Avoiding Unbounded Priority Inversion in Barrier Protocols Using Gang Priority Management

Harald Röck
University of Salzburg
hroeck@cs.uni-salzburg.at

Joshua Auerbach
IBM Research
josh@us.ibm.com

David F. Bacon
IBM Research
dfb@watson.ibm.com

Christoph M. Kirsch*
University of Salzburg
ck@cs.uni-salzburg.at

ABSTRACT

Large real-time software systems such as real-time Java virtual machines often use barrier protocols, which work for a dynamically varying number of threads without using centralized locking. Such barrier protocols, however, still suffer from priority inversion similar to centralized locking. We introduce *gang priority management* as a generic solution for avoiding unbounded priority inversion in barrier protocols. Our approach is either kernel-assisted (for efficiency) or library-based (for portability) but involves cooperation from the protocol designer (for generality). We implemented gang priority management in the Linux kernel and rewrote the garbage collection safe-point barrier protocol in IBM's WebSphere Real Time Java Virtual Machine to exploit it. We run experiments on an 8-way SMP machine in a multi-user and multi-process environment, and show that by avoiding unbounded priority inversion, the maximum latency to reach a barrier point is reduced by a factor of 5.3 and the application jitter is reduced by a factor of 1.5.

1. INTRODUCTION

The execution of multi-threaded programs is often separated into phases in which the computation differs in some important way. At the entry of each phase, a barrier ensures that a set of threads has completed a phase before continuing to the next. In other words, a barrier ensures that no thread advances beyond a certain point before all other threads in the set have arrived at that point. A barrier protocol determines which threads participate in a barrier and coordinates their progress.

Barrier protocols have not received a lot of attention in the real-time literature. However, recently developed real-time Java VMs, e.g. [18], use barrier protocols to coordinate the execution of application threads and system functionality such as garbage collection, logging, and software loading. A Java program executing in a Java VM, for example, switches regularly between a phase of running application threads that potentially allocate memory, and a garbage collection phase that frees memory not referenced anymore. The transition between these phases is managed by a barrier, which guarantees that no application thread is allocating memory while the garbage collector is running. If the Java program uses threads with real-time deadlines, the VM has to make sure that the garbage collection phase and barrier protocol is not interfering with the timely execution of the application threads.

In addition to real-time Java VMs, there is a trend towards the use of middleware to support real-time programming [25, 31]. Such real-time middleware systems include real-time CORBA implementations [28, 27] and real-time CLI environments [12]. More recently, even real-time frameworks and middleware systems are implemented on top of Java [26, 24]. In addition to managing application threads with real-time deadlines, middleware software provides system functions that usually impose some invariant on the application threads, which could involve a barrier protocol.

The logic and actual implementation of a barrier protocol is usually application-dependent, but barrier protocols follow a certain pattern. A barrier uses some shared state, which is visible to all participating threads. The initiating thread, which could be a signal handler or an asynchronous alarm thread, sets an indication in the shared state that a barrier is active. The next time a thread reaches a barrier point and notices that the barrier is active, it marks itself to be at the barrier point and then blocks on the barrier's resumption condition. Additionally, the last thread reaching the barrier marks the barrier state as completed, and posts a completion condition before it blocks.

Application-dependent aspects of barrier protocols include at least the following. First, there can be arbitrary logic linking the completion and resumption conditions. In the simplest case these are the same, meaning that the arrival of the last thread immediately releases all threads. In other cases, the completion condition releases a different set of threads than those blocked on the resumption condition and it is up to this second set of threads to release the original ones that remain blocked on the resumption condition. Second, there are numerous ways in which the set of threads that participate in each execution of the barrier can be determined. In the simplest case, this set is statically determined, but in other cases the membership can only be determined dynamically at runtime when the barrier commences. Hence, threads participating in the barrier have to cooperate.

*Supported by a 2007 IBM Faculty Award, the EU ArtistDesign Network of Excellence on Embedded Systems Design, and the Austrian Science Fund No. P18913-N15.

JTRES'09, September 23-25, 2009 Madrid, Spain

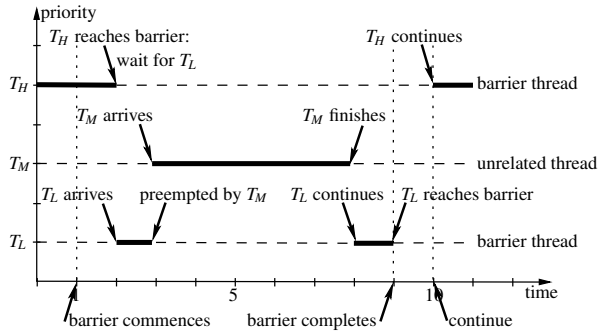


Figure 1: Priority Inversion in Barriers

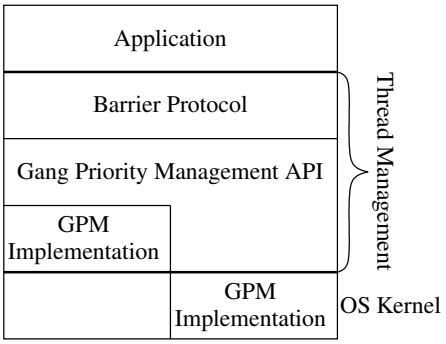


Figure 2: Barrier and GPM Software Stack

A priority inversion occurs if the threads participating in the barrier protocol have different priorities (high and low) and additional unrelated threads with a medium priority are running on the same system. Figure 1 depicts a scenario with three threads running at different priorities: a high-priority thread T_H, a medium-priority thread T_M, and a low-priority thread T_L. The threads T_H and T_L are participating in the barrier protocol, whereas T_M is not. First, the high-priority thread T_H runs. At time instant 1 the barrier is started, which T_H recognizes at time instant 2. Now, T_L is scheduled, but then preempted by thread T_M at time instant 3 before it notices that a barrier protocol is in progress. The medium-priority thread T_M delays T_L until time instant 8. At time instant 9, T_L reaches the barrier, and signals its completion. Now, a system method is running that releases the threads blocked in the barrier at time instant 10, at which point T_H is rescheduled and continues. In this scenario, T_M delays the completion of the barrier and, as a consequence, the execution of the high-priority thread T_H by 5 time units. Moreover, the delay introduced by threads not participating in the barrier protocol is unbounded and therefore, barrier protocols suffer from unbounded priority inversion. Section 7 analyzes different barrier algorithms found in the literature.

While it might seem attractive to provide a single barrier protocol implementation that avoids unbounded priority inversion, the application-dependent aspects of barrier protocols would limit the usefulness of such a solution, and it would also fail to generalize to somewhat different protocols like Query Suspend and Resume. Instead, we propose a general mechanism called gang priority management (GPM) to boost priorities of a set of threads (a gang) temporarily and for the boosted threads to subsequently revert their priorities and inform a master thread that they have done so. With GPM it is possible to modify an existing barrier implementation or other coordination protocols to avoid unbounded priority inversion.

Figure 2 shows the barrier and GPM software stack. On top is the application that uses barriers. The barrier protocol is implemented as part of the thread management below and uses the GPM API to control the priorities of threads participating in barriers. Note that the GPM system is not aware of the barrier protocol. The GPM system is either implemented as a library on top of an operating system that supports dynamic priority adjustments, or is incorporated into the OS kernel.

The first contribution of this paper is the GPM system (Section 2). The key property of the system is that the time from boosting a gang's priority until the priority is reverted (e.g. to signal a barrier's completion) is determined by the gang and any threads with higher priorities than the gang, but not by threads with lower priorities. The second contribution is the integration of a generic barrier protocol into the GPM system (Section 3). Based on the above property, our GPM-enabled barrier protocol guarantees that the time from initiating a barrier until the barrier's completion is determined by the threads participating in the barrier and any threads with higher priorities than the participating threads, but again not by threads with lower priorities. The third contribution is a kernel-space (K-GPM) and an alternative library-based (L-GPM) implementation of the GPM system (Section 4). The fourth contribution is a detailed case study of the integration of our GPM-enabled barrier protocol into the garbage collection safe-point barrier protocol of IBM's WebSphere Real Time Java Virtual Machine (Section 5). The fifth contribution is a set of experiments with the code from the case study, which demonstrate that the time from initiating a barrier until the barrier's completion is indeed independent from lower-priority threads resulting in substantially improved application responsiveness (Section 6).

Related and future work are in Sections 7 and 8, respectively. Conclusions are in Section 9.

2. GANG PRIORITY MANAGEMENT

GPM allows a set of threads to have their priorities temporarily boosted, and for the boosted threads to revert to their previous priorities subsequently and explicitly, as well as to inform a master thread that they have done so. The GPM API is listed in Listing 1. There are calls to allocate and deallocate a gang structure, three calls that manage a thread's *passive* membership in the gang, and three calls that provide a priority management mechanism. A thread's *active* membership in a gang (assuming the thread is a passive member) is managed without any call overhead using a control word shared by the thread and the GPM system. The priority boosting is performed only on the active members of a gang. The priority to which the active members are boosted is calculated by considering all members of the gang, including the passive members since any passive member can change to active membership at any time.

A gang is created by invoking `gang_create`, which takes a pointer to the gang attributes as its argument and returns a new gang identifier. Participation of individual threads is enabled using `gang_insert`. Such threads will become passive members of the gang. The arguments of `gang_insert` are the gang identifier, the process identifier of the thread to be inserted, and a pointer to a 32-bit control word that is unique to the thread. The control word jointly represents

```
/* allocate gang structure */
long gang_create(struct gang_attributes *)
void gang_close(long gang)

/* membership API */
long gang_insert(long gang, pid_t thread,
                 uint32_t *controlWord)
long gang_remove(pid_t thread)
int gang_get(pid_t thread)

/* priority API */
long gang_run(long gang, unsigned long mask)
long gang_wait(long gang,
               struct timespec *timeout)
long gang_notify()
```

Listing 1: Gang Priority Management API

the thread's membership status in the gang (passive or active, in the lower 28 bits) and the priority inheritance status of the thread (in the upper four bits). Usage scenarios are presented below. The motivation of the control word is to avoid system call overhead as threads dynamically change their membership status from passive to active and back.

The membership of a thread in the gang is canceled by invoking `gang_remove`. When a thread terminates, it is automatically removed from its gang. A thread can be a member of at most one gang. The identifier of this gang (if any) is returned by `gang_get`. The `gang_close` operation indicates that the gang structures should be destroyed when the gang has no more members.

The `gang_run` operation is invoked by an application-determined thread (not necessarily a gang member) or signal handler. It atomically boosts the priorities of the active members of the gang and marks the priority change in the members' control word. A thread, which may or may not be the one that issued `gang_run`, can then use `gang_wait` to wait for all active members of the gang to invoke `gang_notify` as described below.

The arguments of `gang_run` are the gang identifier and a bitmask, which determines the active members among the members of the gang. A thread is an active member of the gang if the bitwise `and` of the bitmask and the thread's control word is non-zero. Thus it is possible to exclude a thread from the effects of the `gang_run` operation without the overhead of executing a system call. Using atomic compare-and-swap a thread can simultaneously set the bits in its control word and learn its membership status as well as whether a `gang_run` is already in progress.

An invocation of `gang_notify` atomically restores the original priority of the calling thread, modifies the control word to reflect the new state, and discounts the thread for implementing `gang_wait`. It is safe to call `gang_notify` even if the thread was not an active member of the gang at the time instant `gang_run` was invoked. Moreover, the thread will be correctly accounted for regardless of whether it invoked `gang_notify` before or after the supervising thread's invocation of `gang_wait`. Removing a thread with `gang_remove` does the equivalent of `gang_notify` if the thread was an active member of the gang.

The `gang_wait` operation has a timeout argument, and returns upon an invocation as soon as all active gang members have invoked `gang_notify`, or the timeout, if specified, elapsed.

2.1 Gang Priority Inheritance

The `gang_run` operation determines the priority of a gang and temporarily boosts the effective priorities of the active members of a gang. The effective priority of a thread is the priority a scheduler applies when scheduling the thread. A gang's priority is the maximum of the user priorities of all passive and active members of the gang. The user priority of a thread is the priority provided by the thread's application program. For compatibility with the priority inheritance protocol for locking in the Linux kernel, `gang_run` sets the effective priority of each active member to the maximum of the member's lock-inherited priorities and the gang's priority. The lock-inherited priorities of a thread are the priorities the thread temporarily inherited from other threads due to priority inheritance. Note that, in the priority inheritance protocol for locking, the effective priority of a thread requesting a lock (i.e., not its user priority) is inherited to the thread holding the lock, also if the effective priority is a gang's priority. Moreover, the effective priority of an active member of a gang releasing a lock is reverted to the maximum of the member's remaining lock-inherited priorities, if there are any, and the gang's priority. The `gang_notify` operation reverts the effective priority of the invoking thread to the maximum of the thread's lock-inherited priorities, if there are any, and the thread's user priority. The following invariant holds for the GPM system on the Linux kernel.

INVARIANT 1. *Between the time instants when `gang_run` is invoked on a gang and when a thread that is an active member of the gang invokes `gang_notify` the thread's effective priority is the maximum of its lock-inherited priorities and the gang's priority. At any other time, the thread's effective priority is the maximum of its user priority and its lock-inherited priorities.*

From the invariant immediately follows the following property of the GPM system when using a scheduler that always executes the thread with the highest effective priority.

PROPERTY 1. *The duration from an invocation of `gang_run` on a gang until a subsequent invocation of `gang_wait` on the gang returns is bounded by the sum of the uninterrupted execution time each active member of the gang needs to invoke `gang_notify`, and any interference from threads with equal or higher effective priority than the gang's priority.*

Note that any invocation of `gang_run` on a gang after a previous invocation of `gang_run` on the gang but before all active members of the gang have invoked `gang_notify` will fail and not modify the gang's priority.

3. BARRIER AND GPM INTEGRATION

Consider a background system service that examines the call stacks of some threads in the system, e.g., a garbage collector. As an invariant, the service requires for examining the stacks that all threads are at a consistent state and are not modifying their stacks. Before starting the system service, a signal handler initiates a barrier by setting a bit that will be visible to each thread. It then starts the service thread, which immediately blocks on the barrier's completion condition. As each thread reaches a place where it is safe to pause

execution, i.e., its next barrier point, it checks the bit, notices that a barrier is in progress, and blocks on the barrier's resumption condition. Furthermore, when the last thread reaches its barrier point, it posts the completion-condition, which starts the service thread. When the service thread is done, it restarts all blocked threads by posting the resumption condition.

The GPM system avoids unbounded priority inversion when starting the system service as follows. All threads that are potentially examined by the system service are inserted into a gang G, and removed from G on termination. The signal handler initiates the barrier as described above, but also invokes `gang_run` on G before starting the system service thread. When the system service thread starts executing it first invokes `gang_wait` on G to wait for the completion of the barrier. When a thread that is actually (as opposed to potentially) participating in the barrier notices that a barrier is in progress it invokes `gang_notify` just before it blocks on the resumption condition. The difference between actually and potentially participating threads is usually application-dependent. An example is explained below. When the last thread invokes `gang_notify`, the service thread's `gang_wait` returns and the barrier is completed. Note that the GPM mechanism replaces the barrier's completion condition. Using the GPM mechanism as a replacement for the barrier's completion condition is sound as long as the active gang membership accurately reflects the set of threads that are actually (as opposed to potentially) participating in the barrier.

The barrier for the system service is complicated by the fact that some threads may be sleeping or blocked in I/O waits and hence not in a position to notice a barrier's inception. Such threads are therefore only potentially participating in a barrier. This complication can be turned to an advantage, though, because threads that are waiting are not executing and it is safe to examine their stacks. Thus, such threads withdraw their active membership before waiting and regain it just after their wait is complete, but before doing anything that would modify their stack. To maintain correctness, any thread resuming execution must first check if a barrier is in progress and block on the resumption condition if necessary. Any thread that is about to do a sleep or perform I/O must withdraw its active gang membership using an atomic compare-and-swap operation on its control word. If it discovers that a `gang_run` was initiated while it was still a member, it issues `gang_notify` (to ensure correct counting) even though it withdraws voluntarily.

Our implementation of the described GPM-enabled barrier protocol has the following property, which immediately follows from Property 1.

PROPERTY 2. *The duration from the initialization of a barrier until the completion of the barrier is bounded by the sum of the uninterrupted execution time each thread actually participating in the barrier needs to reach its next barrier point, and any interference from threads with an equal or higher effective priority than the highest user priority of any thread potentially participating in the barrier.*

Note that this property bounds the time to reach the completion condition, not the resumption condition (unless they are the same). The barrier is considered complete when its completion condition is signaled.

4. GPM IMPLEMENTATION

We have implemented two GPM versions: a kernel patch applied to the Linux kernel (K-GPM) and a Linux user library that executes entirely in user space (L-GPM).

4.1 Kernel-based GPM Implementation

We implemented K-GPM as a patch to a recent real-time version of the Linux kernel [22]. In K-GPM, the gang identifier is a file descriptor because file descriptors are valid process-wide, as are gangs, and because the kernel provides a well-defined and efficient interface to create, find, and manage file descriptors. Like any other file descriptor, a gang file descriptor should be closed with `close`, which releases the file descriptor associated with the gang. Closing the file descriptor will not deallocate or destroy the gang data structures inside the kernel until all threads are removed from the gang. Additionally, the gang file descriptor supports the standard polling interface `poll`, `select`, and `epoll`. Using `poll`, the gang file descriptor returns the event `POLLIN` (data ready to read) if an invocation of `gang_wait` would immediately return.

Another feature of K-GPM is its integration into other priority boosting mechanisms of the Linux kernel. The K-GPM priority boosting implementation follows the pattern established by already existing priority boosting mechanisms such as RCU priority boosting or traditional priority inheritance protocols for locking. More precisely, the calculation of effective priorities by the Linux scheduler is adjusted to include the gang's priority if the thread is an active gang member and a `gang_run` is in progress. Hence Invariant 1 holds for the K-GPM implementation.

Furthermore, K-GPM requires only one system call to boost the priority of all threads that are active members of a gang, whereas a library implementation such as L-GPM has to execute a system call for each thread. Using K-GPM requires fewer system calls than L-GPM to execute a `gang_run`. However, to manage active and passive memberships with a naive GPM design, K-GPM would end up requiring far more system calls than L-GPM because the kernel needs to know when threads change their membership status. Issuing a system call for each membership change would incur unacceptable overhead. Consequently, our GPM design avoids using a kernel call for this purpose, in favor of a 32-bit control word in user memory that can be changed (using atomic compare-and-swap) by both the application and the kernel.

4.2 Library-based GPM Implementation

The L-GPM version mimics the kernel implementation using the POSIX thread API. It involves no kernel changes and thus works on any Linux kernel by modifying the threads' user priorities using `pthread_setschedparam`. However, the standard Linux kernel does not provide an interface to change the user priorities of a set of threads. Hence `gang_run` in L-GPM iterates through all threads and invokes `pthread_setschedparam` once for each active member of a gang. Moreover, L-GPM needs to maintain some additional state. It saves the original user priority of each thread in order to restore it later. It also needs one lock per thread and one lock to protect the gang, and uses a condition variable to implement the `gang_wait` operation.

Since L-GPM manipulates the user priority of a thread, it is safe to use it on a Linux system where traditional priority inheritance for locking is done in the kernel, and as long as no other mechanism is manipulating user priorities. If any other mechanism manipulates user priorities, the changes have to be synchronized with the

GPM system. In other words, Invariant 1 holds for the L-GPM implementation since priority inheritance for locking is done by the kernel and not by some user-space threading library.

5. REAL-TIME GARBAGE COLLECTION: A CASE STUDY

As a case study of priority inversion in barrier protocols we studied the problem of starting up a garbage collection quantum in IBM's WebSphere Real Time Java Virtual Machine (RT-JVM) [18, 1]. We believe this Java middleware system has characteristics that are likely to be shared by other middleware systems and complex real-time applications. Furthermore, the RT-JVM used in this case study has already been adopted to develop new complex real-time software systems, such as a next-generation U.S. Navy destroyer, or financial and telecommunication applications.

The RT-JVM employs the Metronome garbage collector [4]. Metronome garbage collections employ "quanta" that should, if at all possible, not last longer than 500μs. Each quantum begins by ensuring that every heap-allocating thread has reached a "safe point" where both its stack and other RT-JVM data structures that it uses are in a consistent state. Once at a safe point, a thread is blocked from proceeding further until the quantum ends. Therefore, the quantum length corresponds to the pause time an application experiences when running in the RT-JVM.

Achieving safe points at quantum startup is a barrier protocol as defined in this paper. The completion condition is achieved when all threads are at safe points. The resumption condition is the end of the quantum. If the barrier takes more than 500μs to reach the completion condition, Metronome has violated its contract. If it takes a substantial portion of the 500μs to reach completion, then there will be too little time between the completion condition and the deadline by which Metronome must post the resumption condition. The RT-JVM will run out of memory because Metronome will not have had enough time to collect garbage. Therefore, timely completion of the barrier protocol is of critical importance.

The Metronome safe-point barrier is like the barrier described in Section 3 in two senses. First, the completion and resumption conditions are separated by a period during which a system service is performed. Second, threads that are waiting or sleeping need to be excluded from active participation in the protocol since they are not in a position to participate. Our introduction of GPM into this barrier therefore followed the pattern described in Section 3. However, the specifics of the Java language added some additional factors. In Java, all waiting happens inside *native methods* written in a non-Java language (e.g. C). In general, native methods contain arbitrary code, so a thread cannot be counted on to notice the inception of a barrier while executing a native method (even if it is not actually waiting). By distinguishing between Java and native stack frames, however, the RT-JVM can also ensure that a (correct and non-malicious) native method does not modify any RT-JVM data structures. Native methods written using the Java Native Interface (JNI) [13] may call back into the JVM to modify Java objects or to execute Java methods. During such callbacks, the code being executed is no longer arbitrary so the thread is again in a position to notice barrier inceptions, although it is also in a position to modify RT-JVM data structures. The RT-JVM models this situation by simply declaring that all threads that are in native methods (and have not called back into the JVM) are at safe points while others are not and must actively reach safe points. Consequently, our introduction of GPM into this barrier generalizes the practice described in Section 3. Threads are required to withdraw from the gang when entering a native method and to rejoin the gang upon returning from a native method. They rejoin the gang when calling back into the RT-JVM using the JNI interface, and they withdraw from the gang when returning from a JNI callback (thus reentering a native method).

Although all waiting happens in native methods, not all native methods wait, and even those that do, may do computation in addition to waiting. Computation performed in native methods is therefore one important source of priority inversion because these methods may have a priority in the range of priorities of the threads attempting to reach safe points. A second source of priority inversion comes from processes besides the RT-JVM (or from additional instances of the RT-JVM running on the same machine). If any threads in these processes employ real-time priorities that are in the same range of priorities of those threads attempting to reach safe points, threads in the RT-JVM may again experience unbounded priority inversion. Since the economics of modern real-time applications does not always lend itself to the simple model of one application per machine, running multiple independent real-time applications on one machine is a use case we want to consider. In any case, the OS itself and system management entities on the machine perform tasks that can interfere, at least occasionally.

For the experiments of this paper, we used a slightly modified version of the RT-JVM in which the data structure that represents a thread in the VM (`VMthread`) was modified to include an additional word. It should also be noted that the experimental RT-JVM was built from source in between formal releases of the product and so does not correspond exactly to any version of the released product.

The extra word in the `VMthread` was used as the thread's control word for the API. As required by the GPM API, all modifications of this word were done with compare and swap, to ensure atomicity. The first time a thread starts executing Java code, it becomes a passive gang member by calling `gang_insert`. Thereafter, whenever the thread is executing Java code (or is inside a JNI callback) it registers itself as an active gang member by atomically modifying its control word. When it is about to enter a native method or to return from a JNI callback, it leaves the gang by atomically modifying its control word. As described in Section 3, it first blocks on the resumption condition whenever it is about to reenter the Java world while a quantum is in progress. As required by the analysis of Section 3, a thread performs a `gang_notify` upon leaving the gang if its atomic modification of the control word discovers that it has already been counted in a `gang_run` operation.

When initiating a quantum, the Metronome scheduling thread sets the global state to cause threads to wait on the resumption condition. It then walks through the list of threads and marks the threads that are not at safe points with a request to reach a safe point as soon as possible. Then it calls `gang_run` using a mask that selects all threads which have not voluntarily left the gang. Eventually `gang_wait` is called on the gang to wait until all gang members reach a safe point and have called `gang_notify` (all such threads will then block on the resumption condition). At this point, the quantum is launched and Metronome can collect garbage, until the end of the quantum, at which point it posts the resumption condition to resume the application threads.

Our experimental version of the RT-JVM tests whether the GPM API is available in the kernel and, if so, uses it. If it is not avail-

able, it executes an alternative implementation of the same design pattern using the library version of GPM. This allows us to evaluate both implementations against each other as well as against the case where there is no attempt to avoid unbounded priority inversion.

6. EXPERIMENTS

The experiments were conducted on a machine with two quad-core 2GHz AMD Opteron CPUs (eight cores per machine). Two different versions of Linux kernel 2.6.24 were employed. Both had the real-time preemption patches installed and activated. In addition, one of the kernels had our implementation of the GPM API installed and activated.

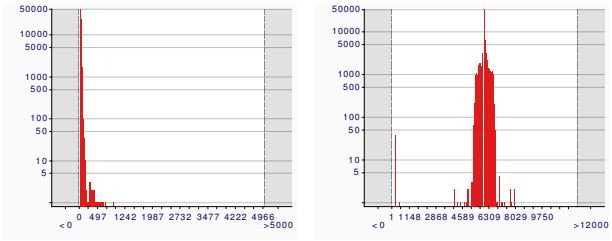
We also employed two different versions of the WebSphere Real Time Java VM (RT-JVM). Both were generated from the same source snapshot and both had the additional word added to the `VMthread` as described in Section 5. One RT-JVM version (no-GPM) used the unmodified original safe-point barrier without using GPM. Thus, in this version, there was no attempt to address priority inversion (the additional `VMthread` word was unused). The other RT-JVM version modified the safe-point barrier to implement priority inheritance using either K-GPM or L-GPM.

We used the two kernel versions and the two RT-JVM versions to create three configurations. (1) The no-GPM configuration ran the non-GPM version of the kernel with the no-GPM version of the RT-JVM. (2) The L-GPM configuration ran the non-GPM kernel and the GPM version of the RT-JVM, causing L-GPM to be used. (3) The K-GPM configuration ran the GPM enabled kernel and the GPM version of the RT-JVM, causing K-GPM to be used.

The real-time application was a simulation of an unmanned lunar lander module (available as a sample application with the RT-JVM release [18]). The lander uses vertical and horizontal thrusters to adjust its position on the x and y axis. Additionally, it uses a radar to calculate the module's position, by measuring the time taken by the radar pulses to return. This data is used to calculate the adjustments to the lander's position by firing the thrusters. If the radar pulse reception is delayed, for example by a GC pause, the lander's position is not calculated correctly, and the controller makes wrong adjustments. A more detailed description of the simulation is available in [14].

We modified the original controller to generate more load on the machine. The simulation and controller in our experiments use a shorter period than the original code (6ms instead of 20ms). Additionally, the controller is implemented as a Java `RealtimeThread` instead of a standard Java `Thread`. Note that we did not introduce any other Java real-time features like NHRTs [6] or Eventrons [32]. Furthermore, all controller runs are performed in interpreter mode, to avoid non-deterministic delays introduced by the JIT compiler. Finally, the controller was modified to run additional allocation threads that allocate memory and produce garbage at an average rate of 340MB/s, with the effect that the GC is basically always active. Although, the lunar lander is a simple real-time application, especially for the machine it runs on, the underlying virtual machine that executes it is a relatively complex software system. Moreover, the priority inversion occurs inside the virtual machine and not in the application. Nonetheless, the experiments in this section show that the application performance could also be affected by the priority inversion in the safe-point barrier.

To demonstrate the problem of priority inversion in the safe-point barrier, we started an additional instance of the RT-JVM, which runs one or more unrelated Java real-time threads at medium priority. The medium priority threads periodically perform some computation for about 5ms and sleep for 20ms. We performed similar experiments, by running the medium priority threads in the same virtual machine instance but invoking native methods (using JNI) that perform the computation. Regardless of whether a separate RT-JVM instance was used or JNI threads were used, we varied the number of real-time threads in preliminary experiments. In this section we present the results with 16 real-time threads. The results of the JNI experiments and the experiments with different numbers of interfering threads are similar to the ones shown in this section.



(a) Safe-point barrier duration (b) Controller event interarrival times

Figure 3: Single RT-JVM instance performance

We also performed runs using the no-GPM configuration but with no interference from any other threads (neither a separate RT-JVM instance nor any JNI threads). These runs were used to establish a baseline for how the RT-JVM safe-point barrier performs "normally" when there is nothing to cause priority inversion.

During all runs we collected trace data using TuningFork [17]. The Metronome GC in the RT-JVM is instrumented with TuningFork and provides detailed trace information of its execution. We are interested in the duration of the Metronome safe-point barrier. In addition, we instrumented the lunar lander application to trace when the control calculation is done and how long it takes, and to trace all the calculated data and the actual position of the lander module.

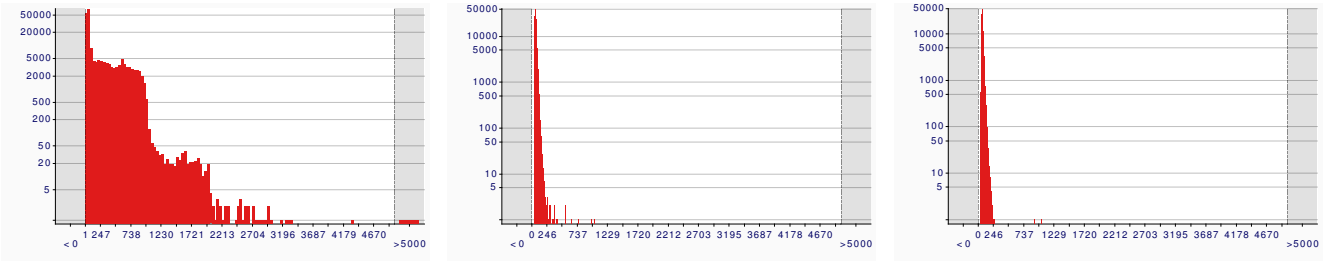
6.1 Results

We first show the results for the baseline, a single, no-GPM RT-JVM instance with no interfering medium priority threads, thus no priority inversion. We then show how this behavior changes when priority inversion is stimulated.

6.1.1 Single RT-JVM instance

Figure 3 depicts the results of the baseline run. The histogram of the safe-point barrier duration (Figure 3(a)) depicts the distribution of the duration from the time instant when the GC initiates the safe-point barrier until the time instant when the threads have reached a safe point. During this experiment there were 206871 barrier operations. On average safe-point barrier takes 56μs (median 63μs) to complete, with a standard deviation of 11μs. There is, however, one outlier at 933μs. These values suggest that the normal behavior of the protocol is quite acceptable.

We also analyzed the application performance in terms of its latency and jitter. The application runs a periodic controller computation that should execute every 6ms. Our measure of jitter is the standard deviation of the interarrival times of controller events.



(a) no-GPM (b) L-GPM (c) K-GPM

Figure 4: Safe-point barrier duration

	baseline	no-GPM	L-GPM	K-GPM
# events	206871	207442	213006	188131
Mean μs	56	214	67	70
StdDev μs	11	272	14	14
Max μs	933	5406	1005	1013
Min μs	31	25	38	39

Table 1: Safe-point barrier duration

	baseline	no-GPM	L-GPM	K-GPM
# events	89592	88388	88433	89245
Mean ms	6.0	6.0	6.0	6.0
StdDev ms	0.261	0.390	0.282	0.251
Max ms	7.9	13.0	9.9	12.0

Table 2: Controller event interarrival times

Figure 3(b) depicts the distribution of the interarrival times. The x-axis range is from 0ms to 12ms and the y-axis scale is logarithmic. There are 89592 controller events during this run and the interarrival times are centered around 6ms with a jitter of 261μs. The symmetry of Figure 3(b) and other interarrival histograms presented in the next section can be explained by the fact that, when an event happens later than it should, the interval before the event is lengthened and the interval after the event is shortened by the same amount of time. The small bump on the left side in the figure and in the subsequent interarrival histograms reflect some startup anomalies of the RT-JVM, which occur only during the initialization of the RT-JVM.

Note that these experiments were run with additional allocation threads that kept the GC busy by allocating memory at an average rate of 340MB/s. When this rate is reduced to a more realistic value of 25MB/s, the jitter of the interarrival time of the controller event is about 150μs with a maximum outlier of 635μs. These values are consistent with earlier results that were published about the RT-JVM [1].

6.1.2 Multiple RT-JVM instances

We now present the results of experiments in which an additional instance of the RT-JVM was running on the same machine. The second instance runs 16 independent periodic medium-priority real-time threads. The period of a thread in the second RT-JVM instance is 25ms, in which the thread does some computation for about 5ms.

Figure 4 shows histograms equivalent to that in Figure 3(a), depicting the distribution of the safe-point barrier's duration. The presented data is from 9 minute runs, during which time the GC performed about 200000 safe-point barriers. The values of the x-axis are between 0ms and 5ms. Outliers are shown in the gray shaded bars on the left and right side of the graphs, and the scale on the y-axis is logarithmic. Table 1 lists exact numbers of these runs.

These experiments demonstrate that the safe-point barrier suffers from priority inversion. In the RT-JVM instance that is not using GPM, the duration of the safe-point barrier is very unpredictable: on average it takes 214μs with a standard deviation of 272μs. Furthermore, there is even an outlier that needed 5.4ms. In contrast, in the RT-JVM instances that use GPM, either L-GPM or K-GPM, the average duration is about 70μs with a standard deviation of only about 14μs. Moreover, the maximum for both L-GPM and K-GPM is at 1ms. The improvements over not using GPM are: average duration of the safe-point barrier 3 times faster, standard deviation 19.4 times smaller, maximum time 5.3 times faster.

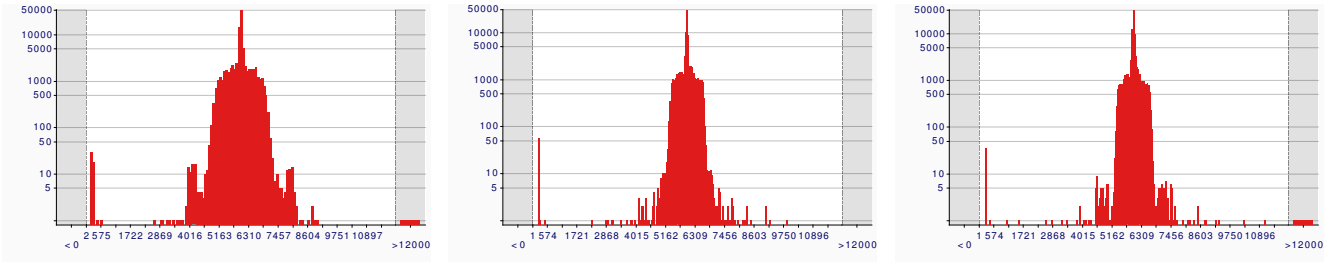
As previously mentioned, the controller period is 6ms, and we analyze the application performance by looking at the interarrival times of the controller events. Figure 5 shows the distribution of the interarrival time of controller events and Table 2 summarizes the statistical data. The average in all runs is about 6ms, but the standard deviation and therefore the jitter, vary between the different configurations. Not using any GPM results in jitter of about 390ms. The jitter is reduced to 282μs and 251μs when using L-GPM and K-GPM, respectively. The baseline jitter is 261μs.

The experiments demonstrate that unbounded priority inversion in barrier protocols can occur, and that a priority inheritance protocol implemented on top of GPM effectively avoids unbounded priority inversion. Furthermore, avoiding unbounded priority inversion results in similar application performance than the baseline, which is not influenced by any interfering medium priority threads, and thus not affected by priority inversion.

7. RELATED WORK

The present work extends the recognition of the priority inversion problem and the priority inheritance solution to a new class of situations, namely, barrier protocols. Thus, the literature on both barrier protocols and priority inversion is relevant. In addition, the present work is related to a broader class of protocols that dynamically adjust a real-time schedule, and approaches to integrate it in general-purpose operating systems.

To incorporate real-time scheduling into general-purpose operating systems, techniques using proportional-share scheduling have



(a) no-GPM (b) L-GPM (c) K-GPM

Figure 5: Controller event interarrival times

been proposed as being more compatible with operating systems that must mix real-time and non-real-time tasks [33, 10]. Dynamic adjustment to the shares (weights) has also been studied, including in multiprocessor contexts [5]. Brandt et al [7] implemented a rate-based EDF scheduler in the Linux kernel. It also allows dynamic adjustments to both the period and the utilization of tasks. However, this family of approaches generally does not consider the needs of groups of real-time tasks acting in concert; instead, adjustments to share, deadline, and so on are done at the level of individual tasks.

The first time the priority inversion problem was discussed in the literature was in relation to monitors and priority scheduling [19]. Later on, a formal definition of the problem and solutions, called priority inheritance protocols, were presented and were proved to solve the unbounded delay induced by priority inversion [30]. The latter paper introduces the priority inheritance protocol and the priority ceiling protocol, and it shows that both protocols provide an upper bound on the total delay a high priority thread can encounter due to priority inversion. Additionally, the authors prove that the priority ceiling protocol can prevent deadlock.

Four different sources of priority inversion in priority-based real-time systems are discussed in [9]. First, semaphores and critical sections, second software queues for data buffering, third Ada tasking and rendezvous, and fourth hardware queues on communication buses. Recently, more different sources of priority inversion were identified in object request broker middleware [29] and in multiprocessor systems [23]. All these sources, however, can be classified as resource sharing problems: multiple threads with different priorities share and access a common resource like a semaphore, queue or a communication bus. In contrast, the priority inversion problem discussed in this paper occurs during a synchronization protocol that does not involve a common resource. A barrier is merely a synchronization point in time which all threads have to reach before the program is allowed to advance.

Different barrier algorithms have been proposed in the literature. Among them are tree-based barriers [34, 21], tournament barriers [20, 16, 15], and multi-stage barriers [11]. These algorithms are designed for high performance computing and do not consider priorities among the different threads. Mellor-Crummey and Scott [21] evaluate barriers in terms of critical path length, number of network transactions, space requirements and the need for atomic operations. They conclude that a central barrier is the best choice if the number of threads change from one phase to the next (as in the RT-JVM safe-point protocol). The internal reorganization that occurs in the algorithms as computation moves from phase to phase outweighs any performance advantage in the barrier itself.

Although the RT-JVM safe-point protocol described in this paper is similar to a traditional central barrier algorithm, it differs in two aspects. First, the threads participating in the safe-point protocol are blocked until another unrelated thread, i.e., the GC, releases them. Second, threads can have different priorities and not all threads in the system are involved in the barrier. It is this second property that makes the protocol vulnerable to priority inversion. We believe that any barrier protocol would suffer from priority inversion if it was designed without priorities in mind and is then used in a multi-process environment with different priorities among the participating and non-participating threads in the barrier algorithm. To the best of our knowledge, there is no barrier algorithm designed and evaluated for real-time systems that respects priorities among the threads involved in the barrier. However, we believe that there will be increased use of general purpose multi-thread software in contexts where real-time priorities are important (for example, the emergence of real-time Java VMs).

Priority inversion can occur in other protocols such as the "Query Suspend and Resume" protocol for database management systems [8]. In the Query Suspend and Resume protocol high-priority transactions (e.g. real-time decision-making queries) should be able to suspend low-priority transactions by scheduling a suspend event in the low-priority transaction thread. The low-priority transaction thread has to process the suspend event successfully, before the high-priority transaction can continue. However, an unrelated medium-priority thread on the same machine could prevent the low-priority transaction thread from making progress, which again leads to priority inversion.

The tax-and-spend scheduling policy relies on a new incremental and concurrent GC architecture [2]. The new GC is not stopping mutator threads and hence does not depend on a barrier protocol. However, the authors identified other sources of priority inversion that would prevent the GC from making progress. Their solution uses manual priority setting instead of the GPM mechanism, because the authors would not accept a solution that involves changes to the underlying operating system kernel. Furthermore a bug in the Linux kernel prevented the use of LGPM.

8. FUTURE WORK

We provide 28 bits for the application to use in matching masks to gang-control words, which we hope will provide flexibility when an application uses multiple barrier protocol types involving different subsets of the threads. However, our case study in this paper used only one bit, and so the question of how to best handle these more complex cases is still open. Future investigations of using GPM to avoid unbounded priority inversion should study a wider set of

barrier protocol use cases, and would begin by surveying practices in other general-purpose real-time systems to find more use cases. Such an investigation would reveal whether the GPM should be changed to support a more flexible notion of gang membership.

Controlling higher-level priority inversion is not the only case we know of where control over priorities by the application is needed and where the standard kernel semantics may be insufficient. The original motivations for GPM included use cases beyond priority inversion that we have not yet explored. For example, the Exotasks system [3], supports pluggable real-time schedulers. At present, the design of such schedulers is limited by the need to run the scheduling threads at particular priorities and scheduling policies offered by the operating system. For example, it would be infeasible at present to implement a true EDF scheduler in the Exotasks system using only the kernel facilities in standard Linux. Whether the GPM API can be used as-is to support such efforts is uncertain, but future work can reveal what additions will be needed. In the end, the goal is a flexible interface to priority management that covers the majority of known use cases and assigns the optimal roles to both the kernel and the application.

9. CONCLUSIONS

We have introduced gang priority management (GPM) as a generic solution for avoiding unbounded priority inversion in barrier protocols. We have implemented GPM in the Linux kernel and in a user-space library, and rewrote the garbage collection safe-point barrier protocol in IBM's WebSphere Real Time Java Virtual Machine to exploit it. We provided empirical results, which confirm that the GPM-enabled safe-point barrier protocol successfully avoids unbounded priority inversion resulting in substantially improved application responsiveness.

10. REFERENCES

[1] AUERBACH, J., BACON, D. F., BLAINEY, B., CHENG, P., DAWSON, M., FULTON, M., GROVE, D., HART, D., AND STOODLEY, M. Design and implementation of a comprehensive real-time Java virtual machine. In *Proc. EMSOFT* (2007).

[2] AUERBACH, J., BACON, D. F., CHENG, P., GROVE, D., BIRON, B., GRACIE, C., MCCLOSKEY, B., MICIC, A., AND SCIAMPACONE, R. Tax-and-spend: democratic scheduling for real-time garbage collection. In *Proc. EMSOFT* (2008).

[3] AUERBACH, J., BACON, D. F., IERCAN, D. T., KIRSCH, C. M., RAJAN, V. T., ROECK, H., AND TRUMMER, R. Java takes flight: time-portable real-time programming with exotasks. In *Proc. LCTES* (2007).

[4] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proc. POPL* (2003), pp. 285–298.

[5] BLOCK, A., AND ANDERSON, J. H. Task reweighting on multiprocessors: Efficiency versus accuracy. In *Proc. IPDPS* (2005).

[6] BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., HARDIN, D., AND TURNBULL, M. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, 2000.

[7] BRANDT, S. A., BANACHOWSKI, S. A., LIN, C., AND BISSON, T. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proc. RTSS* (2003).

[8] CHANDRAMOULI, B., BOND, C. N., BABU, S., AND YANG, J. Query suspend and resume. In *Proc. SIGMOD* (2007).

[9] DAVARI, S., AND SHA, L. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *SIGOPS Oper. Syst. Rev. 26*, 2 (1992), 110–120.

[10] DUDA, K. J., AND CHERITON, D. R. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. SOSP* (1999).

[11] EUGENE D. BROOKS, I. The butterfly barrier. *Int. J. Parallel Program. 15*, 4 (1986), 295–307.

[12] GOH, O., LEE, Y.-H., KAAKANI, Z., AND RACHLIN, E. Schedulable garbage collection in cli virtual execution system. *Real-Time Syst. 36*, 1-2 (2007), 47–74.

[13] GOSLING, J., JOY, B., AND STEELE, G. L. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[14] GOUGH, C., HALL, A., MASTERS, H., AND STEVENS, A. Real-Time Java, Part 5: Writing and deploying real-time Java applications. `http://www.ibm.com/developerworks/java/library/j-rtj5/index.html`.

[15] GRUNWALD, D., AND VAJRACHARYA, S. Efficient barriers for distributed shared memory computers. In *Proc. IPPS* (1994).

[16] HENSGEN, D., FINKEL, R., AND MANBER, U. Two algorithms for barrier synchronization. *Int. J. Parallel Program. 17*, 1 (1988), 1–17.

[17] IBM CORP. TuningFork Visualization Tool for Real-Time Systems. `www.alphaworks.ibm.com/tech/tuningfork`.

[18] IBM CORP. *WebSphere Real-Time User's Guide*, first ed., 2006.

[19] LAMPSON, B. W., AND REDELL, D. D. Experience with processes and monitors in mesa. *Commun. ACM 23*, 2 (1980), 105–117.

[20] LUBACHEVSKY, B. D. Synchronization barrier and related tools for shared memory parallel programming. *Int. J. Parallel Program. 19*, 3 (1990), 225–250.

[21] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (1991), 21–65.

[22] MOLNAR, I., AND GLEIXNER, T. The rt-preempt patch set for Linux. http://rt.wiki.kernel.org.

[23] NAESER, G. Priority inversion in multi processor systems due to protected actions. *Ada Lett. XXV*, 1 (2005), 43–47.

[24] PLSEK, A., LOIRET, F., SEINTURIER, L., AND MERLE, P. A component framework for java-based real-time embedded systems. In *Middleware* (2008).

[25] POLZE, A., PLAKOSH, D., AND WALLNAU, K. Corba in real-time settings: A problem from the manufacturing domain. In *Proc. ISORC* (1998).

[26] RAMAN, K., ZHANG, Y., PANAHI, M., COLMENARES, J. A., KLEFSTAD, R., AND HARMON, T. Rtzen: Highly predictable, real-time java middleware for distributed and embedded systems. In *Middleware* (2005).

[27] SCHMIDT, D. C. Middleware for real-time and embedded systems. *Commun. ACM 45*, 6 (2002), 43–48.

[28] SCHMIDT, D. C., AND KUHNS, F. An overview of the real-time corba specification. *Computer 33*, 6 (2000), 56–63.

[29] SCHMIDT, D. C., MUNGEE, S., FLORES-GAITAN, S., AND GOKHALE, A. Software architectures for reducing priority inversion and non-determinism in real-time object request brokers. *Real-Time Syst. 21*, 1-2 (2001), 77–125.

[30] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput. 39*, 9 (1990), 1175–1185.

[31] SHANKARAN, N., SCHMIDT, D. C., KOUTSOUKOS, X. D., CHEN, Y., AND LU, C. Design and performance evaluation of configurable component middleware for end-to-end adaptation of distributed real-time embedded systems. In *Proc. ISORC* (2007).

[32] SPOONHOWER, D., AUERBACH, J., BACON, D. F., CHENG, P., AND GROVE, D. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *Proc. PLDI* (2006).

[33] STOICA, I., ABDEL-WAHAB, H., JEFFAY, K., BARUAH, S. K., GEHRKE, J. E., AND PLAXTON, C. G. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proc. RTSS* (1996).

[34] YEW, P.-C., TZENG, N.-F., AND LAWRIE, D. H. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput. 36*, 4 (1987), 388–395.