A Wait-Free Queue for Multiple Enqueuers and Multiple Dequeuers Using Local Preferences and Pragmatic Extensions

Philippe Stellwag, Alexander Ditter, Wolfgang Schröder-Preikschat
Friedrich-Alexander University Erlangen-Nuremberg
Computer Science 4
Martensstr. 1
91058 Erlangen, Germany
{stellwag,ditter,wosch}@cs.fau.de

***Abstract*— Queues are one of the most commonly used data structures in applications and operating systems [1]. Up-and-coming multi-core processors force software developers to consider data structures in order to make them thread-safe. But, in real-time systems, e.g., robotic controls, parallelization is even more complicated as such systems must guarantee to meet their mostly hard deadlines. A considerable amount of research has been carried out on wait-free objects [2] to achieve this. Wait-freedom can guarantee that each potentially concurrent thread completes its operation within a bounded number of steps. But applicable wait-free queues, which supports multiple enqueue, dequeue and read operations, do not exist yet. Therefore, we present a statically allocated and statically linked queue, which supports arbitrary concurrent operations. Our approach is also applicable in other scenarios, where unsorted queues with statically allocated elements are used. Moreover, we introduce 'local preferences' to minimize contention. But, as the response times of our enqueue operation directly depends on the fill level, the response times of a nearly filled queue still remain an issue. Moreover, our approach is jitter-prone with a varying fill level. In this paper, we also address all of these issues with an approach using a helping queue. The results show that we can decrease the worst case execution time by approximately factor twenty. Additionally, we reduce the average response times of potentially concurrent enqueue operations in our queue. To the best of our knowledge, our wait-free queue is the best known and practical solution for an unsorted thread-safe queue for multiple enqueuers, multiple dequeuers and mulitple readers.**

I. Introduction

Multi-core platforms have arrived in the sector of embedded systems, like Intel's Atom series 300. This means that it is imperative to refactor and restructure existing software to further increase the performance of future processor generations with multiple cores. In the past, consecutive processor generations have been used to integrate new technologies into software and make them applicable in terms of performance. Now, the free lunch is literally over [3], which means that software developers must parallelize their software in order to achieve further performance gains. But, in the embedded and real-time sector, e.g., for robotic controls (RC), parallelization is even more difficult than for the desktop area. But, missing this trend would inevitably lead to a drop in the sales figures.

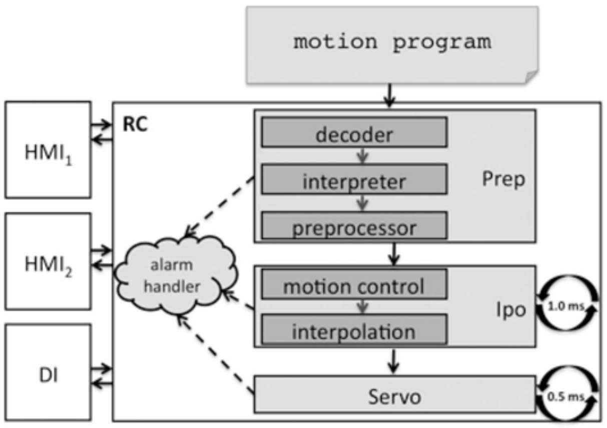


Fig. 1. RC interfaces to clients reading out and visualizing active alarms

A. Our Motivation

Our ongoing attempts to parallelize a robotic control kernel forced us to make the alarm handler thread-safe. Therefore, we present a wait-free queue for multiple enqueuers, multiple dequeuers and multiple readers. The alarm handler stores alarms (e.g., drive errors) and transfers the alarm information to units outside the robotic control (e.g., human machine interface (HMI) or debugging interface (DI)), which do not have any real-time constraints. The alarm information is stored in a common unsorted queue, which is concurrently accessed. As illustrated in Fig. 1, inside the robotic control (RC) there are three concurrent threads ($Prep$, Ipo and $Servo$). In the following we call these threads actuators, which are potentially concurrent reading, enqueuing and dequeuing alarms. Clients outside the RC only read-out the alarm handler's queue. Every thread consists of multiple processing station objects, which realize the information flow of the motion program through the RC kernel to the physical drives. A respective alarm reaction (e.g., to stop the drives) is handled on the basis of the return values of each processing station. But, the alarm information is enqueued inside the processing station, which means that the response time to synchronously triggering the alarm information will delay the respective alarm reaction. Dequeue operations are performed by an additional background task inside the RC kernel, which can delete dedicated or groups of alarm information. For example, a group of alarms can be such alarms with a specific axis number.

Our solution of an optimized wait-free queue for multiple enqueuers, dequeuers and readers is also applicable in other concurrent scenarios, where statically allocated arrays are used and where potentially high contention occurs.

B. Contribution

Previous approaches to synchronize linked queues with locks can suffer from a whole host of problems. These can include priority inversion, deadlocks, livelocks and starvation, which are not tolerable for real-time systems. Furthermore, we have to take into account the worst case execution time of synchronization methods leading us to design a wait-free [2] approach, which does not suffer from any of the mentioned problems. But previous work in this area, as described in Section VI, does not satisfy our real-time and/or scenario specific requirements or is not applicable in practice.

The subject of this paper is a statically allocated and statically linked wait-free queue with local preferences. As actuators firstly use their statically allocated local elements to trigger alarms, which appear to be placed in a static array, we called this behavior 'local preferences'. Furthermore, we must handle both multiple enqueuers and multiple dequeuers, respectively. We do not want any jitter-prone compare-and-swap based retry loops or helping schemes, as presented in [4], [5], nor (potentially expensive) kernel lock objects; our protocol needs atomic test-and-set (TAS), test-and-reset (TAR), fetch-and-add (FAA) and bit-test (BT) operations, which are available in the most multi-core processors. Additionally, we present a helping queue mechanism and an improved traversal algorithm to reduce the time for traverse the alarm queue to find a free element. This guarantees that enqueue operations have short response times and hence the delay of the alarm reaction is minimal. To the best of our knowledge, our approach has not been previously studied.

C. Outline

The paper is organized as follows: Section II analyzes the status quo solution and describes the requirements, which must be fulfilled by a thread-safe alarm queue. In Section III we present our approach for a data structure and its associated wait-free protocol for concurrent operations on it. Section IV describes a helping queue mechanism to improve our wait-free queue in terms of determinism and performance. This is followed by the description, discussion and evaluation of an improved traversal algorithm in Section V. Related work is shown in Section VI. And we summarize our results and experiences in Section VII.

II. Analysis

A. Status Quo

Currently the data structure used by the alarm handler is a linked queue. All queue elements are allocated statically when the RC boots. In order to increase determinism and avoid unnecessary overhead during the runtime, the system internally maintains two queues: one with free elements and one with the active alarms. When a new alarm is issued the alarm handler takes a free element, fills it with the alarm data and inserts it into the queue of active alarms. In order to ensure consistency, the enqueue and dequeue operations are performed with disabled interrupts. If no free element is available, the new alarm will be lost. This only means that the new alarm will not be displayed on the client-side (e.g., HMI or DI). As mentioned, the alarm reaction (e.g., to stop the drives) will nevertheless be handled based on the return value of the processing station. Currently, the alarm handler stores up to 600 alarms, which is more than enough for our robotic control.

Changes in the queue cause all active clients to re-read the queue of active alarms in order to update their displays. Dequeue operations, e.g., removing one specific alarm or deleting all alarms from a dedicated station, can be performed from any actuator in the RC kernel.

B. Requirements for Transition to SMP

The following points have to be considered for possible solutions running on multiprocessor systems. They are the basis of our solution presented in the next Sections.

Correctness. This seems to be a self-evident point, but conventional locking strategies with mutexes, spinlocks or semaphores suffer from multiple problems. The incorrect use of locks can lead to unbounded priority inversion, starvation, deadlocks, livelocks and race conditions. Additionally, locks can lead to convoy effects and introduce jitter. Nonblocking lock protocols, such as the priority inheritance or priority ceiling protocol [6], [7], can result in high overhead regarding time and space. The priority inheritance protocol [6] cannot guarantee to be free from starvation.

Compatibility is split into two sub-items. First, the RC's internal compatibility with today's interfaces in order to guarantee that all existing alarm handler calls are valid for the SMP-compliant version. The second sub-item concerns the external compatibility, which is much more important. Our RC controls industrial production robots in the automotive industry. To understand the requirement of external compatibility, we have to keep in mind that this is what the customers will see on their HMI. Hence, future software releases must be totally backwards compatible to previous versions.

Scalability. In the future we expect a continuous increase in the number of available execution cores. In order to take advantage of this development, the number of actuators has to grow as well. This requires a queue that is able to deal with a growing number of enqueuers and dequeuers.

No assumptions about timing. The robotic control system can trigger alarms asynchronously at any time. This means that we cannot make any assumptions regarding the timing information of when operations will actually occur.

Bounded response time. In hard real-time systems it is essential to fulfill the timing specifications, which as a

status bits, counter	semantics
init	This bit reserves an alarm element for an enqueue or dequeue operation.
use	This bit indicates whether the alarm element is 'in use' and contains valid alarm data.
del	This bit indicates if the element has been 'logical removed' and is free for further enqueue operations.
r_cnt	This read counter indicates if and how many readers are attachted to this alarm element.

TABLE I
SEMANTICS OF THE STATUS BITS

consequence, makes the worst case execution time of any operation an important parameter.

Minimal jitter. Jitter becomes noticeable when the axis moves and is therefore a critical point. Hence, we have to monitor the jitter introduced by our protocol.

III. A Wait-Free Multiple Enqueuers Multiple Dequeuers Queue Using Local Preferences

A. Idea

Figure 2 gives an overview of the structure of our wait-free alarm queue. A set of local elements is assigned to each actuator; the allocation is done during the boot process of the RC. These 'local queues' are connected with each other. The last element of an actuator's queue points to the first element of the next actuator's queue. The last element of the last actuator points to the first element of the first queue. As a result we have one large queue with a statical circular structure that can be traversed using the next pointers. Traversal of the queue, e.g., to find a free element, always starts at the first element of the corresponding actuator and ends at the last element of the previous actuator. We call this behavior 'local preferences' of actuators. If actuators do not exceed their local elements, there is no mutual impact of other actuators. Furthermore, we have relocated the contention from an ordinary dynamic queue, where enqueue and dequeue operations induce contention on the queue itself, to the queue elements. Our approach obviously induces no contention on the queue, because it is statically linked. But, actuators can dispute about the queue elements. As an example, if there are only a few free elements, a scenario can occur where actuators compete for one element. But if there are only a few alarm elements in use, the level of contention on the free alarm elements is very low. Hence, both the behavior of local preferences and the statically linked structure of our queue ensures minimal contention.

Each element has its own set of status bits (*init*, *use*, *del*), which are used to coordinate and synchronize the parallel disjoint-access [8] enqueue and dequeue operations on each element. Disjoint-access parallelization means that our enqueue and dequeue operations concurrently work on disjunctive elements. The read counter *r_cnt* handles multiple simultaneous read operations with potentially concurrent enqueue and/or dequeue operations on one element. The semantics of the status bits and the read counter are depicted in Table I. The entire queue is statically linked, this means no pointers have to be modified at runtime.

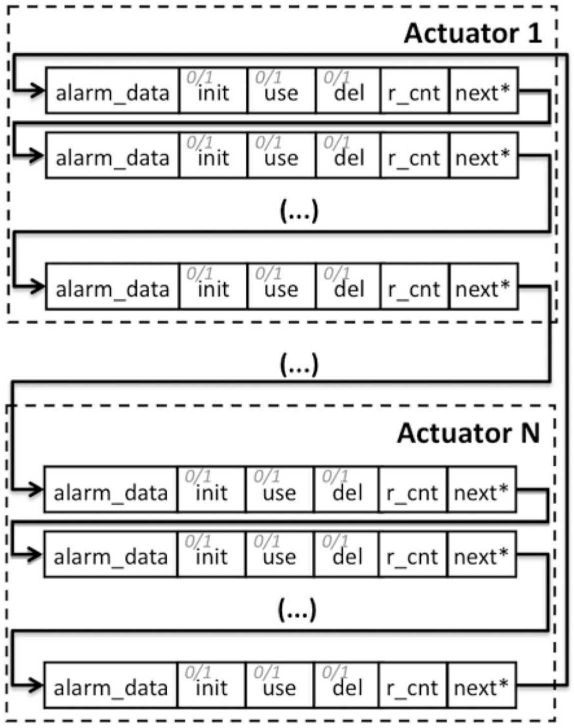


Fig. 2. Structure of our wait-free queue

B. Protocol

For synchronization and coordination of the queue we use three status bits and one counter at each local element, as described in the previous section. The following protocol guarantees consistent data during (potentially concurrent) enqueue, read and dequeue operations.

1) Enqueue: The enqueue operation always starts traversing the queue at the first element in the local queue of the actuator issuing the alarm. It checks each element whether it is freely available or marked for deletion. In the second case, the element can be used, if no read action is performed on the element.

The protocol steps are illustrated in Figure 3. At first, the enqueue operation tries to reset the *del* bit using TAR. If it succeeds, the element is marked for deletion. Then the enqueue function must decide whether there are still actuators reading this element or not. If there are readers still attached to this element (*r_cnt* $>$ *0*), the enqueue operation sets the *del* bit again using TAS and returns for further traversal of the queue. However, if *r_cnt* is equal to zero, the new alarm data is written to the element. After that, the alarm data is released by the actuator. Therefore, the *use* bit is set and the *del* bit as well as the *init* bit are reset using one assignment to the binary mask, which stores this three bits. This step represents the linearization point [9] LP_1 of the enqueue protocol, because at this point the changes become visible for all of the other

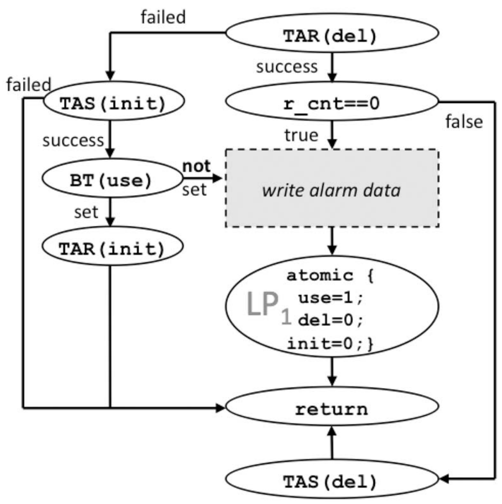


Fig. 3. Flow chart of the enqueue operation

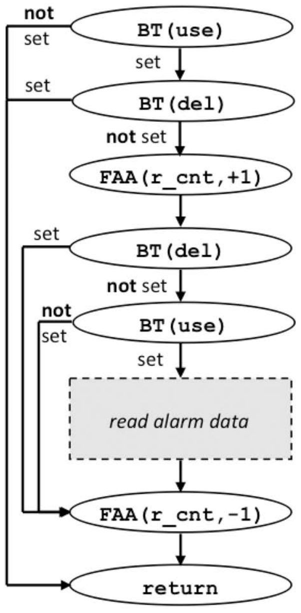


Fig. 4. Flow chart of the read operation

actuators. Now, the changes are active and become visible to the readers, which are notified using the observer pattern [10].

If the reset of the *del* bit fails at the beginning, the actuator tries to set the *init* bit using TAS. If TAS fails, a concurrent enqueue or dequeue operation is presently using this element and the enqueue operation returns. If TAS succeeds, the actuator checks (BT) as to whether the element's *use* bit is set. If it is not, the alarm data gets written without interleavings of other operations on this dedicated slot element. The actuator then releases the written data by setting the *use* bit and resetting the *del* bit as well as the *init* bit — using an assignment as mentioned above — and returns. If the *use* bit is set, the element is still 'in use' and the actuator resets the *init* bit via TAR and returns. Then further traversal of the queue is required. If the traversal of the queue ends up again at the first local element again without finding a free element, then the alarm is lost.

2) Read: Reading an element's data requires consistent alarm data. This is ensured by using the counter *r_cnt* for each element, it is incremented using FAA at the beginning of each read operation and decremented using FAA after the operation has been performed. This counter ensures that the element will not be overwritten by an enqueue operation until the last client has finished reading the data. Data is only read if the *del* bit is not set and the *use* bit is set. The first bit-tests on the use and del bits ensure that continual read operations cannot influence a concurrent enqueue operation on an element that has already been removed. The read protocol is shown in Figure 4.

3) Dequeue: The dequeue function deletes alarms from the queue. Therefore, dequeue has to traverse the queue in order to find alarms to be deleted. To decide whether an alarm should be removed or not, dequeue compares its parameter values, e.g., alarmID or stationID, with the data stored in *alarm_data*.

The following protocol steps are visualized in Figure 5. First of all, the dequeue operation checks the *use* bit. If it is set, the dequeue operation checks whether the deletion criterion matches its parameter values. If the parameter values do not match, the dequeue operation returns. In the case of a match, the actuator tries to set the *init* bit using TAS. If it fails, the element is reserved by a concurrent enqueue or dequeue operation. Therefore, the dequeue operation returns. If the *init* bit is successfully set, the actuator has to check again for a matching deletion criterion. This is because concurrent operations can, in the meantime, potentially change the element's data. If the parameter values do match again, the actuator atomically resets the *use* bit and sets the *del* bit using one assignment to the binary mask, which stores these two bits. This represents the linearization point LP_2, because at this point the state changes from 'in use' to 'removed'. Now, no further action is needed and the dequeue function returns. If the parameter values no longer match, the actuator resets the *init* bit via TAR and returns.

C. Verification

We implemented our protocol in PROMELA and validated our state transitions with the SPIN model checker [11]. All possible states for each alarm element are depicted in Table II.

We will now describe the possible states 1. to 16. of an alarm element and discuss why invalid states cannot occur. Firstly, there is an inital state 1., where all status bits and the read counter *r_cnt* are at zero. If an element is in the inital state, the element is free. Read operations on free elements were rejected by the first bit-test and hence cannot change the state of the element. Hence, also state 9. cannot occur. Dequeue operations on a free element cannot change the state of this element, because the first bit-test will fail, if the *use* bit is not set. Once, an alarm element has left the inital state, the inital state cannot occur twice.

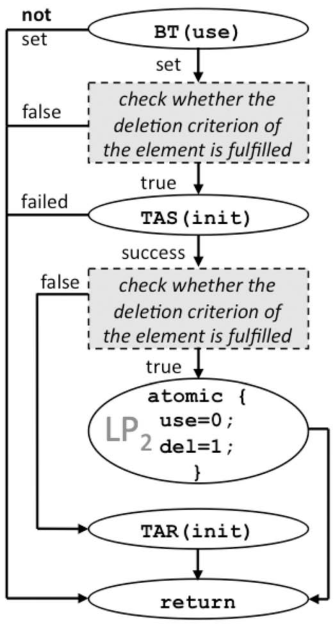

not set
BT(use)
set
false
check whether the deletion criterion of the element is fulfilled
true
failed
TAS(init)
success
false
check whether the deletion criterion of the element is fulfilled
true
atomic { use=0; del=1; }
LP2
TAR(init)
return

Fig. 5. Flow chart of the dequeue operation

	r_cnt	*init*	*use*	*del*	short state description
1.	0	0	0	0	inital state
2.	0	0	0	1	*invalid state*
3.	0	0	1	0	element (elem.) in use
4.	0	0	1	1	*invalid state*
5.	0	1	0	0	elem. in process by enqueue/dequeue
6.	0	1	0	1	elem. is free for overwriting
7.	0	1	1	0	before LP_2 in dequeue
8.	0	1	1	1	*invalid state*
9.	≥ 1	0	0	0	*invalid state*
10.	≥ 1	0	0	1	*invalid state*
11.	≥ 1	0	1	0	reading an element's alarm data
12.	≥ 1	0	1	1	*invalid state*
13.	≥ 1	1	0	0	elem. in process by enqueue/dequeue
14.	≥ 1	1	0	1	readers still reading a removed elem.
15.	≥ 1	1	1	0	before LP_2 in dequeue
16.	≥ 1	1	1	1	*invalid state*

TABLE II
STATES OF AN ALARM ELEMENT

The invalid state 2., where only the *del* bit is set, cannot occur because if a dequeue operation was successfully performed, the element is always free for overwriting — this is state 6. If the atomic step LP_2 (see Figure 5) of the dequeue operation is performed, the *init* bit is always set. Hence, this state cannot occur.

If an enqueue operation was successfully executed, the element's *use* bit is always set — this is state 3.

The invalid state 4. cannot occur because the atomic linearization points LP_1 and LP_2 (see Figures 3, 5) always set the *use* and *del* bit to different values. LP_1 and LP_2 cannot be interleaved as the *init* and *del* bits ensure that parallel enqueue and dequeue operations are mutually excluded (at the element level).

If an element has state 5., an enqueue or dequeue operation has reserved this element by setting the *init* bit. An element has state 6., if a dequeue operation was successfully performed. If a dequeue operation is located in front of the linearization point LP_2, the *init* and *use* bit are set and the *del* bit must be zero — this is state 7.

State 8. is invalid. It cannot occur for the same reason that state 4. cannot occur.

If actuators are presently reading the alarm data (*r_cnt* $\geq$ *1*), the element's state is 11.

If an element is removed (*del=1 & init=1 & use=0*) and readers are still attached to this element (*r_cnt* $\geq$ *1*), the element resides in state 14.

The descriptions for the remaining states are identical to the explanations above.

D. Evaluation

For a preliminary evaluation of our protocol, we measured the response time in processor cycles against the fill level of the alarm queue. Furthermore, we compared our wait-free solution to the status quo implementation in a sequential test scenario.

1) Test Setup: As hardware environment we used an octa-core PC with two Xeon E5440 quadcore processors, running at a 2.83 GHz clock frequency, 256 KB L1 cache per core for instructions and data, 6 MB L2 cache per core pair (that is 12 MB per CPU) and 1333 MHz FSB.

Today, our RC kernel running on different powerful Intel processors. Hence, we are using two of Intel's E5440 for an appropriate test environment, even though a Xeon processor creates too much heat to integrate it into our target system.

2) Testing Method: We implemented a test environment using Windows XP on the hardware platform mentioned above. In this test environment we triggered highest priority threads on the basis of the multimedia timer with a minimum resolution of 1 ms. We validated our implementation with Intel's Thread Profiler.

To interpret the results shown in the next section, we have to keep in mind that the cores, which execute our periodic threads, do not have to deal with incoming interrupts. This is because we have set the *IntAffinity* boot option in *boot.ini* to force interrupts to the highest numbered execution core, which we do not use. The jitter shown in the next section arises from cache effects. Furthermore, it depends on the strategy as to how changes in the cores' caches are written back (e.g., write-through, write-back). We analyzed the influence of Windows in our test scenarios, which is insignificant.

For collecting our measured values, we used the 64-bit model-specific register RDTSC [12]. Because of the out-of-order execution of the E5440 processors, we had to flush the processor pipeline before reading out RDTSC to get suitable measured values. The CPUID instruction was used to flush the pipeline, which introduces some jitter (min = 220 cycles, max = 284 cycles, σ = 12 cycles, c_v =$\sim$ 0.05 cycles; evaluated with 1,000 test runs) against what is remaining in the pipeline; where σ stands for the standard deviation and c_v quantifies the coefficient of variation.

3) Test Scenarios, Results and Discussion: Figures 6(a) and 6(b) show the results of our analysis. We performed two tests using different scenarios. For every test we use 600 global elements. In the following, we use the acronym $|E_l|$ for the number of local elements and $|E_g|$ for the number of elements for the entire queue.

The first scenario compares the sequential status quo implementation with our wait-free queue. We sequentially enqueued alarms until no free element was left in the queue and measured the response times with increasing fill level. As visualized in Figure 6(a) the status quo solution shows some jitter but has quite a constant response time, which is independent of the fill level. In contrast the new SMP-aware alarm queue shows that the response time scales linearly according to the fill level. Since the queue is always traversed starting at the first local element, with each iteration one more step is needed to find a free element.

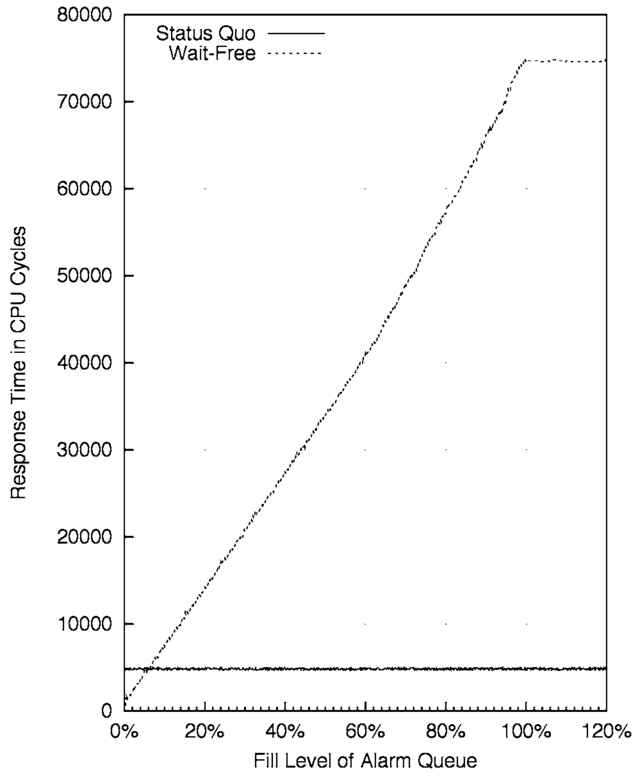
The second scenario shows six concurrent enqueue operations working on our wait-free queue. For this purpose we instantiated six highest priority threads on different execution cores and concurrently enqueued alarms until no free element was left in the queue. We measured the response times with increasing fill level, as shown in Figure 6(b). It can be seen that the enqueue operations have no influence on each other, while they are using their local elements. All six threads operate at about the same speed and hence only trigger alarms in their local queue elements. When the fill level of the queue reaches about 98 percent, the contention on the last remaining free elements increases because a few actuators have exceeded their local queue and a few have not. This leads to the outliers, because now actuators with an exceeded local queue must traverse the entire queue in order to find a free alarm element. Furthermore, the interchange of data through the cache hierarchies and the main memory induces an additional overhead. All of these increase the response times.

There are also several scenarios for measuring the response times of dequeue operations started from a completely filled queue. Based on its parameter value the dequeue function decides whether an element has to be removed or not, as mentioned in Section III-B.3. Since the dequeue function does not affect the alarm reaction, we will not consider this use case any further.

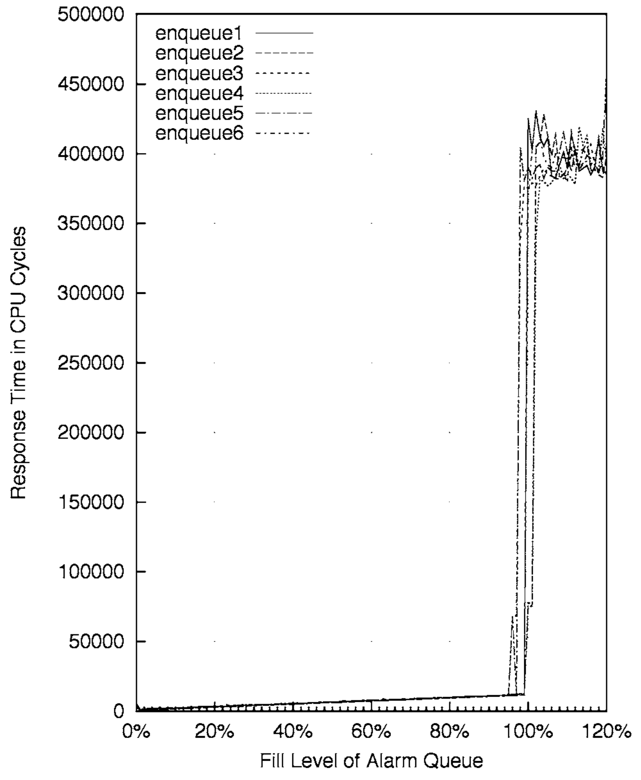
E. Conclusion

The results in the previous section show that our wait-free solution scales linearly with the fill level. This of course depends on the fragmentation of the queue. As Jayanti et al. [1] have already argued for their implementation of a wait-free (FIFO) queue, the use of helping schemes also interchanges time and space complexity — their solution also scales linearly. Due to fact that we did not choose a complex algorithm to traverse the queue, there is still room for improvements in terms of response time and determinism in our solution.

Our algorithm mostly fulfill the requirements described in Section II-B. We checked the state transitions of our protocol



(a) Sequential test case to compare the status quo alarm queue to our wait-free solution using only one highest priority thread; $|E_l| = 600$, $|E_g| = 600$



(b) Test case of concurrent enqueue operations with six highest priority threads running on different execution cores; $|E_l| = 100$, $|E_g| = 100 * 6 = 600$

Fig. 6. Results of our two test scenarios

using the SPIN model checker [11]. Furthermore, our protocol does not suffer from priority inversion, deadlocks, livelocks and starvation since this is inhibited by the wait-free property [2]. Because we are dealing with a system operating under hard real-time constraints an upper limit for the execution time is essential. This is guaranteed by the fact that incoming alarms can be rejected without influencing the real-time system if the alarm queue is full. But jitter is still an issue with our solution since it can only be determined in an experimental manner and also depends on the fill level of the queue. Moreover, the response times for enqueue operations on a (nearly) filled queue are still too expensive for our use case.

IV. Helping Queue

A. Motivation

As the response times directly depend on the fill level of the alarm queue, there are some issues which we take into consideration in this section. If the fill level varies, the response times of enqueue operations also vary. Moreover, the fragmentation of the queue is also an issue and also leads to varying response times because of the linear traversal of the queue. If the alarm queue is nearly full, the response times increase exponentially because of contention regarding the last free elements. And the response times of enqueue operations on a full queue take up a lot of time, since an actuator must traverse the entire queue just to identify that it is full. In Figure 7, a situation is illustrated, where six actuators performs 80 percent enqueue operations and 20 percent dequeue operations with groups of alarm elements. On the average, one performed dequeue operation removes 3.5 percent of all of the used elements. The response times on the y-axis depict the response times for enqueue operations only. The downward outliers come from performed dequeue operations, which remove elements in favor of further enqueue operations. This means that the costs for traversal of the alarm queue by further enqueue operations are influenced by dequeue operations.

To a certain extent, the jitter and the response times for enqueue operations on a nearly full queue are a problem. This is because the enqueue operation delays the respective alarm reaction. Therefore, in this section we present a wait-free helping queue mechanism on the basis of our solution described in Section III. Our helping queue mechanism minimizes the costs for traversing the queue in order to find a free element as well as the varying response times of enqueue operations.

B. Idea

Each local alarm queue is separated into two halves and for each half of the local alarm queue we save the number of free elements in *free_at_top* and *free_at_bottom* respectively. The size of both halves can differ by one alarm element, if the size of the local alarm elements is an odd number. We use a statically allocated array with alarm elements for each actuator and abstain from connecting the arrays of alarm elements among each other in a circular structure, as illustrated in Figure 8. Instead of this, one helping queue element is assigned to each

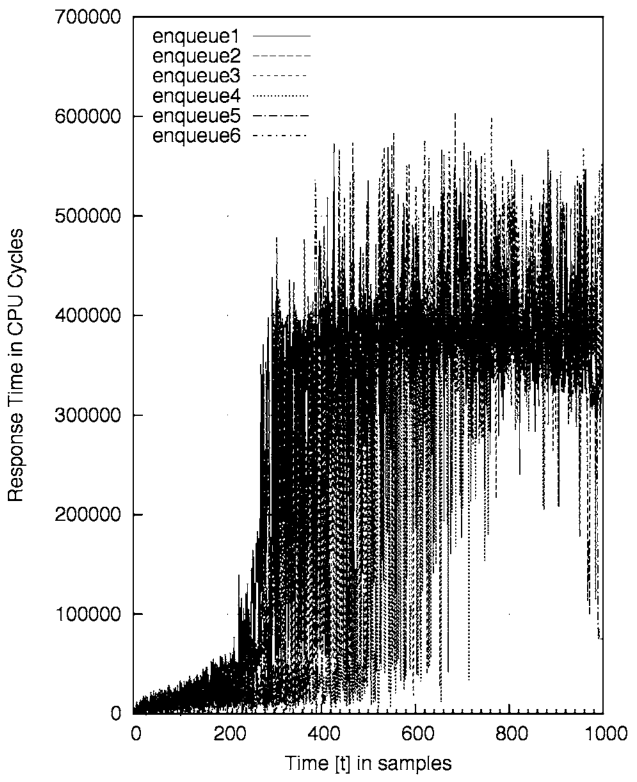


Fig. 7. Test case where six actuators perform 80 percent enqueue and 20 percent dequeue operations; $|E_l| = 100$, $|E_g| = 100 * 6 = 600$

actuator, which stores a *local_queue** pointer to its local alarm element array, a constant integer variable *local_middle* for the first element of the second half of the alarm array, the integer variables *free_at_top* and *free_at_bottom* as described above and a pointer *next** to the helping queue element of the next actuator. The next pointer of the last actuator's helping queue element statically points to the first actuator's helping queue element. Hence, we have a helping queue with a static circular structure that can be traversed using their next pointers.

C. Protocol

To enqueue an alarm, the actuator must traverse the helping queue in order to find a free alarm element. Each actuator starts traversing the helping queue at its local helping queue element to ensure our local preferences and to avoid contention. For each helping queue element the actuator consecutively checks the two counters *free_at_top* and *free_at_bottom*. If both counters are less or equal to zero, the actuator continues to traverse the helping queue by using the next pointer. If the helping queue is traversed and ends up at the first element again without finding a free element, the alarm is lost. If the actuator found a counter which is greater or equal than one, then it has found a potentially free alarm element. Then the actuator decrements this counter by using FAA. If the new value of the counter, which is returned by FAA, is greater or equal to zero, then the actuator has found a free element. The decremented counter ensures that the actuator has reserved

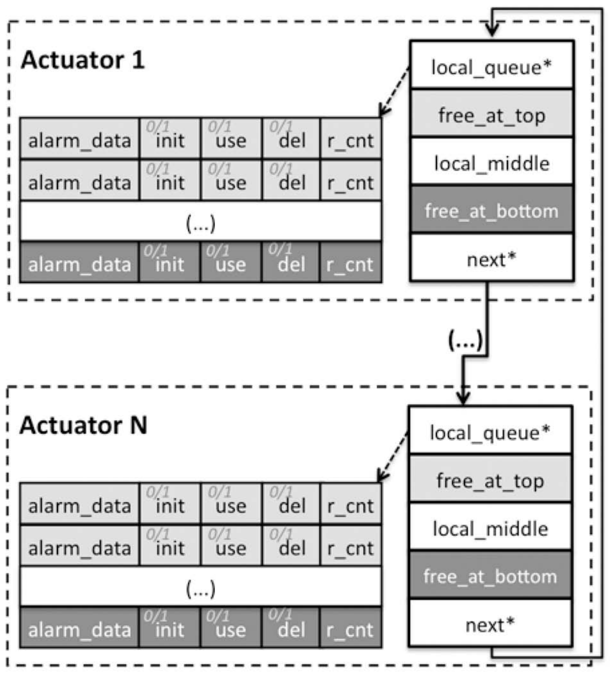


Fig. 8. Structure with our wait-free helping queue

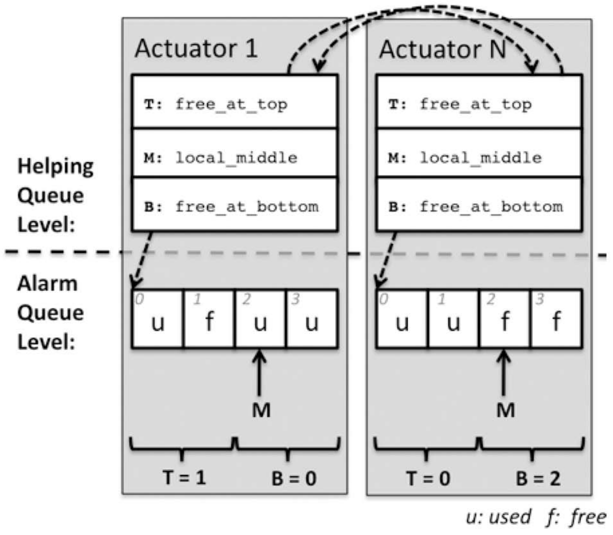


Fig. 9. A possible state of the alarm and helping queue.

one of the free elements on the respective half of the alarm element array. If the returned value of FAA is less than zero, the actuator must increment the counter back via FAA and continues traversing the helping queue. If the actuator has decremented a counter and the return value of FAA is greater or equal one, then it must traverses the half of the respective alarm queue as described in Section III to find the free alarm element.

To ensure that the number of free elements in the alarm queue is represented by the respective counters of the helping queue elements, the dequeue operation described in Section III-B.3 must increment the respective counter using FAA <u>after</u> an element is removed by atomically setting the *use* bit to zero and the *del* bit to one. This is LP_2 in Figure 5. This means that after a dequeue operation has removed an alarm element at the alarm queue level, the alarm element is released at the helping queue level (see Figure 9).

D. Validation

Since we are using FAA to increment and decrement the two counters *free_at_top* and *free_at_bottom*, a race situation cannot occur at these state transitions. Each counter represents the number of free elements for a dedicated half of a local alarm array. For enqueue operations we use the counters to decide whether there is a free alarm element or not. For a successful dequeue operation we increment the respective counter after the linearization point LP_2 has been executed. This is because new enqueue operations can potentially enter a half of the list as they decremented the respective counter and FAA returns the new value with a number greater or equal zero. The alarm element must have already been removed so that the enqueue operation can use it. Therefore, we must remove the element by executing LP_2 before we increment a helping queue counter. Hence, the step of atomically incrementing a helping queue counter in the dequeue function is the linearization point for a helping queue element, because at this point we release the removed alarm element for further enqueue operations.

The helping queue procedure ensures that if an enqueue operation has decremented a helping queue counter and FAA returns a new value which is greater than zero, then it is guaranteed that there is at least one free element for the enqueue operation at the respective half of the alarm array. A possible scenario is depicted in Figure 9.

If an enqueue operation traverses the entire helping queue without finding a counter which is greater or equal to one, then the alarm is lost.

E. Evaluation

In this section we discuss whether we have to take into consideration the issues mentioned at the beginning in Section IV-A. We minimized the time involved for traversing the alarm queue as described in Section III as we integrate two counters for each helping queue element. These counters count the free alarm elements in the local alarm array. Additionally, traversing the helping queue elements decreases the time for traversing the entire alarm elements. As depicted in Figure 10, we also reduce the contention on the last remaining free elements. On the other hand, traversing the helping queue also results in additional overhead.

Figure 10 shows that actuators initially fill the first part of their local alarm arrays until the fill level of the entire alarm elements reaches 50 percent. At this point the helping queue protocol realizes that the first halves of the alarm arrays are full (*free_at_top == 0*) and now starts to use the second halves of the alarm arrays as the *free_at_bottom* counters are still greater than zero. The significant reduction in the response times of 50 percent fill level of all alarm elements mainly comes from

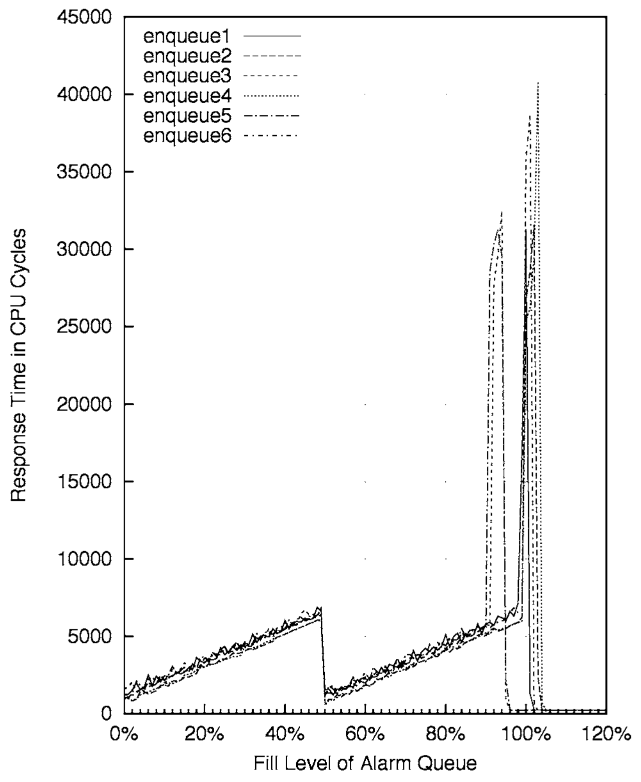


Fig. 10. Test case with helping queue, where six actuators perform enqueue operations; $|E_l| = 100$, $|E_g| = 100 * 6 = 600$

the time involved with traversing the alarm array as described in Section III. This is because now every enqueue operation starts to traverse the alarm array starting from *local_middle*. If the fill level of all alarm elements reaches about 95 percent, contention on the counters as well as the alarm elements occurs. After the fill level reaches about 100 percent, the average response times reach 226 cpu cycles — as shown in Figure 10, which are required to traverse the full helping queue once.

F. Further Partitioning

Our approach to partition each local alarm queue is not limited to two parts. It is straightforward to partition each local queue up to $|E_l|$ parts (in the most finely-grained case). But, it is a tradeoff between reducing the relatively high time it takes to traverse the alarm queue and the additional (relatively low) costs for traversal the helping queue. In the worst case, if we partition each local alarm queue in $|E_l|$ parts, which is element-wise, the costs for traversing the helping queue are approximately identical to those when using the alarm queue as described in Section III without helping queue.

G. Optimization of the Search Path

In this section we show a mathematical approach to determine the optimal number of partitions. It is assumed that the number of actuators A is known as well as the number of alarm elements $|E_g|$ that are needed. For our use case $A = 6$ and $|E_g| = 600$. Neglecting the hardware costs at this point, we further assume that the costs for traversing one element at alarm queue level are 1 and the costs for checking one partition for free alarm elements at helping queue level are also 1.

Additionally, we establish the following acronyms for the two unknowns:

- p: Number of partitions for each local alarm queue
- $|E_p|$: Number of alarm elements per partition

It is imperative:

$$A * p * |E_p| = |E_g| \quad (1)$$

As equation 1 shows, $|E_g|$ is the worst case search path to find a free alarm element without using a helping queue. The worst case is, if there is only one free alarm element left and we must check every alarm element to identify whether it is free.

Now, we must determine the unknowns p and $|E_p|$ so that we minimize the search path for finding a free alarm element. The costs for this worst case search path with helping queue are $\Omega : A * p + |E_p|$, where $A * p$ stands for the worst case costs for checking every partition for free elements at the helping queue level and $|E_p|$ stands for the worst case costs for traversing the alarm elements inside the partition.

The minimum of the worst case search path is the equation of the costs for checking every partition for free alarm elements $A * p$ and the number of alarm elements per partition $|E_p|$ (see equation 2).

$$A * p = |E_p| \quad (2)$$

Solving equation 1 for $|E_p|$ yields:

$$|E_p| = \frac{|E_g|}{A * p} \quad (3)$$

Inserting the result of equation 3 into equation 2, we get:

$$A * p = \frac{|E_g|}{A * p} \quad (4)$$

Now, we can also solve equation 4 for p:

$$p = \lfloor \sqrt[2]{\frac{|E_g|}{A^2}} \rfloor \quad (5)$$

With this result of equation 5, we can determine p and $|E_p|$. The following steps are inevitable for our particular use case:

1) $p = \lfloor \sqrt[2]{\frac{600}{6^2}} \rfloor = 4$
2) Insert p in equation 3: $|E_p| = \frac{600}{6*4} = 25$
3) Determine the minimal worst case costs for $p = 4$ and $|E_p| = 25$: $\Omega = 6 * 4 + 25 = 49$[1]

If we shift the values of p and $|E_p|$, e.g., $p = \{5, 3\}$ and $|E_p| = \{20, 33\}$, the costs increase $\Omega = \{50, 51\} > 49$. Note that for $p = 3$ and $|E_p| = 33$ there are only 594 alarm elements available.

[1]see also Figure 13

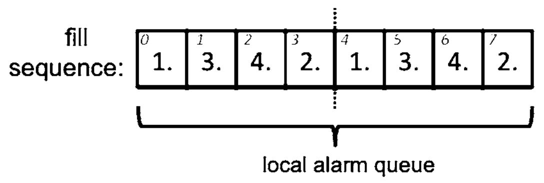


Fig. 11. Improved traversing sequence in turn from left/top and right/bottom for each half of the local alarm queue

H. Conclusion

Our wait-free helping queue approach is an extention for our wait-free queue mentioned in Section III. It is an improvement relating to determinism and performance. Indeed, the response times still depend on the fill level of the alarm queue since we must traverse a half local alarm queue in order to find a free element, but the costs therefor were reduced dramatically. Additionally, we significantly reduced the jitter induced by varying fill level. By reducing the contention by dividing each local alarm queue into two halves, we have also reduced the cache-based outliers. Consequently, we also decrease the costs for enqueue operations, if the alarm queue is full.

Moreover, the helping queue mechanism allows us to divide each local alarm queue into arbitrary parts. This means that we can raise the costs for traversing the helping queue and reduce the costs for traversing the alarm queue and hence minimize the accumulated costs for enqueue operations.

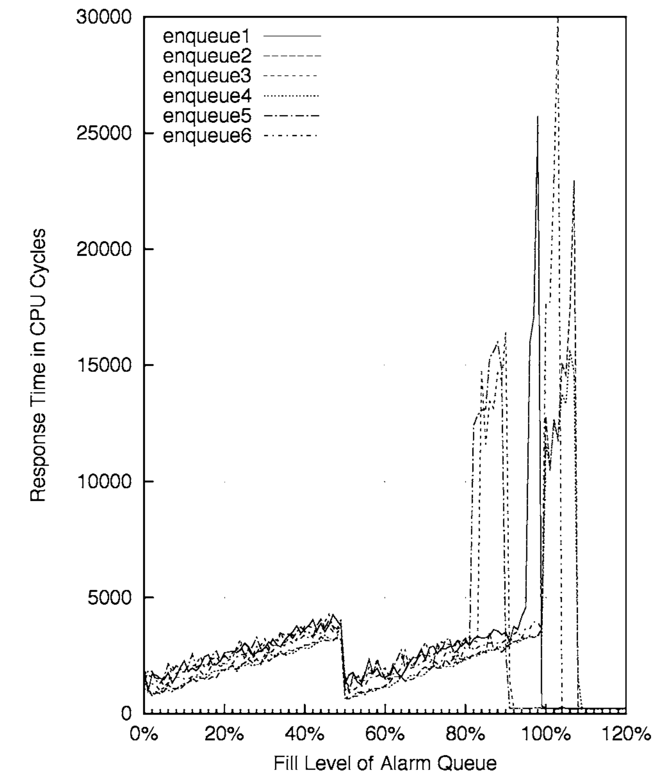


Fig. 12. Test case with helping queue and improved traversal strategy; six actuators perform enqueue operations; $|E_l| = 100$, $|E_g| = 100 * 6 = 600$

V. Improved Traversal Algorithm

A. Motivation

In Section IV we have shown a mechanism to modify the worst case execution time to find a free alarm element. This is done by reducing the search path. In this section we discuss a simple algorithm to traverse the alarm queue to find a free element. This does not change the worst case execution time, but it reduces the average overhead for traversing the alarm queue.

B. Idea and Protocol

The idea is straight forward. Each actuator performing an enqueue operation traverses the alarm queue in turn from the left and the right in order to find a free element. Note that the alarm queue is traversed after the actuator has found a free element by using the helping queue as described in the previous section. Assuming that only one actuator enqueues alarms, then each half of the local alarm queue is filled as illustrated in Figure 11. We do not make assumptions about the preference of actuators, which half of the local alarm queue is used first — which means that we have numbered the filling steps from 1. to 4. On the average this traversal algorithm leads to a reduced traversal overhead for the alarm queue, since actuators firstly use their local preferences and now filling each half of the local alarm queues from the left and from the right in turn. However, in the worst case, an actuator must nevertheless traverse the complete half of the local alarm queue in order to find a free element. This represents the same effort as incurred when using a linear traversal strategy. This improved traversal algorithm needs an additional bit per actuator to determine the traversal direction; this does not involve additional space or time overhead.

C. Evaluation

Figure 12 shows the identical concurrent test scenario as described in Section IV-E with our simple improved traversal algorithm. If the fill level of the entire alarm queue reaches 50 percent, which means that the first halves of the local alarm queues are full, further enqueue operations of actuators use the second half of each local alarm queue. The strategy as to how the elements of each half of the local alarm queue are alternately used occurs from the left/top and from the right/bottom. As we can see in Figure 12, until the alarm queue reaches an fill level of approximately 81 percent, actuators only use their local alarm elements for triggering alarms. After the alarm queue has been filled to about 81 percent, contention on the last free alarm elements increases. Since we do not use the linear traversal mechanism as described in Section IV the contention is obviously a little bit lower. Now, the accumulated response time of each actuator has been halved as illustrated in Figure 10. For this reason the possible different runtimes of each actuator are of more significance; hence we moved the cache-based outliers a little feed forward.[2] However, the

[2]see also Figure 13

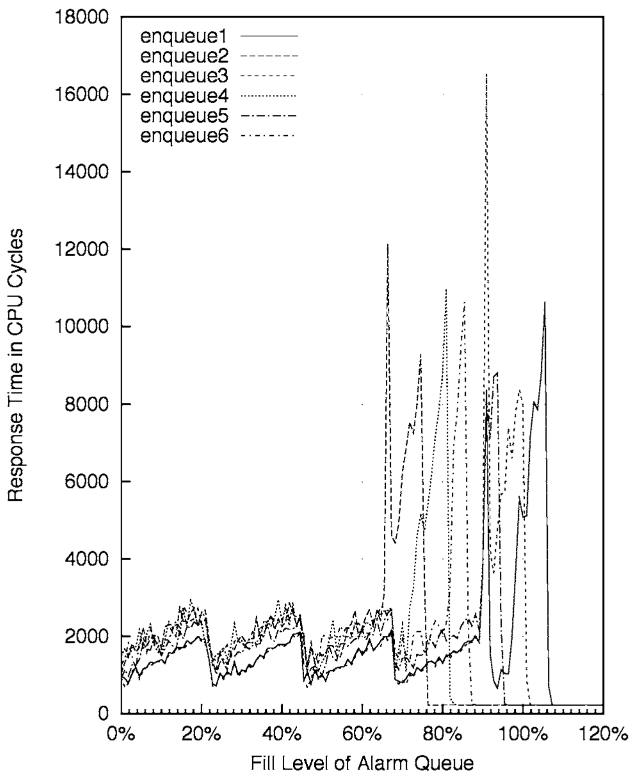


Fig. 13. Test case with helping queue, improved traversal algorithm and local alarm queue divided into four parts, where six actuators perform enqueue operations; $|E_l| = 100$, $|E_g| = 100 * 6 = 600$

average response times are reduced by approximately 50 percent in the best case compared to Section IV.

Actuator *enqueue2* and *enqueue3* are obviously slower than the others, because their response times fall to the overhead of the helping queue protocol after their last outliers by approximately 91 percent.

D. Conclusion

In real-time systems developers must ensure that resources are not exhausted. In our RC kernel the alarm queue has approximately 20 percent above the capacity of the most common failure scenarios. New incoming alarms are only rejected under worst case conditions. In this section we had to take into consideration the average response time for triggering alarms. This ensures that the respective alarm reaction — which is triggered on the basis of the return values of the processing stations — has the shortest possible delay. In comparision to a linear traversal mechanism as described in the previous section, this improved traversal strategy reduces the average response times by approximately 50 percent in the best case. In the worst case we do not perform as well as a linear traversal algorithm.

If we divide each local alarm queue into more than two parts (e.g., four parts as illustrated in Figure 13), we can reduce the costs of our approach for a wait-free alarm queue so that on the average it is faster than today's sequential solution as illustrated in Figure 6(a). The cache effects are caused by the hardware design (e.g., cache synchronization over 2nd level cache or over main memory) and cannot be completely eliminated if contention occurs.

VI. Related Work

Lamport developed the first single writer and multiple reader algorithm without locks in [13]. However, his approach is not wait-free [2] and hence does not satisfy our real-time requirements. Lock-freedom only guarantees that always *some* process completes its operation within a bounded number of steps on a lock-free data structure. This can lead to starvation of other processes. Also other nonblocking approaches of (FIFO) queue algorithms, such as [5], [14]–[17], only guarantee lock-freedom.

Furthermore, the obstruction-free approach from Herlihy et al. in [18] also does not satisfy our real-time requirements. Obstruction-Freedom only guarantees progress 'in isolation' of all other processes.

There are only a few wait-free queues, such as [1], [19], [20]. Tsigas et al. — who proposes in [19] wait-free queue class implementations for synchronization of real-time and non-real-time threads in an unidirectional manner. Therefore their generic queue implementations use the priorities of the scheduler to guarantee progress. In [20] David presented a single enqueuer, multiple dequeuer approach of a wait-free queue implementation. Jayanti et al. showed in [1] a multiple enqueuer and single dequeuer scenario for wait-free queues and stacks. However, our scenario requires multiple enqueuers and multiple dequeuers, which is not satisfied by [1], [20]. Furthermore, FIFO queues do not address our concerns.

VII. Summary

We designed a wait-free solution of a queue to handle multiple concurrent enqueue and dequeue operations using local preferences to reduce contention. Additionally, we presented a helping queue mechanism to significantly reduce the costs to traverse the entire queue in order to find a free element. Furthermore, we presented a simple traversal algorithm, which — in the best case — results in significantly shorter response times.

All of this allows us to extend the alarm handling in our RC to SMP systems without requiring much further effort. The wait-freedom property ensures that developers do not have to worry about lock orders to avoid deadlocks and additional protocols to avoid unbounded priority inversion scenarios. The wait-freedom property also guarantees that each actuator makes progress at every time.

This makes our approach the best known and practical solution for an unsorted thread-safe queue for multiple enqueuers, multiple dequeuers and multiple readers.

References

[1] P. Jayanti and S. Petrovic, "Logarithmic-time single deleter, multiple inserter wait-free queues and stacks," *in Proc. of 25th International Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 408–419, 2005.

[2] M. P. Herlihy, "Wait-free synchronization," *in ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 124–149, January 1991.

[3] H. Sutter, "The free lunch is over," *in Dr. Dobb's Journal*, vol. 30, no. 3, March 2005.

[4] M. M. Michael, "ABA prevention using single-word instructions," IBM Research Division, RC23089 (W0401-136), Tech. Rep., January 2004.

[5] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems," *in Proc. of the 13th ACM Symposium on Parallel Algorithms and Architectures*, pp. 134–143, 2001.

[6] R. Rajkumar, L. Sha, and J. P. Lehoaky, "Real-time synchronization protocols for multiprocessor," *in Proc. of Real-Time Systems Symposium*, pp. 259–269, 1988.

[7] V. Yodaiken, "Against priority inheritance," FSMLabs, Tech. Rep., July 2002.

[8] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," *in Proc. of Symposium on Principles of Distributed Computing*, pp. 151–160, 1994.

[9] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trnasactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[10] E. Gamma, R. Helm, and R. E. Johnson, *Design Patterns. Elements of Reusable Object-Oriented Software*. Longman, Amsterdam: Addison-Wesley, March 1995.

[11] G. J. Holzmann and D. Bošnački, "The design of a multi-core extension of the SPIN model checker," *in IEEE Trans. on Software Engineering*, vol. 33, no. 10, pp. 659–674, 2007.

[12] J. Muir, "Using the RDTSC instruction for performance monitoring," Intel Corporation, Tech. Rep., 1997. [Online]. Available: http://cs.smu.ca/ jamuir/rdtscpm1.pdf

[13] L. Lamport, "Concurrent reading and writing," *in Communications of the ACM*, vol. 20, no. 11, pp. 806–811, 1977.

[14] J. D. Valois, "Lock-free linked lists using compare-and-swap," *in Proc. of the Fourteenth ACM Symposium on Principles of Distributed Computing*, August 1995.

[15] M. M. Michael and M. L. Scott, "Simple, fast and practical non-blocking and blocking concurrent queue algorithms," *in Proc. of the 15th ACM Symposium on Principles of Distributed Computing*, pp. 267–275, 1996.

[16] D. Fober, Y. Orlarey, and S. Letz, "Optimised lock-free FIFO queue," TR010101, GRAME – Computer Music Research Laboratory, Tech. Rep., January 2001.

[17] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," *in Lecture Notes in Computer Science*, pp. 300–314, 2001.

[18] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," *in Proc. of the 23rd International Conference on Distributed Computing Systems*, pp. 522–529, 2003.

[19] P. Tsigas and Y. Zhang, "Efficient and simple implementations of the wait-free queue classes of the real-time specification for java," 2002-01, Department of Computer Science, Chalmers University of Technology, Tech. Rep., 2002.

[20] M. David, "A single-enqueuer wait-free queue implementation," *in International Symposium on Distributed Computing*, pp. 132–143, 2004.