A Summary of Research in System Software and Concurrency at the University of Malta: Multithreading

Kevin Vella

Department of Computer Science and AI,
University of Malta

1 Introduction

Multithreading has emerged as a leading paradigm for the development of applications with demanding performance requirements. This can be attributed to the benefits that are reaped through the overlapping of I/O with computation and the added bonus of speedup when multiprocessors are employed. However, the use of multithreading brings with it new challenges. Cache utilisation is often very poor in multithreaded applications, due to the loss of data access locality incurred by frequent context switching. This problem is compounded on shared memory multiprocessors when dynamic load balancing is introduced, as thread migration also disrupts cache content. Moreover, contention for shared data within a thread scheduler for shared memory multiprocessors has an adverse effect on efficiency when handling fine grained threads.

Over the past few years, the System Software Research Group at the University of Malta has conducted research into the effective design of user-level thread schedulers, identifying several weaknesses in conventional designs and subsequently proposing a radical overhaul of the status quo to overcome these deficiencies. Various results have been published in academic conferences and journals [1–4]; this brief report highlights the principal findings. The related problem of communication and I/O bottlenecks in multithreaded systems and contemporary computer systems in general is discussed elsewhere in these proceedings [5].

2 The Old Testament

Many multithreaded environments utilise kernel-level threads as a lightweight alternative to heavyweight operating system processes. Kernel-level threads share a single address space to enable sharing of data and to minimise thread creation and context switching times. Kernel entry is still required when threads are created, whenever they communicate or synchronise, and at every context switch. Furthermore, the threading model is implemented within the kernel and is therefore dictated by the kernel. Systems such as Microsoft .NET make direct use of kernel threads as a vehicle for driving multithreaded applications.

The expense and inflexiblity of kernel involvement can be eliminated through the use of user-level threads, which operate entirely at the user level. Thread operation times as low as a few tens of nanoseconds can be achieved using user-level threads. Since the kernel is oblivious to user-level threads, a two-level thread hierarchy, with several user-level threads riding over a small pool of kernel-level threads, is employed to gain access to multiple processors and other kernel resources. This model, which is increasing in popularity, is utilised in this research, as it provides sufficient flexibility for experimentation with alternative scheduling strategies and yields by far the fastest implementations.

In this section we mention efforts at implementing uniprocessor and multiprocessor user-level thread schedulers based around the traditional single shared run queue approach.

2.1 Uniprocessor Thread Scheduling

Most uniprocessor user-level thread schedulers in existence utilise a single FIFO queue to hold descriptors for runnable threads. Cooperative round robin scheduling is used, with threads being descheduled whenever inter-thread communication or synchronisation occurs, or when an explicit 'yield' operation is invoked (of course, such descheduling points may be inserted automatically by a preprocessor). Thread operation times can be as low as 20 nanoseconds, particularly when the compiler passes register usage information to the scheduler to restrict thread state saving overhead. Examples of this genre include our Smash [6], Kent's KRoC [7] and CERN's MESH [8].

The problem with such a naive scheduling strategy is that frequent context switching (a characteristic of fine grained threads) disrupts the operation of the locality principle, on which cache hits depend. The expense of repopulating the processor's cache with a newly dispatched thread's footprint becomes significant when viewed in relation to the shortened thread dispatch time. To make matters worse, an individual thread is unlikely to accumulate a significant cache footprint by itself: only when threads are considered in groups can a long term cache footprint be identified.

2.2 Multiprocessor Thread Scheduling

The obvious way of extending a scheduler to operate on a shared memory multiprocessor is to wrap a monolithic lock around the scheduler's entry and exit points to protect against corruption arising from concurrent access to the scheduler's internal data structures. Through further refinement, the sizes of the critical sections may be reduced, and independent internal structures may be guarded by separate, fine-tuned locks to reduce contention. It emerges that the shared run queue is the data structure that will experience most contention as the multiple processors compete to fetch the next thread to execute. The locking method used must exhibit low latency, thus excluding the use of kernel-assisted locks such as semaphores. Usually a spin-lock variant designed to minimise memory bus traffic is used. SMP-KRoC [2] was implemented in 1998 following this design. Subsequently, work undertaken at the University of Malta adapted CERN's MESH [8] in a similar fashion to produce SMP-MESH [9]. A variant of our scheduler, Shared-Smash [6], also follows this design. We daresay that most multiprocessor user-level thread schedulers that balance thread load across processors operate on the same lines.

While schedulers adopting this approach exhibit perfect balancing of thread load across processors, the shared run queue causes threads to be indiscriminately migrated to processors on which they would not have recently executed. As a consequence, newly dispatched threads are unlikely to find any of their data footprint in the local cache. Moreover, when fine grained threads are being executed and the thread dispatch frequency is high, contention amongst the processors for access to the shared run queue inevitably spirals, despite efforts to minimise the critical section size.

3 The New Testament

Having glanced at what may be considered to be 'traditional' thread scheduler design and identified the following shortcomings:

- poor cache utilisation on uniprocessors;
- worse cache utilisation on shared memory multiprocessors; and
- high levels of contention for the run queue on shared memory multiprocessors;

we now provide a brief overview of our proposed solutions.

3.1 Thread Batching

Thread batching, first proposed in [10], involves grouping threads together into coarser grain entities termed batches. In batch schedulers, the scheduled entity is a batch of threads rather than an individual thread. A processor obtains one batch at a time from a batch pool and services the threads on the batch for a fixed number of thread dispatches, before depositing the batch back on the batch pool and acquiring the next batch. If the threads within a batch are scheduled repeatedly within the same batch dispatch and the combined memory footprint of the threads in the batch fits within the processor's cache, cache exploitation will naturally improve. In practice, this approach regains some of the data access locality which multithreading disrupts in the first place, even on uniprocessors.

On shared memory multiprocessors, batching algorithms yield even better results. Since the dispatch time of an entire batch is large when compared to the dispatch time of an individual thread, sufficient cache reuse is carried out within a single batch dispatch to dwarf the cache-related expense of thread or batch migration across processors. Moreover, if processors fetch batches from a shared run queue, the relatively large batch dispatch time reduces the frequency of access to this shared data structure, thus reducing contention for it while maintaining a balanced batch workload.

Batching can also be used to decrease the incidence of false sharing, since threads accessing data in a common cache line may be batched together. Moreover, when balancing load across processors, migrating threads in batches (whether in a shared batch run queue environment or otherwise) reduces the contention that arises when migrating multitudes of individual threads.

Batching gently nudges the memory access patterns of multithreaded applications to fit into a more sensible regime. It also coarsens the granularity of real concurrency in a manner which is dynamic and totally transparent to the developer.

Uniprocessor thread batching was first implemented in an experimental version of uniprocessor KRoC [10], with encouraging results: performance of fine grained multithreaded applications was improved by as much as 28%. As the gap between processor and memory speeds grew, this figure was only bound to improve. In fact, two years later, Unibatch-Smash registered improvements of up to 100%. On shared memory multiprocessors, additional benefits were made apparent using SmpBatch-Smash, a shared run queue multiprocessor batch scheduler. Various other scheduling arrangements utilising batching were implemented with positive results. Detailed accounts of the inner workings and performance of the thread batch schedulers implemented at the University of Malta may be found in [6, 4, 1].

3.2 Appling Lock-free Data Structures and Algorithms

Traditionally, access control to concurrent data structures relies on the use of locks. An oft-overlooked alternative method of synchronisation is available through the considered use of lock-free structures and algorithms, which dispense with the serialisation of concurrent tasks. Lock-free data structures rely on powerful hardware atomic primitives and careful ordering of instructions to protect them from unsafe concurrent access.

Valois [11] discusses lock-free techniques in detail and supplies various definitions of relevance. Lock-free data structures may have a further two properties: they may be non-blocking and wait free. A lock-free data structure is termed non-blocking if some operation on it is guaranteed to complete in finite time. When used for thread scheduling and inter-thread synchronisation, non-blocking data structures have other advantages, including stronger fault tolerance, deadlock freedom, removal of the extended spinning problem on multiprogrammed multiprocessors, and in priority-based schedulers, elimination of priority inversion within the scheduler routines. Unfortunately,

the non-blocking algorithms usually rely on retries to recover from unexpected alterations performed concurrently by other processors. This can result in unpredictable delays and starvation under high contention. Furthermore, the use of COMPARE-AND-SWAP in most of the algorithms brings about the *ABA* problem [11], which necessitates complex support for memory management to avoid it.

If every operation on the data structure is guaranteed to complete in a fixed number of operations the structure is said to be wait-free. Wait-free data structures, as discussed by Herlihy [12], do not suffer from the *ABA* problem and do not ever require retries. As a consequence, starvation is eliminated and the maximum number of instructions executed in the algorithms is fixed at compile-time.

Lock-free algorithms and data structures are well documented in academia, but an account of the application of this technique in a scheduler is hard to come by. We were unable to locate publications describing a wait-free scheduler implementation other than our own effort. The use of wait-free techniques within Wf-Smash [3, 1], a multiprocessor thread scheduler designed and implemented locally, removes all of the locks and critical sections that protect internal data structures. This scheduler also utilises thread batches to coarsen the granularity of thread migration. As a result, high levels of efficiency are sustained at very fine thread granularities on shared memory multiprocessors, where all other thread scheduling strategies break down due to contention and cache misses.

4 Conclusion and Further Work

We have conducted an investigation into the effectiveness of cache-conscious scheduling using batches, and the exclusive use of wait-free synchronisation techniques in a thread scheduler for shared memory multiprocessors. The experimental results obtained indicate that a significant reduction in execution times can be gained at fine levels of thread granularity through the use of such techniques, both individually and in combination.

The implications of wait-free thread scheduling are numerous. Any path through the scheduler involves the execution of a fixed number of instructions, thus enforcing an upperbound on the time spent in it. In multiprogrammed systems, the extended spinning that typically ensues when a kernel thread is descheduled whilst holding a lock is no longer an issue. In general, contention for shared structures within the scheduler is reduced considerably. We believe that a substantial performance improvement can be achieved on shared memory multiprocessors by avoiding any form of locking in such scenarios, including blocking locks as well as spin locks. This approach should be at least considered as a practical alternative to the status quo.

Further experiments with real applications are required to gather information about the performance of batching under more realistic conditions. It should be noted that batch-based thread scheduling as presented here may be subject to problems when the pre-set batch size limit is greater than the total number of threads being executed in the application, since all threads would be serviced by a single processor. While this can be advantageous in identifying situations where parallel processing is not worth the effort, pathological cases may well occur in specific applications. Automatic modification of the batch size limit could be envisaged, whereby the batch limit is dynamically set to match the current application's needs. At the moment, threads are grouped into batches by locality and indirectly through communication, so that threads created on a common processor are placed onto the same batch. An additional grouping criterion could be based on the frequency of inter-thread communication or rely on object-affinity [13, 14]. Furthermore, the application programmer could be given the opportunity to manually specify viable thread groupings to override the automatic batching arrangements adopted by the scheduler.

The investigations presented here fit into the wider context of a general purpose server-side parallel processing system composed of a cluster of shared memory multiprocessors with high speed user-level CSP communication over Gigabit Ethernet between nodes in the cluster, as well as gigabit speed user-level TCP/IP connectivity to the outside world [15, 16, 5]. Many of the constituent components have already been developed or are at an advanced stage of development.

References

1. K. Debattista, K. Vella, and J. Cordina. Wait-free cache-affinity thread scheduling. *IEE Proceedings - Software*, 150(2):137–146, April 2003.
2. K. Vella and P.H. Welch. CSP/occam on shared memory multiprocessor workstations. In B.M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering*, pages 87–120. IOS Press, April 1999.
3. K. Debattista and K. Vella. High performance wait-free thread scheduling on shared memory multiprocessors. In H.R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1022–1028, June 2002.
4. K. Debattista, K. Vella, and J. Cordina. Cache-affinity scheduling for fine grain multithreading. In J.S. Pascoe, P.H. Welch, R.J. Loader, and V.S. Sunderam, editors, *Proceedings of Communicating Process Architectures 2002*, volume 60 of *Concurrent Systems Engineering*, pages 135–146. IOS Press, September 2002.
5. J. Cordina. A summary of research in system software and concurrency at the University of Malta: I/O and communication. In G. Pace and J. Cordina, editors, *Proceedings of CSAW 2003*, July 2003.
6. K. Debattista. High performance thread scheduling on shared memory multiprocessors. Master's thesis, University of Malta, February 2001.
7. D. Wood and P. Welch. KRoC – the Kent Retargetable occam Compiler. In B. O'Neill, editor, *Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166. IOS Press, March 1996.
8. M. Boosten, R. Dobinson, and P. van der Stok. MESH: Messaging and scheduling for fine-grain parallel processing on commodity platforms. In *Proceedings of PDPTA*, June 1999.
9. J. Cordina. Fast multithreading on shared memory multiprocessors. B.Sc. I.T. Final Year Project, Department of Computer Science and Artificial Intelligence, University of Malta, June 2000.
10. K. Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, University of Kent at Canterbury, December 1998.
11. J. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, May 1995.
12. M. Herlihy. Wait-free synchronisation. *ACM Transacations on Programming Languages and Systems*, 13(1):124–149, January 1991.
13. L. Kontothanassi and R. Fowler. Mercury: Object-affinity scheduling and continuation passing on multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Languages, Europe*, 1994.
14. S. Abela. Improving fine-grained multithreading performance through object-affinity scheduling. B.Sc. I.T. Final Year Project, Department of Computer Science and Artificial Intelligence, University of Malta, June 2002.
15. J. Cordina. High performance TCP/IP for multi-threaded servers. Master's thesis, University of Malta, March 2002.
16. S. Busuttil. Integrating fast network communication with a user-level thread scheduler. B.Sc. I.T. Final Year Project, Department of Computer Science and Artificial Intelligence, University of Malta, June 2002.