Abstract—We describe a scheduler that processes a high number of typed events per second while enabling certain event types to be allocated more resources than others in a work-conserving fashion. The scheduler is the core of a high volume messaging system, it uses a lock-free approach allowing it to scale with increasing number of processors. The scheduler threads coordinate using a lock-free approach allowing it to scale with increasing number of processors. As far as we know, no formal analysis of this lock-free data structure has been given so far in the literature. We analyze the expected behavior of the scheduler and report on its actual performance on a multiprocessor machine.

Index Terms—Scheduling, Event System, Lock-Free, Middleware

I. INTRODUCTION

Queued event systems detach the production of an event from its handling. Structuring applications through the use of such abstractions allows a looser form of coupling than via function calls. When the queues themselves are lock-free then there is no synchronization at all between event producers and handlers. In a multiprocessor environment this helps in proving the correctness of the program and allows better parallelization. As the operational semantics of placing an event in a remote queue are the same as that of placing it in a local one, they do not suffer the same impedance as remote procedure calls where the distributed system attempts to give the illusion that a remote network invocation has the same semantic as pushing a new frame onto the stack [1]. For these reasons we expect a queued event approach to become the unifying abstraction for communication within distributed middleware, whether the communication is intra or inter machine.

Events are typed by the function used to handle them. As distinct event types may be of different importance to the application it is advantageous if the relative importance of the events can be expressed to the dispatching mechanism such that it can be taken into account when allocating resources to them. The entity that dispatches events in the typed event queues is a task scheduler in which the tasks comprise the event queue and the associated function to execute. The scheduler chooses the next task to schedule based on the importance assigned to the corresponding event type.

We shall use a distributed messaging system built in Java as the context in which we describe the rest of our work. In a messaging system, messages on a named topic are carried from one or more publishers to one or more subscribers. Messages can either be out-going from local applications or in-coming from the network. In the first case, the associated task transmits them over the network; in the latter, it executes the associated application-specific callback.

The selection of the scheduling mechanism is determined by the needs of this messaging system. The arrival distribution of events and their dispatch time are not in general known a priori. As a messaging system may be used by different applications in different contexts, the means by which importance is attached to topics should be as general as possible. The scheduler must be work-conserving and parallelizable, allowing it to make best use of the available resources and to scale with increasing numbers of processors. We don’t assume any particular support from the underlying OS and assume that thread scheduling is non-preemptive.

In the next section we motivate our choice of a time-based credit scheduler for the messaging system and compare it with other alternatives. We then describe the implementation of the system showing how coordination between scheduling threads can be achieved using an appropriate lock-free data structure. We give the invariant for this data structure and explain why this invariant is strong enough for use within the messaging system. We describe what fairness means for a work-conserving scheduler on a multiprocessor system. Finally, we investigate various aspects of the overall system in particular in relation to throughput, delay, scalability with processors and performance isolation.

II. DESCRIPTION OF THE SCHEDULER

At a first glance the scheduling of message queues has strong similarities with algorithms used for giving different service rates to flows within network routers. For example, the Weighted Fair Share (WFS) queuing discipline [2] allows different service rates to be allocated to different queues according to some requested share. WFS was developed in the context of packet-switched networks to protect one network flow from another. It belongs to a set of virtual-time-based algorithms that provide fairness and/or delay bounds. Message scheduling is distinct from packet scheduling in that the time a message takes to be serviced cannot be easily estimated; in a packet scheduler it is a simple function of the packet length. This means that the guarantees given by the message scheduler are by nature less precise.

Our proposal is to use time-based credit-scheduling within the message scheduler, in which resources are partitioned...
amongst tasks by the allocation of credits. The total credits allocated to a task is a fraction (defined by the *share* given to the task) of the time period $T$. The scheduler always chooses to schedule the task with the highest number of remaining credits that has a non-empty queue, i.e. the scheduler is work-conserving. Each time a task is scheduled, the scheduler measures how long it runs before it completes. This amount is deducted from the total number of credits for that task. We now describe the guarantees that this time-based credit scheduler gives with respect to throughput.

Assume that $0 \leq v_i \leq 1$ is the share allocated to task $i$ and that $s_i$ is the number of messages arriving at the task queue per second. The question we wish to address is how many messages $r_i$ will actually be serviced for task $i$. It is important to realize that due to the work-conserving nature of the scheduler, no one task can be viewed completely in isolation, i.e. the amount of messages serviced for any given task share depends on the messages arriving for other tasks at other queues. The number of messages serviced for a task can be viewed as containing two components: that which is guaranteed by the share and any spare capacity that a task can take advantage of. We denote the total maximum achievable rate by $R^{\text{max}}$. $R^{\text{max}}$ is a function of the system settings, but for the moment we consider it as a constant.

For example, suppose we have three tasks $T_0$, $T_1$ and $T_2$ with shares $v_0 = 0.5$, $v_1 = 0.3$ and $v_2 = 0.2$, suppose further that we know that $R^{\text{max}} = 10,000$. We would like to know what values are achieved for the three tasks if $s_0 = 4,000$, $s_1 = 6,000$ and $s_2 = 5,000$. It is clear that $r_0$ is 4,000 because task $T_0$ is sending less than its share. That leaves 6,000 to be shared among $T_1$ and $T_2$. Both $T_1$ and $T_2$ will get their guaranteed share (3,000 and 2,000 respectively) plus some fraction of the spare capacity (1,000) that $T_0$ has reserved but is not using. The unreserved capacity is shared proportionally to the shares $v_1$ and $v_2$, so $T_1$ gets 60% and $T_2$ gets 40%, meaning that $r_1 = 3,600$ and $r_2 = 2,400$.

We now give the general formula for calculating $r_i$ for an arbitrary resource allocation between a set of tasks and an arbitrary attempted sending rate on those tasks on a single processor. We consider the effect of multiprocessors in Section III-E. We define $u_i$ to be the fraction of resources that a task is attempting to consume and we define $x_i$ to be the ratio of $v_i$ to $u_i$.

$$u_i = \frac{s_i}{\sum_j s_j}$$

$$x_i = \frac{v_i}{u_i}$$

An $x_i$ value of 1 means that a task is attempting to use exactly what it has reserved. A value greater than 1 means that it has unused capacity and a value less than 1 means that it is trying to use more than its has reserved, i.e. it is trying to take advantage of any unused capacity. For a given setting of $v_j$, $s_j$ and $R^{\text{max}}$ the expected receive rate $r_i$ is:

$$r_i = \operatorname{Min}(s_i, v_i, R^{\text{max}} \cdot \frac{\sum_{j \neq i} x_j \cdot r_j}{1 - \sum_{j \neq i} x_j \cdot v_j})$$

This states that the share that a task will get is its *adjusted* share of what is left over after tasks with higher $x_i$ have been allocated. The amount of additional unused capacity available to a task is that which is left over by tasks which are less speculative than itself. This is equivalent to the Weighted Max-Min fairness allocation developed for the Available Bit Rate (ABR) service within ATM [3] where $u_i$, corresponding to the demand or rate and $v_i$, to the normalized weight.

### III. Implementing the Scheduler on a Multi-Core Architecture

![Diagram](image)

**Thread** SchedulingThread

1. while true do
2. $t \leftarrow \text{SchedulableTasks.get()}$
3. if $t.\text{credit} = 0$ then
4. $\text{resetAllTaskCredits()}$ /* the Task with the most credits has 0. Period $T$ is over. */
5. continue while
6. end if
7. $e \leftarrow t.\text{eventQueue.get()}$
8. $T_o \leftarrow \text{getTime()}$
9. $t.\text{processEvent}(e)$
10. $t.\text{credit} \leftarrow t.\text{credit} - (\text{getTime()} - T_o)$
11. if $\text{queueSize} > 0$ then
12. $\text{SchedulableTasks.put}(t)$ /* Task $t$ has more events, so it is put back for scheduling */
13. end if
14. end while

![Diagram](image)

A task $t$ contains a queue $t.\text{eventQueue}$ into which messages or timer events are placed. In addition, it contains control information such as the allocated share and the amount of outstanding credits $t.\text{credit}$. A task is ready to run when it has credit and its queue contains at least one event. Tasks are passive data structures; the scheduler employs a number of scheduling threads to execute tasks. Figure 1(b) shows the basic operation of a scheduling thread. The core of the scheduler is a priority queue containing the set of schedulable tasks. Figure 1(a) shows the schema of the interaction in which tasks are ordered by their number of threads in the queue (shown as a binary heap) and scheduling threads add and remove elements from this queue concurrently.
We refer to this priority queue as \textit{SchedulableTasks}. The task with the most credit has the highest priority and is therefore at the head of the priority queue, from where it is removed by a scheduling thread for execution (see Figure 1(b)). Once invoked, the task processes exactly one event. The scheduling thread measures the execution time and uses this to decrease the credit of the task, thus changing its priority. If the highest priority task is out of credit, then no task has credit left and the scheduling thread resets the credits of all tasks, i.e. the scheduler is work-conserving. This situation occurs at the latest when the scheduling period $T$ expires; but may occur earlier if some of the tasks had less work than their allocated share.

Using multiple scheduling threads allows the scheduler to run on multiple CPU cores. For efficient operation, it is important that a scheduling thread is not blocked by another. Two contention points exist, the event queue used by each task (see Section III-A) and the priority queue used at the core of the scheduler (see Section III-B).

\textbf{A. Coordination between writer threads and scheduling threads}

The event queue of tasks is implemented as a Michael–Scott [4] lock-free FIFO queue. This queue allows scheduling threads (the readers) and input threads or application threads (the writers) to concurrently access the same queue without the need for locking. Whereas in [4] the size of the queue is not limited, we trivially extend the algorithm such that a writer blocks if the queue reaches its capacity and then is woken up by the next reader. All the information about the credits a task has is maintained within the task itself, and because it is only updated by the scheduler thread that took the task out of \textit{SchedulableTasks}, this information does not need to be thread-safe.

Writer threads and scheduling threads coordinate in deciding whether a task is schedulable or not. A writer thread that adds a new event to a task that is currently not scheduled will add it to \textit{SchedulableTasks}. Likewise, a scheduling thread that finds the event queue of a task empty will remove the task from the set.

Figure 2 shows how this coordination takes place. In Figure 2(a) the writer thread recognizes that a task is not in the schedulable task set and adds it; simultaneously, a scheduler thread accesses the same data structure and removes the tasks with the most credits. In Figure 2(b) the task is scheduled, the writer thread adds events to the event queue while the scheduler thread removes the first one. In Figure 2(c) the scheduler thread puts the task back into the schedulable task set after updating its credits, while the writer thread continues to write. In Figure 2(d) the scheduler thread has emptied the queue and does not return the task to the set; the next time the writer writes it will notice this and perform the action described in Figure 2(a) again.

The scheduler thread removes the task from \textit{SchedulableTasks} and returns it if there is still work to do (i.e., an event is in the task’s queue). If the scheduler thread detects that there is no more work to do for a task, it is not returned to the queue. It is the responsibility of the writer thread to put the task back into \textit{SchedulableTasks} when the task has again events in its queue. This coordination is achieved without locking by using an atomic counter $t.queueSize$ for the number of events in the task’s event queue. The writer increments $t.queueSize$ after having written an event, whereas the scheduler thread decrements it after reading an event. The scheduler thread will only return the task to the set if the value is non-zero. A writer recognizes that a scheduler thread did not return a task if the value before it succeeded in incrementing the atomic counter was zero.

\textbf{B. Lock-free access to the concurrent priority queue}

The priority queue that implements \textit{SchedulableTasks} is accessed concurrently by writer threads and scheduler threads. Because of their importance in scheduling on multiprocessor operating systems, much work has been done on algorithms for concurrent priority queues. This work covers both blocking [5], [6], [7] and non-blocking approaches [7], [8]. While in the event-system different tasks can have an arbitrary number of time credits, meaning that in theory we need to use an algorithm that supports arbitrary priorities, in practice there is little to be achieved by distinguishing between tasks that have a number of credits which are within some range of each other. This observation allows us to quantize the credits such that there is a fixed number of priority levels.

A blocking quantizing priority queue algorithm has been described in [7]. The range of priorities is divided such that every valid priority falls into one of $N$ buckets. Elements in the same bucket are retrieved according to their arrival order. Shavit et al. call this a \textit{SimpleLinear} priority queue and report through the use of simulation that its performance compared with a range of other approaches is best for small numbers of processors (fewer than 16). They propose the use of a funneling mechanism, whereby processes accessing the same bucket can recognize and avoid contention by having only one of them perform the required operations.

Our implementation follows the same idea as \textit{SimpleLinear}, but unlike \textit{SimpleLinear}, it is non-blocking. Each bucket is associated with a lock-free FIFO queue. These buckets are placed in an array such that the highest priority (most credits) bucket is at the extreme left and the priority reduces as we move to the right. The scheduler thread starts at the highest priority bucket and moves to the right until it finds a bucket with a non-empty queue. It then attempts to take the highest priority task in this queue. If the scheduler thread succeeds, it dispatches this task, if not it moves to the right again. If it reaches the end of the structure without finding work it returns a null value. Figure 3 shows the lock-free priority queue. The precision of the share can be traded off against the efficiency of the system by increasing or decreasing the number of buckets.

The total number of operations required to identify into which bucket to add a task is constant, but the number to find the first non-empty bucket increases linearly with the number of buckets, $N$. The thread determines whether a bucket is non-empty by checking an atomic counter. For a given number of buckets the worse case removal time is constant, i.e. when the element is found in the least priority bucket.
If no task overruns its allotted time slice then we can guarantee that no task goes unscheduled for longer than 10 ms by setting the value $T$ over which the share is respected to 10 ms. Now assume we wish to be able to distinguish tasks at a time granularity of 100 \( \mu\)s then we need 100 buckets. This in turn requires on average 50 integer comparisons each time we schedule a task. To reduce this overhead we group buckets into bucket groups of size $M$. An additional atomic counter is kept for the entire bucket group as well as for the individual buckets. The average number of comparisons is then $Y = M/2 + N/(2M)$. The optimal value of $M$ for fixed $N$ is now the value at which the derivative of $Y$ with respect to $M$ is zero, which is given by $M = \sqrt{N}$. The expected number of operations is $\sqrt{N}$ and the worst case $2\sqrt{N}$, i.e. for 100 buckets we expect to have 10 additional integer comparisons for removing a task and in the worst case we have 20. The same technique can be performed at multiple different levels, i.e. by creating groups of bucket groups etc. In practice trying to distinguish between messages at very fine granularities makes little sense as the variance in other factors beyond the messaging system’s control dominate, e.g. network delay. We find that for time granularities that make sense in a messaging system running over a LAN (in the 100-1000 \( \mu\)s range) one level of bucket groups is adequate.

When no schedulable task has work to do, the scheduler resets the credits allocated to all tasks in the \textit{SchedulableTasks} set. The scheduler recognizes that this has occurred when the task returned from the \textit{SchedulableTasks} set has zero or less credit. The scheduler removes all the tasks from the set, resets their credit as described in Section III, and returns them to the set. Multiple scheduler threads can identify the need for credit resetting and perform this operation in parallel. This is a consequence of the fact that a task can be read by exactly one scheduler thread from the set and that all credit information about the task is contained within it.

It may be the case that one or more tasks with credit are
Currently being serviced by other scheduler threads when a given scheduler thread identifies that no current task in the `ScheduleableTasks` has work. A thread that was servicing a task while the credits were being reset by another thread needs to recognize that this has occurred and update this task’s credit before putting it back into `ScheduleableTasks`. This is achieved using an atomic counter that is incremented after each reset. Each thread keeps the value of this counter before getting a task and checks whether it has changed before it puts the task back.

C. Linearizability and the concurrent priority queue

Herlihy’s linearizability condition [9] can be stated informally as follows. A data structure is linearizable if and only if:

- all possible histories over that data structure have a legal sequential equivalent;
- the order of operations that do not execute concurrently are respected within the sequentially equivalent history.

If a concurrent data structure is linearizable we can then reason about it using its sequential equivalent. Paper [7] claims that the algorithm described in the preceding section is linearizable, however this is not the case. We show that it is not by the means of a counter example. As for any lock-free concurrent priority queue, because elements are being simultaneously added and removed, the get operation may not retrieve the task with the highest priority in the queue at the moment the operation completes. In our implementation, it is possible that another thread adds a task to a bucket that the reading thread has already passed over. The following describes a valid sequence of operations in our implementation using the notation of [9], where x is a task with higher priority than task y:

\[
\text{Get()}A; \text{Put}(x)B; \text{Ok}()B; \text{Put}(y)B;
\text{Ok}()B; \text{Ok}(y)A; \text{Get}()C; \text{Ok}(x)C;
\]

Each operation consists of a start of invocation and its completion. For example, `Put(x) A` stands for the start of the put invocation of task x into the priority queue by thread A, and `Ok() A` corresponds to its completion. The history is not linearizable when the priority of x is higher than the priority of y as no legal sequential history can respect the fact that the addition of x completed before the addition of y but was removed after.

The practical consequence of this is that thread A retrieves a lower priority task than thread C, even though the get operation of thread A precedes the get of thread C and task x was put in before task y. Figure 4 illustrates this inversion of priorities.

The advantage of linearizability is that linearizable data structures can be composed such that the resulting combined history is also linearizable. Showing that a data structure is linearizable allows it to be used in many different contexts. We are interested in the use of the concurrent priority queue specifically in the context of the credit scheduler. Therefore we must demonstrate that the priority queue behaves appropriately in that context. We now give the invariant of the concurrent priority queue and show that it is adequate for our purposes.

D. The concurrent priority queue invariant

Let a put operation be defined as follows:

\[
\text{PUT} ::= [e : \text{Element}, \text{start} : \text{Time}, \text{end} : \text{Time}]
\]

where `start` and `end` are the times that the operation started and completed at and e is the element added with priority e.prio. Let a get operation be defined as:

\[
\text{GET} ::= [p : \text{PUT}, \text{start} : \text{Time}, \text{end} : \text{Time}, e : \text{Element}]
\]

where `start` and `end` are the times that the operation started and completed at and p is the put operation whose element the get operation retrieves. It must be the case that an element is retrieved only after it has been added, i.e.:

\[
\forall g : \text{GET}, g.\text{end} > g.p.\text{start}
\]

We define a history of the observed get operations ordered by their time of completion, i.e.:

\[
H : \text{SEQ of GET}, \forall i < j \quad H[i].\text{end} \leq H[j].\text{end}
\]

Then the following must hold \( \forall i, j \ i < j \):

\[
H[i].\text{e.prio} < H[j].\text{e.prio} \Rightarrow H[i].\text{start} < H[j].\text{p.end}
\]

This simply states that if an element e was removed from the queue with a higher priority than one removed before it, then the operation that placed said element e in the queue must have completed after the preceding removal had already started. This is sequentially consistent according to Lamport’s definition in [10], i.e. an equivalent sequential history can always be produced, but because that sequential history would involve inverting the order of certain put and get operations, it is not linearizable.

Although the data structure is not linearizable and hence does not have the same behavior as a sequential priority queue, this does not exclude its use within the scheduler. In the specific context of the scheduler, a given scheduling process always gets a task from the queue and then puts it back. A given process history is therefore an interleaved series of get’s and put’s that are always sequential with respect to each other.

![Figure 4. The interleaving of get and put operations (t4 > t3 > t2 > t1).](image-url)
in the complete history. This means that, while it is possible that the data structure occasionally does not behave like a sequential priority queue, the duration of this discrepancy is bounded: the process that placed the task \( T_a \) that suffers from the priority inversion will immediately attempt to get the current highest-priority task. If no other task has been added, \( T_a \) will be chosen; if a higher priority \( T_b \) has been added then \( T_a \) will be chosen and the process that added \( T_b \) will in turn attempt to get the current highest-priority task and so on. It follows that even allowing for occasional priority inversions the overall share that a task receives over a given time period will be respected.

E. Fair shares on a multi-core machine

The scheduler guarantees that when a scheduler threads executes it will choose the available task that best fits the schedule. Suppose we have a one CPU machine in which the scheduler thread is the only thread that runs, then all tasks will always be available and the share allocated to the task will simply be a fraction of the CPU.

In a real system there are many other threads running. Within the messaging system alone, we have timer threads, threads monitoring the I/O, and threads supporting the control part of the messaging protocol. The JVM itself typically runs several daemon threads, e.g. for garbage collecting, and then there is everything else that runs on the machine, e.g. the application. In short, in reality the scheduler threads are not always running, and the share allocated to a task on a single processor machine is a share of the fraction of the time the scheduler runs.

The situation is more complicated on a multi-core machine because when a scheduler thread is serving a task, that task is unavailable to other scheduler threads. This is simply a consequence of the fact that a task is allocated to at most one scheduler thread at any given moment.

Assume that the probability that a scheduler thread is scheduled by the OS is independent and denoted by \( P \) and that there are \( N \) processors and \( N \) scheduler threads in the system. If \( P = 1 \), then no task can ever get more than \( 1/N \) of the total time the scheduler's threads run. So for example if \( N = 1 \) the maximum share that a task can receive is 100%, if \( N = 2 \) the maximum is 50%, etc. More generally, the probability of there being \( k \) scheduler threads running simultaneously is given by the binomial distribution whose expected value is \( N \cdot P \).

Hence we expect the number of processors on which scheduler threads run to be the product of the number of processors and their probability of getting scheduled. The maximum percentage of total scheduling time that any task can get is then given by

\[
\frac{100\%}{\text{Max}(1, N \cdot P)}
\]

For fixed \( N \) this approaches 100% as \( P \) gets smaller, and for fixed \( P \) it approaches \( 1/N \) as \( N \) gets bigger.

Note that the scheduler is not fair in the sense used in WFQ [2], i.e. it is possible for a task to get more than it requested while another gets less than it requested. To make the scheduler both fair and work-conserving, it would be necessary to allow multiple scheduler threads to dispatch events from the same event queue simultaneously. This would not only complicate the scheduler, but, more importantly, would mean that FIFO delivery of messages within a given topic is no longer guaranteed by the scheduler.

IV. Performance Evaluation

In all of the tests the following configuration has been used unless otherwise stated. The time period over which the shares are valid is set to 10 ms. The number of buckets used in the priority queue is set to 100. We use a number of scheduler threads equal to the number of cores on the machines. The communication between machines is always Gigabit Ethernet, and we use TCP for the transport layer with the socket size set to 128 kbytes. The message size is 128 bytes. All machines are running a 2.6 Linux Kernel with Java 6. The machine on which the subscribers run has \( 2 \times 2.3 \) GHz 4-core processors, i.e. 8 cores in total. All publishing machines have \( 2 \times 3.0 \) GHz HyperThreading processors.

A. Bandwidth/delay for one publisher to one subscriber

![Figure 5](image_url)

Fig. 5. Bandwidth/delay for one publisher to one subscriber.

We measure the maximum number of messages per second we can send between a single publisher and a single subscriber. We also measure the effect on the average delay of increasing the sending rate. To avoid having to synchronize clocks on distinct machines, the end-to-end delay is measured by
marking some random sample of the sent messages at the publishing application and having the subscribing application echo these packets back to the publisher over UDP. This method over-estimates the end-to-end delay for low-sending rates (fewer than 1,000 msg/s) as the additional network latency is a significant fraction of the total delay, but is a good approximation for higher rates where the network latency plays a lesser role.

Figure 5 shows the evolution of the effective throughput and the end-to-end delay as a function of the publishing rate. The system sustains a rate of 120,000 msg/s. The average end-to-end delay rises from below 1 ms for 1,000 msg/s to slightly above 10 ms at 120,000 msg/s. Above this figure the system is no longer sustainable, and the average delay rises dramatically.

B. Scalability with increasing number of topics

We measure how the messaging system behaves with increasing number of topics. We run a set of subscribers within a single JVM. Each subscriber subscribes to a distinct topic. We run each publisher on a different machine. Each publisher attempts to send at the same fixed message rate. Figure 6(a) shows how the cumulative throughput at the subscribers evolves as the number of publishers increases for a per-publisher sending rate of 70,000 msg/s. We find an almost linear scaling with increasing number of topics up to the number of cores on the machine, demonstrating that topics can be serviced in parallel. The average delay for the sets of topics behaves similarly to that reported in Section IV-A, with an average delay of less than 50 ms for a cumulative throughput of 500,000 msg/s.

An interesting effect is observed when we increase the fixed sending rate to 80,000 msg/s (see Figure 6(b)) and 90,000 msg/s (see Figure 6(c)). The cumulative throughput initially increases linearly and then collapses, meaning for example that the total throughput for 8 publishers at 90,000 msg/s is significantly less than at 70,000 msg/s. We currently assume that this is an artifact of the Linux kernel when subjected to high interrupt rates, but have not yet been able to verify this.

The promising performance in Figure 6(a) shows that our design is able to take advantage of a multi-core architecture. The reader may better grasp the importance of these results by comparing them with Figure 7: there we perform exactly the same experiment as before, but we replace the lock-free SchedulableTasks in the scheduler with a simple queue accessed using a lock. The lock is necessary to avoid having the queue corrupted by threads manipulating the data structure concurrently. Figure 7 shows that the traditional approach does not scale. Using more than one scheduler thread improves performance slightly, but for increasing number of threads this effect is offset by the increased contention on the lock (8 scheduler threads perform worse than 2, even if the machine has 8 cores).

C. Task shares

Section II described the mathematical model of the scheduler for a single scheduling thread. Section III-E described how the actual share allocated to a task is influenced by the number of processors over which the messaging system runs and by the probability of scheduler threads being able to execute on those processors.

We test the actual measured received rates when ten topics are allocated distinct shares on a two-processor machine running either one or two scheduler threads. The system is pushed to saturation by attempting to send an aggregate sending rate twice that of the maximum measured achieved rate for the configuration. The share allocated to a topic and the fraction of the aggregated total it attempts to send at, are independent random variables from two distinct Poisson distributions. Thus, it can occur that a topic attempts to send at a very high rate with a very low share. The test is run until new results do not significantly change the average. The test was repeated 500 times, each time with a different randomly chosen configuration. We then measure the actual aggregate throughput \( R_{\text{max}} \) achieved at a given setting and used it to calculate the predicted received rates of the individual topics at those setting according to the model in Section II. Finally, we calculate the error as the distance between the vector of predictions and the vector of measured values. The relative error is defined as that error divided by the magnitude of the vector of measured values.

The CCDF (Complementary Cumulative Distribution Function) of the relative errors is given in Figure 8. For one scheduler thread, 85% of all runs have an error smaller than 10%; for two scheduler threads, the same is achieved for 80% of the configurations. The results in Figure 8(a) show the performance of the scheduler at saturation for randomly chosen settings. When the system is not at saturation the fidelity of the scheduler with respect to the model is almost exact (see Figure 8(b) for 100,000 msg/s total sending rate).

V. CONCLUSION

We have shown how a messaging system supporting different qualities of service for different topics can be built using a event dispatching model. We have motivated our choice of time-based credit scheduling within the messaging system and given a statement of the guarantees that it provides. We have described how the scheduler is parallelizable allowing it to scale on a multi-processor system through the use of
a lock-free approach and given a formal invariant for the concurrent priority queue used. We have motivated our use of this data structure proving that it is not linearizable but that in the context used, its invariant is strong enough to allow the required scheduling behavior. We have reported on the performance of the messaging system showing that on a real multi-core architecture the throughput scales with number of cores, we have also described the measured delay and the consequences for topic isolation on running the scheduler on a multiprocessor machine.

REFERENCES


