High Performance Staged Event-Driven Middleware

Alan Cassar and Kevin Vella

Department of Computer Science, University of Malta
{acas034|kevin.vella}@um.edu.mt

Abstract. In this paper, we investigate the design of highly efficient and scalable staged event-driven middleware for shared memory multiprocessors. Various scheduler designs are considered and evaluated, including shared run queue and multiple run queue arrangements. Techniques to maximise cache locality while improving load balancing are studied. Moreover, we consider a variety of access control mechanisms applied to shared data structures such as the run queue, including coarse grained locking, fine grained locking and non-blocking algorithms. User-level memory management techniques are applied to enhance memory allocation performance, particularly in situations where non-blocking algorithms are used. The paper concludes with a comparative analysis of the various configurations of our middleware, in an effort to identify their performance characteristics under a variety of conditions.

1 Introduction

In this paper, we investigate the construction of highly efficient and scalable staged event-driven middleware for shared memory multiprocessors. The staged event-driven architecture (SEDA) that we focus on is a design introduced in [Wel02] for developing massively concurrent services which behave well under heavy loading. Our multiprocessor implementations of SEDA middleware are designed to be highly efficient, through the use of a variety of event queue structures, and a range of access control mechanisms including non-blocking algorithms.

The paper kicks off with a brief overview of SEDA and possible event queue arrangements. We proceed by investigating a selection of our shared queue designs and multiple queue designs. Following this, we consider the memory management techniques used in our design, and our handling of blocking system calls. We conclude with a comparative analysis of the performance of the various configurations of our middleware.

2 Background

The following section gives a brief introduction to some of the techniques applied to our staged event-driven middleware, including scheduling techniques and performance enhancement techniques.

2.1 Staged Event-Driven Architecture

The staged event-driven architecture (SEDA) is a design introduced in [Wel02] for developing massively concurrent services which behave well under heavy loading. A SEDA application consists of a network of event-driven stages connected by event queues. Dynamic resource controllers are used to ensure the efficient behaviour of stages irrespective of loading variations.

2.2 Shared Run Queue

A technique widely used in scheduling algorithms is based on a shared run queue. Cordina [Cor00] chose this technique to implement a SMP version of the MESH user level thread scheduler, which the author called SMP MESH. Debattista [Deb01] experimented with this technique in his user level thread scheduler implementations. In [Vel98], a similar technique is used for an SMP implementation of KRoC [WW96]. In a shared run queue environment, a single queue shared between all processors contains all the schedulable entities and each processor will have to "fight" to acquire one or more items from the queue. Since we have one queue shared between all processors, the bigger the number of processors, the higher the contention on the queue. This queue can easily become the bottleneck of the system especially when the number of processors is relatively huge.

2.3 Per-Processor Run Queue

In a per processor run queue environment, each processor has its own run queue where the schedulable entities are placed. The biggest advantage of this scheduler is that it does not matter how many processors are accessing the run queue, since each processor has its own, no synchronisation techniques are required to get exclusive access to the queue, hence there is no need to implement complicated and not so trivial lock-free, non-blocking or wait-free algorithms.
Anderson *et al.* [ALL89] experimented with a per processor run queue thread scheduler, however the authors kept a global pool for the reason of load balancing. Their experimentation showed that their per processor scheduler performed better than a shared run queue scheduler, mainly due to no contention on a shared run queue and to the locality achieved in scheduling the same threads on the same processor [ALL89].

2.4 Locality and Load Balancing

Locality is the notion of keeping the processes as close to their data as possible [Deb01]. One of the best ways to achieve this is to exploit the processors' cache. Schedulers which use per processor run queue technique tend to favour locality [Deb01]. On the other hand, shared run queue schedulers usually do not emphasise on locality. Some explicit techniques have to be applied such as batching [Vel98] or cohort scheduling [LP01].

Both load balancing and locality are key factors in performance, however, most of the time tend to be mutual exclusive. In a per-processor run queue, automatic locality management is provided, but when migration policies are applied to load balance the work, locality is lost. Shared run queue offer automatic load balancing, but special techniques have to be applied to maximise locality.

2.5 Batching

Vella [Vel98] argues that much of the performance enhancements that were being studied and evaluated for the KRoC scheduler focused on how to reduce the explicit cost of context switching, but no one actually tried to improve on other implicit costs such as the effect of cache memory, which can easily boost software performance when used correctly.

A technique called batching, which tries to maximise the cache locality in fine grained multithreaded systems, is introduced in [Vel98]. In batching, the run queue does not hold schedulable entities such as threads or processes. Instead, it holds queues of threads, referred to as batches of threads. When a processor requests an entity to be executed from the run queue, the processor is given a batch of threads, where each batch has a dispatch time much longer than the dispatch time of a single process/thread.

It is argued that if the batch size is relatively small with respect to the batch's dispatch time, each thread inside the batch will be scheduled several times on the same processor (within the dispatch time of the batch), helping each thread to find most of its relevant data in the cache memory. If a batch contains a relatively small number of threads, these will not manage to push all the data of other threads from cache, improving performance drastically.

3 Shared Run Queue Middleware

The Shared Run Queue middleware embraces only one run queue for the set of available processors, each of them demanding work from the same shared queue. Once this middleware is launched, one thread for each processor is created and bound to a separate CPU. Figure 1 depicts the design of this scheduler.

Each processor is responsible of dequeuing an available stage from the shared run queue using some synchronisation technique. If the inbound queue of the dequeued stage contains executable events, the processor will remove a set of events, reappend the stage to the run queue, and execute the events as a batch in order to maximise both data and instruction locality. Stages with empty inbound queues are removed from the run queue and added only when their event queue is populated by at least one event.

Once the run queue is short of work, each processor will block after incrementing sleepCounter. This value is consulted every time a new stage is added to the run queue, and if greater than zero, a signal is sent in order to wake up the sleeping processors to resume their execution.

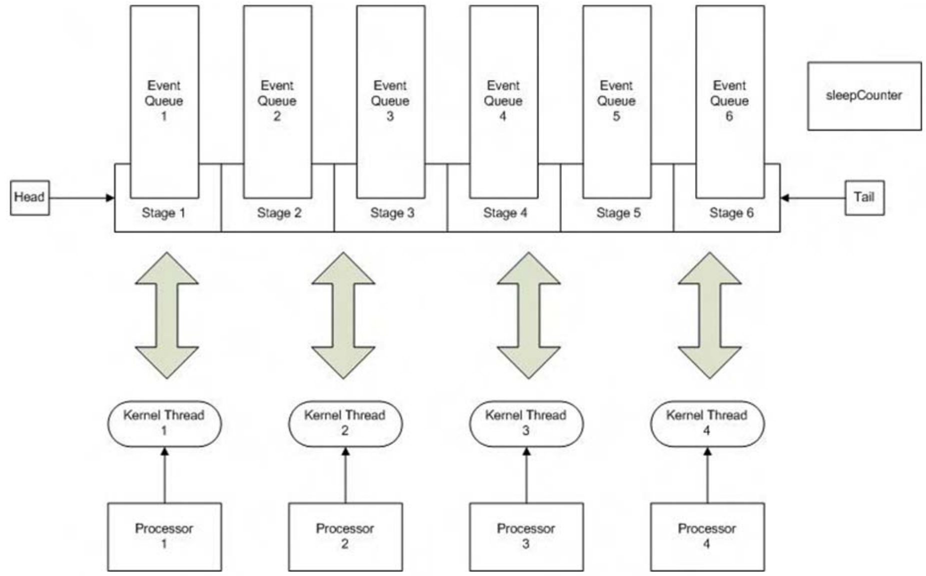

Fig. 1: Shared Run Queue Middleware

The shared run queue scheduler is configurable in several ways, the run queue and the event queues can be configured to use simple queue or dummy head queue. The locking mechanism is also configurable from the operating system-provided locks and our implementation of the *test&test&set* algorithm. Moreover, a non-blocking shared run queue implementation is also available.

3.1 Event-Driven Scheduler

In the event-driven scheduler, instead of scheduling stages, events are directly placed on a shared run queue, each processor dequeues these events and executes them as shown in Figure 2. This behaviour can be simulated by setting the batch size of the previously discussed scheduler to one, however, the above scheduler requires three operations to dequeue one event: dequeue a stage, dequeue an event and enqueue the stage back on the run queue, where this scheduler would require only one. The aim of this scheduler implementation is to prove whether stage batching is actually a better solution or not.

4 Per-Processor Run Queue Middleware

Every processor has exclusive access to an associated run queue in this scheduler implementation. Moreover, each processor has an associated migration queue used for migration of stages between the different processors in the system. Figure 3 shows the architecture of the Multiple Run Queue Staged Event-Driven Middleware.

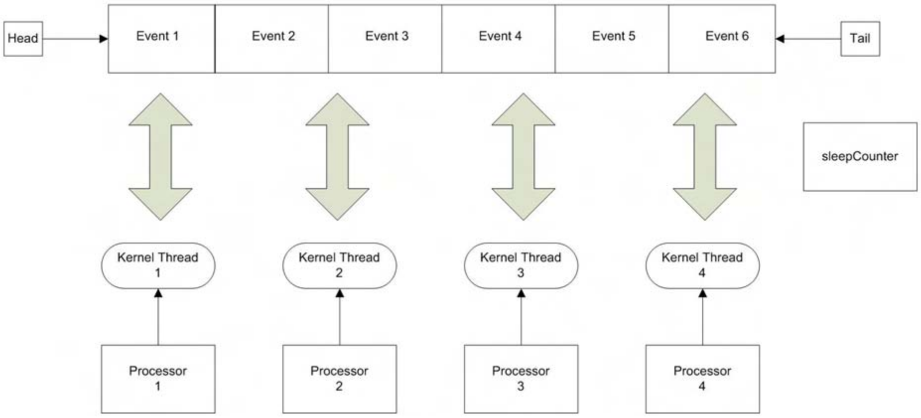

Fig. 2: Event-Driven Shared Run Queue Middleware

Each processor will start off an execution cycle by moving any stage found on the migration queue to the run queue. Once the operation is complete, a stage is obtained from the associated run queue with no use of synchronisation and a set of events are executed as a batch. When no work is available both on the run queue and on the migration queue, the processor blocks waiting for new work to reach the migration queue, at which point it will be signalled by the thread which provided the work.
Several different per-processor run queue schedulers are implemented, each with its own characteristics. The following list gives a brief description of each scheduler.

Basic Model This is the most primitive per-processor run queue scheduler implementation. When a stage changes state from non-schedulable to schedulable, it will be placed on the run-queue of the processor which triggered the operation. Load balancing is an issue with this scheduler implementation.

Maximising Cache Locality This scheduler binds a stage to one processor where it will execute for its entire life. Cache locality is improved because each time an event is executed, it has a greater probability of finding necessarily data in the cache which was pulled in by the execution of similar previous events. This scheduler however tends to suffer from poor load balancing if the system is not designed well.

Sender Initiated Migration The processor having extra workload triggers the routine of sender initiated migration in order to try to move some of his extra workload to other less loaded processors. This routine is initiated by a simple counter and a threshold configurable by the user.

Receiver Initiated Migration This is triggered by the processor which is short of work, in other words, both the run queue and the migration queue are empty.

Hybrid A scheduler which applies both sender initiated migration and receiver initiated migration to keep load balanced.

The run queues, event queues and the migration queues can be configured as simple queues or dummy head queues. The locking mechanism used for the event queues is also configurable from, *pthread* locks, *test&test&set*, or non-blocking.

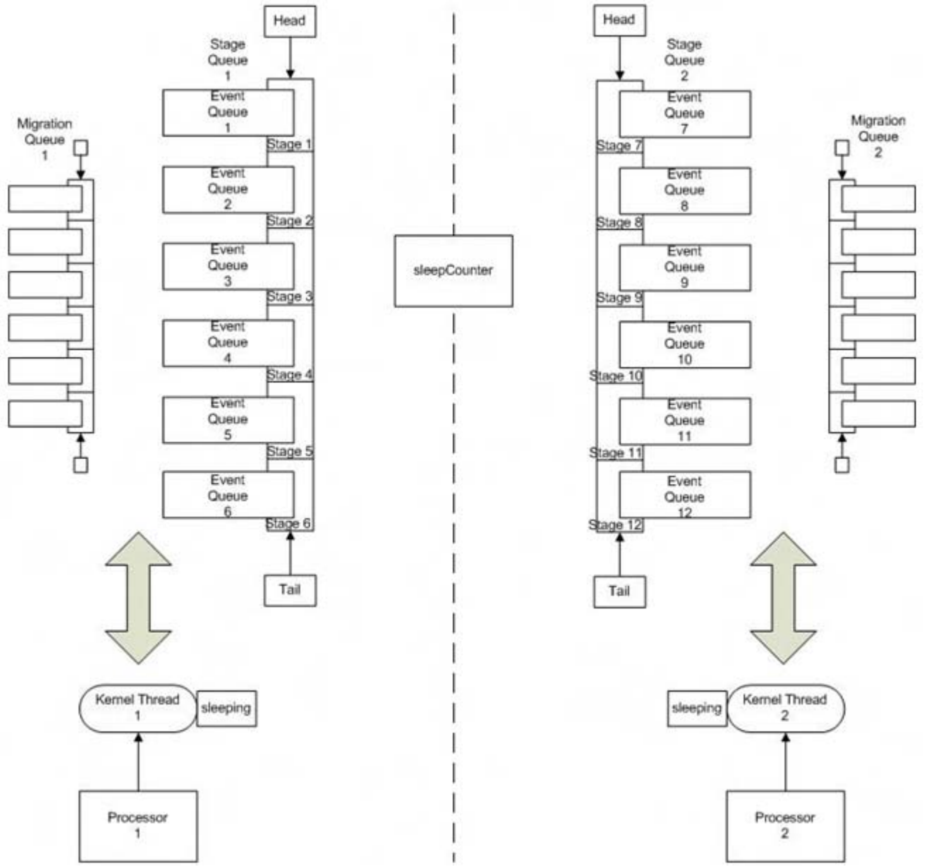

Fig. 3: Per Processor Run Queue Middleware

5 Memory Management

The memory management used in our Staged Event-Driven Middleware is very similar to the one presented by Valois [Val96] for the implementation of his non-blocking data structures. However, a race condition in the memory management of the same author was identified by Michael and Scott in [MS95] where a solution is also given which was carefully applied to our memory management module. Each block of memory to be reused has associated a reference count integer value referred to as *refCount*. This has a dual purpose: to store the number of references pointing to the block in question multiplied by two, or to contain the value of one to denote that the memory block is deleted and ready for reuse.

Memory blocks which are safe to be freed without causing the ABA problem [Val96] are actually never deleted using the *free* system call. Instead, a stack (called the *free stack*) is defined which keeps track of all free blocks in the system which can be reused by the application whenever required.

If a pointer p is referencing a particular block, whenever a function will need to read or modify that location, it does not simply copy the value of p into a temporarily pointer and perform the required operations. Instead, it will need to safely read that block by first incrementing the reference count, followed by the reading or modification of the contents of the block. When all operations on that block are ready, the process cannot plainly continue its execution without first releasing that block by decrementing its reference counter. Furthermore, if the reference count results in the value of zero after being decremented, that block must be freed and returned to the free stack to be reused by other operations.

One can easily see how the *malloc* and *free* system calls are avoided with this solution, but how is the ABA problem solved? If processor n has a reference to block p, processor n is guaranteed that any operation can be carried out safely on block p, even a *compare&swap*. This statement is supported by the definition of our memory management. When processor n obtains a reference to block p, the reference count of p will be incremented by two. Since in our memory management blocks will only be freed when their reference count reaches zero, the safe read executed by processor n on block p halts the reference count to reach zero, and never freed. If the same block is not reused, the ABA problem cannot possibly arise. This is the same solution used by Valois [Val96] to avoid the ABA problem in his non-blocking data structures and modified by Michael and Scott [MS95] to eliminate the race condition determined by the same authors.

6 Blocking Operations

In an event driven system, event handlers are executed until *completion*. Since only one kernel thread per processor is used by the middleware, blocking one of these threads stalls a processor until the blocking call returns. Most I/O operations present such problem including file I/O and socket I/O. In this section we present the approaches taken to solve this issue and discuss the methods applied to integrate for both file I/O and socket I/O in our middleware.

6.1 File I/O

Since we are aiming to achieve asynchronous I/O operations in order not to block our scheduler threads, why not use the technology provided by the operating system itself? The Linux kernel 2.5 and higher provides the support for asynchronous non-blocking I/O operations which they call AIO.

Welsh in SEDA [Wel02] wrapped each I/O call in a SEDA stage [Wel02], and in our staged event-driven middleware, we also prepare the main file I/O operations in stages. When the user requires a file read operation, a file read stage has to be created, where only one stage is required for any number of operations. The operation starts off by sending a special event data structure to the file read stage which includes data such as the file descriptor, the stage to be notified when the operation is completed, and other important information. The file read stage immediately starts off an AIO read operation which upon completion executes a call-back function. The job of the call-back function is that of notifying the appropriate stage that the operation is complete by handing over the data read from the file (or any errors which occurred during the operation). The file write operation is handled in almost the same way as the file read operation.

6.2 Socket I/O

At the time the staged event-driven middleware was being implemented, the socket AIO system was not a standard feature of the Linux kernel. Due to this fact, it was decided that the socket I/O had to be implemented using a different approach from the one used by File I/O.

The model used resembles the solution used by the Flash [PDZ99] web server. For a blocking operation, Flash uses a helper process which blocks until the I/O event is completed, and notifies the original process when such operation completes. Instead of using a separate process, we opted for a thread pool from which a thread is acquired and handled the blocking operation to be performed. Furthermore, Flash [PDZ99] uses IPC channels for communication between the main polling loop and the helper processors which entails a system call each time an event is completed. In our middleware implementation, when a blocking operation terminates, the stage to be notified is sent an ordinary event just as any stage would send an event to any other stage in the system in order to communicate, and everything is done in user space.

7 Performance Testing

The following section presents some tests which were performed in order to compare the different schedulers implemented throughout this project. All tests were performed on a dual Intel Xeon processor machine supporting the Hyper-Threading technology, each CPU running at 3.80Ghz with 2MB of level 2 cache per processor. This Intel processor family supports the EM64T (Intel Extended Memory 64 Technology) architecture and was running an x86-64 operating system, a Linux RedHat with kernel version 2.6.9-42.ELsmp. All memory allocation was done apriori of the start of the tests to avoid system calls.

7.1 Queue Comparison

The following test was performed on a synthetic benchmark where a total of 502 stages had to be scheduled. The job of each event handler is that of reading 8 bytes out of a 32-byte cache aligned data structure and writing 4 bytes into the same structure. The scheduler was configured as an event-driven scheduler and the results can be seen in Figure 4.

Our *test&test&set* implementation performed better than the operating system provided locking mechanisms in both the coarse grained locking queue and the finer grained locking queue. Moreover, the dummy head queue performed better than the coarse grained locking queue due to the separation of the enqueue operation from the dequeue operation. The non-blocking queue was outperformed by the *test&test&set* implementation, however we believe that with a larger number of processors available, the non-blocking queue would have performed better than any other implementation available for the event-driven scheduler.

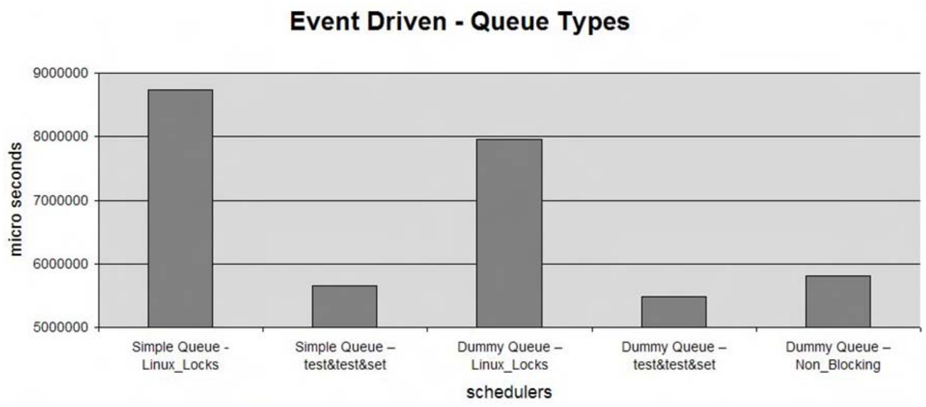

Fig. 4: Event-Driven Scheduler — Different Event Queues

7.2 Batching

The event-driven scheduler never exploits batching, therefore the time taken for the test to complete remains constant independently of the batching value. It is however more efficient than the staged shared run queue scheduler with the batching size set to one due to the fewer number of queue operations performed to execute an event.

With a low batching value, the per-processor run queue scheduler outperforms the shared run queue scheduler by almost 6 seconds, and their are two main reasons for this behaviour, the first one being the contention on the shared run queue with a low batching value. Cache locality is the second reason. Binding stages to the same processor helps in decreasing the total amount of cache misses issued during the execution of the application. As the batching value is increased, both schedulers' performance increases.

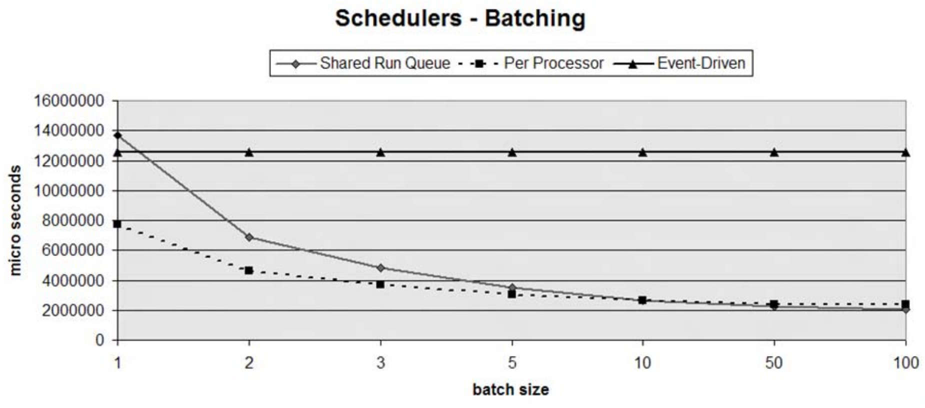

Fig. 5: Scheduler Comparison — Batching

8 Conclusion

We have presented a framework which is highly configurable and general purpose, usable with any staged event-driven application. We have also shown from the results obtained that our middleware implementation is scalable and efficient. File I/O is also integrated within our middleware with the help of the operating system provided AIO while socket I/O is available through the help of thread pools.
We have also conducted a series of performance tests to identify which scheduler performs the best, but we can conclude this document by noting that each application has different requirements and characteristics, and will perform differently on different scheduler implementations.

References

[ALL89] T. E. Anderson, D. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. In *In SIGMETRICS '89: Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, page 4960, 1989.

[Cor00] Joe Cordina. Fast multi-threading on shared memory multiprocessors. Technical report, Department of Computer Science and Articial Intelligence, University of Malta, May 2000.

[Deb01] Kurt Debattista. High performance thread scheduling on shared memory multiprocessors. Master's thesis, University of Malta, February 2001.

[LP01] James R. Larus and Michael Parkes. Using cohort scheduling to enhance server performance (extended abstract). In *In LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 182–187, 2001.

[MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *In Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[Val96] John David Valois. *Lock-free data structures*. PhD thesis, Troy, NY, USA, 1996.

[Vel98] Kevin Vella. *Seamless Parallel Computing on Heterogenous Networks of Multiprocessor Workstations*. PhD thesis, University of Kent at Canterbury, 1998.

[Wel02] Matthew David Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California at Berkeley, 2002.

[WW96] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments – Proceedings of WoTUG 19*, pages 143–166, Nottingham-Trent University, UK, March 1996. World occam and Transputer User Group, IOS Press, Netherlands. ISBN 90-5199-261-0.