on separate processors and concurrently accessing different data sequences. MIMD structures are considered the most useful for general purpose parallel computing. We will focus our attention on this type of parallel architecture throughout the rest of this document.

2.1.2 Memory models

Another important aspect of multiprocessor architecture, is the memory model. There are two abstract types of memory model, the shared memory model and the no remote memory access model. Furthermore the shared memory models can be subdivided into uniform access and non-uniform access models. Each model could be briefly described as follows:

Uniform Memory Access (UMA) UMA multiprocessor systems offer CPUs equal access to memory and I/O devices. Connections between CPUs and memory usually take place on a single bus. See Figure 2.1.
Non Uniform Memory Access (NUMA) In NUMA systems, each CPU has local memory and is capable of accessing memory local to other CPUs as well, as can be seen in Figure 2.2. Remote memory is usually slower than access to local memory. Hybrid NUMA models allow a set of CPUs to have access to the same main memory and remote access to a remote set’s memory. See Figure 2.3.

No Remote Memory Access (NORMA) NORMA systems connect different CPUs, each of which have their own local memory, over a high speed network connection. Figure 2.4 is an example of a NORMA system connected by a high speed network. Memory access is not allowed across the network.

The most popular type of memory model is UMA [82]. However, it may be noted that in most of these machines each processor would have an individual cache. This property makes them similar to the NUMA memory model. The UMA model that interests us is the tightly coupled, shared memory, symmetric multiprocessor (SMP) [88]. CPU, I/O and memory are all tightly coupled, usually interconnected over a high speed bus and all units are located at short physical distances from each other. Memory consists of one single and global memory module. CPUs store all instructions and data in shared memory, which becomes immediately accessible to all other CPUs and I/O devices. All memory access is symmetric, all CPUs and I/O use the same physical address to reference the same data and are ensured fair access.

2.1.3 Cache Consistency

Modern chips are large enough to contain not only the processor but also high speed memory called cache. The role of the cache is to diminish the relatively low speed access between memory and processor. The cache acts as an instruction and data buffer. Processing on a system with cache would bring data and instructions from main memory when they are first needed (at the slow speed of main memory access), however when the data or instructions are reused, they can be obtained directly from the cache at higher speeds than if main memory had to be used. Caching exploits the locality of reference principle [88] providing a significant increase in performance. The cache capacity is usually much smaller than that

---

1 Most CPUs usually have access to local memory in terms of a local cache.
of main memory, therefore the cache data and instructions are replaced when others are needed.

On multiprocessor machines, caches present a problem. Since copies of a shared variable lie within the individual caches, changes to the variables must be reflected in all the other caches and in main memory. There are various solutions that solve cache inconsistency problems [1, 28, 44, 51]. Describing the various solutions in detail is beyond the scope of this survey. However, we briefly describe the most common cache consistency mechanisms:

**Write through protocol** There are two different methods of implementing this protocol. The first, known as write through with update protocol, involves the processor performing a write to data inside its cache also updating the value of the data inside the memory module. The simplest method of updating other caches in this case is by broadcasting the modified data to all the processors, allowing each one to update the contents of the changed cache line. In the second version, when a cache line is written to, the value is written into the memory, and the cache lines of other processor’s caches accessing the same data, is invalidated.

**Write back protocol** Whenever a processor needs to change information in a cache block, it first needs to acquire the rights to that block. It acquires the rights from the memory module that owns the block. When another processor needs to read the block, the data is sent to the processor by the current owner that also informs the memory module of the change. The memory model then re-acquires ownership. This technique is considered faster than the write through protocol, since a processor would perform more than one write operation on a cache block before giving it up to some other processor [44].

**Snoopy caches** Since on single bus systems all operations are performed via the bus, each processor can monitor all transactions. When a processor performs a write on a variable, the processor’s cache block is marked as dirty, the change is broadcast on the bus and memory and the other caches invalidate their copies. The processor now can perform as many write operations as it likes on that cache block. When another processor needs to read a variable in the dirty cache block (the main memory module has an invalid copy) the owner of the cache block notices from the bus transaction and sends a valid copy to the requesting processor’s cache and to the main memory module. The processor that owns the cache block now marks his cache block as clean.

**False sharing**

Problems due to caching still however arise on multiprocessors. One common problem that can severely affect performance is false sharing (see Section 5.2.4 for an example of the potential performance hit that can arise from false sharing). False sharing arises when more than one processor is accessing data which is located on the same cache line. If one processor writes to any word on the cache line, the same cache line in the other processors’ cache will be invalidated, even though the word they where accessing was not written to. This may lead to situations where more than one processor are accessing different words on the same cache line continually and inevitably in each case the process involves performing one of the operations which were described in the above protocols. The simplest solution is to ensure that any shared data is buffered by at least a cache line of non-used words.
Another solution [58] involves the use of compile time checks to identify and transform potential structures that are susceptible to false sharing.

### 2.1.4 Pentium III Xeon

The Pentium III Xeon processor forms part of the P6 family of processors [52]. The Xeon processor differs mainly from the standard Intel Pentium III in that it caters for a multiprocessor environment [55]. The Xeon architecture provides SMP support for up to four processors, by means of a 100MHz system bus. It can however be scaled to 8-way and larger SMPs by means of specialised chip sets and clustering techniques. The Xeon supports up to 2MB non-blocking level 2 cache and 32KB (16KB instructions, 16KB data) level 1 cache.

The Xeon (and other Intel P6 processors) support multiprocessing by means of bus locking, cache consistency and serialisation of instructions [54]. Certain instructions (such as reading or writing a byte, word/doubleword/quadword on an aligned 16/32/64-bit boundary) are guaranteed to be atomic. Other operations (such as XCHG instructions [53], setting the busy flag and updating segment registers [52], acknowledged interrupts) automatically lock the bus. Other instructions, are allowed to use the lock prefix [53] to explicitly lock the bus (e.g. bit test and modify instructions, exchange instructions, certain logical and mathematical instructions [53]). The P6 family of processors also support atomic instructions (see Section 2.2.1) and have equivalents of compare and swap, test and set, swap amongst others [53]. The P6 memory model does not enforce strong ordering. Performance is enhanced by such variations such as allowing reads to be executed before buffered writes. However, writes are not speculative and the memory model in any one given processor is self-consistent [54]. Memory ordering can be strengthened using the lock prefix, by serialising instructions and by means of memory type range registers. Certain instructions, known as serialising instructions, ensure that the processor has completed all modifications to flags, registers and drains all buffered writes to memory before they are executed [54].
Algorithm 2.1 Attempting to lock a resource.

R1  if (flag == 0) { // read flag - if flag 0 resource is available
R2    flag = 1; // if resource is available lock resource
R3    return TRUE; // return resource obtained
R4  }
R5  else return FALSE; // resource not obtained

2.2 Synchronisation

The use of synchronisation primitives in a multiprocessor environment serves a dual purpose [82], to provide for mutual exclusion and for event ordering. Synchronisation primitives such as semaphores were first introduced by Dijkstra [31]. Usually high level constructs such as semaphores and monitors are used to provide synchronisation at the user level. However, we are not interested in the interfaces, (these primitives are discussed in literature [65, 92, 102]) but on how to cater for synchronisation.

For mutual exclusion based synchronisation, before entering a critical section a decision has to be taken on whether the current process (or thread) is to relinquish processor control or busy wait until the resource is free and it can acquire it. Since yielding a process requires a vertical context switch, and busy waiting wastes processor cycles, the general rule of thumb is to busy wait if the time expected to wait is less than that of blocking and resuming the task, and if no other work is available [4].

Since we will be working in a fine grain multithreaded environment where critical sections are usually small and we would not like to involve any kernel calls, we are more concerned with the implications of spin locks and alternatives such as lock free algorithms.

2.2.1 Synchronisation support

Synchronisation in a multiprocessor system relies on hardware support [102]. In such a system two different processors could attempt to access the same resource. If a processor attempts to run Algorithm 2.1 and is executing between R1 and R2 and another processor concurrently reads R1, both processors would have found the resource available and attempt to lock it concurrently. The above situation can be avoided if the hardware provides a primitive which can perform part of the above code in a single operation. Most modern processors contain in their instruction set one or more instructions that can perform said function. All these operations are atomic, therefore no other operation can interfere in its running, not even interrupts can pause their execution.

Atomic read and write instructions

The simplest form of atomic instructions are read and write operations from a CPU, memory or I/O device (using DMA). These operations cannot be interrupted once begun. Although they can be used to implement mutual exclusion constructs [30, 89], they provide limitations (e.g. the memory model required for Dekker’s algorithm [30]). Atomic read and write instructions cannot be used in more complicated synchronisation algorithms [105].
CHAPTER 2. SMP SYNCHRONISATION AND SCHEDULING OVERVIEW

Algorithm 2.2 Compare and swap pseudo code.

```c
BOOL CAS (destination, expected, new) {
    if (destination == expected); // is destination what we expect
        destination = new; // set destination to new
        return TRUE; // return success
    else return FALSE // return failed
}
```

Atomic “read-modify-write” instructions

Probably the simplest atomic “read-modify-write” instruction is the atomic swap instruction [88]. The atomic swap simply swaps a value in a register/memory with a value in register/memory and returns the original value.

Atomic test and set provides the reading of a flag (usually a single bit in memory), setting the flag to 1 and returning the previous value of the flag. The main use of test and set is for spin locks (see Section 2.2.2).

Atomic compare and swap, was first made available for IBM system 370 [22]. compare and swap accepts three parameters, the destination, an expected value and a new value. It compares the destination with the expected value and if equal, sets the destination equal to the new value and returns successful, otherwise it returns failure. Algorithm 2.2 describes a non-atomic, C pseudo code version of compare and swap. compare and swap is used mainly in the construction of lock free algorithms, which we discuss in Section 2.2.3.

There are other more powerful versions of compare and swap. One version of compare and swap is the double compare and swap which performs two compare and swap operations simultaneously. This version is available for the Motorola M68030 processor. Another implementation is the double word compare and swap which is available on Intel P6 and SPARC machines. Both versions are used to avoid the ABA problem (see Section 2.2.3).

Bershard [13] presents an implementation of compare and swap using test and set or swap and some operating system intervention.

The load linked and store conditional pair are another set of atomic instructions. The load linked instruction loads the value from memory into a register and sets a flag. The flag is reset if another processor writes to the memory location. The store conditional instruction checks if the flag is still set and stores a new value in the monitored location. It also sets the value of another register on success. The load linked, store conditional pair can be used to implement compare and swap [79] on machines that do not have support for compare and swap.

A hierarchy of primitives

Herlihy [45] establishes a hierarchy of atomic primitives based on wait free concepts (see Section 2.2.3). An object in a higher position in the hierarchy can implement a wait free version of a primitive that has a lower position. Atomic read and write operations reside at level 1 of the hierarchy. Primitives such as test and set and swap are at level 2. Herlihy [47] defines compare and swap as a universal primitive which sits at the top of the hierarchy and can be used to implement any lock free implementation of a concurrent data object. Michael and Scott [76] argue that although store conditional/load linked pair would be at level $\infty$ of the hierarchy just like compare and swap the simulation of compare and swap is less efficient than if there was a a single atomic instruction. However they also
Algorithm 2.3 Acquire spin lock.

```
spin:  lock ; lock on bus for btsl instruction
   btsl lock,0 ; bit test and set on variable lock, copy previous to carry
   jnc cont ; continue if carry is 0
   jmp spin ; else spin
cont:  nop
```

Algorithm 2.4 Release spin lock.

```
lock ; lock on bus for btrl instruction
btrl lock,0 ; reset lock
```

state that the compare and swap algorithms given by Herlihy [46] are more complex than the load linked/store conditional implementations [48].

2.2.2 Spin locks

Spin locks are considered the simplest locking primitives [82, 102]. Two operations act on a lock. When a task requires access to a resource protected by a spin lock, it attempts to acquire the resource by means of a `spinlock()` operation. If the resource has already been acquired, the task busy waits by continuously checking the contents of a variable (which is usually set to 0 if the resource is available and 1 when the resource is locked) until the resource is released. A lock is released by means of a call to a `spinrelease()` function which sets the contents of the tested variable to show that the resource is available. Although spin locks can be implemented by means of Dekker’s algorithm [30] which uses only reads and writes, this requires a specific memory model and is considered inefficient [107]. The traditional spin lock is therefore implemented using atomic instructions such as test and set and fetch and store. Algorithms 2.3 and 2.4 demonstrate versions of spin lock acquisition and release [25] using Intel P6 instructions [53]. Since test and set monopolises the memory bus and invalidates any cache line that contains the tested variable, a preferred approach due to Segall and Rudolph [90] could be used. Segall and Rudolph propose employing a test and test and set algorithm composed of two loops, to acquire the lock. Algorithm 2.5 [25] first checks if the lock is acquired, if it is, it spins on a read, until the value of the variable is changed. On shared memory machines with many processors, this algorithm still suffers from the “thundering herd problem” [102], since the cache of a

Algorithm 2.5 Acquire spin lock, using test and test and set.

```
spin:  lock ; lock on bus for btsl instruction
   btsl lock,0 ; bit test and set on variable lock, copy previous to carry
   jnc cont ; continue if carry is 0
loop:  btl lock,0 ; copy on carry
   jc loop ; loop if carry is set
   jmp spin ; else try and acquire lock again
cont:  nop
```
processor containing the tested variable needs to be updated [4]. Queue locks such as the
MCS lock [75], allow for each processor to spin on a different memory location, therefore
only the cache line of the next process to enter a critical section is invalidated. Anderson
et al. [4] propose an exponential back-off algorithm to solve the problem.

One of the main problems with spin locks occurs in multiprogrammed environments. If a
kernel thread is de-scheduled while holding a spin lock, other kernel threads waiting for the
lock to be released will have to wait for the kernel thread holding the lock to be rescheduled.
Preemption safe locking techniques attempt to solve this problem by either trying to avoid
a kernel thread holding a lock to be preempted or by recovering if such a situation occurs.
Edler et al’s [36] Symunix system allows a thread holding a lock to set a flag, so that it won’t
be preempted. The kernel will not preempt the process that set the flag until either the flag
is reset or a reasonable amount of time has elapsed. Marsh’s [72] first class thread interface
allows the kernel to inform the application (by means of a flag that the kernel thread can
read) sometime before the process will be preempted. The process that is about to acquire
a lock can then decide whether to enter the critical section or relinquish processor control.
Ousterhout [83] proposes a recovery based technique where processes waiting to acquire a
lock busy wait for a certain amount of time. If the time allocated expires and they still
haven’t acquired the lock, they block. Lim [62] discusses the possibility of a hybrid between
busy waiting and blocking, depending on run time factors.

2.2.3 Non-blocking and wait free synchronisation

An alternative method of synchronisation which dispenses with serialising concurrent tasks
is the concept of non-blocking and wait free synchronisation. Lock free algorithms rely
on powerful atomic primitives and careful ordering of instructions to protect corruption
of resources and ADTs from concurrent access. Valois [105] describes the relationship of
synchronisation methods as those that use mutual exclusion techniques (which include busy
waiting, blocking and hybrids) and lock free techniques. Lock free structures may have two
properties. They can be non-blocking, in which case an operation on the lock free data
structure is guaranteed to complete in finite time. Moreover, if all such operations are
guaranteed to complete (thus avoiding starvation) the structure is said to be wait free.
Note that any data structure that uses mutual exclusion is neither non-blocking nor wait
free.

Various methods were used to create lock free versions of data structures such as the
universal logging method [45, 47] and copying method [46, 48] and other more general
techniques such as blocking algorithms [74, 98] and non-blocking [67, 77, 79, 85, 103, 104]
algorithms amongst others. Valois [105] argues that most of the universal methods impose
a significant amount of overhead that make the resulting algorithms less efficient than the
equivalent algorithms that use locks. Valois goes on to describe a general method that
could be used to create non-blocking algorithms. Lock free algorithms generally use a
“read-modify-(check-write)” cycle for modifying the data structure. The “(check-write)”
operation must be done atomically. From Herlihy’s consensus hierarchy [45] we know that
certain atomic primitives can be used to implement all types of wait free data structures
(see Section 2.2.1). compare and swap lies at level \( \infty \) of the hierarchy. The “read-modify-
(check-write)” cycle could thus be implemented as a “read-modify-\texttt{compare and swap}” cy-cle.
Algorithm 2.6 demonstrates a general outline of an operation on a non-blocking data
structure [105].
Algorithm 2.6 General outline of an operation on a non-blocking data structure.

```
L1 void adt_operation(adt * ptr) { // operation on adt structure
L2     adt * old; // local pointers
L3     adt * new;
L4     do {
L5         old = ptr; // assign old to pointer
L6         calc_new(new); // calculate new value
L7         while(!CAS(ptr,old,new)); // if no other operations
L8     } // have interfered swing ptr to new
```

The ABA problem

A problem that arises from using compare and swap in Algorithm 2.6 is the ABA problem. In L7 we intend to swing ptr to new if the value of ptr hasn’t changed (i.e. if old and ptr still have the same value) since we read it at L5. There may however be a case when the value might have changed and the subsequent compare and swap operation might still be successful. This happens when the value of ptr is changed, and then is changed back to old. This would be sufficient to corrupt our concurrent data structure. One solution to the ABA problem involves associating a counter with every pointer [77]. Every successful compare and swap increments the counter. To implement this solution either a double word version of compare and swap is used (see Section 2.2.1) or array indices are used instead of pointers. Valois [105] uses a single word compare and swap, however he supports it with memory management techniques which are impractical. Michael and Scott report running out of memory many times when experimenting with Valois’ queue implementations [78]. Mellor-Crummey’s [74] lock free queue avoids the ABA problem since it is implemented using a “fetch and store-modify-(check-write)” cycle instead of the usual “read-modify-compare and swap” cycle. However, this removes the non-blocking property from the data structure.

2.3 Scheduling

In this section we present scheduling techniques. We first introduce the traditional UNIX process and the concepts of threads at the kernel and user level. We conclude the section by having a look at some applications which make use of efficient thread scheduling.

2.3.1 Processes

Traditional UNIX operating systems (and most modern operating systems) define the process entity as an instance of a running program, generally providing an execution environment, the program code (also known as the text section), the program counter, the contents of the processor’s register and a stack (containing temporary data). More than one process may run the same program at the same time (or at least give the impression that they are running the same program at the same time).

A process is a schedulable entity, only one process can be run on one CPU at a time. Processes also have to contend for various resources such as memory and peripheral devices. Processes are also responsible for executing system services by means of system calls which
are executed by the kernel. Processes usually exist in a hierarchy. The main process under most UNIX systems is called the `init` process [102]. Most other processes are children of the `init` process or children of some other process in the hierarchy. All processes have one parent and can have one or more child processes. Processes create child processes by means of the `fork()` or `vfork()` system calls [97]. Processes end when the system call `exit()` is called. More than one program can be executed in the lifetime of a process. New programs can be run through the `exec()` system call. Processes also provide interprocess communication so that independent processes can communicate with each other. Figure 2.10 illustrates the role of the traditional UNIX kernel and their relation with the kernel and underlying hardware. Figures 2.5, 2.6, 2.7, 2.8 and 2.9 serve as an introduction for the figures related to thread scheduling in the rest of the document.

The process properties mentioned above, i.e. process creation, process termination and interprocess communication, are expensive and are only meant to be run a few times a second. These type of calls usually involve system calls into the kernel. This type of context switch is known as a vertical switch. UNIX operating systems and most other modern operating systems schedule runnable processes by means of time slicing. Every process is allocated a certain amount of time to run on one given CPU. When the time slice expires the next runnable process is scheduled onto the CPU. This gives the user the illusion that more than one process is running concurrently, even on a uniprocessor machine. This type of context switching, known as horizontal switching, incurs the overhead of the clock interrupts for the time slicing, the saving and restoring of the register’s context and program counter and changing memory protection boundaries.

### 2.3.2 Scheduling policies

Scheduling policies allocate time or processors to processes. Multiprocessor operating system scheduling strategies could be divided into time sharing and space sharing.

Time sharing involves allocating processor time between different tasks. A traditional time sharing scheduling strategy is time slicing. Another strategy is gang scheduling or coscheduling [42, 83, 100]. Gang scheduling attempts to execute processes from the same job simultaneously. Gang scheduling assigns as many processes from the same job as possible to processors simultaneously. It also preempts them simultaneously. Gang scheduling is suitable for frequently communicating processes. Since processors from the same job run concurrently, if a process holds a lock, it is preempted together with all the other processes, thus avoiding extended spinning.

Space sharing on the other hand involves dedicating processors (or a group of processors) for a particular job. An example of space sharing is processor partitioning [41, 101], in which groups of processors could be assigned to a given job for a given period of time. This solution does not disturb the cache. Static processor partitioning involves assigning processes to processors and if need be assigning exclusive use. The `psets` library [63] is an example of static processor partitioning. Dynamic processor partitioning is a scheduling policy proposed by Tucker and Gupta [101] that attempts to use the same number of processes as there are processors for running an application. An application polls a scheduling server to find out how many processes it should run. If the number is less than the number of processes it suspends some processes. This solution involves interaction between the kernel and the application.
CHAPTER 2. SMP SYNCHRONISATION AND SCHEDULING OVERVIEW

Figure 2.5: Key for forthcoming figures. Subsequent figures illustrate the functionality of some of the items.

Figure 2.6: Our diagrams show details at the hardware level, kernel level and user level. In this case two kernel threads are bound to a single CPU.
Figure 2.7: A single user thread is running on a single kernel thread bound to a processor.

Figure 2.8: Threads are blocked on a communication construct. Communication constructs are not specified in diagrams, they are assumed be either a CSP channel or a semaphore.
CHAPTER 2. SMP SYNCHRONISATION AND SCHEDULING OVERVIEW

Figure 2.9: Four kernel threads each of which is executing a user thread, are attempting to acquire access to the same shared resource.

Figure 2.10: Traditional UNIX processes and their relation with the kernel.
2.3.3 Threads

A thread also known as a light weight process (LWP) is also an instance of execution of a program, but there can be more than one for a given program. A thread consists of a program counter, a register context and a stack [92]. It shares the address space and other operating system resources with fellow threads in a given process. A traditional UNIX process can be thought of as a process with one single thread of control. Unlike the heavy weight process, since LWPs share resources between them, creation of threads and horizontal switching time is severely reduced. Threads share many properties with processes. Threads are created, scheduled, destroyed etc... as UNIX processes are. They can synchronise between each other and create child threads just like there heavy weight counterparts. Threads share similar running states to processes too. Some threads will be blocked, others running, others will be terminated etc... One important aspect about threads is that they offer no memory protection over shared data. Memory protection must be taken care of through the application itself. Threads are also advantageous since they can be used at an application level to express, create and control parallelism [6].

2.3.4 Multithreading in modern operating systems

Modern operating systems offer different support for threads. We now describe the association between processes, kernel-level threads and user-level threads.

Kernel threads

Kernel threads offer a level of thread management at the kernel level. Kernel threads usually carry with them the same overhead of processes except with the added advantage that no memory protection needs to be offered between kernel threads that share the same address space and memory mappings do not need to be flushed for every context switch. Kernel threads can be used to implement various traditional processes such as the pagedeamon process or more recently on Solaris to implement network daemons. Implementations of kernel-level threads in multiprocessing environments can provide single applications with true parallelism. Kernel threads can switch between various processors or be assigned statically to particular processors.

The main disadvantage of kernel threads is that they involve the kernel by means of system calls on creation, destruction, scheduling and in some cases synchronisation between each other. A context switch between kernel threads while not as expensive as that of a process still incurs an overhead in the millisecond range. Furthermore if switching has to occur regularly, or if the threads need to share data and synchronise frequently, kernel threads are not so suitable [6, 15].

User threads

User-level threads provide mechanisms for thread creation, destruction, scheduling, inter-thread communication etc... at a user level. The kernel need not know anything about them. It is also possible to create multithreading environments on operating systems that do not have support for thread scheduling.

User-level thread libraries provide an abstraction from the kernel. A user-level thread library provides a powerful utility for application programmers to run jobs which can be
created, scheduled and maintained in general without expensive system calls to the operating system’s native kernel. This makes the user-level thread library a mini-kernel on top of the operating system [102]. As was mentioned in Section 2.3.3, a thread as an entity is made up of a program counter, a stack and a register context. It is possible to create and handle these thread properties without any means of intervention from the kernel. A context switch would just involve saving the register context, stack and program counter of the currently running thread and loading the thread properties of the next runnable thread.

The main advantage of user threads is performance. The following table [102] gives a clear indication of the benefits of using threads at the user level. The table shows latency of user thread, kernel thread and process for creation and synchronisation time (using semaphores), in microseconds, on a SPARCstation 2 under Solaris.

<table>
<thead>
<tr>
<th></th>
<th>Creation Time</th>
<th>Synchronisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>User thread</td>
<td>52</td>
<td>66</td>
</tr>
<tr>
<td>Kernel thread</td>
<td>350</td>
<td>390</td>
</tr>
<tr>
<td>Process</td>
<td>1,700</td>
<td>200</td>
</tr>
</tbody>
</table>

User-level thread implementations include POSIX pthreads [56, 81], Mach Cthreads [27], Sun OS threads [84], MESH [17, 18, 25], the KRoC scheduler [108, 110], Filaments [37].

2.3.5 High performance multithreading

Multithreading serves as a means of harnessing concurrency for high performance applications. In this section we attempt to demonstrate the importance of efficient thread scheduling by describing its use in a few select applications.

Scientific calculations

As an example of high performance scientific calculations which rely on high performance user-level thread scheduling infrastructure, we briefly describe Boosten’s [19] work which lead to the development of the MESH scheduler (see Section 3.1).
CERN [23], the largest laboratory for particle physics in the world, is expecting to have their new Large Hadron Collider (LHC) available for experiments by 2005. One of the experiments expected to make use of the LHC is ATLAS [8], an experiment which attempts to discover the Higgs particle. The ATLAS detector is responsible for recording the head-on collisions of protons, in the form of three dimensional cylindrical digital data images. Out of the 40 million 1MB images produced per second only a few images are relevant. The ATLAS trigger’s job is to filter the required images at a rate of 40 TB/s. The trigger consists of three different levels, but Boosten’s work provides a solution for level 2 (and partly for level 3). After level 1, which is done in hardware, filtering at levels 2 and 3 are expected to processes images at a rate of 100kHz. The selected infrastructure for the level 2 trigger consists of a NOW, where each workstation has to make efficient use of both multitasking and communication due to the fine grained nature of the application. Communication is expected to occur approximately once every 4,000 to 8,000 instructions. Boosten investigates two possible infrastructures for the development of the NOW. In terms of scheduling, the first project, DSNIC uses the Linux OS, the second uses a purpose build user-level thread scheduler MESH. MESH is preferred over DSNIC because of its user-level nature both in terms of scheduling and communication. Of particular relevance to our discussion is Boosten’s identification of one of the major shortcomings of the DSNIC project; the Linux OS’s context switch time which is recorded at 12\mu s making it relatively slow compared to MESH’s context switch time of 55\mu s.

**Server side software**

Multithreading has become an increasingly favourable method for developing server side software. Hu, Mungee and Schmidt [50] compare the performance of web servers that use different mechanisms to handle client requests. Results demonstrate that the traditional process-per request mechanism used by popular web servers such as Apache [7] does not perform as well as thread-per request solutions, whereby a new thread is created to handle a new request. The cost of creating a thread in this experiment was estimated to be 20 times faster than when using fork(). Moreover, thread pools, a mechanism in which threads are created upon initialisation and re-used provided even better performance than the thread-per request solution. Schechter and Sutaria [87] demonstrate that context switching between processes can reduce web server performance and use POSIX threads to alleviate the problem. The threads used in the above experiments were kernel-level threads. The use of user-level threads is meant to improve performance further (see Section 2.3.4 for a discussion of user-level threads and kernel-level threads). However, when using user-level threads for server software, one must be aware of user threads issuing blocking system calls that have the potential of blocking the entire scheduler (solutions for this problem are discussed in Section 2.6.1). Borg [20] presents ActServ, a static web server that supports HTTP/1.0, which is implemented by means of combination of two of our own uniprocessor and SMP smash schedulers, and a scheduler activations patch for Linux [32]. Borg’s results demonstrate that ActServ outperforms Apache’s web server substantially. Barnes [11] obtains similar results for his web server implemented by means of a modified KRoC scheduler that avoids blocking system calls.
CHAPTER 2. SMP SYNCHRONISATION AND SCHEDULING OVERVIEW

Real-time systems

Real-time operating systems (RTOS) are common in a majority of applications ranging from electrical appliances to airplane navigation systems. In RTOS the time at which results are produced is as important as the results itself. Threads in RTOS usually need to meet a series of deadlines imposed upon them by the thread scheduler. In certain cases, the failure to conform to the deadlines (e.g. in an air traffic control system) might lead to loss of lives. Since the required response times for such systems can typically be within the microsecond [21], making use of traditional operating systems that have thread management overheads of hundreds of microseconds is impractical for such tasks. Consequently, a large amount of time and effort is spent in ensuring that RTOS’s thread scheduling is efficient and reliable. This has led to the development of a large number of commercial RTOSs with fast thread management mechanisms such as QNX [86] and ThreadX [99], which advertise high performance context switching, of well below a microsecond, as their strong selling points.

2.4 Locality and load balancing

Parallel applications execute well when all processors are sharing the workload and the running tasks are located close to their data. Lack of load balancing leads to wasted CPU cycles since not all processors would be running for the length of an application. When a processor cannot reference data close to it, it needs to communicate, and this inevitably leads to contention [3, 68].

Load balancing techniques attempt to distribute workload around all the processors. Jobs must be shared evenly among all processors. Normally tasks are selected from a shared run queue and distributed equally amongst processors. When a processor is idle it is assigned a job from the queue, when it is preempted (or the job finishes) it selects another from the shared queue. The jobs can be set statically at load time or dynamically at run time. If the jobs are allocated dynamically this could involve migrating jobs to other processors, when one of them is idling and there is an unbalanced workload.

Locality tends to group together a task and the data it frequently accesses. At its simplest form, on a uniprocessor machine, the locality based mechanisms try to exploit the cache. Results by Devarakonda and Mukherjee [29] have shown that cache exploitation is not suitable for fine grain multithreading, since Devarakonda and Mukherjee point out that to be effective, cache affinity must consist of long lived, frequently synchronising processes that access large areas of memory. Vella [107] introduces a scheduling strategy which caters for both frequently scheduling long lived threads at a user level. Vella proposes to batch threads into groups of threads which synchronise with each other frequently and have a run queue of batches of threads instead of individual threads. Each batch could be considered a run queue of threads. Vella concludes that performance is improved by using this technique, especially when granularity is low and threads access the same data more than once before de-scheduling. On a multiprocessor machine, there are other advantages to localising data. Besides exploiting caching, locality policies attempt to minimise communication across the bus or switched networks [71].

Load balancing and locality both lead to improved performance, however both policies interfere with each other. For example, a system that favours load balancing would assign initial tasks to individual processors, while a locality based scheduling strategy would place the processors close to their data or to their parent [3]. During execution load balancing
would migrate tasks across processors, while a locality friendly policy would prefer to re-
execute tasks on the processor they where initially running on. Clearly a conflict exists
between load balancing and locality. Eager et al. [35] resolve the discussion in favour of
locality in distributed systems. Bellosa [12] arrives to similar conclusions regarding NUMA
multiprocessors. However the general trend on UMA multiprocessors was to base scheduling
policies on load balancing mechanisms, usually by means of a shared run queue [25, 108].
Squillante and Lazowska [95] demonstrated how a process develops an affinity to a proces-
sor. On the other hand Gupta et al [42], and Vaswani and Zahorjan [106] results showed
little improvement when use cache affinity scheduling. Vella [107] argues that these results
could be due to both experiments relying on processes that do not synchronise frequently.
Markatos and LeBlanc [70] observe the improvement in CPU speed as opposed to memory
access will lead to a change in scheduling strategies that will make further use of locality
based techniques. They previously [68] compared the conflicting policies and suggested
that locality management has greater benefits. They propose techniques to increase per-
formance of fine grain thread scheduling using “memory conscious scheduling” [69], which
groups threads that access the same data areas.

2.5 User-level thread issues

In this section we discuss the various issues effecting user-level threads.

2.5.1 Co-existence with existing platforms

One of the first points to be aware of is that user-level thread libraries can provide high
performance computing on commodity platforms [18, 66, 80]. Besides creating high per-
formance scheduling environments on already established platforms, user-level thread en-
vironments co-exist with operating systems designed for these platforms. The user thread
scheduling environment therefore acts just like a native operating system process. Fur-
thermore the user-level thread library can also use the native operating system drivers,
file system, compilers, libraries, debuggers etc... Boosten [19] describes these ideas with
regards to MESH and the Linux operating system.

2.5.2 Performance issues

Anderson et al. [6] argue that three factors contribute to the cost of thread parallelism in a
program.

Thread overhead Thread management must be less than the work performed by the
thread.

Communication overhead The cost of sharing information among threads must be less
than the cost of computing the the information in the same thread.

Programming overheads This reflects the amount of overhead involved by the program-
mer to write the program.

Another important aspect Anderson et al. discuss is the granularity factor of concurrency.
Granularity is determined by the frequency of synchronisation and management of
threads. Fine grained programs will potentially have thousands of threads being created,
synchronised and switched between in rapid succession. On the other hand coarse grained scheduling deals with threads that do not synchronise frequently. It is obvious that the thread management and communication overhead must be very low to support fine grain multithreading.

We now discuss these issues separately. The programming overhead is a factor influenced by the programmer and the ease of use of the associated parallel programming language. We will therefore discuss the choice of language as a factor of performance.

### 2.5.3 Language implementation

A programmer writing a parallel application has a choice of three types of languages with which to write his application:

1. A language in which to write the program in a standard sequential manner and have the compiler “parallelise” the program.

2. An extension to an already existing sequential language.

3. A language written directly for parallel computing.

Almesi and Gottlieb [1] give a detailed description of the various programming language options available to potential parallel programmers. We will be mainly interested in the second item, an extension for an existing programming language. In our case C [60]. C is already an established programming language and is relatively portable among various platforms, especially across UNIX platforms. Certain techniques such as active context switching (see Section 2.6.3) are achieved by means of the gcc compiler\(^2\), which is available for most modern operating systems. Although a user thread library is more than just an extension of a programming language, in the case of MESH, pthreads, Solaris’ user threads, smash etc., from the application programmer’s point of view this may indeed be the case.

### 2.5.4 Communication

Most communication overhead is reduced by supporting user threads at the user level avoiding expensive kernel-level access. [14, 18, 80] provide examples of such an implementation.

**Communication with external events**

Traditionally communication with external events is handled by means of hardware interrupts. Interrupts require that current execution context is saved and that the operating system scheduler is invoked. An alternative to interrupting thread execution is polling. Polling requires that the application programmer specifies the places where preemption can take place and have the external handlers handle synchronisation of external events at these points. The advantage is that the compiler will know the context state at these points.

One disadvantage of polling as described above is that there might be long sections of code in which the scheduler is not invoked. This would result in a long list of unhandled external events. A solution to this problem is automated polling. Automated polling involves the insertion of polling statements automatically. We would typically do

\(^2\)Techniques for using active context switching without gcc do exist [80], but these are not as efficient.
this in long computation sections of code. The same technique could be used for automatic preemption techniques (see Section 2.6.2). Barnes [9] describes how to implement such a technique.

**User-level hardware access**

User-level hardware access offers the same advantages that user-level thread scheduling does, in the sense that it avoids expensive kernel access. Unfortunately hardware access at the user level is not protected. The only protection is that the user-level thread scheduler operates in its own memory space, so this should prevent the user-level hardware access from interfering with other OS processes [16, 18].

**Zero copy protection**

Zero copy protection provides a mechanism of accessing the memory only once [18]. When the NIC card receives a message it writes it to memory and when sending, it reads it from memory. More conventional techniques would require a NIC receiving data to place it in a temporary buffer, then to a register then to main memory (this process is known as single copy protection). Zero copy technique removes most of the CPU load associated with single copy protection. Zero copy protection does however have certain constraints since data must be placed in a consecutive area of memory. This might mean that on certain NICs all the data must be sent in a single packet.

### 2.6 Thread management

Besides scheduling strategies which will be discussed in the next section, thread management and scheduling performance can be improved further with a number of techniques we discuss below.

#### 2.6.1 Interaction between kernel and user threads

There are many types of implementations for how user-level thread libraries interact with the operating system. On operating systems that are not multithreaded, there is really only one solution, that of having the thread library as a separate program with no kernel interaction whatsoever. However, by means of interaction between kernel-level threads and user-level threads, powerful operating environments can be created. Kernel-level threads could be used by the user-level thread library to multiplex user-level threads [96]. Figure 2.11 provides an example of such a configuration. When dealing with multiprocessor machines this type of configuration provides true parallelism between the various threads. When a user thread calls a blocking system call, the entire kernel thread blocks inside the kernel not allowing user-level threads to use the blocked kernel thread. Giving the user-level thread package control over how many kernel threads should be used, might allow another kernel thread to be run instead of the blocked kernel thread [43]. One disadvantage is that if more kernel-level threads than there are processes are used, there will be a vertical switching overhead [107].

The lack of communication between kernel threads and user threads prompted Anderson [5] to come up with a solution in the form of scheduler activations. Scheduler activations offer user-level thread libraries user thread functionality as well as kernel-level interaction (similar to those available in kernel-level threads). The implementation introduces a new
entity to the thread entities, i.e. a scheduler activation, which is similar to a kernel thread. The scheduler activations model also introduces a call from the kernel to the user-level thread library called an upcall. The upcall passes the activation to the library. Each processor has one activation. An interesting feature of this architecture is the handling of user threads that block in the kernel. When a user thread blocks the kernel informs the library by means of an upcall function and passes it a new activation, on which to launch a new user thread. The old activation is passed back to the kernel. When the original thread completes another upcall function with a new activation informs the library that the blocking operation has completed. Scheduler activations provide very fast abstractions since they do not involve the kernel but their kernel interaction provides flexibility that aids the user-level thread library.

Vella [107] suggests using wrappers around blocking system calls, so that when a user-level thread calls a blocking system call, it will in effect be calling a function that will handle the blocking call before calling the appropriate system call. While such a solution would work well in an environment such as KRoC [11], since the compiler can easily capture the system calls, implementing it on top of C based threads package is a harder task as it would imply either informing the application programmer of new system calls he/she should use or re-writing the system calls API.

Inohara and Masuda in [57] propose using a read only memory area, called the c-area, for user threads to monitor kernel-level thread status and take adequate action at the scheduler’s own time. This mechanism helps avoid vertical switching since the user can check kernel status from the c-area. Yet another solution for user threads that block in the kernel is proposed by Koppe in [61]. The thread library parks spare threads in the kernel, when a user thread blocks in the kernel, the mechanism can switch to one of the spare threads. Solaris also provides a facility by means of the SIGWAITING [84] signal to wake up a kernel thread if all the kernel threads in an address space are blocked in the kernel. One of the kernel threads is awoken to receive the signal and can take the required action. Marsh et al. [72] propose interactions between user and kernel for the Psyche operating system by means of shared data structures, a scheduler interface and software interrupts.

An alternative approach is to have the underlying kernel cater for blocking system calls, by means of a call-back mechanism. User threads will be temporarily suspended when making such calls, but the kernel thread itself will never block. Windows NT provides such a service by means of an asynchronous procedure call (APC) [92].

Linux support for multithreading

Linux from version 1.3.56 offers support for kernel-level threading [109]. Linux defines a thread as a “context of execution”, therefore not keeping the thread entity separate from that of a process. Only one process table is required for both threads and processes. Linux achieves this functionality by means of the clone() function. clone() could be considered an extension of the traditional UNIX fork() system call. clone() like fork() creates a child process/thread. clone() will allow the parent and child to share various resources, memory, file descriptors, signal information and file system information. These are set by flags that determine the sharing mode. clone(0) i.e. clone() with no sharing is equal to fork(). LinuxThreads is an implementation of POSIX pthreads [56, 81] for Linux that also supports kernel threads. Recently support for processor sets psets [63] and scheduler activations [32] have been implemented for Linux.
2.6.2 Context Switching

As was previously mentioned the main advantage of user-level threads over kernel threads and processes is that user-level threads do not have the same overhead as the latter two entities, when creating, synchronising and most importantly when switching between jobs, since there is no interaction with the kernel. Both kernel threads and processes use preemption to interrupt a currently running task and schedule another one. In user-level thread scheduling we have the option of using non-preemptive multitasking. The main advantage of using preemption is that each thread would be given a relatively equal time-slice of execution. The main disadvantages of preemption is that handling clock interrupts and invoking the scheduler is expensive. Moreover the threads context must be saved. The full context of a thread could be very large. On a Pentium processor, which has a relatively small register set, the register information would be about 150 bytes. On a SPARC, which has a comprehensive set of registers, about 900 bytes would need to be saved on every context switch. An alternative approach would be to have de-scheduling points in the program that would be responsible for de-scheduling the currently running thread. We will thus be using non-preemptive multithreading. A function such as yield() would be responsible for labelling the de-scheduling points. In this way the compiler would be able to control the amount of context to be saved. This technique is known as active context saving and was presented by Moores [80] and further extended by Boosten for MESH [17]. The facility to yield() could be either left to the discretion of the programmer or inserted in the compiler at certain points that are considered suitable for yielding. Such a solution is implemented by Barnes in [9] for communication (Barnes does not use the compiler but pre-parses the program to be run inserting polling calls where appropriate before compilation). However this solution can be applied for yielding thread execution also.

2.6.3 Active context switching

Active context switching uses gcc’s inline assembly [94] function asm() which allows assembly code to be inserted into C code. The asm() function has the following format:

\[
\text{asm("assembly code" : input parameters : output parameters : corrupted registers);}\]

Of special interest in active context saving is the corrupted registers parameter. If all registers are listed here then the compiler is informed that after the assembly code all registers will be corrupted. The variables in the registers need to be reloaded from memory and those already there need to be saved in memory before the assembly code. Moreover, this forces the compiler to save only the active registers. This can be accomplished since the compiler decides which registers are going to be used during code generation. Therefore context switching will not need to save the entire context of the registers. The only registers that need to be saved are the stack and base pointer registers. Furthermore if context switching does not take place between floating point operations, floating point registers will not need to be saved, providing considerable performance gain. Active context switching could be used in conjunction with automated insertion of yield functions discussed above.

2.6.4 Inline context switching

Another technique use by Boosten [18] and implemented for MESH, is inline context switching. The context switch is not implemented as a function call, but as an inline function
2.7 Scheduling Strategies

Our interest in synchronisation, optimisation, load balancing and locality is mainly fuelled by the desire to find a high performance scheduling strategy to drive our threads package.

2.7.1 Shared run queues

On SMP workstations it was believed that since migration of processes is not too high, it would be inexpensive to implement a central run queue that will feed all the processors equally [68]. Figure 2.12 demonstrates how a shared run queue services all processors. Processors take threads from the queue, execute them to completion or place them back onto the queue, and pick up another thread. The principal of load balancing is favoured here. Vella [108] in his SMP implementation of the KRoC [110] scheduler and Cordina’s SMP MESH [25] make sure that no processor will ever be idle if the number of processes is greater or equal to the number of processors.

Concurrent queues

Shared run queue performance at fine granularity depends highly on the synchronisation mechanism of the queue. Fine grain thread schedulers employing concurrent queues using
single lock synchronisation techniques have already been implemented [25, 108], we now have a look at other possible concurrent queue implementations. Mellor-Crummey’s lock free queue implementation [74] differs from the normal “read-modify-compare and swap” cycle and instead implements a “fetch and store-modify-(check-modify)” cycle which however makes the queue blocking. Stone [98] presents a lock free queue that is non-blocking and non-linearisable (i.e. the semantics would seem to be different than that of a normal queue). Massalin and Pu [67] develop a lock free operating system kernel and present an efficient lock free concurrent queue, however their implementation uses a double compare and swap (see Section 2.2.1) instruction only available on Motorola’s M68030. Valois [103, 104, 105] developed a lock free queue that was efficient and did not use any compare and swap variations. The use of a dummy head (first suggested by Sites [93]) simplified special case handling. Valois also developed memory management techniques to combat the ABA problem, see Section 2.2.3. Michael and Scott [78] corrected race conditions in Valois’ memory management. Michael and Scott [77] present a concurrent queue which uses two spin locks, one for every dequeue operation and one for every enqueue operation. This is made possible using the same dummy head technique employed by Valois [103]. Michael and Scott’s results show the dual lock queue performs better than the single lock queue on SMP machines with a substantial amount of processors (more than five). In the same paper [77] Michael and Scott present a non-blocking concurrent queue algorithm. They use a dummy head similar to Valois’ [103] and sequences of reads to re-check the value of the pointers’ earlier values. In a separate paper [79], Michael and Scott compare their algorithms performance with that of various concurrent queues mentioned above (Valois, Mellor-Crummey, Prakash et al. [85], single locks, their own two lock queue and the locked queues using Elder et al’s [36] preemption safe locks). They run three tests, with different levels of multiprogramming. Their non-blocking queue resulted the fastest in all the tests when three or more processors are used.

2.7.2 Per processor run queues

Per processor run queue schedulers use a separate run queue for every processor. See Fig. 2.13. Markatos and LeBlanc [68] argue that per processor run queues favour the principle of locality and Anderson et al [3], when comparing results with other run queue structures, found per processor run queues to be more efficient than shared run queues in reducing contention. Anderson’s scheduling strategy involves placing new threads on the run queue of the processor on which it was created. Processors first check their own run queue for threads to execute, if none are found they scan through other run queues. Threads that are created on one processor spend their entire execution time on that processor’s run queue, thus maintaining cache affinity. Rarely do threads migrate to other processors. Based on the above results thread packages such as Filaments [37] adopt per processor run queues as the scheduling strategy.

The choice of run queue implementation is not as important as the choice for the shared run queue. Shared run queue strategies rely on fast synchronisation techniques to reduce contention. Per processor run queue structures reduce contention naturally [3]. When protection is required spin locks are usually used. Markatos and LeBlanc [69] observe while using per processor run queues in their “memory conscious scheduling” techniques that fine grain thread applications are comparable to coarse grain thread applications.
Figure 2.13: Per processor run queues. The migration details in this diagram are not specified.
CHAPTER 2. SMP SYNCHRONISATION AND SCHEDULING OVERVIEW

Batching

Vella [107] proposes a technique to not only exploit cache affinity but to reduce contention. Vella’s technique was implemented successfully for the uniprocessor version of the KRoC scheduler (see Section 2.4) and demonstrated that this technique improves performance by exploiting the cache. Vella proposes grouping up common threads into batches. The scheduling strategy involves maintaining a shared batch queue (see Figure 2.14). Each processor then selects a batch from the shared batch queue (as was seen in Section 2.7.1 in the shared run queue case). When acquired by a processor, the batch in itself would represent a run queue. When a thread is created, it is placed on the currently running batch of the processor. Batches have a fixed size, so when a batch is full, newly created threads are placed on an overflow queue. When the overflow queue is full, it is placed onto the shared batch queue (other events might trigger placing the overflow batch on the shared batch queue). When a batch is running it is assigned a counter representing a number of possible dispatches. When the counter reaches a certain value the batch is placed back onto the shared batch queue and another batch is serviced. The rationale behind batching is that batches act as a per processor run queue, therefore have the advantages associated with per processor run queues, reducing contention [3] and improving performance by means of cache affinity [69]. The shared batch queue also reduces contention because it is accessed less frequently than normal threads and also helps to maintain a balanced workload.

2.8 Summary

In this chapter we have introduced parallel architectures and techniques for synchronising jobs on multiprocessor systems. We introduced schedulable entities in the form of UNIX processes, kernel threads and focused our attention on user-level threads, where we highlighted techniques to improve the performance of user-level thread management. We discussed the issues of load balancing and locality as related to scheduling on multiprocessor systems. Finally we discussed the advantages and disadvantages of possible SMP scheduler configurations.
Figure 2.14: A quad processor implementation of SMP batching.
Chapter 3

Uniprocessor scheduling

This chapter introduces our scheduler in its uniprocessor incarnations. We will discuss the techniques used to implement three different uniprocessor schedulers. We evaluate our uniprocessor schedulers by comparing them with already existing schedulers such as MESH and with each other, and show by means of examples how the different run queue configurations are more suited for certain applications than others.

We begin by detailing features common to the three schedulers, then we introduce our three schedulers sequentially, explaining the motivation behind each one, we describe implementation details, presenting results and discussing the advantages and disadvantages of the schedulers compared to each other. This chapter also serves as an introduction to the smash schedulers, whereby we specify the internal scheduler calls and the general API.

3.1 MESH

The uniprocessor smash is based on Boosten’s MESH [18]. MESH itself is based on previous work by Moores [80]. MESH is a non-preemptive priority based user-level thread scheduler. Techniques used to speed up MESH where outlined in section 2.6. smash borrows many of the ideas used in MESH, including active context switching, thread management and synchronisation techniques. smash strips off the external communication interface, priorities and some of the MESH’s API function calls. In some cases smash also improves slightly on MESH’s performance, improving the context switch time from 69 ns for the latter to 57 ns1.

3.2 Scheduler details

We now present the underlying architecture which is the same for all uniprocessor schedulers, describe scheduler startup and shutdown procedures and specify the internal scheduler functions and the API.

3.2.1 Underlying architecture

Before delving into the individual scheduler’s implementation, we present the underlying architecture of all three uniprocessor smash schedulers. The user-level thread scheduler exists in one process, i.e. the address space is accessed by only one kernel thread. No multiplexing of kernel threads is used. Figure 3.1 demonstrates the underlying architecture,

1Performance results and the tests used to calculate them will be discussed in section 3.3.4
Figure 3.1: The underlying architecture of the uniprocessor schedulers. Specific run queue details will be presented for every uniprocessor scheduler in the appropriate sections.
without specifying any run queue details. Context switching relies on the use of two functions. `JmpBuf_Set()` is used to save the current context. `JmpBuf_Jmp()` is used to restore a saved context. Active context switching is used to speed up performance (active context switching is discussed in Section 2.6). Each thread has its own context and stack. The scheduler does not have its own stack, but borrows the last user-level thread’s stack when the application is executing scheduler functions. This method avoids unnecessary switching to the scheduler’s stack every time we enter the user-level thread scheduler.

**Scheduler startup and shutdown**

We use the same scheduler initialisation and shutdown routines for all three schedulers. Certain details are not exactly the same for all schedulers. For example run queue initialisation would vary from scheduler to scheduler. The system specification we introduce here, in terms of startup and shutdown, will remain intact for the SMP routines. Implementation details will however be more complex (see Chapter 4).

**Scheduler initialisation**

Scheduler initialisation involves initialising the run queue and creating the main thread, which will execute `cthread_main()`. The application programmer’s traditional `main()` function is replaced by the libraries’ `cthread_main()`. `cthread_main()` serves as `main()` for all intents and purposes. The `main()` function is linked in at compile time and launches, the thread representing `cthread_main()`. All other threads are children of `cthread_main()` and must be explicitly created by the application programmer.

**Scheduler shutdown**

Scheduler shutdown occurs when `cthread_main()` finishes executing. It is the application programmer’s responsibility to ensure that all threads he/she intends to execute would have been executed before execution is halted (functions such as `cthread_join()` are sufficient to ensure all threads run to completion).

### 3.2.2 The thread descriptor

The thread descriptor or workspace is the structure that contains the thread’s state and properties, in particular our thread descriptor is composed of:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>JmpBuf</code></td>
<td>Registers and pc context.</td>
</tr>
<tr>
<td><code>stack</code></td>
<td>Pointer to thread’s stack.</td>
</tr>
<tr>
<td><code>pointer</code></td>
<td>Pointer to data buffer or thread, used in channel communication.</td>
</tr>
<tr>
<td><code>chan_length</code></td>
<td>Length of message, used in channel communication.</td>
</tr>
<tr>
<td><code>joiner</code></td>
<td>Pointer to thread to wake up after termination, used in barrier construct.</td>
</tr>
<tr>
<td><code>next</code></td>
<td>Pointer to next thread.</td>
</tr>
<tr>
<td><code>prev</code></td>
<td>Pointer to previous thread, for double-linked run queues only.</td>
</tr>
</tbody>
</table>

### 3.2.3 General functions specification

The general functions of the scheduler can be divided into two, the internal functions (which the application programmer does not have access to) and the API functions. Most of the
API functions make use of the internal scheduler functions. Indeed most of the API function implementations are identical across schedulers. The internal scheduler functions however vary widely between schedulers and are in the most part implemented as macros to ensure both efficiency and portability across schedulers. We begin by introducing the internal scheduler functions.

**Scheduler jump**

The scheduler jump takes care of the context switch. It makes use of the `JmpBuf Jmp()` call (see Section 2.6.3), to switch execution to the next running thread. This is the only routine in this section that remains unchanged across schedulers.

**Thread insert**

The thread insertion routine inserts threads into the scheduler. In the uniprocessor case this simply equates to a thread enqueue onto the run queue. Since there are times when we are sure a thread exists on the run queue, for certain schedulers, there are two versions of the thread insert routine. One version checks for an empty run queue, and the other assumes that the run queue is not empty. The advantage of assuming a non-empty queue is that we make use of one less conditional statement during a thread insert operation.

**Thread remove**

The thread remove routine removes threads from the scheduler. In the uniprocessor case it equates to a thread dequeue operation. Threads are removed from the run queue(s), if the thread terminates or if the thread blocks on a communication construct.

**Thread yield**

The yield call de-schedules the current thread and executes the next thread on the run queue. It is implemented as a scheduler macro not directly in the API implementation to cater for the different versions of the schedulers.

### 3.2.4 The smash API thread functions

In this section we introduce the API routines that the application user can access from the application program. The implementation is the same for all schedulers, however the different run queue configurations affect the performance of an individual function considerably. We will discuss the run queue affects on the API functions in the individual scheduler’s sections.

**User thread initialisation**

User-level thread initialisation involves preparing the thread’s descriptor. Each thread when initialised is associated with a function. Any parameters to the function are also passed as parameters to this routine. The routine also initialises the thread’s context and assigns a stack to each thread. `cthread_init()` is the function responsible for thread initialisation. There are two implementations of `cthread_init()` one which allocates memory both for the thread block and for the stack, and another which assumes that memory was allocated previously. We will explain in Section 3.3.4 why this could be useful.
Thread execution

Thread execution involves the application programmer explicitly launching the thread by calling `cthread_run()`. Thread execution in our uniprocessor scheduler simply inserts the thread to be executed onto the run queue. The thread will not be executed until all threads before it are executed and either terminate or relinquish control by some form of inter-thread communication. Thread execution uses the thread insert routine, described in the previous section.

Thread yield

A call to `cthread_yield()` relinquishes execution of the current thread to the next thread on the run queue.

Thread barrier

`cthread_join()` represents our thread barrier construct. When a thread requests to join with another thread, it suspends its own execution until the other thread terminates. The terminating thread wakes up the original joiner when it terminates. In our implementation a thread can be joined by only one caller.

Thread termination

Although not a specific API call, thread termination occurs when the function representing the thread finishes executing. Thread termination requires the current thread to be removed from the run queue. If the terminating thread’s joining pointer points to another thread, the suspended thread is placed back onto the run queue. The next executable thread is run, unless the terminating thread was `cthread_main()`.

3.2.5 The smash API CSP functions

Our smash schedulers provide inter-thread communication by means of CSP constructs [49]. As was the case with the thread API functions the CSP implementations do not differ between the three schedulers, however they make use of the underlying internal scheduler functions. CSP communication performance is therefore different among the various schedulers. We provide support for three possible CSP constructs, channel input, channel output and channel alternative input.

Channel input

Channel input is the construct which allows point-to-point communication by passing data of any particular type. `channel_in()` accepts as parameters the channel to return the input from, a buffer where to place the message and the message size. We note that both channel input and output copy messages of any type and length, however we could provide alternative variations which are optimised for passing specific types of data. We experiment with versions of channel input and output using only integers (see Section 3.3.4).

`channel_in()` checks the channel word to see if another thread has already tried to output on the channel. If the word is not `NULL`, the channel word contains the address of the thread descriptor address on the other side of the channel. The message is then copied from the outputting thread’s workspace to the buffer, the channel word is reset and the
outputting thread is placed back onto the run queue. Message copying is implemented by
copying the minimum number of bytes specified by the input length and the output length.
If the channel word was initially `NULL`, the channel word is made to point to the current
running thread, which has its context saved and is removed from the run queue, until the
thread communicating on the same channel wakes it up.

**Channel output**

The `channel_out()` at its most basic performs a function similar to `channel_in()` above. How-
ever the direction of the communication is reverted in this case. `channel_out()` is also
required to handle inputs from the alternative input construct.

In both cases when `channel_out()` detects a `NULL` channel word it sets the channel word
to point to the current thread and de-schedules the current thread. In the case of a non-
`NULL` channel word, we check the content of the buffer pointed to by the channel word which
contains either a pointer to a buffer, in which case `channel_out()` acts like `channel_in()`
above (it however copies the message from the buffer the user passed to the buffer in the
thread workspace pointed to by the channel word), or the state of the alternative input
routine. If the state is `READY` or `ENABLING` we set the channel word to point to the current
thread, we set the current threads message buffer to that passed to `channel_out()` and
de-schedule the current thread. If the state is `WAITING` we place the thread pointed to by
the channel word back onto the run queue, we set the state to `READY` and perform the same
operations we do in the case of the state being set to `READY`.

**Channel alternative input**

Channel alternative input is a powerful construct which allows a thread to receive input
from more than one channel. The limitations of the C programming language did not permit
us to force onto the programmer a proper `alt` construct as is the case with `occam`, however
we provide constructs to allow for the implementation of `alt`.

The alternative input algorithm [73], is composed of five distinct routines. The first
routine, alternative start, sets the alternative input state to `ENABLING`. The second routine,
channel enable, prepares all the channels for input, if one is already ready the state is set
to `READY`. The third routine, alternative wait, sets the state to `WAITING` and suspends the
current thread, unless a thread is ready to output (in which case the state would be `READY`).
The fourth routine, disable channel, selects the first channel to have been ready to output
and sets a jump address to execute the code from that stage. Alternative end, the fifth and
last stage, executes the code starting from the jump address.

We provide the application programmer a `channel_select()` routine which selects a
channel from a list of channels that are passed as parameters. The `channel_select()` uses
the alternative input algorithm to return a channel which gets selected. The use of a case
statement would then allow the programmer to input from the channel that got selected and
run the associated code also. As an alternative we also provide the five routines which make up
the `alt` algorithm. A pre-compiler could be used to provide a new `alt` C construct that
would enforce the proper use of `alt`. The code would be converted from the new language
into our C based code and the new `alt` construct would be built out of the functions `smash`
provides.
3.3  Circular run queue smash implementation

MESH uses a circular concurrent queue as its run queue. We adopt this feature for our first scheduler. In this section we give a brief description of the smash architecture, discuss the main advantages and disadvantages and present a set of results. We also introduce benchmarks that we will use to test all schedulers, and discuss fair methods of calculating performance.

3.3.1  Introduction

The circular uniprocessor version constitutes one double-linked circular run queue and can accommodate any number of communication constructs. The current thread on the circular run queue represents the currently executing thread. In this way the current thread always remains on the run queue. This technique improves context switch performance times, since context switching in general would involve just moving the current pointer to the next thread on the run queue, instead of removing and inserting threads onto the run queue. Figure 3.2 demonstrates the circular run queue scheduler.

3.3.2  Internal scheduler functions

In this section and the equivalent sections for other schedulers we will describe the specific internal function implementations of the given scheduler. This section and the next will serve to differentiate between the scheduler implementations. The results section will then highlight the advantages and disadvantages of the given implementation.

Thread insert

Thread insertion on the run queue in the general case, involves first checking for an empty run queue. If the run queue is empty we assign the current pointer to the thread to be inserted and make the run queue point to itself in both directions. In the case of a non-empty run queue, we insert the thread before the current pointer and update the previous and next pointers of both the current and new thread. Figure 3.5 describes the operation of inserting a thread onto the circular run queue.

Thread remove

As mentioned before the thread dequeue routine does not check for an empty run queue. In the case of the circular run queue, we check if the thread to be removed is the last in the queue. If it isn’t, we remove the current thread and update the pointers of the next thread on the run queue. We also update the pointer of the last thread in the queue. Figure 3.4 illustrates the thread remove operation.

Thread yield

Thread yield involves saving the current context, updating the current thread pointer to point to the next runnable thread and jumping to that thread. The circular run queue is at an advantage here, since only one assignment is needed to update the current pointer.
Figure 3.2: Circular run queue scheduling.

Figure 3.3: Insert thread onto circular run queue. Note that C represents the current thread pointer in this case, not a communication construct.
Figure 3.4: Remove thread from circular run queue. Note that $C$ represents the current thread pointer in this case, not a communication construct.

Figure 3.5: Circular run queue yield. Note that $C$ represents the current thread pointer in this case, not a communication construct.
3.3.3 Thread API functions

Due to the circular run queue implementation, it is clear that the best use of this scheduler is for applications that call \texttt{cthread\_yield()} frequently (see results, Section 3.3.4). Other API functions such as \texttt{cthread\_run()}, \texttt{cthread\_join()}, all the CSP based functions and the thread termination function which make use of at least one enqueue and dequeue suffer slightly from the fact that we have to update more than one pointer per thread, due to the double linked nature of the queue. This is however necessary since if we used a single linked queue, we would either lose the FIFO nature of the run queue or scheduler enqueue and dequeue would become a linear cost operation since it would involve traversing the whole list.

3.3.4 Results

In this section we introduce the benchmarks on which we perform our calculations. All results in this chapter were taken using a PII 350MHz with 512KB second level cache (16KB instruction and 16KB data first level cache) and 64MB of physical memory, running Linux 2.2.12-20. Swap space is set at 256MB, for a total of 320MB virtual memory. When comparing results with multiprocessor results (in forthcoming chapters), we will be sure to specify what type of machine we will be taking the benchmark results on. Results were taken with all possible optimisation parameters passed to the compiler.

Accuracy of results

External factors, due to the system architecture’s memory management, both in terms of software and hardware, influence the benchmark results. In terms of systems hardware, cache misses degrade performance, and at the operating system level page faults can degrade performance. When an operation accesses data in memory, there are four (in our case) possible places where a copy of the data can be retrieved from (see Figure 3.6). The first and second level caches are the first places where a copy of the data can be found, followed by physical memory and finally virtual memory. When a copy of the data resides in either
of the caches or physical memory, it stays there until flushed away by other data. When used frequently, a copy of the data will easily be accessed in the lower levels of the hierarchy. Fastest results are obtained when data is found in the first level cache. The second level cache is accessed if a cache miss occurs when trying to access the data in the first level cache. It is important to note that both the first and second level caches do not copy data byte by byte or word by word. Instead they copy chunks of 32 bytes (on Intel P6 architecture) which we refer to as cache lines. We can take advantage of this situation by placing frequently used data on the same cache line. Physical memory is then accessed if a copy of the data is not found in the second level cache. Moreover, in modern operating systems, data might not be in physical memory. If the data is not found in physical memory, a page fault is generated and the operating system retrieves the page, usually from a secondary storage device (in our case the hard disk). As is the case with the cache, memory is retrieved from secondary storage in chunks of data we refer to as pages. Pages in Intel architectures are allocated 4,096 bytes each. It is important to note at this stage that Linux does not place newly allocated memory into physical memory until the memory is accessed.

We use the standard UNIX function `gettimeofday()` to calculate the time at the start of execution and at the end. `gettimeofday()` is meant to be accurate up to the microsecond, however to make sure of our calculations we perform a considerable number of operations and average out the result. In certain benchmarks (e.g. thread placement on run queue), running a test for thousands of times, means initialising thousands of threads. When using large numbers of threads to perform the benchmarks, the thread data might no longer be available in the first levels of the memory hierarchy, therefore every operation will also incur the cost of the cache misses and/or page faults. Our benchmarks try and account for the possible performance loss due to cache misses by presenting all results in two different guises, benchmarks that are cache friendly and always find the data in the cache, and benchmarks that do not always (if ever) find the data in the cache. In one particular benchmark, we offer results due to page faults, cache misses and cache hits to emphasise the difference in results, and how the hardware and operating system memory management effects the performance of fine grain applications.

**Thread initialisation**

Our thread execution benchmark calculates the time taken to initialise 25,000 threads. We conduct the test in various ways. The first test can be seen in Algorithm 3.1. As can be seen we initialise 25,000 threads. The initialisation routine we use allocates space for each thread block and stack. Allocation involves the use of calls to `malloc()` which in turn calls `sbrk()` which is a system call. When accessed none of the data in the initialisation routine is resident in cache since it is the first time we access it. Moreover, none of the data is resident in physical memory either. Although the benchmark suffers from three performance degrading issues (system calls, cache misses and page faults), it is mostly effected by page faults. The net result of the operation is 211,684\(\mu\)s for the entire operation. The result of a single operation averages out to 8.5\(\mu\)s per thread initialisation.

**Eliminating system calls**

We systematically try to remove all the causes of poor performance. We start off with system calls. To remove system calls, we provide the application programmer with an alternate function to `cthread_init()`, which does not allocate memory. The application programmer
Algorithm 3.1 Thread initialisation benchmark.

```c
#define THREADNUM 25000 // Number of threads

cthread_main(cthread * cmain, int argc, char ** argv) {
    int i;
    cthread * c[THREADNUM];
    struct timeval t1, t2;

    gettimeofday(&t1, NULL); // begin timer
    for (i = 0; i < THREADNUM; i++) // initiate threads
        c[i] = cthread_init(do nothing, 1024, 0);
    gettimeofday(&t2, NULL); // end timer
}
```

can choose to use static variables and pass them to `cthread_init_static()` or dynamic variables which he must be responsible to allocate memory for. Algorithm 3.1 was modified to cater for static variables and call `cthread_init_static()` instead of `cthread_init()`. The benchmark took 171,950\(\mu s\) to complete. A single initialisation would therefore complete in 6.9\(\mu s\). There is an improvement to be gained from not using system calls, however the overall cost of initialising a thread is still expensive.

Eliminating page faults

One solution that eliminates page faults is to use an area of contiguous physical memory. We can allocate space directly into this physical area. We adapted our scheduler to use a kernel extension for Linux [64], that provides an area of contiguous physical memory. Linux treats this area as a device. When we allocate area in this part of memory we can access it immediately without worrying about page faults. We re-run our initialisation benchmark (see Algorithm 3.1) and instead of using `malloc()` in `cthread_init()` we allocated portions of the contiguous physical area. The benchmark using physical memory took 17,183\(\mu s\) to complete. Every operation thus taking 687\(ns\).

An alternative solution that eliminates both page faults and system calls would be to allocate large chunks of memory at startup and whenever it is required (i.e. right before memory runs out). The advantage is that for most of the time we do without system calls. Moreover, each page in the allocated area can be initialised, and thus be moved to physical memory. We have tested this with our scheduler, and the result is similar to that of physical memory (ca. 687\(ns\)), if enough memory is initialised beforehand. The advantage over using physical memory is that we do not need third party packages. The main disadvantage is that when we run out of memory, the scheduler will be forced to slow down to service the memory initialisation routine. Pre-compilers could be used to give an indication of how much memory would be required.

Eliminating cache misses

Our third issue related with performance deals with cache misses. To prove our hypothesis correct, we call `cthread_init()` 25,000 times, however we use the same 10 threads instead
Algorithm 3.2 Cache friendly thread initialisation benchmark. Internal loop only.

```c
gettimeofday(&t1,NULL); // begin timer
for (k = 0; k < 2500; k++) // repeat a total of 25,000 times
    for (i = 0; i < 10; i++) // initiate the 10 threads
        cthread_init_static([i],do_nothing,
            stack[i],1024,0);

gettimeofday(&t2,NULL); // end timer
```

of 25,000 different ones. In this way we ensure that every time we run `cthread_init()` the thread’s work space lies inside the cache. Algorithm 3.2 demonstrates the internal loop of our benchmark. Note that we use `cthread_init_static()` to avoid making system calls. We also initialise the stack beforehand to avoid any page faults. The time reported by the benchmark was 6,773\( \mu \)s, with each operation averaging 271\( \mu \)s. The table below summarises the various benchmarks. The improvements cascade (i.e. each test inherits all the improvements of the tests before it).

<table>
<thead>
<tr>
<th>Improvements</th>
<th>Time (( \mu )s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>8,467</td>
</tr>
<tr>
<td>Eliminated system calls</td>
<td>6,878</td>
</tr>
<tr>
<td>Eliminated page faults</td>
<td>687</td>
</tr>
<tr>
<td>Eliminated cache misses</td>
<td>271</td>
</tr>
</tbody>
</table>

Thread insert and remove

Our thread insert benchmark is similar to the thread initialisation benchmark in that it calculates the time taken to place 25,000 threads into our scheduler, ready for execution. In reality, in the uniprocessor case we just calculate the time to place 25,000 threads onto the run queue. The threads are not executed. We simply calculate the time taken to place all the threads onto the run queue then exit. Algorithm 3.3 demonstrates this benchmark.

The net result was 5,797\( \mu \)s to execute the program. The average time taken to place a thread onto the run queue is 231\( \mu \)s. As was the case with the initialisation benchmark, the thread placement benchmark suffers from cache misses. Although no page faults will arise since we do not calculate the time taken to initialise the threads, since we use 25,000 threads, the thread’s data will no longer be in the cache when we place the threads onto the run queue.

We therefore opt to run the test in a cache friendly manner as we did in the previous case. As we did before we run the test for 10 threads, each one being inserted 2,500 times, for a total of 25,000 operations (see Algorithm 3.4). The execution time for the 25,000 operations was 1,264\( \mu \)s, with an average of 51\( \mu \)s per operation.

Note the benchmark cannot be considered a real application since threads that are already on the run queue could be re-inserted, thus corrupting the run queue. However, this benchmark gives an indication of the proper performance of the operation when a copy of the data is found in the cache. A more realistic test would involve modifying both Algorithm 3.3 and its cache friendly version (Algorithm 3.4), by inserting a dequeue
Algorithm 3.3 Thread insert benchmark.

```
#define THREADNUM 25000 // Number of threads

cthread_main(cthread * cmain, int argc, char ** argv) {
    int i;
    cthread * c[THREADNUM];
    struct timeval t1,t2;

    for (i = 0; i < THREADNUM; i++) // initiate threads
        c[i] = cthread_init(do_nothing,1024,0);
    gettimeofday(&t1,NULL); // begin timer
    for (i = 0; i < THREADNUM; i++) // execute threads
        cthread_run(c[i]);
    gettimeofday(&t2,NULL); // end timer
}
```

Algorithm 3.4 Cache friendly thread insert benchmark. Internal loop only.

```
gettimeofday(&t1,NULL); // begin timer
for (k = 0; k < 2500; k++) { // repeat a total of 25,000 times
    for (i = 0; i < 10; i++) { // execute the 10 threads
        cthread_run[i];
    }
}
gettimeofday(&t2,NULL); // end timer
```
Algorithm 3.5 Thread execution benchmark.

```c
#define THREADNUM 25000 // Number of threads

void answer(cthread * ct) { // thread function
    int ans;
    ans += 42;
}

cthread_main(cthread * cmain, int argc, char ** argv) { // perform some calculation
    int i;
    cthread * c[THREADNUM];
    struct timeval t1,t2;

    for (i = 0; i < THREADNUM; i++) // initiate threads
        c[i] = cthread_init(answer,1024,0);
    gettimeofday(&t1,NULL); // begin timer
    for (i = 0; i < THREADNUM; i++) // execute threads
        cthread_run(c[i]);
    for (i = 0; i < THREADNUM; i++) // join threads
        cthread_join(c[i]);
    gettimeofday(&t2,NULL); // end timer
}
```

operation after every `cthread_run()`. Neither version of the benchmark corrupts the run queue and they serve as an indicator of real application performance, such as inter-thread communication, where there are a lot of enqueues and dequeues. The non-cache friendly version of our benchmark took 6,203\(\mu s\) to complete, with an average of 248\(\mu s\) per operation (which is very close to the results of Algorithm 3.3) and the cache friendly version completed in 2,746\(\mu s\) (an average of 110\(\mu s\) for every `cthread_run()` and dequeue operation).

Thread execution performance

Perhaps the best indicator of run queue insertion and removal in terms of performance in a real application environment, is calculating the amount of time taken to execute a number of threads. Our benchmark calculates the time taken to execute 25,000 threads, that means the time taken to place them all on the run queue, execute the associated function until it terminates (including executing the termination code for each thread). As in the previous cases we perform a cache friendly test and a non-cache friendly test.

Algorithm 3.5 demonstrates our non-cache friendly algorithm. Notice timings also take into account the barrier synchronisation routine, in this way we wait for all threads to terminate. The thread execution benchmark took 27,788\(\mu s\) to complete (average of 1.112\(\mu s\) per execution) to run all 25,000 threads.

The cache friendly version runs the same 10 threads 2,500 times. We use a new API call `cthread_reuse()` which resets termination flags in the thread block. Algorithm 3.6 demonstrates the benchmark’s internal loop. The time taken by the benchmark was 7,212\(\mu s\) or 288\(\mu s\) per thread execution.
Algorithm 3.6 Cache friendly thread execution benchmark. Internal loop only.

```c
gettimeofday(&t1,NULL); // begin timer
for (k = 0; k < 2500; k++) {
    for (i = 0; i < 10; i++) {
        cthread_run([i]);
    }
}
gettimeofday(&t2,NULL); // end timer
```

Context switch performance

Our context switch benchmark, calculates how long it will take to switch between 20 threads for a total of 10,000,000 context switches. The benchmark can be seen in Algorithm 3.7. The `cthread_join()` operations ensure that each thread runs to conclusion.

The results from running this benchmark, averaged 572,768μs. We conclude that one single context switch takes around 57ns to complete. The same benchmark under MESH took an average of 697,893μs, with a average context switch time of 69ns. The difference in performance is due to `smash` using macros to replace functions in frequently used routines and since `smash` does not need to look out for external communications and real time events.

It is clear from the previous discussion that the context switch benchmark is cache friendly, in that every time we access the thread’s data a copy is found in the cache. For completeness sake we run the same algorithm, this time with 25,000 threads yielding for a total of 10,000,000 times. We also modified Algorithm 3.7 to remove the `cthread_join()` part, since 25,000 barrier operations would interfere with the overall result. Instead we maintain a global yield counter which is incremented before every yield. When 10,000,000 yield operations are performed the program returns the time taken and exits. Note that the same program run with 20 threads (to be cache friendly) gave the same results as the test above (i.e. 57ns per context switch). The non-cache friendly version took 4,529,089μs to complete, meaning that every context switch operation costs 453ns.

Inter-thread communication performance

The benchmark we use for our inter-thread communication is the `commstime` [111] benchmark developed to test the performance of the KRoC scheduler. The `commstime` benchmark calculates the time taken to perform a series of channel communications. We have adapted the `commstime` benchmark used in MESH for our benchmark. The MESH benchmark makes use of the sequential `delta` routine. We had to change the code slightly since MESH only passes integers along channels. Our version of the `commstime` benchmark outputs two results, the time taken for 1,000,000 loops of the communications to complete and the time taken for each context switch in the communication loop. The Linux KRoC `commstime` (using sequential `delta`) performs 1,000,000 loops in 1,778,149μs, and the context switch
Algorithm 3.7 Context switch benchmark.

```
#define THREADNUM 20 // Number of threads
#define YIELDNUM (10000000 / 20) // Number of yields per thread

void thread_test(cthread * ct) {
  int i;
  for (i = 0; i < YIELDNUM; i++) {
    cthread_yield();
  }
}

cthread_main(cthread * cmain, int argc, char ** argv) {
  int i;
  cthread * c[THREADNUM];
  struct timeval t1, t2;

  for (i = 0; i < THREADNUM; i++) { // initiate threads
    c[i] = cthread_init(thread_test, 4096, 0);
    gettimeofday(&t1, NULL); // begin timer
    for (i = 0; i < THREADNUM; i++) { // execute threads
      cthread_run(c[i]);
    }
    for (i = 0; i < THREADNUM; i++) { // join threads
      cthread_join(c[i]);
    }
    gettimeofday(&t2, NULL); // end timer
  }
```
time is $222\text{ms}$. It must be noted that the KRoC scheduler occasionally preempts the current thread.

We conduct two different tests. The first test involved using the standard `channel\_in()` and `channel\_out()` (with the alternative input code). The test took 1,553,877$\mu$s to execute every 1,000,000 loops and an average of 194$\text{ns}$ per context switch. Our second test involved using channel constructs which are type specific. In our case we use `channel\_out\_int()` (without alternative input code) and `channel\_in\_int()`. Results improve in this case also. The loop stage takes a total of 1,004.964$\mu$s, and the context switch time averages 125$\text{ns}$.

We justify our optimisation in two ways. An application programmer that knows how his application will behave can choose to disable alternative input code from `channel\_out()` by setting a flag. He can also use specific data type channel communication if he is aware that it will improve the performance of his program. A compiler (or pre-compiler) can detect that no alternative input is being used and link in the `channel\_out()` code that does not have the alternative input code. The compiler can also decide which channel function to use in the type specific case.

3.3.5 Analysis of results

One of the main discussions of the previous section was due to problems with page faults and cache misses. We have attempted to solve some of the problems in some of the benchmarks (and will attempt to solve more, using different run queues), and in others we have accepted them as external conditions which affect our results. We have attempted to present all the results to allow the reader to get a global picture of the situation.

Results, in particular the context switch benchmark, demonstrate that the circular nature of the run queue is designed mainly for fast context switch times for threads that relinquish control voluntarily (or possibly by automatic yield insertion). In real life applications the result would equate to very smooth response times. However, as we shall see in the forthcoming sections, the circular run queue suffers from relatively slow thread enqueue and dequeue routines, which affect inter-thread communication and direct thread execution. Another problem, which affects even its strong point (i.e. thread yields), is performance degradation due to cache misses. We intend to solve both problems in the forthcoming sections.

3.4 Scheduling for faster execution and inter-thread communication

We now introduce a scheduling technique which improves thread execution, termination and inter-thread communication.

3.4.1 Introduction

The underlying scheduler architecture is the same as the one described earlier (see Section 3.2.1). The run queue configuration is different from the first of our uniprocessor schedulers where we used a double linked circular queue to represent our run queue. The advantage was that since the current pointer on the circular run queue accessed the scheduler’s current thread, the context switch routine just involved saving and restoring the

---

2We used the same machine that we used to calculate our benchmarks on, to perform this test.
context and assigning a pointer. On the other hand thread enqueue and dequeue routines were more complicated since we were dealing with a double linked queue.

Our solution involves using a linked list with a dummy head. The run queue is an adaptation of the one Valois [103] uses for his non-blocking concurrent queue, and that Michael and Scott [79] use for their concurrent run queues. In the above cases however, the run queue implementation makes use of a node for each item of data (in our case a thread), therefore dequeue operations involve removing the head. Since we do not want to use nodes to avoid allocating and freeing memory during every enqueue and dequeue operation, and also wish to avoid the extra level of indirection nodes present, we had to change the original implementation. Our dummy head is fixed and is never changed during the entire program execution. The scheduler’s current thread would be represented by the thread which is the real head. This is what the dummy head will point to. Dequeues in our case involve removing our current and making the fixed dummy head point to the next thread on the run queue. Figure 3.7 demonstrates the run queue configuration.

We will now describe the various functions as was done in Section 3.3.2, we then conclude the section comparing performance results, so that the reader might appreciate the advantages and disadvantages of this system.

### 3.4.2 Internal scheduler functions

This section highlights the difference between this scheduling strategy and the previous, and attempts to demonstrate the reasoning behind this method.

#### Thread insert

The thread insert routine for this scheduler involves placing threads on the run queue. In this respect it is similar to the circular run queue implementation. The number of instructions is however reduced. Where the double linked circular run queue involves the need of four assignment operations, this version only needs two assignment operations. All the enqueue operation does is point the next tail pointer to the new thread and update the tail pointer to the new thread. The dummy head removes the need of checking both for the case of an empty run queue (although this is not always needed) and the case of a run queue with one element. Thread insertion is illustrated in Figure 3.8.

#### Thread remove

The thread remove routine involves only checking if the thread to be removed is the last on the run queue. If it is, the scheduler is shut down. If we are going to remove the thread from the run queue though, the situation becomes similar to the thread enqueue routine described above. The number of operations involved in a dequeue operation are two assignments against the four of the circular run queue scheduler. The thread remove operation is described in Figure 3.9.

#### Thread yield

Yielding is not as efficient in this case. Since our run queue is not circular, we must perform an enqueue and a dequeue operation to place the current thread at the end of the queue. This involves a conditional check and four other assignments.
Figure 3.7: Dummy head run queue scheduling.
3.4.3 Thread API functions

The main advantages this scheduler has over the circular run queue scheduler, is when using functions that perform enqueue and dequeue operations. `cthread_run()`, `cthread_join()`, the thread termination routine and in particular the channel input and output functions all improve in performance, as the results will demonstrate. However, `cthread_yield()` performs poorly since we need to perform the equivalent of a dequeue and enqueue operation for each `cthread_yield()` call.

3.4.4 Results

We will be comparing the performance of our scheduler with the performance using the same benchmarks first introduced in Section 3.3.4 and an application that highlights performance improvement using this run queue configuration. Thread initialisation will not be discussed since it is the same as in the circular run queue scheduler’s case.

Thread insert and remove

Thread insert and thread remove operations are the dummy head run queue’s strong points. To calculate our thread execution we use the same benchmark used previously and outlined in Algorithm 3.3 and Algorithm 3.4 for the cache friendly version. The time taken to insert 25,000 threads in the non-cache friendly version was measured at 4,722\(\mu\)s. On average the execution time of one thread is 189\(\mu\)s. The cache friendly benchmark completed in 508\(\mu\)s, averaging 20\(\mu\)s per operation. We also measured the performance of the aforementioned benchmark where every `cthread_run()` is followed by a dequeue. The non-cache friendly
version took an average of 200 ns to complete and the cache friendly version took an average of 49 ns to complete.

**Thread execution**

Our thread execution benchmark was outlined in Algorithm 3.5. This benchmark, like the one before it, demonstrates the potential of the dummy head based run queue, since there are a lot of enqueue and dequeue operations. Moreover unlike the previous benchmark, this test could be considered a real life application. The non-cache friendly version takes 25,587 μs to execute 25,000 threads (average 1 μs per thread). The cache friendly version takes 5,913 μs (average 237 ns per thread). Notice the time taken for the non-cache friendly version is similar to the time taken for the benchmark when using the circular run queue. The reason for this is that the overhead of cache misses and that of executing the function overshadows the performance gained through the use of the dummy head run queue.

**Context switch performance**

This benchmark demonstrates the main disadvantage of our dummy head run queue over the circular run queue. We measure context switch performance by measuring how long it takes for 20 threads to switch between each other for 10,000,000 times (see Algorithm 3.7). The net execution time was 1,015,084 μs, resulting in a average context switch time of 102 ns.

The context switch performance for the non-cache friendly benchmark is 4,953,293 μs (495 ns per context switch), which is similar in performance to the same benchmark when using the circular run queue. As was the case with the thread execution benchmark, the cache misses overshadow the advantage the circular run queue had in this benchmark.

**Inter-thread communication benchmark**

We use the `commstime` benchmark to measure the performance of our inter-thread communication. In the circular run queue `commstime` benchmark we performed two calculations. A test that uses our general channel input and output routines and one that uses the integer specific communication functions. The general `commstime` benchmark computes in 1,234,723 μs per 1,000,000 loops, and the context switch time measures in at 154 ns. The integer specific `commstime` executes in 680,029 μs and has an average context switch time of 85 ns. The results show that the dummy head scheduler is without doubt the better CSP thread scheduler.

**Matrix multiplication**

Our matrix multiplication benchmark involves the multiplication of two matrices. We create $N^2$ threads for an $N$ by $N$ matrix multiplication. Each thread calculates the result of one element in the matrix. Algorithm 3.8 demonstrates the duty of each thread.

Our benchmark calculates how long it takes to place the threads onto the run queue, and wait for all threads to execute the entire multiplication and terminate. Algorithm 3.9 demonstrates the benchmark. Results for a matrix multiplication of a 10 by 10 matrix measured in at 86 μs for our dummy head run queue version and 94 μs for our circular run queue version. As the matrix size increases the number of calculations performed by `elt_mult()` increases and so performance improvement is not so notable.
Algorithm 3.8 Calculating an element in matrix multiply.

```c
void elt_mult(cthread * ct, int i, int j) { // i and j represent row and column
int k;
for (k = 0; k < SIZE; k++) { // calculate an element
    Z[i][j] += (X[i][k] * Y[k][j]);
}
}
```

Algorithm 3.9 Matrix multiplication benchmark.

```c
cthread_main(cthread * cmain, int argc, char ** argv) {
    int i,j;
    cthread * c[SIZE][SIZE]; // a thread for each element
    struct timeval t1,t2;
    for (i = 0; i < SIZE; i++) { // initiate threads
        for (j = 0; j < SIZE; j++) // initiate threads
            c[i][j] = cthread_init(elt_mult,4096,2,i,j);
    }
    gettimeofday(&t1,NULL); // begin timer
    for (i = 0; i < SIZE; i++) { // execute threads
        for (j = 0; j < SIZE; j++)
            cthread_run(c[i]);
    }
    for (i = 0; i < SIZE; i++) { // wait for threads to terminate
        for (j = 0; j < SIZE; j++)
            cthread_join(c[i]);
    }
    gettimeofday(&t2,NULL); // end timer
}
```
3.4.5 Analysis of results

The above results clearly show that for certain applications our alternative run queue configuration improves performance. The matrix multiplication benchmark demonstrates a nearly 10% improvement in performance that is very relevant. If the application program intends to make heavy use of CSP constructs (as is the case with occam programs), the dummy head scheduler again is the one to go for. However, when scheduling threads with short dispatch times, the context switch measurement shows that the dummy head run queue can be up to 50% slower.

3.5 Uniprocessor batching

Batching is an alternative scheduling strategy that attempts to make better use of the cache. As was mentioned in Section 2.4, cache exploitation is improved when long lived threads access the same data in memory for a considerable amount of time.

3.5.1 Introduction

Traditionally, fine grain thread schedulers run threads on a first in first out basis (FIFO), and by the time a thread is rescheduled, its memory footprint would have been flushed out of the cache. Batching attempts to provide a solution by means of a scheduling strategy which will allow threads to find the data they access still in the cache.

Section 2.4 also introduced results by Vella, in his implementation of a uniprocessor batching KRoC scheduler. We attempt to implement the same scheduling strategy for our smash schedulers, to see if we can reproduce Vella’s results and to highlight the advantages of cache exploitation due to batching on uniprocessors rather than SMPs (SMP batching will be introduced in Chapter 5).

The underlying architecture of our uniprocessor batch scheduler remains unchanged from the previous two schedulers. The run queue structure is however modified to handle batches. Batching involves grouping together a fixed number of common threads into coarser entities we call batches. A batch is in itself a double-linked circular queue. A batch has another two properties, a dispatch counter and a size counter. The dispatch counter is incremented on every dispatch. When the dispatch counter reaches a certain pre-defined threshold, the batch relinquishes processor control. The batch size monitors the size of the batch, ensuring that all batches do not exceed a pre-set size limit. Instead of a run queue of user threads, batching involves using a batch run queue (see Figure 3.10). All our batches are placed onto the batch run queue, and the user-level thread scheduler, instead of servicing one user thread at a time in sequence until it has run through the whole run queue, acquires a batch, and services the batch for a fixed number of dispatches. When the entire memory footprint of the batch fits in the cache, each thread will be guaranteed to find the data it accesses in the cache for however long the dispatch size is. The reader will note that it is advantageous for the number of dispatches to be considerably greater than the the number of threads in a batch. At the same time the ratio must not be too large, so as to avoid a single batch monopolising the processor for considerable amount of time.

3.5.2 Internal scheduler functions

As we did with the other schedulers we provide a description of the main scheduler functions so that the reader can get an idea of how batch scheduling works and what overheads are
Figure 3.10: Uniprocessor batch scheduling.
involved.

Thread insert

The thread insert routine is responsible for placing user threads in the appropriate place in the complex run queue data structures. The routine first attempts to place the thread in the currently running batch. Each batch has a size threshold, and when this is met, any new thread is placed onto an overflow batch (see Figure 3.10). The overflow batch also limits size to the maximum batch size threshold. When the overflow batch is full, it is placed onto the batch run queue and a new overflow batch is created. It can be seen that this operation is far more complex than the same operation for other schedulers.

Thread remove

The thread remove routine like the insert routine, has to cater for the batch queues. Initially a thread is removed from the currently running batch. If the batch is empty the scheduler attempts to service another batch, if no batches exist, there is nothing else to run so the scheduler shuts down. If there are more threads on the current batch they are executed.

Thread yield

Each batch is allowed to run for a certain number of dispatches before yielding control to the next batch. Since each batch is a circular queue, when a batch is being serviced the threads inside the batch relinquish execution between each other in a similar way to the circular run queue version of our scheduler. For each thread yield, a dispatch counter is incremented. When the counter reaches a predefined threshold, control passes on to a new batch. Since the batch run queue itself is a circular queue, this operation involves just making the current batch pointer point to next and resetting the dispatch counter. Yielding control only suffers from the overhead of checking and incrementing the dispatch size, the only other operation is an assignment to update the current thread pointer (and in the case when the batch dispatch count expires, resetting the counter and updating the batch pointer).

3.5.3 Thread API functions

Although there is considerable overhead in all of the internal scheduler functions (which reflects itself in all the API function calls), performance does not degrade too much, since given a decent dispatch threshold and batch size the special cases are far between, and the only difference in implementation between the circular run queue scheduler would be the conditional checks for the special cases (to check if the dispatch threshold and size threshold have been met). Moreover for applications which use more threads than can be fitted into cache, the performance of certain functions such as cthread_yield() improves since the thread data itself will most probably remain still in the cache between dispatches.

3.5.4 Results

We discuss the results due to batching in a similar fashion to what we did before. We compare performance for thread enqueue, thread dequeue, thread execution and for context switch time. Finally we present an application where the user threads access the same areas
Figure 3.11: The uniprocessor batch scheduling thread insert process. In case a) the new thread NU is placed directly onto the running batch. In case b) the current batch is full, so the new thread NU is placed onto the overflow batch. In case c) the overflow batch is full, so the overflow batch is placed onto the batch queue (not shown in diagram) and the new thread is placed onto the now empty overflow batch.
of memory consistently and demonstrate the usefulness of batching. As was the case with the dummy head run queue we do not display results for thread initialisation, since the results are similar to that of the circular run queue version.

**Thread insert and remove**

Our thread insert benchmark calculates the length of time taken to execute 25,000 threads (see Algorithm 3.3). The batch size is another variable which needs to be considered when performing the benchmark. A batch size of 12,500 would mean only two batches being created, while a batch size of 5 would mean 5,000 batches need to be created. We conduct the test using 5 different batch sizes. We provide results for the extreme case, *i.e.* a batch size of 1 meaning all threads exist in their own batch, and a batch size of 25,000 where all threads are created in the same batch. We also provide results for what we consider to be normal batch sizes. What follows is a table with performance results for the non-cache friendly version of the benchmark:

<table>
<thead>
<tr>
<th>Batch size</th>
<th>Net time taken (µs)</th>
<th>Average time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17,197</td>
<td>687</td>
</tr>
<tr>
<td>100</td>
<td>6,159</td>
<td>246</td>
</tr>
<tr>
<td>250</td>
<td>6,095</td>
<td>243</td>
</tr>
<tr>
<td>1,000</td>
<td>6,040</td>
<td>241</td>
</tr>
<tr>
<td>25,000</td>
<td>5,815</td>
<td>232</td>
</tr>
</tbody>
</table>

We also present results for the cache friendly version of the benchmark, below:

<table>
<thead>
<tr>
<th>Batch size</th>
<th>Net time taken (µs)</th>
<th>Average time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14,767</td>
<td>590</td>
</tr>
<tr>
<td>100</td>
<td>2,315</td>
<td>92</td>
</tr>
<tr>
<td>250</td>
<td>2,209</td>
<td>88</td>
</tr>
<tr>
<td>1,000</td>
<td>2,188</td>
<td>84</td>
</tr>
<tr>
<td>25,000</td>
<td>1,999</td>
<td>80</td>
</tr>
</tbody>
</table>

The results demonstrate the difference in performance between creating only 1 batch and creating a batch per thread. The batch sizes of 100, 250 and 1,000 represent more realistic batch sizes and the performance figures are very similar, mainly because only a few special cases need to be taken care of in all cases except for the first. Results in the case of the cache friendly benchmark are in general slightly slower than the 51ns performance of our first scheduler, but considerably slower that the 20ns of our dummy head scheduler. In the case of the non-cache friendly benchmark, results are closer to those of the previous schedulers especially in the case of the circular run queue (231ns). We note that the batches could have been linked lists using a dummy head, instead of circular queues. The main use of the batching scheduler is for long lived threads that de-schedule frequently, so the circular run queue structure is more suitable for faster yielding times.

For completeness sake we also present the thread insert followed by dequeue benchmark and the thread execution benchmark. In this case we fix the batch size to 250. We perform both benchmarks making sure all data is in the cache. The former took an average of 159ns and the latter averaged 361ns.
Context switch performance

We perform the same context switch performance benchmark as with the other schedulers (see Algorithm 3.7). We measure the time taken for 20 threads to yield for a total of 10,000,000 times. We also have to pay attention to the dispatch size and the batch size when measuring results. We attempt to execute the benchmark using two different batch sizes, a more realistic batch size of 250 threads, and another with a batch size of 8 (we use a batch size of 8 so that the 20 threads, will be distributed among 3 batches). Furthermore we perform four measurements. The first measurement takes into consideration the extreme case of having each batch only perform one dispatch before allowing another batch to continue. The other extreme of allowing a batch to continue until all threads terminate is also taken into account, by assigning a dispatch counter limit of 10,000,000. The other two dispatch thresholds of 1,000 and 2,500 are more realistic and what we would expect from a practical application.

We begin by providing the reader with the results of the first experiment, where we measured benchmark times by using only one batch (with a size of 250 threads). The table below summarises the results:

<table>
<thead>
<tr>
<th>Dispatch threshold</th>
<th>Net time taken (µs)</th>
<th>Average time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,256,767</td>
<td>126</td>
</tr>
<tr>
<td>1,000</td>
<td>916,367</td>
<td>92</td>
</tr>
<tr>
<td>2,500</td>
<td>914,597</td>
<td>91</td>
</tr>
<tr>
<td>10,000,000</td>
<td>913,500</td>
<td>91</td>
</tr>
</tbody>
</table>

For our second group of measurements we set the batch size to 8. While this might not be a realistic batch size, it reflects the type of results we would achieve when using multiple batches instead of just one batch. The following table demonstrates the results:

<table>
<thead>
<tr>
<th>Dispatch threshold</th>
<th>Net time taken (µs)</th>
<th>Average time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,255,933</td>
<td>126</td>
</tr>
<tr>
<td>1,000</td>
<td>100,101</td>
<td>100</td>
</tr>
<tr>
<td>2,500</td>
<td>991,161</td>
<td>99</td>
</tr>
<tr>
<td>10,000,000</td>
<td>956,125</td>
<td>95</td>
</tr>
</tbody>
</table>

In both cases the results are fairly similar. The second set of results is slightly slower, due to the changing of batches. When we use a dispatch size of 1 the context switch time is calculated at 126 ns, which is the worst possible performance we would expect from the scheduler. The other results reflect what we would expect the performance to be for most applications. The 99 ns context switch time is a good estimate for most applications. While far slower than the 57 ns we achieve from the original smash scheduler, it is still slightly faster than the 101 ns performance of the dummy head run queue scheduler.

When we run the non-cache friendly context switch benchmark, we come across the main advantage of the batching scheduler. Since we schedule in batches, the threads will keep finding a copy of their thread descriptor inside the cache for the amount of times of the dispatch size. However, every time a new batch is scheduled we still incur the costs of the cache misses. The following table summarises the results (the batch size is 250):

<table>
<thead>
<tr>
<th>Dispatch threshold</th>
<th>Net time taken (µs)</th>
<th>Average time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7,808,669</td>
<td>781</td>
</tr>
<tr>
<td>1,000</td>
<td>3,950,988</td>
<td>395</td>
</tr>
<tr>
<td>2,500</td>
<td>3,452,142</td>
<td>345</td>
</tr>
<tr>
<td>10,000,000</td>
<td>3,040,880</td>
<td>304</td>
</tr>
</tbody>
</table>
Realistically we would say that the context switch time for threads that do not fit inside the cache is between 330\(\text{ns}\) and 400\(\text{ns}\). The results compare favourably with the 453\(\text{ns}\) incurred by the circular run queue scheduler, demonstrating that batching could be useful when dealing with very fine grain applications.

**Inter-thread communication benchmark**

We perform the *commstime* benchmark in its two incarnations. We set the batch size to 250 and the dispatch count to 2,500. *commstime* using the standard communication constructs performed 1,000,000 loops in 1,867,561 with an average context switch time of 233\(\text{ns}\). The integer specific benchmark performed 1,000,000 loops in 1,356,701\(\mu\text{s}\), averaging 170\(\text{ns}\) per context switch. Note the *commstime* benchmark does not take advantage of batching, since there are not that many threads and they all communicate with each other.

**Cache re-use benchmark**

We now focus our attention on a benchmark developed by Vella [107] which highlights the advantages of using batching even on a uniprocessor machine. Our application, performs a series of operations on the same array of data over a period of time. Three variables are used, an array size which represents the data that will be handled, a variable representing the process length, which gives an indication of the life time of each thread (the larger the variable, the longer lived the thread will be) and a granularity variable which represents the number of operations that each thread will perform on the array. We set the dispatch count to 2,500 and the array size to 250.

Function *cache_test()* in Algorithm 3.10 displays what each thread’s duty will be. Each thread runs for a certain amount of times (depending on the PROCESS_LENGTH variable) and always performs the given operations on the same data in the array. Algorithm 3.10 demonstrates how the array is partitioned between each of the threads. The finer the granularity (set by the GRANULARITY variable) the more threads are created. A thread is assigned a part of an array and performs its operations on that part of the array only. We perform two series of tests. In the first series we fix the array size to 50,000 words, which exhausts the first level cache. In the second we set the array size to 300,000 words, which is more than enough to occupy all the space of our first and second level cache. We vary both the granularity and the process length and run the benchmark using our three schedulers.

Our results echo the results of Vella [107]. The batching scheduler outperforms both other schedulers when the process length is greater than 1. Graphs in Figure 3.12 demonstrate the performance gain when accessing an array of 50,000 words. Performance gain is maintained up to a granularity of around 20. However, when accessing an array of 300,000 words, the situation is far better and performance gain exists even when running the benchmark with coarser grain parameters.

### 3.5.5 Analysis of results

As was the case with the other schedulers, benchmark results highlight cases in which the batching scheduler, performs significantly better than the other schedulers. The main advantage is for fine grain applications where threads access the same data over long periods of time. The context switch benchmark is also very encouraging since the common data benefiting from the batching is the threads’ own work space data, thus scheduler performance improves at very fine granularity even for general applications. The main disadvantage
Figure 3.12: Results from Algorithm 3.10, for an array size of 50,000 words.
Figure 3.13: Results from Algorithm 3.10, for an array size of 300,000 words.
Algorithm 3.10 Cache affinity benchmark.

```c
#define YIELDNUM PROCESS LENGTH
int array[ARRAY_SIZE];

void cache_test(cthread * ct, int i) {
    int j,k;
    for (j = 0; j < YIELDNUM; j++) {
        for (k = 0; k < GRANULARITY;k++) {
            array[(GRANULARITY * i) + k] = array[(GRANULARITY * i) + k] + 1;
        }
    }
    cthread_yield();
}

cthread_main(cthread * cmain, int argc, char ** argv) {
    int i;
    int thread_num = ARRAY_SIZE / GRANULARITY;
    cthread * c[thread_num];
    struct timeval t1,t2;
    for (i = 0; i < thread_num; i++) // initiate threads
        c[i] = cthread_init(cach_test,1024,0);
    gettimeofday(&t1,NULL); // begin timer
    for (i = 0; i < thread_num; i++) // execute threads
        cthread_run(c[i]);
    for (i = 0; i < thread_num; i++) // join threads
        cthread_join(c[i]);
    gettimeofday(&t2,NULL); // end timer
}
```
is the overhead due to maintaining batching, which is even more apparent at such fine granularities. However, the batching scheduler’s overall performance is positive and results further highlight the importance of locality in thread scheduling, even at the uniprocessor level.

### 3.6 Conclusion

In this chapter we have introduced the three different versions of our uniprocessor scheduler. We showed how external elements such as system calls, page faults and cache misses can degrade performance of some applications and provided appropriate solutions for each of them. We demonstrated by means of results the advantages and disadvantages of our various run queue implementations. We present a summary of all the results in the following table:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Circular</th>
<th>DH</th>
<th>Batches</th>
<th>MESH</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialisation</td>
<td>687</td>
<td>687</td>
<td>687</td>
<td>9,215</td>
</tr>
<tr>
<td>initialisation (cache)</td>
<td>271</td>
<td>261</td>
<td>271</td>
<td>9,215</td>
</tr>
<tr>
<td>insert</td>
<td>231</td>
<td>189</td>
<td>243</td>
<td>559</td>
</tr>
<tr>
<td>insert (cache)</td>
<td>51</td>
<td>20</td>
<td>88</td>
<td>138</td>
</tr>
<tr>
<td>enqueue dequeue</td>
<td>248</td>
<td>200</td>
<td>261</td>
<td>580</td>
</tr>
<tr>
<td>enqueue dequeue (cache)</td>
<td>110</td>
<td>49</td>
<td>159</td>
<td>367</td>
</tr>
<tr>
<td>execution</td>
<td>1,112</td>
<td>1,023</td>
<td>1,162</td>
<td>1,958</td>
</tr>
<tr>
<td>execution (cache)</td>
<td>288</td>
<td>237</td>
<td>361</td>
<td>549</td>
</tr>
<tr>
<td>context switch</td>
<td>453</td>
<td>495</td>
<td>345</td>
<td>507</td>
</tr>
<tr>
<td>context switch (cache)</td>
<td>57</td>
<td>102</td>
<td>99</td>
<td>69</td>
</tr>
<tr>
<td>commstime</td>
<td>222</td>
<td>154</td>
<td>233</td>
<td>NA</td>
</tr>
<tr>
<td>commstime int specific</td>
<td>125</td>
<td>85</td>
<td>170</td>
<td>123</td>
</tr>
<tr>
<td>matrix multiplication (in μs)</td>
<td>94</td>
<td>86</td>
<td>101</td>
<td>150</td>
</tr>
<tr>
<td>cache re-use (in secs)</td>
<td>3.4</td>
<td>3.7</td>
<td>2.5</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Each benchmark is followed by its cache friendly version (where this applies). In all cases we are presenting the average time taken for the operation to complete in nanoseconds, unless otherwise specified. The thread initialisation function displayed, does not incur loss in performance due to page faults or system calls. The *commstime* benchmark reports the context switch average. The cache re-use benchmark uses an array of 50,000 words, a process length of 200 and granularity of 2 (see Section 3.5.4). Best performance results for each operation are highlighted in bold. All results due to batching are based on a batch size of 250 and dispatch threshold of 2,500. We also include the performance of MESH using the same benchmarks and on the same machine. The MESH results are only here to act as a guideline on which to compare the results, since MESH is already an established scheduler (and since our work is based on the MESH scheduler). We remind the reader that MESH performs more functions than our scheduler, and the overhead of supporting the extra functionality affects performance. Note the initialisation function of MESH does not use anything to remove page faults and system calls, therefore the non-cache friendly thread initialisation benchmark and it’s cache friendly counterpart incur the same cost since the cache misses are overshadowed by the page faults and system calls.

The reader will note from the results that the dummy head version of *smash* performs better than the other schedulers in most operations. The reason for this is that most benchmarks are dependent on enqueue and dequeue operations. All different versions of the
**smash** scheduler are suitable for different applications though. The cache friendly context switch benchmark demonstrates that the circular run queue version is best used in multi-threaded environments where not too many threads are used and smooth response times are required. The dummy head is best suited for applications that use CSP constructs for communication or for fast execution. The batching based scheduler is ideal for applications that access the same data over long periods of times or when using many threads, since the batching management overhead does not have a negative impact on the overall performance of the application.

Finally we conclude the chapter by bringing to the attention of the reader a possible enhancement that would allow any application to benefit from the advantages of all the schedulers. A pre-compiler could be used to parse the application program and decide which scheduler would be ideal to execute the program, based on a number of factors, such as the number of threads, the number of yields, the number of insert and remove operations and the amount of data re-use. The pre-compiler would then suggest which version of the scheduler would be most suitable to run the program on. Another similar solution involves running an application for a number of times on all three schedulers using different parameters and concluding which scheduler is most suitable depending on the parameters passed.
Chapter 4

Shared run queue SMP scheduling

In this chapter we will focus our attention on the implementations of various shared run queue SMP user-level thread schedulers. We also introduce issues common to all the SMP smash variants such as underlying software architecture, the idling process, SMP context switching, kernel thread identity and scheduler startup and shutdown. We experiment by implementing schedulers which use locks at four different granularities and provide a lock free scheduler implementation that uses non-blocking algorithms and data structures. The individual SMP thread scheduler implementations will be presented in the same way we presented the uniprocessor schedulers.

4.1 The shared run queue

The shared run queue implementation is probably the most common strategy used for SMP scheduling. Any thread that is created is placed onto a shared run queue and any processor looking for a task accesses the shared run queue, dequeues a thread and executes it. Care must be taken to provide access control for the shared queue amongst processes. Concurrent access is usually provided by means of a spin lock (see Section 2.2.2). The potentially fine grain nature of our applications means that we will access the shared resources very frequently, and in particular the shared run queue could be accessed millions of times per second. The run queue can therefore become the major bottleneck of the shared run queue implementations. We try and diminish the amount of time each processor busy waits by holding locks for the least amount of instructions possible and by varying the granularity of the locks.

Load balancing is perhaps the strongest property of shared run queue schedulers. The scheduling strategy can be implemented in such a way as to avoid any processes from being idle when there is work to be done on the run queue. Figure 4.1 demonstrates the possible scheduler states.

4.2 Scheduler details

All our SMP schedulers share a common underlying architecture. Together with the uniprocessor version of smash they also share a common API and lower level scheduling functions. Most of these were discussed in the previous chapter. The specification of the API and the internal scheduler functions remain the same as those discussed previously. However, the implementation differs between the uniprocessor schedulers and SMP versions.
Figure 4.1: Load balancing properties.
In the next section we demonstrate implementation details of the software architecture of the SMP schedulers. We will then present the individual implementations of the internal scheduler calls, API calls, run queue configurations and scheduler synchronisation techniques.

4.2.1 Underlying architecture

All our SMP thread schedulers share the same underlying architecture, where we create an address space of $N$ shared memory processes for $N$ processors. The shared memory processes are equivalent to kernel level threads under Linux. We initialise the kernel level threads by means of the `clone()` Linux system call, setting a flag to share memory (CLONE_VM) and to share signals (CLONE_SIGHAND) between individual kernel level threads.

We discussed the issues of multiplexing user-level threads on top of kernel level threads, which can themselves be multiplexed on top of the CPUs in Chapter 2. We have chosen to have one kernel thread per processor and moreover, we have the option to bind the kernel thread to that process. We decide against having more than one kernel thread per process because we would like to avoid excessive horizontal switching. We bind kernel threads to processors by means of an extension to the Linux kernel, psets [63] which is an adaptation of SGI’s psets. Figure 4.2 demonstrates our general configuration.

4.2.2 A case of identity

A processor, which is represented by an individual kernel thread, will need to access information specific to that kernel thread. The kernel thread identity is needed every time the user-level scheduler kernel is accessed, so care must be taken to ensure that there is no excessive increase in overhead to identify the kernel thread. Since all memory is shared, it
CHAPTER 4. SHARED RUN QUEUE SMP SCHEDULING

is impossible to have a single identical variable for every processor which can be used to determine the current thread. Solutions do however exist. We present a few here.

Kernel thread identity could easily be identified by means of a call to \texttt{getpid()}. Unfortunately, \texttt{getpid()} is a system call, if we had to call \texttt{getpid()} every time we enter \texttt{smash} kernel we would defeat the scope of running the threads at the user level.

Cordina [25] proposes a solution for his SMP implementation of MESH. Since MESH uses the currently running user thread stack when in the scheduler, he suggests comparing the stack pointer being used when inside the user-level scheduler, with all the user-level thread stacks currently running. If the stack being used exists within the bounds of one the current threads’ stack the correct kernel thread identity is determined. This solution involves a worst case of $O(N)$ and an average case of $O(N)$ for a scheduler using $N$ kernel threads. This is acceptable for a small number of processors, however it must be noted that when inside the user-level scheduler the kernel thread identity might need to be verified more than once, since the stack might be changed (stack changing and problems related to it are described in Section 4.2.4).

Our solution is based on the solution used by Glibc [40]. We use the system call \texttt{modify_ldt()} to setup a segment for each thread. We store information in the segment relative to our kernel thread. In our case we store a pointer to a kernel thread structure containing the kernel thread identifier (0 for the first processor, 1 for the second etc..., thus making our system compatible with \texttt{psets}) and the \texttt{pid} for every kernel thread (used for cleaning up before shutting down the scheduler). Note we could have opted to just keep the integer in the data segment. We can also use this area for creating a user space that can be seen by both the user and the kernel for adding facilities such as preemption-safe locking (see Section 2.2.2).

4.2.3 Idling around

On occasions some of our scheduler’s kernel threads could end up with nothing to do. One of the processors will not have any threads to run. This situation might arise when the current thread finished executing and the shared run queue is empty. Since the kernel thread has nothing to do, it could busy wait until a user thread is available for execution or it could relinquish the processor and let other processes use the current CPU. Since \texttt{smash} is also intended to be run in multiprogrammed environments we prefer the second option. Even when \texttt{smash} is the only application running, performance would deteriorate only in the most unlikely cases (when there are continuous sleeps and wake ups).

For our SMP schedulers we would choose to set the kernel thread to sleep on a number of occasions. These occasions vary from scheduler to scheduler and will be discussed in the individual implementation sections, but a common reason for sleep and wake up is for handling situations when the run queue is empty. When the schedulers finds an empty run queue it would set the running kernel thread to sleep. On the other hand when a thread is inserted onto an empty run queue, and a processor lies idle, the kernel thread associated with the idle processor is awoken.

Traditional wakeup and sleep functions entail a potential race condition known as the “lost wakeup problem” [102]. We could illustrate the “lost wakeup problem” in our case if we were to send a wake up signal to inform that we have placed a thread onto an empty run queue. If a kernel thread were to find the run queue empty it would attempt to sleep, however if between checking for an empty run queue and sleeping, another kernel thread sends a wake up signal the kernel thread will sleep for nothing. Moreover, this operation
Algorithm 4.1 Context switch race condition for kernel thread 0.

```c
R1 if (JmpBuf_Set(sched.current[0]-&gt;jmp) { 
R2 runqueue_enqueue(sched.current[0]);
R3 sched.current[0] = runqueue_dequeue();
R4 JmpBuf_Jmp(sched.current[0]-&gt;jmp);
R5 }```

could result in the kernel thread sleeping indefinitely. A solution introduced by Vella [108] for the KRoC SMP scheduler, is to use semaphores provided by the operating system. Semaphores do not suffer from the “lost wakeup problem”.

On initialisation we set up a structure containing semaphore information for each of our kernel threads. When a kernel thread needs to be set to sleep the kernel thread’s identity is verified, and a call to `semop()` sets the kernel thread to sleep. When a kernel thread needs to be awaken we use `semop()` to wake up the appropriate kernel thread.

Since our idling system involves system calls, we have also implemented some techniques which would speed up the process. Before waking up a kernel thread we set the thread that it is meant to execute directly as the processor’s current thread, to relieve the scheduler from the laborious task of first inserting the thread into the run queue and then waking up the kernel thread which would then have to remove the thread from the run queue (this also solves the problem of having another kernel thread remove the thread setting the run queue state back to empty and calling two excessive semaphore operations). We also make sure the scheduler does not sleep unnecessarily. During a yield operation there are times a thread relinquishes control, but the run queue is empty, in this case we continue execution of the original thread. Another optimisation technique occurs when a thread terminates. If the terminating thread is supposed to awaken another user-level thread, we let the current thread run the joining thread instead of re-inserting the joining thread onto the run queue which could cause excessive sleep and wakeup calls.

4.2.4 Context switching

In the uniprocessor version of `smash` when the scheduler switches the running thread to another thread the current thread’s context would be saved by means of a call to `JmpBuf_Set()` . The task related to the function is then performed (e.g. a `cthread_yield()` would point the current pointer to the next thread on the run queue) then the context of the scheduler is switched to that of the next thread by means of a call to `JmpBuf_Jmp()` . Both `JmpBuf_Set()` and `JmpBuf_Jmp()` operate on the entire set of registers (or on the ones that are required at that time, by means of active context switching; however, the program counter, stack pointer and base pointer are always saved/loaded). Another important thing to bear in mind is that `smash` “borrows” the last user thread stack used. These two factors give rise to a potential race condition.

To understand how this race condition can occur consider Algorithm 4.1 which represents a traditional yield call, where the scheduler relinquishes the execution of the currently running thread, inserts the current thread into the run queue (R2), and removes another thread from the run queue (R3). In this case we assume the yield occurred on the processor
Algorithm 4.2 Solution for context switch race condition by switching to kernel thread stack.

1. \text{id} = \text{kernelthread\_id}();
2. \text{if} (\text{JmpBuf\_Set}(\text{sched.current}[\text{id}]->\text{jmp}) \{ \\
3. \quad \text{scheduler\_jmp2stack}(\text{id}); \\
4. \quad \text{id} = \text{kernelthread\_id}(); \\
5. \quad \text{runqueue\_enqueue}(\text{sched.current}[\text{id}]); \\
6. \quad \text{sched.current}[\text{id}] = \text{runqueue\_dequeue}(); \\
7. \quad \text{JmpBuf\_Jmp}(\text{sched.current}[\text{id}]->\text{jmp}); \\
8. \} \\

Algorithm 4.3 Change stack to scheduler stack.

\begin{verbatim}
int index = \text{kernelthread\_id}();  // return thread identifier
asm volatile (  // asm inline function
    "movl (%0),%%esp"  // move stack pointer
    "movl 4(%0),%%ebp" // move base pointer
    : : "b" (&sched.jmp[index]), // the kernel thread stack
); 
\end{verbatim}

represented by kernel thread 0. The race hazard arises right after the current thread is inserted onto the run queue (R2). After the enqueue operation, another kernel thread is free to take our relinquished thread from the run queue and run it itself. If say, kernel thread 1 acquires the thread and begins to execute it, and kernel thread 0 has not yet changed context (R4), both kernel thread 0 and kernel thread 1 would be using the same stack. This situation could lead to a corrupt stack.

One solution is to keep an extra stack for each kernel thread. When we enter the scheduler we first switch to the kernel thread’s stack. Thus before inserting a current thread onto the run queue the scheduler switches to the stack associated with that kernel thread. Since each kernel thread has its own stack there are no concurrency issues. Algorithm 4.2 illustrates how this could be applied to a yield routine. At S3 the stack is set to point to the kernel thread’s own stack. Since the information inside the scheduler’s stack is of no use to us after we execute the next thread, there is no use in saving the kernel thread stack’s context and we opt to always jump to the original values of the stack and base pointers. Using this technique we save on precious time since we avoid saving the entire scheduler’s thread context. Algorithm 4.3 demonstrates how we use the least number of instructions required.

Another solution involves re-structuring the order of events when switching to another thread’s context. In Algorithm 4.1 the current thread is placed onto the run queue (R2) before a new thread is removed from the run queue (R3). By removing the next runnable thread from the run queue before we insert the current thread, we can switch directly to the new thread’s stack before releasing the current thread. Using this method we ensure that the stack the kernel thread is running on cannot be used by any other kernel thread.
Algorithm 4.4 Solution by switching to thread stack for race condition

```
S1    id = kernelthread_id();
S2    if (JmpBuf_Set(sched.current[id]->jmp) { 
S3        tempcurrent[id] = runqueue_dequeue();
S4        cthreadjmp2stack(id);
S5        id = kernelthread_id();
S6        runqueue_enqueue(sched.current[id]);
S7        sched.current[index] = tempcurrent[id];
S8        JmpBuf_Jmp(sched.current[id]->jmp);
S9    }
```

Algorithm 4.4 illustrates this solution. We use a temporary pointer as a handle for the thread we dequeue from the run queue (S3). The scheduler then switches to the dequeued thread’s stack (S4) before placing the current thread onto the run queue (S6). The current thread is then set to the thread pointed to by the temporary pointer (S7). The kernel thread identity needs to be verified again (S5) since the identity was maintained on an automatic variable and when the kernel thread switches to the new thread’s stack, the value of the automatic variables is not on the new stack (S5). The jump routine (S8) does not change the value of the stack and base pointers but of all other registers. In this way we maintain the same number of instructions as for a context switch on the uniprocessor schedulers. This method is slightly faster than the kernel thread stack switching method illustrated in Algorithm 4.2 which involved an extra switch to the kernel thread’s stack and base pointers besides the new thread’s context switch. However, the kernel thread stack switching is necessary when a thread is terminated or blocks on a communication construct and there are no more threads on the run queue. In this case there are no threads to switch the stack to, so we must switch to the kernel thread’s stack. As a general rule of thumb the scheduler switches to the next runnable thread’s stack unless there are no more threads on the run queue, in which case it switches to the kernel thread’s own stack.

### 4.2.5 Scheduler initialisation

The startup process involves first initialising the various scheduler structures and run queues (depending on which version of smash is being used). We described in Section 4.2.1 the organisation of our system relative to the number of processors. The original kernel thread creates an individual kernel thread for each processor, and waits for each processor to signal back to it that is about to sleep\(^1\). Each kernel thread then binds itself to a process, modifies an LDT entry (used for retrieving identity, see Section 4.2.2) and signals to the main thread that it is about to sleep\(^2\). The kernel thread sleeps waiting for a potential thread to run. Since we are creating a kernel thread for each processor, the parent kernel thread that creates the other kernel threads is not used until shutdown, and is set to sleep, right after it places the first thread onto the run queue. The main kernel thread is also responsible for

---

\(^1\)This process does not involve operating system signalling. The main kernel thread busy waits on a set of flags (one per processor).

\(^2\)Note that it doesn’t matter if the kernel thread sleeps or not since we are using semaphores.
cleanup and shutdown. This kernel thread is awoken when the main user thread finishes executing.

Using a parent kernel thread is not the only solution for our configuration. We could have had a parent which creates \((N - 1)\) kernel threads for \(N\) processors, and services the remaining processor itself. We feel however that having a child kernel thread taking care of each processor, represents a more symmetric approach, and is better suited for startup, cleanup and shutdown.

4.2.6 Scheduler shutdown

The scheduler is shut down when the main thread represented by `cthread_main()` finishes executing. The main kernel thread is then woken up and signals all child kernel threads to shut down before terminating.

4.3 Traditional shared run queue SMP smash

We now focus our attention on the first implementation of our SMP schedulers. This implementation comes in two guises, one that uses one global lock for the shared run queue, internal scheduler structures, joiner constructs and CSP channels, and another one that uses one lock for all scheduler structures (including the run queue) and individual locks for the communications constructs. The second represents the more traditional approach to SMP scheduler implementations (Vella [108] and Cordina [25] both use this strategy for their fine grain SMP implementations). For clarity we will henceforth refer to the SMP scheduler that uses only one single global lock as the global lock scheduler and the more traditional SMP scheduler that uses one lock for internal scheduler data structures and separate locks for communication as the traditional scheduler. Unless specified we will discuss scheduler implementations for both schedulers.

4.3.1 Introduction

The scheduler’s run queue is composed of dummy head based queue similar to the one used in our dummy head uniprocessor scheduler (see Section 3.4). The reason for this is that we now have \(N\) current threads for \(N\) processors, therefore we cannot afford to keep the current on the run queue anymore and each context switch requires an enqueue and dequeue routine. We recall from the previous chapter that the dummy head queue is the fastest for enqueues and dequeues. We protect any access to the run queue by means of a single spin lock. The same spin lock is also used to protect other internal scheduler structures. In the case of the global lock scheduler, the same lock is used to protect even the communication data structures (see Figure 4.3). In the case of the traditional scheduler we use individual locks for the communication constructs (see Figure 4.4).

We described in Section 4.2.3 the need of an idling system. To facilitate the task of knowing when to wake up kernel threads, the SMP schedulers maintain a counter representing the number of kernel threads that are asleep on a semaphore. The sleeper counter is used each time we need to insert a thread into the scheduler. When it is greater than 0, the scheduler is aware of an idle processor and queries all processors to check which one has no work.

From a performance point of view, we try to reduce the amount of instructions that a processor retains a shared resource as much as possible. In many cases this is a bit more
Figure 4.3: Global lock scheduling.
Figure 4.4: Traditional shared run queue SMP thread scheduling.
than ten instructions. The most frequently used scheduler data structures have cache line gaps between them to keep them on separate cache lines and so protect them from false sharing (see Section 2.1.3). Although notable enough to merit action, the affects of false sharing on shared run queue schedulers in terms of performance are not as evident as when we use per processor run queue schedulers. We will demonstrate in Section 5.2.4 how when using a per processor run queue scheduler performance degrades drastically if we do not take precautions against false sharing.

4.3.2 Internal scheduler functions

The internal scheduler functions differ somewhat from what we have seen in the uniprocessor case, however for the shared run queue versions in general they remain very similar. In all cases the first thing we do is to return the identity of the processor we are running on so as to be able to access the correct data structures.

Thread insert

The thread insert routine in the general case involves placing the thread onto the shared run queue. However, to improve performance and provide proper load balancing the scheduler first checks if there are any processors idle by consulting the sleeper counter. When no processors are idle the counter is set to 0 and the scheduler simply inserts the thread onto the shared run queue. When the number of idle processors is greater than 0 the scheduler cycles through the processors and checks for one that is idle. When an idle processor is identified, the thread is inserted into the current pointer associated with said processor and the associated kernel thread is awoken. Note that the thread insert routine is protected by means of the scheduler’s spin lock.

Thread remove

Thread remove involves discarding the current thread (either because the thread blocks on a communication construct or because it terminates) and trying to obtain another thread from the shared run queue. If there are no threads left on the shared run queue, the kernel thread representing the processor is set to sleep on a semaphore. As is the case with the scheduler’s thread insert, the thread remove is protected by a spin lock.

Thread yield

The thread yield routine requires placing the processor’s current thread onto the run queue and servicing the the thread that lies at the head of the thread run queue. As usual we save the current context before placing our thread onto the shared run queue, and we conclude the routine by jumping to the context of our new current thread. As with the other routines the whole operation is protected by a spin lock.

4.3.3 Thread API functions

The thread API functions are all notably affected mainly by the shared scheduler data structures, in particular the shared run queue. The contention on the shared data structures associated with the spin locks decreases performance of all API calls, when individually
compared with the uniprocessor scheduler. API calls such as \texttt{cthread\_run()} and the communication API calls (such as \texttt{channel\_in()}, \texttt{channel\_out()} and \texttt{cthread\_join()}) that use thread insert routines suffer the overhead of having to check if other processors are idle (so as to maintain load balancing) before inserting the thread back onto the run queue, although the overhead is alleviated by means of conditionals. \texttt{cthread\_yield()} suffers from the cost of an enqueue and dequeue operation. Note that in the traditional scheduler, since we use individual locks for each of the communication constructs, we add the overhead of having two locks for each communication API call. One of the locks is used to protect the communication construct and the other the shared data structures.

### 4.3.4 Results

We now provide results for our first SMP \texttt{smash} scheduler. The shared run queue SMP results focus mainly on the context switch and thread execution benchmarks we introduced in the previous chapter, and two new benchmarks. The first new benchmark calculates a measure of speedup for the particular scheduler and the second benchmark substitutes the \texttt{commstime} benchmark for SMP based inter-thread communications. For all SMP implementations we will be using the same system, a quad-processor machine consisting of Intel PIII Xeons, and 256MB of physical memory, running Linux 2.2.12-20. Each Xeon runs at 700MHz and is equipped with 512KB second level cache and 32KB first level cache (16KB instruction and 16KB data).

#### Context switch performance

We use the same benchmarks that we used for the uniprocessor schedulers to measure context switch performance. Algorithm 3.7 demonstrates our context switch benchmark. We remind the reader that we calculate the time taken to perform 10,000,000 yields using 20 threads. We calculate the performance four times, starting with our uniprocessor (circular run queue) and the SMP using 1, 2 and 4 processors. The results reflect the average amount of time taken to perform a yield operation. We present the average speed of the context switch when the benchmark executes on the given amount of processors in nanoseconds:

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>Traditional(1)</th>
<th>Traditional(2)</th>
<th>Traditional(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>28</td>
<td>159</td>
<td>1,075</td>
<td>1,424</td>
</tr>
</tbody>
</table>

The results at first sight might seem disappointing since not only is there no speedup but the scheduler performance seems to deteriorate as the number of processors increases. The reason for this is that at such a fine granularity there is too much contention on the run queue and as the number of processors increases the contention also increases. The speedup benchmark will demonstrate that as we vary the granularity we do obtain speedup.

In the uniprocessor chapter for each benchmark, we first calculated the performance using a cache friendly application, we then used a non-cache friendly application. In the case of all our SMP shared run queue schedulers, since we insert cache line gaps between the scheduler data structures, and since the threads do not obey the principle of locality (threads migrate across processors regularly during their lifetime), the results of the cache friendly benchmark and the non-cache friendly benchmark are practically identical.
CHAPTER 4. SHARED RUN QUEUE SMP SCHEDULING

Speedup benchmark

A more interesting benchmark on SMPs is a speedup benchmark. We re-run the context switch benchmark from Algorithm 3.7, changing the function thread.test() to include extra computation before every cthread.yield(). We decide the amount of computation by means of a GRANULARITY variable. Algorithm 4.5 demonstrates the speedup benchmark. The idea behind the benchmark is to reduce run queue contention and be able to calculate the minimum granularity at which we can take advantage of multiple processors. Note that our context switch performance benchmark is identical to running the speedup benchmark with GRANULARITY set to 0 (i.e. without any additional computation). The “for loop” where we perform the computation:

for(k = 0; k < GRANULARITY;k++) p = p + 1;

translates to the following assembly language instructions when we compile with full optimisation:

```assembly
movl GRANULARITY,%eax
JUMP: decl %eax
jns JUMP
```

The first instruction initialises the counter. In practice the number of instructions is equal to (GRANULARITY × 2) + 1. We note that the compiler removes the additions of p since they are never used, however if we omitted them from the source code, the compiler would have removed the entire loop.

We execute the benchmark for various granularities. In the case of all SMP schedulers, we plot the results onto two graphs. The first graph compares the time taken by different granularities among each other. The x-axis of the graph represents the GRANULARITY value from Algorithm 4.5 and the y-axis is a measure of the time taken divided by our unit of granularity. The y-axis calculation is necessary to eliminate the extra variable in the form of the thread’s longevity. Another possible benchmark that would lead to the same results would have involved maintaining the thread’s longevity constant and changing the granularity by modifying the number of threads. Algorithm 3.10 is an example of such a benchmark. The second graph represents the amount of speedup against granularity (again represented by GRANULARITY) for the four processor version when compared to the circular run queue uniprocessor implementation. The constant line at speedup 4 represents linear speedup.

Results demonstrate how scheduling at fine granularities suffers from the overhead of the thread scheduler implementation. In particular Figure 4.5 shows how when using four processors for the finest of granularities, performance decreases dramatically. We expect performance to decrease even further as the number of processors increases. The cause of the performance drop is related to the contention for the shared run queue. At very fine granularities the processors spend more of the time attempting to access the shared run queue than computing useful work. This is all the more evident at finer granularities since most of the work that needs to be done is related to the run queue. At coarser granularities the situation improves due to the higher computation load performed by each processor and the reduced frequency of accesses to the run queue. Figure 4.5 demonstrates how at granularities of around 5,000 our scheduler running on four processors starts to approach a good level of speedup (over 3.6). At a granularity of 10,000 (not shown in graph) the speedup rises to 3.85.
Algorithm 4.5 Speedup benchmark.

```c
void thread_test(cthread * ct) { // thread function
    int i, k, p;
    for (i = 0; i < YIELDNUM; i++) { // yield 500,000 times
        for (k = 0; k < GRANULARITY; k++) p = p + 1; // computation
        cthread_yield();
    }
}

cthread_main(cthread * cmain, int argc, char ** argv) {
    int i;
    cthread * c[THREADNUM];
    struct timeval t1, t2;

    for (i = 0; i < THREADNUM; i++) { // initiate threads
        c[i] = cthread_init(thread_test, 4096, 0);
        gettimeofday(&t1, NULL); // begin timer
    }
    for (i = 0; i < THREADNUM; i++) { // execute threads
        cthread_run(c[i]);
    }
    for (i = 0; i < THREADNUM; i++) { // join threads
        cthread_join(c[i]);
    }
    gettimeofday(&t2, NULL); // end timer
}
```
Figure 4.5: Traditional scheduler, speedup results.
### Thread execution

We re-use the thread execution benchmark, Algorithm 3.5 as a measure of the performance of the API routines that use the thread insert and remove operations of our SMP schedulers. In particular the benchmark will serve to highlight choices related to varying the granularity of the scheduler when dealing with the barrier functions and the ability of a scheduler to execute very short-lived threads.

We primarily execute the benchmark for the global lock scheduler. The results are directly affected by the use of one single spin lock around the 25,000 barrier operations. We present all results in microseconds in the following table:

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>Global(1)</th>
<th>Global(2)</th>
<th>Global(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22,227</td>
<td>39,716</td>
<td>45,062</td>
<td>186,740</td>
</tr>
</tbody>
</table>

In the case of the traditional scheduler, since the barrier construct uses a lock separate from the scheduler structures’ lock, there is an amount of concurrency when executing barrier construct routines. There are times when processors are executing a thread insert routine while other processors perform barrier synchronisation routines. This feature is not available in the global lock scheduler. We present thread execution for fine grain locks (in microseconds) in the following table:

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>Traditional(1)</th>
<th>Traditional(2)</th>
<th>Traditional(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22,227</td>
<td>41,951</td>
<td>47,035</td>
<td>166,627</td>
</tr>
</tbody>
</table>

In all cases the overhead when compared to the uniprocessor scheduler is notable, this is due in part to the overhead of implementing the SMP scheduler but mostly to contention for the shared resources. We note how the global lock scheduler outperforms the traditional scheduler when running on a single processor, this is due to the overhead of adding locks to the thread barrier operation. In the dual processor case the same situation applies. However, in the four processor case we notice a performance improvement on the part of the traditional scheduler. The contention on the global shared lock begins to take its toll in the global lock scheduler’s case and the finer grain spin locks permit the traditional scheduler to perform better for this experiment.

### Inter-thread communication benchmarks

We present a new benchmark to compare the different implementations of our channel constructs, in terms of granularity of locks and implementation (i.e. spin lock and lock free implementations, see Section 2.2.1). We could not use the `commstime` benchmark used to compare the uniprocessor schedulers, since not enough channels are used to get good readings when varying the granularities. We introduce a different benchmark, which launches two sets of 25,000 threads which communicate amongst each other through 25,000 channels. We also run an extra set of threads equal in number to the number of processors. The only task of the extra threads is to yield their turn, so that none of the kernel threads ever go to sleep. Algorithm 4.6 demonstrates the benchmark.

We run the benchmark for both our SMP schedulers. We present results for the global lock scheduler in microseconds, in the following table:

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>Global(1)</th>
<th>Global(2)</th>
<th>Global(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>83,117</td>
<td>135,801</td>
<td>236,462</td>
<td>465,345</td>
</tr>
</tbody>
</table>
Algorithm 4.6 Channel benchmark.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#define THREADNUM 25000
cthread * in[THREADNUM];
cthread * out[THREADNUM];
channel * chan[THREADNUM];
cthread * extra[PROCNUM];

thread_in(cthread *p, int i) {  // input threads
    int value;
    channel_in(chan[i],&value, sizeof(int));
}

thread_out(cthread *p, int i) {  // output threads
    int value;
    channel_out(chan[i],&value, sizeof(int));
}

cthread_main(cthread * cmain, int argc, char ** argv) {
    int i;
    struct timeval t1,t2;

    for (i = 0; i < THREADNUM; i++) {  // initiate threads
        in[i] = cthread_init(thread_in,1024,1,i);
        out[i] = cthread_init(thread_out,1024,1,i);
        chan[i] = channel_init();
    }

    for (i = 0; i < PROCNUM; i++) {  // initiate extra threads
        extra[i] = cthread_init(dont_sleep,1024,0);
        cthread_run(extra[i]);
    }

    gettimeofday(&t1,NULL);  // begin timer
    for (i = 0; i < THREADNUM; i++) {  // execute threads
        cthread_run(in[i]);
        cthread_run(out[i]);
    }
    for (i = 0; i < THREADNUM; i++) {  // join threads
        cthread_join(in[i]);
        cthread_join(out[i]);
    }
    gettimeofday(&t2,NULL);  // end timer
}
```
The following table displays results in microseconds for the traditional scheduler where every channel has its own lock:

<table>
<thead>
<tr>
<th></th>
<th>Traditional(1)</th>
<th>Traditional(2)</th>
<th>Traditional(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNI</td>
<td>83,117</td>
<td>148,430</td>
<td>158,293</td>
</tr>
<tr>
<td></td>
<td></td>
<td>266,374</td>
<td></td>
</tr>
</tbody>
</table>

We note a similar pattern from the previous set of results, further accentuated in this case. The global lock scheduler performs better in the single processor case, but the performance degrades as the number of processors increases. The communication benchmark demonstrates an improvement for the traditional processor even in the dual processor case. The performance of the benchmark on four processors is notable for a close to 100% improvement.

4.3.5 Analysis of results

Our results, in particular the speedup results, are very similar to those obtained by other shared run queue implementations. Cordina’s [25] results in particular attain close to 90% of ideal speedup over a similar range of granularities. The results further demonstrate that when scheduling using shared run queues, a high degree of parallelism is achieved only at quite coarse granularities. The cost of running very fine grain applications on our shared run queue scheduler implementation is prohibitive and if performance is an issue, fine grain applications would be better off using a uniprocessor scheduler. The global lock scheduler demonstrated the extreme of using one global lock to protect all shared data structures from concurrent access. The traditional scheduler on the other hand used the approach of using one lock for internal scheduler structures and individual locks for the communication constructs. We demonstrated how the traditional scheduler’s strategy helps increase performance when a large number of threads communicate among each other.

The overhead due to implementing the SMP version and the contention on the shared run queue for very fine grain applications are the main drawbacks for this type of scheduling. On the other hand performance does improve for coarser grain applications and we are ensured that most applications can attain good speedup and maintain it due to the scheduler’s strong load balancing properties.

4.4 Thread scheduling using fine grain locks

We noted by means of the traditional scheduler the problem of scheduling threads on multiprocessors at very fine granularities is related to the amount of contention on the shared resources. In particular the shared run queue is accessed constantly by all processors for fine grain applications. At these granularities the spin lock affects performance since most of the time kernel threads will be busy waiting to access the shared run queue. We attempt to reduce the amount of time the kernel threads busy wait by using more than one spin lock for the scheduler’s shared data structures.

4.4.1 Introduction

We implement two versions of our fine grain thread scheduler. Both of them have separate locks for each of the communicating constructs (just like the traditional scheduler), one separate lock for the internal structures that handles the individual processor’s current thread pointers and the kernel thread idling system, and also a separate access control
mechanism for the shared run queue. The only difference between the two fine grain lock
schedulers is in the protection of concurrent access to the shared run queue. The scheduler
we shall henceforth refer to as the single lock scheduler\(^3\) has one single lock to protect the
run queue, while the version we will refer to as dual lock scheduler uses a dual lock system to
protect the shared run queue. We noted in the uniprocessor section when using the dummy
head scheduler, how reducing head and tail interaction improved the performance of certain
applications, in particular those that made use of many insert and remove routines. CSP
applications benefited greatly from this type of scheduling. The dual lock scheduler uses
the same ideology and decreases concurrent access to the shared run queue by adapting
Michael and Scott’s dual lock concurrent queue implementation [79]. The implementation
uses separate spin locks for enqueue and dequeue routines, thus if a processor attempts
to perform an enqueue while another performs a dequeue, there is no form of interference
amongst them. Figure 4.6 demonstrates the dual lock scheduler.

### 4.4.2 Adapting Michael and Scott’s dual lock algorithm

Michael and Scott’s dual lock algorithm relied on nodes as a transport vehicle for the
data items on the concurrent queues. Nodes are a supporting structure commonly used in
abstract data type implementations. Their main function is to create a layer of abstraction
between the queue and the data type of the items to be placed on the queue. In the
implementation of concurrent queues nodes\(^4\) serve also as a transport mechanism which
when used with dummy head strategy eliminates the head and tail interaction in the enqueue
and dequeue routines. In Michael and Scott’s algorithm every enqueue operation created a
new node and every dequeue operation freed the node. The overhead of allocating memory
for a node for every enqueue is prohibitive since this would mean entering the kernel every
time. The same argument applies to freeing a node for every dequeue operation. Moreover,
the integrity of the concurrent queue depends on the nodes otherwise the head and tail
would not be truly separate and we would not be able to use separate spin locks for the
enqueue and dequeue routines. In the original algorithm, the initialisation routine creates a
dummy head and makes both the head and tail point to it. The enqueue operation simply
adds an element to the end of the queue and makes the tail point to it. The enqueue routine
removes the node representing the dummy head and returns the data of the node right after
the dummy head when the queue is not empty. If we do not use nodes, when performing
a dequeue routine we would need to remove the item right after the head, which would be
the tail in the case of an empty queue and this would mean protecting the tail also in the
dequeue routine.

We opt instead to make use of nodes, but we avoid allocating and de-allocating memory
for every enqueue and dequeue operation by recycling the nodes. Every time we initialise a
thread descriptor we also initialise a node structure associated with it. For every enqueue
operation we call into play the node structure associated with our thread descriptor. Since
in the dequeue operation we remove the dummy head node but the thread we return is
the thread pointed to by the real head, we cannot re-use the same node for the thread

\(^3\)The single lock scheduler must not be confused with the global lock scheduler. The global lock scheduler
uses one single global lock to protect concurrent access to all shared resources. The single lock scheduler
is a fine grain lock scheduler that uses a separate lock for the shared run queue, communication constructs
and internal scheduler structures. The single lock refers to the single spin lock used to access the run queue
as opposed to using two locks as is the case with the dual lock scheduler.

\(^4\)Valois [103] uses the term cell to represent the very same structure we call nodes, in his implementation
of non-blocking concurrent queues
Figure 4.6: Dual lock shared run queue scheduling.
CHAPTER 4. SHARED RUN QUEUE SMP SCHEDULING

Figure 4.7: Inserting thread \( NU \), which is bound to node \( NN \), onto the dual lock shared run queue.

Figure 4.8: Removing thread \( RU \) from the dual lock shared run queue. The thread \( RU \) is then bound to the removed node \( RN \).

because the dequeued thread’s node becomes the new dummy head. Instead we associate the dequeued thread with the dequeued node. This method adds a certain amount of overhead, however the end result means we can concurrently execute enqueue and dequeue routines without any contention.

The single lock scheduler makes use of the same algorithms outlined above but protects enqueue and dequeue routines with the same lock. Unfortunately the excess baggage renders the version of the scheduler slower than what it would be without nodes, but only very slightly. Results from the single lock scheduler give an insight to the overhead we added compared with the traditional scheduler, so that the reader might note how we make up for the extensive overhead by reducing the contention for the dual lock scheduler.

4.4.3 Internal scheduler functions

The internal scheduler functions are very similar to their traditional scheduler counterparts. The only difference is in the granularity. Differences between the one lock and dual lock version are mentioned only when the case applies. In all other cases the implementations are identical.

Thread insert

The thread insert routine performs the same function as in the previous scheduler’s case. If the sleeper counter representing the number of idle processors is greater than 0, the
scheduler cycles through the processors and looks for one that is asleep, in which case the thread is assigned to that processor and the kernel thread associated with the processor is awoken. This part of the function is protected by one single lock representing the internal data structures. Individual locks could have been used to protect the sleeper counter and each of the processor’s current thread pointers, however we believe this would have been excessive, since only one or two instructions are performed in each case. When the sleeper counter is 0, the thread is inserted onto the run queue. The single lock scheduler uses the single run queue lock to protect the enqueue operation, while the dual lock uses the specific enqueue lock. By the time the enqueue operation is performed the internal scheduler structures’ lock would have been released.

**Thread remove**

When removing a thread the scheduler first performs a dequeue operation using the dequeue lock (or the general queue lock in the single lock scheduler’s case), if the returned result is non-NULL the next thread on the run queue is executed. If the result is NULL the sleeper counter is incremented (in this case only we use the internal scheduler structures’ lock) and the current kernel thread is set to sleep.

**Thread yield**

In the case of the thread yield the scheduler removes a thread from the run queue, enqueues the processor’s current thread and executes the dequeued thread. Unfortunately every yield involves the use of two spin locks, one for the dequeue routine and one for the enqueue routine.

### 4.4.4 Thread API functions

The API functions that make best of this scheduler are the communication constructs when running fine grain applications. Both the channel and barrier based routines benefit because of the finer grain nature of the schedulers’ locks and the ability to perform concurrent enqueue and dequeue operations without contention (in the dual lock scheduler’s case). The yield function suffers mostly due to the fact that we need to acquire two locks in quick succession.

### 4.4.5 Results

We perform our benchmarks in the same sequence we used for the traditional scheduler’s results.

**Context switch performance**

We execute the benchmark outlined in Algorithm 3.7. The first set of results represent the single lock scheduler. All results are displayed below in nanoseconds:

<table>
<thead>
<tr>
<th>UNI</th>
<th>Single lock(1)</th>
<th>Single lock(2)</th>
<th>Single lock(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>264</td>
<td>1,534</td>
<td>2,122</td>
</tr>
</tbody>
</table>

The single lock scheduler performs poorly. This is due to mainly having two spin locks in quick succession in the yield routine, and only a couple of instructions between them.
If the number of instructions between the spin locks was larger we would expect some sort of improvement. Our second set of results below, represent the dual lock scheduler (nanoseconds):

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>Dual lock(1)</th>
<th>Dual lock(2)</th>
<th>Dual lock(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>28</td>
<td>264</td>
<td>1,251</td>
<td>1,503</td>
</tr>
</tbody>
</table>

While our results demonstrate a notable improvement for the dual lock scheduler as opposed to the single lock scheduler, the dual lock scheduler’s results fall just short of the traditional scheduler. The main reason for this, we believe, is the overhead due to having to acquire two spin locks in the yield routine. The single processor results demonstrate the overhead is notable when compared to the traditional scheduler. We note however how the difference in performance improves when the number of processors increases. The performance difference in the two processor case is larger than in the four processor case. In fact we predict that as the number of processors increases the performance of the dual lock implementation will equal and surpass the traditional SMP scheduler.

**Speedup benchmark**

In view of the previous results which are a strong indication of the speedup results, we perform our speedup benchmark only in the case of the dual lock scheduler. We apply Algorithm 4.5 to our dual lock scheduler.

The results from Figure 4.9 demonstrate this benchmark echoes results from the previous benchmark where the performance of the dual lock scheduler falls just short of the performance of the traditional SMP scheduler. Again we expect better results when scheduling on more processors.

**Thread execution**

We run our thread execution benchmark (Algorithm 3.5) for both our fine grain schedulers. In the case of the single lock scheduler we present the following results in microseconds:

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>Single lock(1)</th>
<th>Single lock(2)</th>
<th>Single lock(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22,227</td>
<td>57,518</td>
<td>46,230</td>
<td>149,097</td>
</tr>
</tbody>
</table>

Results are surprising in this case, because even despite the large overhead evident when running on one processor the single lock scheduler outperforms the traditional scheduler in the four processor experiment. Although we had written off the single lock scheduler, the good results are due to using a separate lock for the internal scheduler structures and the shared run queue. Below we present the thread execution benchmark for the dual lock scheduler (in microseconds):

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>Dual lock(1)</th>
<th>Dual lock(2)</th>
<th>Dual lock(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22,227</td>
<td>57,508</td>
<td>45,832</td>
<td>63,911</td>
</tr>
</tbody>
</table>

In the case of the dual lock scheduler we note that in the two processor experiment we equal the best performance of the traditional scheduler, however we achieve overwhelming results in the case of four processors. The reason for the improvement is that this benchmark in particular makes use of many concurrent enqueues (for launching the threads) and dequeues (for the barrier routines and termination). Since there is no contention between the two routines the dual lock scheduler’s results when running the benchmark on four processors achieves more than 250% improvement over the traditional scheduler.
## CHAPTER 4. SHARED RUN QUEUE SMP SCHEDULING

<table>
<thead>
<tr>
<th>Granularity</th>
<th>SMP(4)</th>
<th>SMP(2)</th>
<th>SMP(1)</th>
<th>UNI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (microseconds)</td>
<td>0</td>
<td>500</td>
<td>1000</td>
<td>1500</td>
</tr>
<tr>
<td>Speedup</td>
<td>0</td>
<td>5000</td>
<td>10000</td>
<td>15000</td>
</tr>
</tbody>
</table>

### Figure 4.9: Dual lock scheduler, speedup results.
Inter-thread communication benchmarks

We run the same benchmark outlined in Algorithm 4.6. Results for the single lock scheduler are displayed in microseconds in the following table:

<table>
<thead>
<tr>
<th>UNI</th>
<th>Single lock(1)</th>
<th>Single lock(2)</th>
<th>Single lock(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>83,117</td>
<td>181,437</td>
<td>186,531</td>
<td>261,065</td>
</tr>
</tbody>
</table>

Results in microseconds for the two lock scheduler follow:

<table>
<thead>
<tr>
<th>UNI</th>
<th>Dual lock(1)</th>
<th>Dual lock(2)</th>
<th>Dual lock(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>83,117</td>
<td>183,249</td>
<td>197,486</td>
<td>187,383</td>
</tr>
</tbody>
</table>

Since this benchmark uses many separate enqueue and dequeue operations, the advantage of having separate locks for enqueue and dequeue routines is evident for the dual lock scheduler running on four processors. We again note a slight increase in performance even in the single lock scheduler’s case, due to the finer granularity of the scheduling.

4.4.6 Analysis of results

Both fine grain schedulers help to demonstrate the advantage of using finer grain locks for fine grain inter-thread communication. In terms of the communication benchmarks, the single lock scheduler demonstrates the superiority of fine grain locking, outperforming both the global lock scheduler and the traditional scheduler. However, the overhead due to having to acquire the same lock in succession, for yield routines, is too high to achieve good speedup in our benchmark. In the case of the dual lock scheduler the advantage of having separate head and tail locks is evident for machines with four or more processors that use CSP based applications. In this respect the dual lock scheduler is very similar to the uniprocessor dummy head scheduler. The fact that at four processors the speedup benchmark’s performance begins to approach that of the traditional scheduler is a very good indication of the potential of this scheduler. Although we do not have access to machines with more than four processors, we predict an improvement when scheduling with eight or more processors. If this were the case, the dual lock scheduler would outclass the traditional scheduler in all aspects.

4.5 Non-blocking scheduling

The various fine grain schedulers have demonstrated how locking at finer granularities improves performance for certain applications. In particular CSP programs and applications that make use of short-lived threads benefit from the fine grain locks, in terms of communications constructs and internal scheduler structures. The dual lock scheduler highlights the advantages of maintaining separate head and tail locks and outperformed the more traditional scheduler in some of our benchmarks, and came reasonably close in the others. With this in mind we attempt to remove locks altogether, by adapting Michael and Scott’s [77] non-blocking concurrent queue as our shared run queue, Vella’s [108] wait free CSP channel algorithms for our channel implementations and introducing new wait free algorithms for internal scheduler structures and the barrier construct.
4.5.1 Introduction

The non-blocking queue algorithms and the various wait free algorithms make the entire scheduler non-blocking. We only use locks when allocating memory, though this can be avoided if memory is allocated statically. The non-blocking concurrent queue uses the the same dummy head technique discussed in the case of the dual lock scheduler to separate the head and the tail. However, the modifications we implemented to the dual lock scheduler in terms of associating a node to every thread are not enough in this case, as we shall describe below.

Our non-blocking implementation makes use of the atomic primitives first discussed in Section 2.2.1. The wait free algorithms introduced by Vella make use of an atomic swap instruction, and our internal scheduler synchronisation for migrating threads across to sleeping processors makes use of compare and swap as does our barrier algorithm. The non-blocking concurrent queue introduced by Michael and Scott makes use of a alternative compare and swap that compares and swaps a double word, effectively comparing and swapping two consecutive memory locations. The the double word compare and swap is needed to avoid the ABA problem which we discussed in Section 2.2.3. The use of the
Algorithm 4.7 Wait free thread migration for idle processors.

W1 for (count = 0; count < PROCNUM; count++) {
W2 if (CAS(&sched.current[count], NULL, thread)) {
W3 kernelthread_wakeup(count);
W4 return;
W5 }
W6 }
W7 runqueue_enqueue(thread);

double word compare and swap adds further complications to the thread’s structures since each node must contain be associated with a counter, in a separate structure, Michael and Scott refer to as a pointer. The concurrent queue could be viewed as a queue of pointers, however it is not the pointer structure that points to the next structure in the queue. Each node structure contains a next pointer which references the next pointer structure in the queue. Figure 4.10 illustrates the operation of our non-blocking scheduler.

4.5.2 Internal scheduler functions

We discuss our non-blocking internal scheduler function implementations below. We focus in particular on the thread insert routine, which makes use of our new wait free algorithm.

Thread insert

In the both the single lock scheduler and the dual lock scheduler we recall that we used an independent lock for maintaining the sleeper counter and to protect the processor’s current pointers. We now present wait free algorithms for the above process. We present two versions of the algorithm. The first version does not make use of the sleeper counter, adds overhead (since we always assume we have sleeping kernel threads) but maintains perfect load balance. The second version uses a sleeper counter, which is atomically updated whenever a kernel thread is awakened or put to sleep. However, in between the time we check for sleepers and take the appropriate action the sleeper counter may change, and this may lead to our load balancing rule (there are no idle processors when there is work on the run queue) being broken. The run queue and other internal data structures are not corrupted when this happens. The performance improvement makes up for the occasional loss of load balancing as the results will show.

Consider Algorithm 4.7 for the load balanced version. When inserting a thread the scheduler cycles through all the processors’ current pointers (W1) and if one of them is NULL, it means they are idle (or about to become idle) in which case the thread is inserted there, and the kernel thread associated with that processor is awakened (W3) and the routine is exited (W4). The whole operation is done atomically by means of an atomic compare and swap instruction (W2). When none of the kernel threads are asleep the thread is placed onto the run queue (W7).

The second version of the algorithm (see Algorithm 4.8) is identical to the first except the sleeper number check is added (W1). There is also another test (W3) to see if the current processor is idle, before performing a compare and swap operation, since a compare and swap instruction is more expensive than a conditional. Note that the sleeper counter is decremented atomically (W5) before the kernel thread is awakened.
**Algorithm 4.8** Wait free thread migration for idle processors using a sleeper counter.

```c
if (sleepernum != 0) {
    for (count = 0; count < PROCNUM; count++) {
        if (current[count] == NULL) {
            if (CAS(&current[count], NULL, thread)){
                atomic_dec(sleepernum);
                kernelthread_wakeup(count);
                return;
            }
        }
    }
    runqueue_enqueue(thread);
}
```

**Algorithm 4.9** Wait free barrier algorithm.

```c
void cthread_join(cthread * joiner, thread)
{
    int id = kernelthread_id();
    if (CAS(joiner, thread->joiner, NULL, sched.current[index])){
        if (sched.current[id] = runqueue_dequeue()) sched jmp(id);
        else kernelthread_sleep(id);
    }
}
```

**Thread remove**

The thread remove routine is similar to the other SMP schedulers’ routines except it uses the non-blocking run queue for enqueue and dequeue operations.

**Thread yield**

The thread yield is similar to the fine grain thread yield with the exception of the non-blocking enqueue and dequeue routines.

### 4.5.3 Thread API functions

The thread API functions have the same advantages associated with the dual lock scheduler, in that thread insert and thread remove interaction is avoided. The thread yield routine suffers from high overhead due to the complex nature of the structures involved. The channel calls are implementations of Vella’s wait free CSP channels. We introduce the wait free barrier algorithms below.

**Wait free barrier algorithms**

We recall that our barrier mechanism is specified as a communication construct between two threads whereby the current thread associates itself with another thread and the current thread is set to sleep until the other thread terminates. Only one thread can place itself onto another thread’s barrier construct.
Algorithm 4.10 Wait free barrier algorithm at thread termination.

W1 int id = kernelthread\_id();
W2 if (!CAS(&sched.current[id]->joiner,NULL,sched.current[id])){
    W3 sched.current[id] = sched.current[id]->joiner;
    W4 sched\_jmp(id);
    W5 }

The caller’s barrier algorithm is described in Algorithm 4.9. The contents of the joining thread are compared with NULL and if the result of the operation is true the joining thread is set to the current thread (W3). This operation makes use of atomic compare and swap operation. If the compare and swap operation returns true the scheduler attempts to obtain another thread (W4), if there are no more threads the kernel thread is set to sleep (W5). If the compare and swap instruction returns false the thread continues executing where it left off.

Algorithm 4.10 describes the barrier algorithm in the thread terminate routine. The scheduler atomically checks, by means of a compare and swap instruction, if the status of the terminating thread’s joiner is NULL, and if it is, the same thread is assigned to its joiner (W2) (so that other threads would know the thread has terminated) and the thread termination routine continues (W5). If the compare and swap is false, it means there was a thread assigned to the current thread’s joiner, therefore the joiner thread is run directly (W3).

4.5.4 Results

We present results for the non-blocking scheduler. Unless otherwise specified we will be using the correctly load balanced version of the scheduler.

Context switch performance

We present results for the context switch benchmark (Algorithm 3.7) in nanoseconds:

<table>
<thead>
<tr>
<th></th>
<th>UNI Non-blocking(1)</th>
<th>Non-blocking(2)</th>
<th>Non-blocking(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>28</td>
<td>393</td>
<td>1,256</td>
</tr>
</tbody>
</table>

We note the high overhead of 393\(\text{ns}\) for the experiment running on one processor. This could be attributed mainly to the complex structures used to implement the Michael and Scott algorithm and to avoid the ABA problem. The results for the two processor and four processors do not make up for the initial expense. Moreover, unlike the case of the dual lock scheduler there does not seem to be an improvement in performance, so we are unsure about what to expect when using more than four processors.

Speedup benchmark

Results for the speedup benchmark (Algorithm 4.5) confirm the disappointing results of the context switch benchmark. Results are displayed in Figure 4.11. The non-blocking run queue does not seem to perform as well as expected, and is outperformed by both the traditional scheduler and the dual lock scheduler in this benchmark, which is the essential run queue test. We suspect the problem to be in the overhead due to complexity of the run queue’s structure. Shann, Huang and Chen [91] seem to suggest that the memory
Thread execution

We execute the thread execution benchmark for Algorithm 3.5 for our non-blocking scheduler. Below are the results for the load balanced version (results in microseconds):

<table>
<thead>
<tr>
<th>UNI</th>
<th>Non-blocking(1)</th>
<th>Non-blocking(2)</th>
<th>Non-blocking(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22,227</td>
<td>56,335</td>
<td>41,385</td>
<td>55,986</td>
</tr>
</tbody>
</table>

The results due to the load balanced version of the non-blocking scheduler show an improvement compared to the other schedulers. We execute the results for the version of the non-blocking scheduler that makes use of the sleeper counter:

<table>
<thead>
<tr>
<th>UNI</th>
<th>Non-blocking(1)</th>
<th>Non-blocking(2)</th>
<th>Non-blocking(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22,227</td>
<td>55,514</td>
<td>38,301</td>
<td>49,684</td>
</tr>
</tbody>
</table>

The results from the off-balanced implementation of the non-blocking scheduler outperform all other results demonstrated so far. The results demonstrate that both the wait free algorithms introduced (the barrier algorithm and the thread insert algorithm) are faster than their lock based counterparts.

Inter-thread communication benchmarks

We execute the benchmark from Algorithm 4.6 and present all results in microseconds below:

<table>
<thead>
<tr>
<th>UNI</th>
<th>Non-blocking(1)</th>
<th>Non-blocking(2)</th>
<th>Non-blocking(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>83,117</td>
<td>180,716</td>
<td>176,083</td>
<td>184,683</td>
</tr>
</tbody>
</table>

Again the results confirm the wait free implementation is faster than the lock based implementations, even though running on a slower scheduler.

4.5.5 Analysis of results

The results from this chapter confirm the strength of wait free implementations for the communication constructs and thread execution, but the non-blocking run queue falls short of expectations. Again we could argue this scheduler would be best used for certain applications such as applications using short-lived threads and CSP applications. However, we can confirm that a hybrid scheduler of dual lock shared run queue and wait free constructs would outperform this scheduler.

Although ultimately disappointing the non-blocking nature of the scheduler does have other advantages in that the lack of locks prevents deadlock and provides better fault tolerance. Furthermore, the wait free algorithms prevent starvation.

4.6 Conclusion

In this chapter we have discussed the implementation of shared run queue schedulers. We compared and contrasted distinct run queue implementations and also focused our attention
Figure 4.11: Non-blocking scheduler, speedup results.
on the implementation of other internal scheduler data structures and algorithms, CSP channel algorithms and barrier communications.

We conclude the chapter by comparing results from the various schedulers in Figure 4.12 and in the table below:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Global lock</th>
<th>Traditional</th>
<th>Single lock</th>
<th>Dual lock</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>context switch (ns)</td>
<td>1,424</td>
<td><strong>1,424</strong></td>
<td>2,122</td>
<td>1,503</td>
<td>1,979</td>
</tr>
<tr>
<td>execution (µs)</td>
<td>186,740</td>
<td>166,627</td>
<td>149,097</td>
<td>63,911</td>
<td>49,684</td>
</tr>
<tr>
<td>communication (µs)</td>
<td>465,345</td>
<td>266,374</td>
<td>261,065</td>
<td>187,383</td>
<td>184,683</td>
</tr>
</tbody>
</table>

It would be tempting to conclude the chapter by highlighting the advantages and disadvantages of all schedulers and concluding that each scheduler is ideal for certain applications. Although this line of thought can be applied to the SMP shared run queue schedulers, some of our scheduler implementations are not useful (e.g. the single lock scheduler). There are many factors which influence the performance of the individual schedulers, therefore we prefer to discuss the merits of the individual structures before finally discussing the schedulers.

Out of the four distinct run queue configurations we discussed it was the traditional scheduler’s shared run queue implementation that prevailed. The dual lock scheduler’s shared run queue was close behind, and seems to be more scalable. The dual lock scheduler’s run queue is also more suitable for applications that make use of CSP constructs or short-lived threads. The single lock scheduler’s run queue, performs poorly since at fine granularities there are only a couple of instructions between enqueue and dequeue operations which entail a cost of an extra spin lock and release. The non-blocking run queue
performs poorly and the main reason is possibly due to the high overhead. We have no means of predicting the performance when using more processors. The non-blocking scheduler run queue, like the dual lock scheduler run queue, has the advantage of being able to concurrently execute enqueues and dequeues in the general case.

In terms of internal scheduler routines and communication constructs we have demonstrated that the finer grain schedulers have an edge over the coarser grain implementations. The wait free implementations in particular do without any contention whatsoever and results demonstrate they provide better performance overall.

In terms of the five schedulers, some schedulers would benefit from algorithms and structures we introduced for other schedulers. The global lock scheduler could remain just the way it is, i.e. ideal for applications where only few threads communicate amongst each other. The traditional scheduler would benefit from the wait free implementations introduced for the non-blocking scheduler, and thus communication would be nearly as fast as that of the non-blocking scheduler. The single lock scheduler could also do with the wait free constructs but we have seen that whatever its application the dual lock scheduler would outperform it. The dual lock scheduler would also benefit from the wait free algorithms and would be the fastest for applications that make extensive use of communications and short-lived threads. The non-blocking scheduler still performs well and has the same advantages as the dual lock scheduler but is slightly slower. The non-blocking property could be considered an advantage in itself in terms of deadlock avoidance and fault tolerance.
Chapter 5

Per processor run queue SMP scheduling

In this chapter we focus our attention on scheduler implementations which make use of individual run queues for each processor, and discuss possible strategies for migrating threads amongst each processor’s run queue. We will describe three distinct scheduler implementations. The first implementation introduces the per processor run queue strategy in its most basic form and highlights the highs and lows of the per processor run queue ideology. The second scheduler is an implementation of Vella’s [107] description of SMP batching, where we use batching as first described in Chapter 3, to provide for migration of jobs amongst processors. The third and final scheduler, is a lock free scheduler which uses wait free structures and algorithms to provide for synchronisation, scheduling and migration, making the whole scheduler implementation wait free. We experiment with various forms of migration on our wait free scheduler, namely sender initiated migration, receiver initiated migration and hybrids of the two.

5.1 Per processor run queues

Per processor run queue implementations differ from the shared run queue implementations mainly in that every processor services its own personal run queue and executes threads only from that particular run queue. The main advantages that per processor run queue based schedulers have over their shared run queue counterparts is the reduced contention over the most frequently accessed shared resource, the run queue, and stronger adherence to the principle of locality. On the other hand the per processor run queue strategy suffers mostly from poor load balancing.

5.1.1 The shared run queue bottleneck

As the benchmarks in the previous chapter have demonstrated, our shared run queue implementations did not perform well at very fine granularities. For fine granularities the uniprocessor version outperforms the multiprocessor version running on four processors. The main reason for this is that all our kernel threads continuously access the shared run queue, and very often have to busy wait until the shared resource is accessible. Per processor run queues naturally do without the global shared run queue, thus removing the major bottleneck associated with multiprocessor schedulers.
CHAPTER 5. PER PROCESSOR RUN QUEUE SMP SCHEDULING

5.1.2 The principle of locality

The principle of locality was completely neglected in our shared run queue implementations. We demonstrated the importance of cache-affinity scheduling when we introduced our uniprocessor batching scheduler. On multiprocessors the principle of locality is given further importance due to false sharing (see Section 2.1.3). We will demonstrate in Section 5.2.4 the devastating affects of having processes share data on the same cache lines. For shared run queue implementations, a flag (in the thread’s workspace) denoting the last processor accessed could have been used to improve processor affinity however this would decrease performance due to the overhead of having to check what processor the thread was meant to run on. In the per processor run queue implementation’s, cache affinity is part of the nature of the scheduling structures. Moreover, as we did with the uniprocessor version, we can further enhance cache affinity by means of batching based scheduling.

5.1.3 Load balancing

The major drawback of per processor run queues is poor load balancing. On shared run queues, we never had a run queue with threads waiting on it while processors are lying idle. When using per processor scheduling, thread placement onto the individual run queues must be carefully implemented, so as not to overload a processor while other processors are idle. One situation that we would like to avoid is demonstrated in Figure 5.1. To avoid such situations we need to consider possible migration of threads between processors, even migrating them more than once in their lifetime.

5.2 Simple per processor run queue scheduling

We now introduce the first of our per processor run queue schedulers, which uses the most basic of scheduling algorithms short of running an independent uniprocessor scheduler on each processor. Although this scheduler has a very simple software architecture, when used correctly and for certain applications, readers will be surprised by the results. Under the right conditions this scheduler could be highly efficient. We will refer to this scheduler implementation as the PP scheduler.

5.2.1 Introduction

The PP scheduler’s run queues, are implemented as circular queues similar to those present in the uniprocessor circular run queue version of our scheduler. The philosophy of the PP scheduler is similar to running the uniprocessor version on each of our processors, in that except for the first \(N\) threads (where \(N\) is the number of processors) any thread that is created spends its lifetime on the same processor, unless the thread makes use of a communicating construct. No run-time migration of threads takes place to attempt to improve load balancing dynamically. This scheduling strategy strongly highlights the advantages and disadvantages of multiple run queue implementations.

The scheduler assigns the first thread to be created from the main thread to the first processor, the second thread (again from the main thread) to the second processor \(\textit{etc}…\). Only the main thread assigns threads to other processors, and they must be idle. Any other threads created on the main thread after at least one thread has been assigned to each processor remain associated with that processor. Furthermore each thread created by a
Figure 5.1: Possible gross load imbalances when using per processor run queues.
Figure 5.2: Simple per processor scheduling.
thread other than \texttt{cthread}\texttt{main}() is created on the local processor. Threads are migrated on creation and always from the main thread to other processors. Moreover this happens only once for each processor. The only other reason threads are migrated is when threads are removed from a run queue and placed on some alternative communication structure such as a CSP channel. When a thread is unblocked from the communication construct, it is placed on the run queue of whichever processor woke up the thread. This scheduling strategy helps us do without using atomic primitives or spin locks except in the implementation of the communication routines which must make use of the lock based or wait free algorithms we discussed in Chapter 4.

As mentioned before, this scheduling strategy can lead to gross imbalances, but if the application programmer is made aware of the internal scheduler structures, programs could be constructed to suite the software architecture.

\section*{5.2.2 Internal scheduler functions}

Since the PP scheduler is very similar to the circular run queue uniprocessor scheduler, most of our functions follow the same description, with a few exceptions. In all the routines described below, the first operation to be performed is to check which processor the scheduler is running on.

\subsection*{Thread insert}

As we mentioned before, the thread insert routine, inserts the thread onto the circular run queue corresponding to the the processor on which the insert routine was run. The only case in which this rule is not obeyed is when the main thread inserts one of the first \(N\) threads (where \(N\) is the number of processors), in which case the \(N\)th thread is inserted onto the \(N\)th processor. In this special case the main thread must wake up the kernel thread associated with that processor. This is the only time the scheduler wakes up kernel threads when using this scheduler. The thread associated with the processor then becomes a virtual main thread for that given processor, in that the kernel thread will sleep when this thread terminates and cannot be awaken again.

\subsection*{Thread remove}

Thread remove simply removes a thread from the circular run queue, and checks if there are any other threads on the queue, in which case it sets the kernel thread associated with the current processor to sleep. In this sense our remove routine is just like the circular run queue version except that it sleeps instead of exits when it does not find anything else to run.

\subsection*{Thread yield}

Our yield routine is identical to the circular run queue routine, with the exception of having to check which kernel thread we are running on before commencing. All the routine does is save the current thread’s context and execute the next thread on the local run queue.

\section*{5.2.3 Thread API functions}

Apart from checking which kernel thread they are running on, the thread API functions are affected in the same way as the uniprocessor. The exceptions are the thread execute
function which must cater for the first four threads and the communication and barrier routines which use the wait free algorithms developed for the non-blocking scheduler (see Section 4.5).

5.2.4 Results

For the per processor run queue results we decided to focus our attention on speedup and context switch performance. The results from Chapter 4 due to the thread execution benchmark (Algorithm 3.5) and the inter-thread communication benchmark (Algorithm 4.6) highlighted the superiority of wait free data structures and algorithms for inter-thread communication and therefore we implement those particular algorithms in all our per processor run queue schedulers.

We mentioned before the need to design applications keeping in mind the PP scheduler’s architecture to maximise performance. The speedup benchmark we used in the previous chapter (Algorithm 4.5) will serve as a demonstration. It will also serve as an example of the possible load imbalances we can be subject to due to the design of this particular scheduler. In light of this we begin the results section with the speedup benchmark.

**Speedup benchmark**

If we were to run the speedup benchmark (Algorithm 4.5) setting `GRANULARITY` to 5,000, a figure at which all the SMP shared run queue schedulers tested have always achieved a considerable speedup we would achieve the following results (in seconds):

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>Traditional(4)</th>
<th>PP(1)</th>
<th>PP(2)</th>
<th>PP(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results</td>
<td>143.337710</td>
<td>38.842592</td>
<td>143.423786</td>
<td>136.253501</td>
<td>121.933771</td>
</tr>
</tbody>
</table>

The results clearly show only very slight speedup, when using two and four processors. The traditional scheduler on the same test using four processors completed the benchmark in 38.84 seconds. The reason for the poor performance on the part of the PP scheduler is that the workload is not balanced equally amongst the processors. Consider the case of when we run the benchmark with four processors. We have 20 threads, all doing the same job, i.e. they perform 5,000 additions (since `GRANULARITY` is set to 5,000) and yield after each set of additions for a total of 10,000,000 yields. Ideally we would distribute 5 threads on each of the run queues, however our thread insert algorithm allocates the main thread to the first processor (i.e it is placed on the first processor’s run queue), the next 3 threads to be created are allocated to the remaining processors’ run queue and all the workload that is left (17 threads) are all placed onto the the first processor’s run queue. The imbalanced workload is due to the fact that our scheduler does not migrate threads, a feature we will add to the forthcoming multiple queue based schedulers.

Notwithstanding the poor results due to load imbalance, if we correctly design our programs for the PP scheduler, we can achieve very good performance. We present a speedup benchmark similar to Algorithm 4.5, where we make sure the first four threads are assigned an equal workload. This simple solution works wonders if the threads all do the same job as is the case with our speedup benchmark. Algorithm 5.1 demonstrates our adapted benchmark. Note how we first create a virtual main thread for each processor, which we associate to the function `threadlaunch()`. We use the same `thread_test()` as in Algorithm 4.5. Using the new speedup benchmark, the per processor run queue achieves the following results (in seconds):

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>Traditional(4)</th>
<th>PP(1)</th>
<th>PP(2)</th>
<th>PP(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results</td>
<td>143.337710</td>
<td>38.842592</td>
<td>143.423786</td>
<td>136.253501</td>
<td>121.933771</td>
</tr>
</tbody>
</table>
Algorithm 5.1 Speedup benchmark for per processor scheduler.

```c
#define THREADNUM 20 // Number of threads
#define YIELDNUM (10000000 / 20) // Number of yields per thread
#define TNUM (THREADNUM / PROCNUM) // Number of threads per processor

void threadlaunch(cthread * ct, int id) {
    // virtual main thread
    int i;
    cthread * c[TDNUM];

    for (i = 0; i < TNUM; i++) // initiate threads
        c[i] = cthread_init(thread_test,4096,0);
    for (i = 0; i < TNUM; i++) // execute threads
        cthread_run(c[i]);
    for (i = 0; i < TNUM; i++) // join threads
        cthread_join(c[i]);
}

cthread_main(cthread * cmain, int argc, char ** argv) {
    // a thread per processor
    int i;
    cthread * p[PROCNUM];
    struct timeval t1,t2;

    for (i = 0; i < PROCNUM; i++) // initiate main threads
        c[i] = cthread_init(thread.launch,65536,0);
    gettimeofday(&t1,NULL); // begin timer
    for (i = 0; i < PROCNUM; i++) // execute threads
        cthread_run(p[i]);
    thread.launch(NULL,0); // thread.launch() for processor 0
    for (i = 0; i < PROCNUM; i++) // join threads
        cthread_join(p[i]);
    gettimeofday(&t2,NULL); // end timer
}
```
The results clearly show a huge improvement in performance when using the new benchmark. We plot results from the speedup benchmark at varying granularities on two separate graphs as we did for the other SMP schedulers (Figure 5.3). Results clearly demonstrate the potential of per processor run queues schedulers. This is the first scheduler that obtains close to an $N$-way speedup at the finest of granularities (where $N$ is the number of processors).

**Context switch performance**

As we did with the speedup benchmark we adapt our context switch benchmark to cater for the PP scheduler’s architecture. The benchmark is identical to running Algorithm 5.1 with GRANULARITY set to 0. We display all the results in nanoseconds below:

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>PP(1)</th>
<th>PP(2)</th>
<th>PP(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESULTS</td>
<td>28</td>
<td>36</td>
<td>18</td>
<td>9</td>
</tr>
</tbody>
</table>

In view of this result we note that we obtain near $N$-way speedup ($N$ number of processors) at 0 granularity (when compared with to the PP scheduler running on one processor, not the uniprocessor version). We now deviate slightly and demonstrate how devastating the effects of false sharing can be.

**A note on false sharing**

We mentioned when we first introduced the SMP schedulers that we were taking care of false sharing by making use of cache line gaps between data that would be accessed by more than one processor. A particular case in point is the pointers to the current thread of each processor. It seems trivial to define the current thread pointer as an array of pointers to threads (the size of the array being equal to the number of processors). However, doing so might place all the current pointers onto the same cache line and every time we update one of the words, we need to update the entire cache line. We re-execute the context switch benchmark removing the cache line gaps between our current thread pointers. The current thread pointer is updated at every yield so the shared cache line needs to be updated regularly, and since this particular benchmark performs 10,000,000 yields and not much else it serves as a good indicator of a constantly changing shared cache line. Results are displayed below in nanoseconds:

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>PP(1)</th>
<th>PP(2)</th>
<th>PP(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Separate cache line</strong></td>
<td>28</td>
<td>36</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td><strong>Sharing the same cache line</strong></td>
<td>28</td>
<td>36</td>
<td>99</td>
<td>91</td>
</tr>
</tbody>
</table>

Results demonstrate a decrease in performance for the implementation that defines all current thread pointers on the same cache line, when running on two or more processors, compared with the version that separates the processor’s current pointers onto different cache lines. In particular we note a ten fold decrease in performance in the four processor test. We expect results to deteriorate further when the number of processors increases. We note also that there is no effect on the one processor version of the per processor scheduler.
Figure 5.3: PP scheduler, speedup results.
5.2.5 Analysis of results

The speedup benchmark demonstrates that the PP scheduler, when programmed correctly can produce excellent results and achieve speedup at a fraction of the granularity required by the shared run queue versions. This efficiency comes at a cost, since running a normal program (such as the original benchmarks) can cause gross imbalances in the scheduler’s workload, and thus performance degrades. The solution would be to provide scheduler migration across the different run queues. If we do this directly by either inserting or removing threads into/from other processors’ run queue at run time, we would need to provide access control for each run queue, since the run queues would become shared resources. The same problems we had with the shared run queues would come into play, except that the contention would be split amongst various run queues. The next two per processor run queue scheduler implementations attempt to provide a solution by employing migration techniques which reduce contention from shared resources as much as possible.

5.3 SMP batching

Our first migration strategy involves implementing a scheduling technique outlined by Vella [107]. Vella describes how to use batches to migrate groups of threads across processors. We first introduced batching in the uniprocessor chapter and demonstrated how batching could be put to good use to increase performance by means of cache conscious scheduling.

5.3.1 Introduction

The reader may recall that uniprocessor batching involved servicing a batch for a given number of dispatches. Each batch has a batch size counter, which represents the number of batches on the thread, and a dispatch counter which maintains the number of times the threads in the batch are dispatched. The batch size is not meant to exceed the scheduler’s global dispatch size and threads in a batch are dispatched until the dispatch counter meets a dispatch threshold. For better performance, the batch size is meant to be considerably smaller than the dispatch threshold. The SMP version described by Vella services batches in the same way as the uniprocessor batching scheduler, except each processor services it’s own individual batch. After a batch is serviced it is placed onto a shared migration batch queue. The shared batch queue is protected from concurrent access by means of a spin lock and acts just like the shared run queues described in the previous chapter, except for one important feature. The fact that a processor services batches over a considerable amount of time compared to the time taken to service an individual thread, means that the shared batch queue is accessed very infrequently. The coarser grain nature of batches (compared to individual threads) in the SMP case helps not only in improving locality due to cache affinity, but also in reducing false sharing (see Section 2.1.3) and reducing contention for the run queue. The shared run queue of batches helps improve load balancing since it follows the same rules outlined in the previous chapter, except in term of batches. In our case no processor can be idle when there is a batch on the shared batch queue.

The shared batch queue is implemented as a dummy head based queue to support fast enqueue and dequeue routines. For each processor we maintain a pointer to the current batch and an overflow batch. The overflow batch has the same function it had in the uniprocessor. When the batch size threshold is met threads are placed onto the overflow
Figure 5.4: SMP batch scheduling.
batch. However, in this case, when the overflow batch becomes full we place the overflow batch onto the shared batch queue. Figure 5.4 demonstrates the SMP batching scheduler. The spin lock that protects the shared batch queue is also used to protect the other internal scheduler structures. The lock granularity could be considered similar to that of the traditional scheduler (see Section 4.3).

5.3.2 Internal scheduler functions

We now discuss the internal scheduler function implementation for our SMP batching scheduler.

**Thread insert**

Our scheduler insert routine is a hybrid of the uniprocessor batching scheduler and the traditional scheduler thread insert routines. When inserting a thread, the scheduler first checks if the current batch size has met the batch size threshold, if it hasn’t, the thread is inserted onto the current batch, otherwise the scheduler attempts to insert the thread onto the overflow batch. The overflow batch’s size threshold must also be verified. If it hasn’t been met the thread is placed onto the overflow batch. If the overflow batch is full, the sleeper counter is verified for idle processors. If processors are idle, the overflow batch is migrated directly to the idle processor, otherwise it is placed onto the shared batch run queue. The scheduler’s spin lock is used only if the overflow batch is full, since only in this case does migration to other processors or the shared batch queue take place.

**Thread remove**

When removing a thread, the scheduler first checks if the thread is the last on the current batch. If it isn’t, it is simply removed. Otherwise the scheduler attempts to obtain a batch from the shared batch queue. The scheduler’s spin lock is used only when accessing the shared run queue. If there are no batches on the run queue, the scheduler attempts to run the overflow batch. If the overflow batch is also empty, the kernel thread representing the current processor is set to sleep.

**Thread yield**

The yield routine saves the context, increments the dispatch counter, and if the dispatch threshold has not yet been reached, jumps to the next thread on the current batch. If the batch threshold has indeed been reached, the current batch is inserted onto the shared batch queue and the next batch from the shared batch queue is obtained. The operations on the batch queue are protected by means of the scheduler’s spin lock.

5.3.3 Thread API functions

Since we have drastically reduced the contention on the shared resource (in this case the shared batch queue), this affects all the API functions. The yield function’s performance in particular begins to approach the performance of the PP scheduler’s yield function, except when using small batch and dispatch sizes (in which case performance becomes similar to that of the traditional schedulers). The functions that make use of insert and remove routines have a bit of overhead added to them with respect to the PP scheduler. We opt to
use the wait free algorithms first introduced for our non-blocking scheduler for the channel and barrier algorithms.

5.3.4 Results

We set about presenting results for our batching schedulers as we did with the SMP schedulers. We first present our context switch benchmark results followed by the speedup results.

Context switch performance

For our first test, we adapt our context switch benchmark (Algorithms 3.7) to conform with batching by increasing the number of threads depending on the processor size. In this way all experiments make use of the same amount of cache re-use. We set the batch size to 10 for all the processors and the dispatch counter threshold to 100. We allocate the same number of threads to each processor, so that the uniprocessor and the SMP batching on 1 processor schedule 20 threads, the SMP scheduler on 2 processors schedules 40 threads and the SMP scheduler on 4 processors schedules 80 threads. Results are displayed below in nanoseconds:

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>SMP batches(1)</th>
<th>SMP batches(2)</th>
<th>SMP batches(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>28</td>
<td>52</td>
<td>27</td>
<td>20</td>
</tr>
</tbody>
</table>

We rerun the context switch benchmark, this time by keeping the number of threads constant and setting the batch size to 50 and the maximum batch count to 1,500. The scheduled number of threads is set to 400, for this experiment. We display all results below, in nanoseconds:

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>SMP batches(1)</th>
<th>SMP batches(2)</th>
<th>SMP batches(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>49</td>
<td>54</td>
<td>28</td>
<td>23</td>
</tr>
</tbody>
</table>

Results demonstrate a notable speedup even in the case of four processors. Although not as impressive as the PP scheduler’s results, we note that we are running the same benchmark on which the PP scheduler performed poorly (not the benchmark designed purposely for the PP scheduler). Note the uniprocessor circular run queue scheduler’s decrease in performance is due to caching effects.

Finally we consider a worst case performance benchmark by which we set both the batch size and threshold to 1, thus making our shared batch queue equivalent to a shared run queue, but with added overheads. This result is useful, since it gives as an indication of what the performance would be like if we could adjust the size and dispatch thresholds dynamically at run-time. We schedule 20 threads for all experiments and present results below (nanoseconds):

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>SMP batches(1)</th>
<th>SMP batches(2)</th>
<th>SMP batches(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>28</td>
<td>164</td>
<td>955</td>
<td>1,527</td>
</tr>
</tbody>
</table>

We note that as expected the results are similar to that of the shared run queue SMP.

Speedup benchmark

For the following results we set the batch size to 50 and the dispatch threshold to 1,500. Our speedup benchmark adapts Algorithm 4.5 to execute 400 threads instead of 20. We present
the results in Figure 5.5. We note how the SMP batching scheduler seems to maintain the performance first seen in the case of the PP scheduler. The SMP batching scheduler performance begins to approach $N$-way speedup at the finest granularities.

5.3.5 Analysis of results

The SMP batching scheduler has helped consolidate the potential that batching based scheduling demonstrated in the uniprocessor case. Results show that except for the very finest of granularities the scheduler achieves $N$-way speedup. Moreover, unlike the PP scheduler, the SMP batching scheduler maintains a balanced workload, unless the size of the batches are too large. In terms of comparison with the traditional scheduler, the SMP batching scheduler would only lose out in the case of fine grain applications when there is a small number of threads, that are long-lived and have long dispatch times. A small number of threads would mean that all threads would fit into a single batch, thus not allowing the application to take advantage of all the available processors. Batch sizes that adapt might be a solution since in certain cases the scheduler would become very similar to the traditional scheduler (with batch and dispatch thresholds set to 1), however this would involve an extra overhead on the part of the scheduler’s run time system.

5.4 Wait free scheduling

The final scheduler is a per processor run queue scheduler that uses lock free data structures and algorithms to allow for scheduling and migration. Unlike the shared run queue lock free scheduler that used re-tries which made the algorithms non-blocking, all our algorithms used in this scheduler are wait free (see Section 2.2.3). We present new wait free algorithms for migrating threads across run queues and adopt the wait free inter-thread communication, barrier routines and thread insert routines introduced for the non-blocking scheduler in Section 4.5.

5.4.1 Introduction

The migration routines we are about to describe can be applied directly to threads and used on traditional per processor run queue schedulers (e.g. the PP scheduler we described in this chapter) however, if we use our wait free migration in this case, we would only migrate one thread at time and performance would suffer as a result. The loss in performance in this case is not due to contention of shared resources, but to the cost of having to enqueue and dequeue single threads from the run queues at a regular basis. We opt instead to migrate batches, just as we did with the SMP batching scheduler. Although performance is better than that of other schedulers when migrating one thread at a time, we will show how a batch migration strategy produces even better results (see Section 5.4.5). We will henceforth refer to this scheduler as the wait free scheduler.

While the SMP batching scheduler used a shared run queue to migrate batches, our wait free scheduler adopts a different approach. Our scheduling strategy involves maintaining a batch queue per processor, and a migration batch per processor. The use of the batch size remains the same as for the SMP batching scheduler, but the dispatch counter serves two purposes. Every time a batch’s dispatch counter expires, the scheduler attempts to run the next thread on the batch queue, and also attempts to migrate a batch. We could have maintained two counters which distinctly represent the above conditions, however a
Figure 5.5: SMP batching scheduler, speedup results.
Algorithm 5.2 Sender initiated migration.

\[
\begin{align*}
W1 & \text{ for (count = 0; count < PROCNUM; count++) } \\
W2 & \quad \text{ if (CAS(&sched.cur_batch[count],NULL,migration_batch))} \\
W3 & \quad \quad \text{ kernelthread_wakeup(count);} \\
W4 & \quad \text{ return;} \\
W5 & \}
\]

5.4.2 Wait free migration

We migrate batches over processors in one of two ways, sender initiated migration and receiver initiated migration. We implement three versions of the wait free scheduler. One that uses sender initiated migration, one that uses receiver initiated migration and another that uses both techniques.

Sender initiated migration

In sender initiated migration, it is the processor that has the extra workload that checks if another processor needs the batch. The scheduler attempts to migrate batches using this method when one of two things happen. The first state that triggers sender initiated migration occurs when an overflow batch is full. In this case the scheduler tries to migrate it, to another processor. If the operation doesn’t succeed the batch is placed onto the per processor batch queue. The second reason the scheduler attempts to migrate a batch is when the current batch’s dispatch counter expires and there is more than one batch on the processor’s batch queue.

In both cases the scheduler assumes another processor needs work only when it has gone to sleep and lies idle. If the scheduler finds an idle processor their current batch pointer is set to point to the excess batch and the associated kernel thread is awaken. Another solution (which we do not implement but however propose) that would further improve carefully chosen dispatch threshold, would meet the requirements and decrease the amount of overhead required to maintain two counters. The per processor batch queues are meant to maintain a high degree of cache re-use, since most threads will spend their entire execution time on the same processor. Batches will find their memory footprint in cache even on each thread’s first dispatch in more cases than when using the SMP batching scheduler. Figure 5.6 describes the wait free scheduler’s configuration. The use of the per processor batch queues, reduces the amount of shared data structures to two. The two data structures that can be accessed by more than one processor are each processor’s migration batch and the processor’s current batch. The current batch is accessed when attempting to wake up an idle processor. When the scheduler is aware of an idle processor and there is extra workload in the form of a batch, the scheduler cycles through the processors and if one of them has a current batch pointer which is NULL, the excess batch is swapped into it. This solution is similar to the scheduling routine we use for our non-blocking scheduler. Every processor also maintains a migration batch\(^1\) where they place any extra workload for other processors to access.

---

\(^1\)This depends on the wait free implementation we are using. The sender initiated implementation does not use migration batches.
Figure 5.6: Wait free scheduling.
Figure 5.7: Sender initiated migration. Notice how the second processor is idle. The first processor attempts to send over its own overflow batch.
Algorithm 5.3 Receiver initiated migration.

```
W1 int id = kernelthread_id();
W2 for (count = 0; count < PROCNUM; count++) {
W3   if (migration_batch = SWAP(NULL, &sched.mig_batch[count])) {
W4     sched.cur_batch[id] = migration_batch;
W5     scheduler_jmp(id);
W6   }
W7 }
```

load balancing would be to allow a processor to advertise that it is soon going to run out of work so some other processor could migrate a batch before the processor becomes idle. To check if other processors are idle we maintain a sleeper counter which we atomically decrement right before we wake up a kernel thread and atomically increment right before we set a kernel thread to sleep just as we did with the non-blocking scheduler. The use of the sleeper counter is optional, but it does improve performance since the scheduler does not need to cycle through all the processors every time it needs to migrate a batch. The sleeper counter in this case serves the same purpose as it did with the non-blocking scheduler (see Algorithm 4.8).

Algorithm 5.2 demonstrates the core of the sender initiated routine. When attempting to migrate a batch with a non-NULL sleeper counter, the scheduler cycles through all the processors and attempts to identify the idle processor (W1). We use the fact that when processors are idle their current batch pointer is NULL to check for sleeping processors. The operation is performed by means of an atomic compare and swap instruction, which checks if the current thread is NULL (W2), and only if it is, swaps the overflow batch into it. The routine then proceeds to wake up the associated kernel thread (W3).

Receiver initiated migration

The receiver initiated migration algorithm can be seen in Algorithm 5.3. In receiver initiated migration, when a processor runs out of threads to run, it looks around the migration batches of other processors (W2), and can take extra batches from there. From the receiver point of view, batches are migrated by swapping NULL into the migration batch and returning the old value of the migration batch into a temporary current, by means of an atomic swap operation (W3). A temporary current is used to ensure the current thread does not become NULL, which would give other processors the impression that the processor lies idle. If the returned value is non-NULL the migrated batch is executed (W4), otherwise the scheduler searches for batches on other processors, until either all the processors' migration batches have been checked and no work was found or a batch is acquired.

Batches that are to be migrated are placed onto the migration batch every time a dispatch counter expires and there is more than one batch on the processor's batch queue. The batch that was currently running is placed onto the migration batch if the migration batch is not NULL. The operation is performed atomically by means of a compare and swap instruction. If the migration batch is non-NULL it means that no processor needed any extra workload up to that point, so the batch is left there. For thread fairness sake the scheduler could be set to atomically swap the current batch with the migration batch so that the migration batch will be serviced as frequently as the other batches. The return result of the compare and swap operation is also important since it indicates wether to insert the
current batch back onto the per processor batch queue.

Hybrid migration

Hybrid migration attempts to combine the routines outlined above. The scheduler can use either sender initiated migration or receiver initiated migration depending on the case. When using receiver initiated migration only, there are cases when extra workload is placed onto a migration batch while some processors lie idle. It would be more efficient to use sender initiated migration in this case, and send the surplus batch directly to one of the idle processors. On the other hand when using sender initiated migration there will be times when a kernel thread is about to sleep since it run out of threads to execute, while some other processor has excess workload. The processor that has the extra work might just have checked if someone is idle, and found no one, and therefore didn’t send the batch over. In hybrid migration the scheduler decides on which migration routine to use by consulting the sleepers counter. When there are processors idle the scheduler attempts to migrate batches directly to them, on the other hand if no processors are idle the extra batch is set as a migration batch so that another processor can attempt to migrate it when it needs to.

5.4.3 Internal scheduler functions

We now describe the internal scheduler functions, which are very similar to those of the SMP batching scheduler.

Thread insert

The thread insert routine first attempts to insert the thread onto the current batch. If the batch size is greater than the maximum batch size, the scheduler attempts to insert the thread onto the overflow batch. When the overflow batch itself becomes full, the scheduler does one of two things. If there are any idle processors, the overflow batch is migrated to the idle processor and the associated kernel thread is awaken (see sender initiated migration, Section 5.4.2). Otherwise, there are no idle processors, so the overflow batch is inserted onto the processor’s batch queue.

Thread remove

When a thread terminates or is placed onto a communication structure, the scheduler checks if it is servicing the last thread on the current batch. If there are more threads on the batch, the current thread is removed and the next thread on the batch is executed. When the thread to be removed is the last thread, the scheduler checks if the current batch is the last on the processor’s batch queue. If it isn’t the next batch on the batch queue is serviced, otherwise there are no more batches on the batch queue and the overflow batch is consulted. If it exists the overflow batch is set as the processor’s current batch. Otherwise, the next step depends on the migration policy of the scheduler. When using sender initiated migration the kernel thread is set to sleep until some other processor sends it a batch. If the scheduler policy is one of either receiver initiated migration or hybrid migration the scheduler cycles through the other processor’s migration batches and attempts to migrate a batch over. Note the kernel thread also checks it’s own processor’s migration batch, because there might be a batch there.
Figure 5.8: Receiver initiated migration. The second processor is about to run out of work so it attempts to migrate the migration batch from the first processor.
Thread yield

The thread yield routine depends on the dispatch counter. The yield routine increments
the batch’s dispatch counter and executes the next thread on the batch until the dispatch
threshold is met. When this happens, the scheduler checks if there is more than one batch on
the processor’s batch queue. If the current batch is the only one running the dispatch counter
is reset and the same batch is serviced again. However, if there is more than one batch a
migration routine is attempted, depending on the migration policy the scheduler is using.
In all cases when batch migration is not successful the next batch on the processor’s batch
queue is serviced. In the case of sender initiated migration the sleeper counter is verified. If
there are idle processors, the current batch is migrated to one of the idle processors. In the
case of receiver initiated migration the current batch is removed from the processor’s batch
queue and if there is no migration batch it becomes the migration batch. If there already
is a migration batch the batch is placed back onto the processor’s batch queue. When
using the hybrid migration implementation, if any processors are idle, the sender initiated
migration routine is used and if none of the processors are idle the scheduler executes the
receiver initiated routine.

5.4.4 Thread API functions

The thread API calls have had a substantial overhead added to them, however as is the
case with the uniprocessor batching and SMP batching schedulers, the overhead is related
directly to dispatch counter. In terms of multiprocessor performance the drastic reduction
of contention improves performance notably and even the worst case performance (where we
set a maximum batch size of 1 and dispatch threshold of 1) produces satisfying results (see
Section 5.4.5). We also opt to use the wait free channel and barrier algorithms introduced in
for the non-blocking scheduler. By doing so we have a complete wait free thread scheduler
implementation.

5.4.5 Results

We adopt the same approach we did with the SMP batching scheduler to obtain our results.

Context switch performance

Our first result adapts the original context switch benchmark (Algorithm 3.7) to cater for
the number of processors. We will scheduling $20 \times N$ threads, where $N$ is the number of
processors. We fix the maximum batch size to 10 and the dispatch threshold to 1,000. We
run the results for all three versions of our migration algorithms. All results are presented
below in nanoseconds:

<table>
<thead>
<tr>
<th></th>
<th>UNI</th>
<th>Wait free(1)</th>
<th>Wait free(2)</th>
<th>Wait free(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender initiated</td>
<td>28</td>
<td>51</td>
<td>45</td>
<td>22</td>
</tr>
<tr>
<td>Receiver initiated</td>
<td>28</td>
<td>53</td>
<td>28</td>
<td>16</td>
</tr>
<tr>
<td>Hybrid</td>
<td>28</td>
<td>51</td>
<td>27</td>
<td>14</td>
</tr>
</tbody>
</table>

We re-execute the benchmark, fixing the number of scheduled threads to 400, the batch
size to 50 and the dispatch threshold to 1500. As before we present results for the three
different migration algorithms we have proposed (all results in nanoseconds):
The results provide evidence that the wait free is the fastest scheduler at the finest of granularities (except for the PP scheduler), providing near $N$-way speedup at 0 granularity. The 14ns average context switch performance, in the case of the four processor test, falls just short of a perfect 13ns.

The results also demonstrate the supremacy of the hybrid migration. The sender initiated migration slows down, since the kernel thread will set itself to sleep when it runs out of batches to run and, since it cannot initiate migration, will only be awakened when someone migrates a batch to it. Receiver initiated migration on the other hand can only get batches from the migration queues and if it falls asleep only once it cannot be awaken. When this happens we lose one of the kernel threads. Hybrid migration has the best of both worlds since it checks for possible migration queues before it sets itself to sleep and is can also be awaken by other kernel threads when they have extra workloads.

We re-run the test with a worst case performance (i.e. with a maximum batch size of 1 and dispatch threshold of 1). In view of the previous results, we perform this experiment only for the hybrid migration case. The number of scheduled threads is fixed to 20 (results in nanoseconds):

<table>
<thead>
<tr>
<th>Sender initiated</th>
<th>Wait free(1)</th>
<th>Wait free(2)</th>
<th>Wait free(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>53</td>
<td>33</td>
<td>18</td>
</tr>
<tr>
<td>Receiver initiated</td>
<td>28</td>
<td>53</td>
<td>26</td>
</tr>
<tr>
<td>Hybrid</td>
<td>28</td>
<td>53</td>
<td>26</td>
</tr>
</tbody>
</table>

The results show that as expected, the scheduler degrades much more gracefully than the SMP batching scheduler. The lack of contention allows us to achieve speedup even in the worst of cases.

In view of these results we attempt to run another benchmark, where we set the batch size equal to the dispatch threshold, so that each thread in a batch is dispatched only once before another batch is serviced. This condition allows thread fairness, instead of simply batch fairness. For this experiment we set the batch size to 50 and the batch count to 50. We schedule 400 threads (results in nanoseconds):

<table>
<thead>
<tr>
<th>UNI</th>
<th>Wait free(1)</th>
<th>Wait free(2)</th>
<th>Wait free(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>161</td>
<td>110</td>
<td>82</td>
</tr>
</tbody>
</table>

The results are not very far off from the results shown previously and demonstrate that wait free scheduling can support true thread fairness as demonstrated by the shared run queue schedulers while still maintaining good performance even at fine granularities.

**Speedup benchmark**

It is tempting to use the same type of batch sizes we used for the SMP batching scheduler in our speedup benchmark. However, the wait free scheduler design requires different needs. We demonstrate the disadvantages of using large dispatch sizes, as used in the case of the shared batch queue, as our first example. We explain why this is so and propose faster alternatives. As with did in the case of the SMP shared batch queue scheduler, we run our speedup benchmark (Algorithm 4.5) with a batch size of 50 and a dispatch threshold of 1500. We perform the results only in the case of the hybrid migration.
Figure 5.9: Wait free scheduler, speedup results.
The results are displayed in Figure 5.9. The results could be considered excellent for the finest granularities. However if one looks closely, at coarser granularities the results are somewhat disappointing. At a granularity of 5,000 the traditional SMP scheduler outperforms the wait free scheduler. We discuss the problem in further detail below.

5.4.6 Varying the batch size

We compare results of the batching scheduler, the wait free scheduler and the traditional shared run queue scheduler in Figure 5.10. The graph relates to speedup when running the schedulers on four processors. We note how the wait free scheduler, has the best performance at fine granularities, however performance decreases towards coarser granularities. At a granularity of 1,000 the benchmark has a speedup of 3.6 while the SMP batching scheduler at the same granularity obtains a speedup of 3.9. Moreover, at a granularity of 10,000 (not seen in graphs) the wait free scheduler still has a speedup of 3.6, while the SMP batching scheduler obtains a speedup of 3.95 and even the traditional scheduler outperforms the wait free scheduler since it achieves a speedup of 3.85. We believe the main reason for the performance degradation is due to load balancing. Since the benchmark creates all threads from the first processor, we only pass one batch at a time to the other batch queues. For our benchmark of 400 threads and a batch size of 50 threads, we have 8 batches, of these 3 would run to completion on separate processors while the first processor would have to handle the remaining 5 batches. When the other batches finish, the first processor’s batch queue migrates another 3 batches over to other processors, however it is still left with 2 batches to run. At fine granularities this can prove to be an advantage, however at coarse granularities this problem could prove costly, since for a considerable amount of time only one out of the four processors is being used. Our suspicion is confirmed when we re-run the speedup test using 350 threads instead of 400. In this way we achieve a correct load balance even towards the end. The net result of the operation is a speedup of 3.91 at a granularity of 1,000 and a speedup of 3.95 at a granularity of 10,000. We note that the results of the SMP batching scheduler when using 350 threads instead of 400 threads remains the same.

By setting the number of threads to 350 we have identified the problem. The solution lies in decreasing the batch size and dispatch count to allow for better balancing. We re-run the entire set of results by setting the batch size to 10 and the dispatch threshold to 100 and another experiment where the batch size is 10 and the dispatch threshold is 10 (to support thread fairness). We also add the worst case performance for completeness sake. Results are displayed in Figure 5.11. All results of the wait free schedulers are calculated using the four processor version. We note that the best results at fine grain are due to the first set of results, however the experiment when running with batch sizes of 10 and dispatch threshold of 100 performs very well at fine granularities and outperforms the other two experiments when using coarser granularities. The experiment using a batch size of 10 and dispatch count of 10 does not perform as well as the others at fine granularities, however it catches up with the others and outperforms the first experiment at coarser granularities. The worst case performance of the wait free scheduler is good enough to outperform the traditional scheduler. We also compare the results with the SMP batching scheduler in Figure 5.12. Notice how at finer granularities we achieve speedup faster and that at coarser granularities the performance is similar.
Figure 5.10: Performance comparison from various schedulers.
CHAPTER 5. PER PROCESSOR RUN QUEUE SMP SCHEDULING

Figure 5.11: Comparison of different batch sizes for wait free scheduling.

5.4.7 Scalability

One important issue we have not yet addressed is the wait free’s aptitude to scalability. While the shared run queue schedulers and the SMP batching scheduler scale naturally, the same cannot be said for the wait free scheduler. On machines of up to 32 processors cycling through other processor’s migration batches is not a viable solution.

A simple solution could be to cycle through a number of processors only (e.g. the four adjacent processors) instead of the entire set, however this could cause a certain amount of load imbalance. Another alternative is reducing the number of migration batches. Up until now we have considered each migration batch directly belonging to every processor. The migration batches can be abstracted away from belonging to a particular processor and can be viewed as a separate array of batches, similar to the SMP batching scheduler’s shared batch queue. This idea is illustrated in Figure 5.13. We could then apply the same wait free migration algorithms onto the array. The migration batch array size would be less than the number of processors. A 16 processor machine might have a migration array of 4 batches. Another counter could be introduced to advertise how many batches are currently available on the shared batch queue. The counter would be updated atomically.

When a processor intends to place a batch onto the migration array it would first check how many batches are already in the array, if the migration batch if full, it would not need to migrate, otherwise it would cycle through the array and replace one of the empty slots with the batch it intended to migrate (using an atomic compare and swap instruction). For larger migration batch arrays an alternative to cycling through the entire array would be to associate a set of processors to a single migration batch. In this way a processor checks and updates only one migration batch.
Figure 5.12: Comparison of different batch sizes for wait free scheduling.
Figure 5.13: Scaling the wait free scheduler. Using a migration batch array of size 4, for an 8 processor machine.
When a processor requires work, it can check the migration batches counter and if it is greater than 0 it would cycle through the migration batch array and should (not necessarily) find a batch. As before for large migration batch arrays we could associate a batch with a set of processors, and opt to check the associated migration batch then look for work in other batches. A multiprocessor machine that sports 16 processors using a migration array of 4 processors, would incur the same overhead associated with our 4 processor version, except that migrating a batch might at most require 4 checks. Although we have not implemented such a scheduler, the workload should remain balanced throughout, and performance should be maintained. The use of the migration batch counter helps to reduce the amount of times we would check the array uselessly.

5.4.8 Analysis of results

Results from the speedup benchmark, show that our wait free scheduler is the fastest of all the SMP schedulers since it obtains speedup at the finest of granularities. The reduction of contention is evident, since the worst case results perform better than the traditional scheduler’s results. The problem with load balancing is a minor one, and we have seen that this can be solved by carefully setting the dispatch threshold size, or by means of automatic dispatch control. We hope the scalability issue could be dealt with by means of the techniques we have further outlined. The wait free scheduler’s advantages are not limited to the performance witnessed in the results, but the wait free nature of the scheduler and communication constructs ensure that all operations on the scheduler and communications structures will be completed in finite time. As is the case with the non-blocking scheduler the wait free scheduler can provide better fault tolerance and avoid deadlock. Since the entire scheduler synchronisation is wait free, we also avoid starvation and we can calculate an upper bound on the number of possible instructions the scheduler will perform at run time.

5.5 Conclusion

We compare and contrast the wait free scheduler, the SMP batching scheduler and the traditional scheduler in Figure 5.14. Scaled out and in versions of the same graph can be seen in Figure 5.15. We note how long the traditional scheduler takes (in terms of granularity) to achieve optimal speedup, whereas the wait free scheduler, achieves speedup of 2.19 at a granularity of 10 and over 3.21 at a granularity of 50. The SMP batching scheduler follows closely behind. The results demonstrate the net superiority of the per processor schedulers compared to the shared run queue schedulers at fine granularities. Notwithstanding the PP scheduler’s results which can be misleading, when combined with a strong migration policy, per processor run queue schedulers are definitely preferable to shared run queue schedulers for fine grain applications. In terms of comparison amongst themselves, we choose to discard the simple per processor run queue since it performs too poorly unless we statically map threads to processors, which is not always possible in dynamic environments.

If one problem where to be addressed at the SMP batching scheduler and the wait free scheduler, it would be the very fact that they use batches. Batches do not take advantage of the multiple processors when executing applications which launch very few (less than the batch size) long-lived threads that have long dispatch times, since the threads would be placed into one single batch and not all the processors would be used. A run time system
that varies the batch size directly according to the number of threads should provide a solution. In this case the worst case performance of the thread schedulers does become an issue. We note that the above problem does not apply in the case of the schedulers using a maximum batch size of 1 and dispatch count of 1.

On concluding we have seen that, the SMP batching scheduler performs very well and at worst performs very similar to a shared run queue scheduler, its main disadvantage is that thread fairness is not maintained. The wait free scheduler outperforms all other schedulers at fine granularity and even at its worst outperforms many of the SMP schedulers, thus making it the scheduler of choice for many applications. Moreover, the wait free property of the scheduler is in itself an advantage.

Figure 5.14: Comparison of speedup amongst different schedulers.
Figure 5.15: Comparison of speedup amongst different schedulers. Scaled in and out.
Chapter 6

Conclusion

We conclude the document by reviewing the major contributions of the project and highlight the advantages and disadvantages of the uniprocessor and SMP schedulers. In the SMP case we focus on the distinction between the shared run queue schedulers and the per processor run queue schedulers. In Section 6.2 we summarise the major contributions of our work. We then discuss possible future directions from simple tasks like extending the API to the impact of non-trivial changes that would result from adding extra features, such as priorities to the scheduler, and the possibility of adapting our schedulers to embedded systems or distributed systems.

6.1 Discussion

In this section we discuss the individual merits of all our schedulers in terms of the strategies employed. In the uniprocessor scheduler’s case we also highlight the problems and solutions we discussed related to hardware and operating system interaction. We discuss the SMP schedulers in terms of the insights we gained by varying the inter-thread communication implementations, and the distinguishing factors between shared run queue and per processor run queue scheduling.

6.1.1 Uniprocessor scheduling

Our uniprocessor schedulers served mainly to confirm existing ideas. We presented three uniprocessor implementations, that schedule threads at the user level and and try to do without any kernel interaction. We have seen that though we do not explicitly make use of the kernel we are still bound by the constraints of the system we are running our scheduler on. The hardware system directly affects our schedulers through caching. At fine grunularities performance can degrade drastically due to cache misses. We presented methods that detect the problem and a scheduling strategy that alleviates the problem by improving cache re-use. The hardware/software interaction in terms of page faults can affect performance. We demonstrate how to reduce page faults by making use of physical memory directly or by re-cycling thread data which is usually already stored in physical memory. The operating system can also add overhead in terms of the occasional system call to allocate memory. We considered the use of static memory allocation, however this solution still suffers from problems related to page faults. Instead we suggest the use of a memory management system which would pre-allocate a certain amount of memory from where we
can allocate memory for threads and stacks. We can also ensure the allocated chunk is present in physical memory by setting a bit on every page.

Our uniprocessor schedulers differed mainly from each other in terms of their run queue implementations. We used the MESH scheduler as the basis of the circular run queue scheduler. The results demonstrated fast context switch performance. It would be ideally used in conjunction with a pre-processor for automated yield insertions, and provides the fastest response times. We adopted a structure more commonly used for concurrent queues for our second uniprocessor scheduler. The dummy head scheduler is ideal for CSP scheduling and for applications that use short-lived threads that do not yield frequently. The uniprocessor batching scheduler confirms previous results [107], demonstrating that cache affinity scheduling improves performance for fine grain applications on uniprocessors.

6.1.2 SMP scheduling

Our SMP research focused mainly on trying to implement the fastest scheduling strategy for fine grain applications. We were mainly concerned with gaining speedup at the finest granularity and with finding the most efficient method of implementing already existing communication constructs.

Inter-thread communication

In Chapter 4 we compared and contrasted the possible implementations of inter-thread communication. We showed how when scheduling at fine granularities, the finer grain the locks are, the better the performance was. The absolute removal of locks in terms of wait free structures, proved to be the ideal solution. The wait free implementations not only outperformed the spin lock versions, but the very fact that they are wait free means they remove deadlock, improve fault tolerance, are immune to the priority inversion problem and avoid starvation. The fine grain spin lock versions of the structures also require an extra word in the data structures to cater for the lock, the wait free implementations do not need the extra word and therefore the multiprocessor and uniprocessor versions can share the same implementations.

We also showed how the performance of communication could be directly related to the run queue. The dummy head uniprocessor scheduler first demonstrated the situation by having a run queue that simply performs enqueue and dequeue routines faster than the other schedulers. In the SMP case we took the idea one step further. Both the dual lock and the non-blocking scheduler could perform concurrent enqueue and dequeue routines, as a result of which the performance of our inter-thread communication improved substantially.

Shared run queue SMP scheduling

We implemented and discussed five shared run queue schedulers. The distinguishing factor among this set of schedulers was the method of synchronisation employed for the shared run queue and for communication. We were mainly concerned in discovering which implementation would achieve the best speedup at the finest granularities. Other objectives where also important, namely the performance of inter-thread communication.

In terms of the individual schedulers, the global lock thread scheduler would be best suited for scheduling only a small number of threads. The traditional scheduler performed best in terms of speedup and still seems to be the best alternative from the shared run queue schedulers, at least up until four processors. The overhead associated with fine grain
scheduling, in the single lock scheduler, was only useful for inter-thread communication, since it is not worth relinquishing a lock and attempting to obtain the same lock again just to perform a handful of instructions. The dual lock scheduler demonstrated its use for handling inter-thread communication. The improvement from two to four processors hints at the potential of outperforming the traditional scheduler for five or more processors. This fact cannot be underestimated. The dual lock scheduler could well and truly become a substitute to the traditional scheduler. The non-blocking scheduler was disappointing in terms of performance. The possibility of relative improvement when the number of processors increases, although not evident up to four processors, could indeed be the case given Michael and Scott's [77] results. At this stage combined with the advantages of lock free structures the non-blocking scheduler would be an ideal solution.

On the whole, while benchmarks show that the shared run queue SMP schedulers do achieve a substantial amount of speedup for most applications, this does not hold true for fine grain applications. The main reason for the poor performance at fine granularities is high contention for shared resources, in particular for the shared run queue. Moreover, due to the single shared bus nature of SMPs the situation deteriorates as the number of processors increases. Given the situation it is just not feasible to schedule threads on SMPs that perform only a handful of instructions before relinquishing scheduler control.

Per processor run queue SMP scheduling

The natural structure of per processor run queue schedulers, in terms of both locality and lack of contention from shared resources are the two properties that fuel the difference in performance when compared to the shared run queue schedulers. Although these two properties are not enough, as the PP scheduler demonstrates. Load balancing is a factor which is just as important as locality and lack of contention. Apart from statically mapping threads to processors at compile time (and this will not always suffice), dynamic migration strategies have to be used to ensure a balanced workload. Correctly load balancing a workload by means of migration in real time is no trivial task, however both the SMP batching scheduler and the wait free scheduler, demonstrate how to maintain a correct load balance while still maintaining a strong adherence to locality and maintaining contention as low as possible.

The PP scheduler demonstrated the per processor run queue scheduler's potential in terms of fine grain thread scheduling. We showed that for well crafted applications, this scheduler could outclass all the others. The lack of any sort of run time load balancing mechanism however does not make this scheduler a serious contender. The PP scheduler is ideal in certain cases, when a programmer knows what to expect from his application. The SMP batching scheduler tries to use the advantages of the shared run queue schedulers in terms of load balancing and of the per processor run queue schedulers in terms of locality and reduced contention. The nature of batching reduces the contention (for large batches) and although the threads are migrated regularly, since batches run for a considerable number of dispatches, locality is also exploited. Load balancing is maintained due to the shared batch queue. The main disadvantage that could be levelled at the SMP batching scheduler is in the use of batches, which could fail to maintain thread fairness (although batch fairness is maintained) and for when the total number of scheduled threads is less than the batch size. The wait free scheduler maintains locality even further due to its per processor batch queues. Moreover, locality is maintained even for small batch sizes. The wait free migration not only balances workload well but reduces most contention. The scheduler can perform
well even when dealing with small batch sizes and even without the use of batches at all. The fact that the entire scheduler structure is wait free ensures deadlock freedom, strong fault tolerance and no starvation. The wait free thread scheduler is perhaps the strongest pick of the schedulers due to its flexibility and strong performance.

6.2 Summary of major contributions

We hereby enlist the major contributions of our work:

- **Uniprocessor schedulers:**
  - The circular run queue scheduler has one of the fastest context switch times of all the schedulers we are aware of on Intel-based machines to date.
  - We use a novel dummy-head run queue for our second uniprocessor scheduler. Results demonstrate that the dummy-head queue outperforms other queues in terms of enqueue and dequeue operations. This scheduler is particularly useful for applications in which threads communicate frequently with each other.
  - The uniprocessor batching scheduler confirms Vella’s results [107] that cache-consistency scheduling can improve performance even on uniprocessor machines.

- **SMP schedulers:**
  - Our shared run queue scheduler based on the designs of Vella’s [108] and Cor-dina’s [25] schedulers is believed to be one of the fastest of its kind on Intel architecture.
  - We adapt Michael and Scott’s dual lock concurrent queue [79] as a novel idea for a shared run queue which can perform concurrent enqueues and dequeues thus improving the performance of thread communication. Furthermore, we predict that for SMPs which have more than eight processors this could be the fastest shared run queue SMP thread scheduler.
  - By adopting Michael and Scott’s [79] non-blocking queue, we implement the first entirely non-blocking fine grained thread scheduler.
  - We demonstrate that fine grain critical sections have the edge on coarser grain mutual exclusion when handling both communication and internal scheduler data structures. Moreover, we conclude that wait free synchronisation is to be preferred over mutual exclusion.
  - Our first per processor scheduler which uses static scheduling has very fast thread management due to the low latency achieved from removing thread migration. This scheduler was mainly implemented for testing purposes but is the fastest SMP scheduler for specific applications.
  - As a solution to maintain the principle of locality and reduce contention we introduce the SMP batching scheduler which demonstrates PP run queue schedulers can maintain load balancing and outperform their shared run queue equivalents.
  - The wait free scheduler is the world’s first scheduler that makes use of entirely wait free data structures. We note how the wait free scheduler’s structures and algorithms coupled with batching outperforms all other schedulers in our experiments.
By comparing our suite of SMP schedulers, we conclude that PP run queue scheduling is usually a better choice for fine grained multithreading than schedulers that make use of a shared run queue.

6.3 Future directions

While we have not completely exhausted the possibilities of scheduling that we have set out to survey, future work could be directed towards new paths of investigation for our schedulers, such as priority based scheduling, object-affinity scheduling, preemption, extending the API to include timer events and new communication constructs, and broadening our horizons to surveying thread scheduling on other mediums such as distributed systems and embedded systems.

6.3.1 Extending the scheduler API

The most obvious possible additions are directly related to improving the API. Certain constructs could be added to further enhance the thread scheduling possibilities. Adding constructs to the uniprocessor schedulers and to the SMP schedulers using locks is a trivial matter. However, for the SMP lock free schedulers certain choices would have to be taken to maintain the non-blocking and wait free properties. Moreover, as our research has shown that the wait free implementations should outperform their lock based counterparts.

An immediate addition to the API routines would be an alternative channel input for the SMP implementations. In view of the results of dedicated channel communications we would directly implement Vella’s [108] wait free alternative input algorithm. Since semaphores use queues, the implementation of wait free semaphores would not be trivial, the use of the counter might make a non-blocking version also hard to implement. Binary semaphores however could be easily implemented and would use the same code as the dedicated channel input and output, without the message passing. Message queues could be implemented easily as non-blocking queues using Michael and Scott’s [77] algorithm. Our current research, although still speculative, indicates the potential of a wait free implementation, stemming from the fact that the message queues rely on multiple enqueuers but only one dequeuer.

The batch based schedulers could have extensions to their API’s whereby application programmers can manipulate batches in the same way they manipulate threads. They can create new batches, place them on run queues, and in particularly directly specify which threads are placed in a given batch. If the programmer is made aware of the advantages of batching, he/she will be more likely to know which threads would benefit from being grouped together.

6.3.2 Object-affinity scheduling through batches

Object oriented parallel programming uses a philosophy common to both multithreading and object oriented programming (OOP). The use of OOP for multithreading can help maintain an abstraction from the concept of a thread, providing concurrent objects that have complex details encapsulated [24]. Moreover, in the case of thread schedulers, which are developed as libraries for existing languages, OOP’s encapsulation and hierarchical model provides mechanisms, which can ensure a certain amount of security for the scheduler.
Objects could be used as the criteria for grouping threads into batches. Threads would represent the object’s methods and a batch could be composed of all the methods of an object. The rationale is that the threads representing member methods will very often be accessing the same data, i.e. the object’s member data. Object-affinity thread scheduling has been already researched in a paper by Fowler and Kontothanassis [39], which yielded positive results. However, their approach involved making sure an object’s member functions execute always on the same processor. The batch based approach would involve member functions being scheduled contiguously, which should improve performance even in the uniprocessor case.

6.3.3 Scheduler extensions

The use of a pre-compiler would be useful in a number of cases. If the pre-compiler could foretell the maximum number of threads that would be used, memory management would benefit enormously. Moreover, a pre-compiler could implement static preemption, and might be able to determine which scheduler to link to and find an ideal batch size for the batch based schedulers.

Another system improvement would be implementing non-blocking system calls into the scheduler. We would ensure that when a thread blocks in the kernel another thread is allowed to execute until the blocked thread unblocks. This extension is being added to some of the smash implementations at the time of writing, by using the Linux version of scheduler activations [32].

6.3.4 Changing scheduler requirements

We mentioned in the introductory chapter that we would focus all our attention on priority less, non-preemptive thread scheduling. Our schedulers achieve their best performance using these properties. The issues of preemption and priorities would most definitely introduce new dimensions into the multithreaded applications, and new scheduler strategies would need to be designed to try and best make use of these properties.

Preemption at the user level would require kernel interaction to implement, since the user does not have access to timer interrupts. Vella [107] describes how to implement preemption at the user level using the \texttt{SIGALARM} signal. If we want to maintain performance this solution is not viable and it would require more rigorous protection for internal scheduler structures from concurrent access, in particular for the uniprocessor schedulers, which would further affect performance.

Priority based scheduling can however conform to our user-level ideology. Priority based scheduling requires that a thread of higher priority runs to completion in preference to threads of lower priorities. The thread scheduler we base smash on, MESH successfully implements 32 priority levels where each priority level is implemented as a separate run queue. At the uniprocessor level, implementing priorities onto the circular and dummy head run queues would follow the same path as with MESH, however uniprocessor batching based scheduling is another issue. Since priority based scheduling imposes that higher priority threads would run before the others, some batches would be composed of a single thread, since only it would be in the same priority level. In the SMP case, the shared run queue schedulers are more adapt at coping with priority based scheduling than the per processor run queue schedulers. In the case of the shared run queue implementations we would follow the uniprocessor model of having one shared run queue per priority level.
same fact would also decrease contention on the shared run queue for certain applications. In the per processor run queue scheduler’s case having a per processor run queue for each priority level would not be sufficient, information related to other processor’s run queues would need to be checked. Moreover, threads might have to be migrated more frequently. Priority based scheduling might indeed swing the performance balance towards the shared run queue implementations. Another interesting facet related to priority based scheduling is the priority inversion problem [26]. When a thread of higher priority is blocked waiting for a thread at a lower priority to unblock and a third thread is scheduled, which is of medium priority, the medium priority thread will run to completion before the high priority thread is serviced. There are various solutions to this problem [33], however non-blocking and wait free structures naturally solve the problem since they never block or busy wait. The priority issue could indeed create a new set of scheduling strategies on par to what we have described in this document.

6.3.5 Thread scheduling on other systems

The fact that most of our schedulers are written in C makes them very easy to port onto other platforms. This holds true in particular for the uniprocessor schedulers, since we schedule without any operating system involvement. The SMP schedulers use both the Linux only clone() call and UNIX semaphores. Other operating systems do provide mechanisms similar to Linux’s clone() and most provide semaphores at the kernel level. If no semaphores are available we could still choose to busy wait.

Embedded systems offer the possibilities of interacting directly with the hardware. Our user-level thread schedulers could take the place of an embedded operating system under certain circumstances. Moreover, direct access to the hardware would provide the opportunity to implement preemption without any overheads. Some embedded systems, such as the Alteon AEnic [2], also offer multiprocessor support. We could implement SMP versions of our schedulers on them, however the lack of certain atomic primitives would make non-blocking and wait free scheduling a remote possibility.

A natural next step for our schedulers would be in the field of distributed systems. A similar concept to what Vella [107] did with the KRoC/occam model could be envisaged, where thread scheduling can occur seamlessly across a large number of diverse workstations. Moreover, user-level communication could be used as was done with MESH [18]. An alternative to using message passing as used in MESH would be to use distributed shared memory [59], which attempts to provide the same features familiar to multiprocessors on loosely coupled systems. Our SMP scheduler implementations could be modified to adapt to the distributed architectures, however, many of the issues we discussed would have to be revisited. In particular, the issues of load balancing and locality would have to be discussed again in the distributed systems case, due to the high costs of migration. Generally on distributed systems locality is preferred to load balancing [35].

6.4 Closing note

We have carried out a comprehensive survey for priority less, non-preemptive thread scheduling on uniprocessors and SMPs. Our scheduling strategies have explored a tiny part of the vast plethora of possible scheduler architectures. However, we feel the selection was sufficient to review and provide solutions for some problems commonly attributed to fine grain thread scheduling.
In concluding we remark that although we championed a scheduler for general fine grain thread scheduling, in the form of the wait free scheduler, we have seen that different schedulers perform better than others in certain cases, and if we were to include more features we would indeed end up with further scheduling strategies for different problems, or to paraphrase an old adage, a different tool for every job.
Bibliography


[23] CERN homepage, WWW. URL: [http://www.cern.ch](http://www.cern.ch)


[55] Intel Pentium III Xeon Homepage, WWW. URL: http://developers.intel.com


[63] Linux psets homepage, WWW. URL: http://isunix.it.ilstu.edu/thockin/pset

[64] Linux bigphysarea homepage, WWW. URL: http://www.uni-paderborn.de/fachbereich/AG/heiss/linux/bigphysarea.html


[86] QNX homepage, WWW. URL: http://www.qnx.com


[99] ThreadX homepage, WWW. URL: http://www.threadx.com


[111] WOTUG homepage, WWW. URL: http://wotug.ukc.ac.uk