Real-Time Java and Multi-Core Architectures

Abstract

This paper presents some of the issues related to the use of multi-core architectures for real-time systems, in particular for real-time Java. Currently, the Real-Time Specification for Java (RTSJ) does not account for aspects of parallel computing, but the widespread use of multi-core architectures advocates for the need of a change in this regard. Some of these aspects relate closely to architectural features that need to be properly exported to the application level. We discuss the impact of multi-core processors, both symmetric and asymmetric, on process/thread scheduling. We also analyze the changes needed for some of the important features of RTSJ, such as the priority inversion avoidance, in the new context of multi-core processing. Also, locking and fairness issues are discussed with respect to the use of multi-core processors for real-time Java.

1 Introduction

In the past few years, all of the major processor vendors adopted multi-core architectures as the main way to improve the processor performance. This architectural choice is going to impact the way we think programming in the next decades. Parallel architectures are not new to the computing field and there has been a wealth of research devoted to them in the past decades. But the main lessons drawn from this extensive experience are rather advising against simple solutions like trying to parallelize existing sequential solutions (one of the main reasons being Amdahl's law, even though a more realistic approach given by an equivalent law, Gustafson's law [14], relaxes a bit the view on the problem). As soon as true parallelism is at hand in every single PC shipped out of the factory, most of the algorithms and mechanisms currently in use in software systems will have to undergo tremendous changes, many of them will be completely abandoned and new algorithms, inherently parallel, will have to become common place [5].

From a formal point of view, the multi-core architectures can be thought of as those of Symmetric Multi-Processors (SMP). For instance, the Intel documentation for developers [9] does not make any distinction between cores of the same chip and those of different chips on the same machine from a programming point of view. Nevertheless, the hardware features of these multi-core processors may be different than those of traditional SMPs and, as a result of that, even if the programming model is the same, there might be differences in the perceived performance if the inner characteristics of these multi-core processors are not taken into account.

Up until now, real-time systems have had a separate evolution in the space of processor design, mostly because of their particular features (need for predictibility, robustness, etc.). Therefore, the vast majority of real-time systems uses even today uniprocessor systems, most of which are having architectural features that prevent introducing a factor of hazard in real-time computing (no processor caches, no out of order execution, etc.). In this context, the use of parallel computing has not even been considered for real-time systems until the widespread use of multi-core architectures became an unavoidable evidence. The current trend in real-time research tends to shift the computing paradigm to multi-core processors, but the challenges associated with this step are manifold.

One particular area that naturally seems to fit with multi-core processing is the domain of real-time Java, because the language supports parallelism through threads from its very inception. It is almost a decade now that Java gained full acceptance in the real-time world. The restrictions and conditions that have to be met by a real-time Java implementation are summarized in the Real-Time Specification for Java (RTSJ) [8]. Several implementations of the specification (including commercial ones) are available [2, 12, 19].

In this paper, we discuss some of the issues raised by the use of multi-core processors for real-time Java. We first identify the areas of RTSJ that are sensitive to multiprocessing and then we elaborate on the issues related to these areas. There are two main types of problems, some related to real-time issues and others related to parallelism. They both influence each other. The real-time problems derive mostly from the need to ensure the predictibility of these systems, the feasibility analysis of a given schedule, etc. on parallel architectures. The parallelism problems concern mostly two areas. One tackles the way the capabilities of the hardware processor are exported to the user level

so that applications can decide best what they can do on the underlying architecture (for instance, the architecure of the processor caches, hyperthreading, cores with asymmetric performance, if any, etc.). The second area deals with the harmonization of the decisions taken a various levels of complex software systems (for instance, the need for coordination between user-level schedulers and kernel decisons on synchronization, preemption or blocking).

The rest of this paper is organized as follows. We start out by presenting the features of RTSJ that we identified to be sensitive to parallelism (Section 2). Then we discuss aspects of parallel real-time scheduling with emphasis on asymmetric processing and co-scheduling (Sections 3 and 4). We then present some of the problems caused by a lack of coordination between user-level scheduling and kernel decisions (Section 5). We also discuss an open problem in parallel real-time systems, namely the need for a proper solution for priority inversion avoidance on multi-core architectures (Section 6). Then, we debate fairness in parallel real-time systems (Section 7). Finally, we present our plans for future work (Section 8) and conclude (Section 9).

2 RTSJ features sensitive to parallel computing

The main features of the Real-Time Specification for Java [8] that are interesting from a parallel computation point of view can be grouped into two categories. First, there are low-level issues that influence the performance of memory access, single thread running times, event handling. Second, there are high-level issues related to the way the priority inversion avoidance can be handled in multiprocessors or the way the scheduling of threads and event handlers (in particular, happenings) is being done.

2.1 Processor features relevant for RTSJ

In the case of memory accesses, if two cores of the same processor share on-chip caches (either L2 or L3 caches), then two cooperating Java threads running on these cores can access some memory faster since they do not have to go over the system bus on a cache miss that has been already been solved by the other core. Otherwise, i.e. if the two cooperating threads run on two different processor chips, the overhead of memory access depends on the access type. For instance, a write access can trigger the coherence protocol between the local caches over the system bus. But even a simple shared read can influence the overall memory throughput since it goes over the bus even if one of the cooperating threads has a local copy of the data (which cannot be shared because the two cores do not share L2 or L3 caches). As a result of that, this read will have to compete for the bus with other unrelated reads and the overall memory throughput will decresase.

In order to take advantage of this kind of information, some processors have special instructions that expose the architecture of the hardware caches of the processor. For instance, Intel processors have the *cpuid* instruction that helps find whether the on-chip caches are shard or not [10]. If RTSJ shifts to multi-core architectures, such aspects cannot be avoided if performance is at stake and an appropriate software interface to the hardware needs to be developed.

Other low-level issues that can influence the performance of threads and asynchronous event handling refer to CPU and interrupt affinity. CPU affinity refers to the binding of a running process/thread to a given core and it is supported by many popular operating systems (Linux, Windows, etc). The benefits come from the fact that, over the time, several data of the process/thread may survive in the local core caches (not only data and instruction caches, but also things like the TLBs) and thus the running time of the process/thread gets improved. RTSJ requires support for CPU affinity [8].

Interrupt affinity refers to the possibility of shielding certain cores (or processors) from receiving (and handling) interrupts. Several operating systems support interrupt affinity, including general purpose operating systems like Linux. This is an important aspect for real-time systems as the predictibility of the computation run on shielded cores is improved. RTSJ doesn't currently support interrupt (happening, in RTSJ terminology) affinity, but there has been made a proposal to incorporate it into the specification [22]. Both types of affinity are sensitive to parallel computation if global processor and interrupt scheduling is used.

There are other processor features that can influence RTSJ or its assumptions. Modern processors chips implement logical processors that share hardware components of the same core. This technique is generally known as *simultaneous multithreading* (SMT) or *hyperthreading* (using Intel terminology). Two logical processors of the same core share the execution engine of the core. The techique aims at better usage of the processor hardware (the internal resources of the execution engine) but may induce unexpected behavior in terms of performance. Special instructions help detect whether hyperthreading is supported or not (again *cpuid* in the case of Intel processors [10]). Other special instructions are provided to halt or pause a logical processor [9]. A more detailed discussion on SMT or hyperthreading and their impact on real-time systems can be found in Section 3.

A very recent trend in processor design uses cores with different performances on the same processor chip. The motivation behind this design comes from the need for cost effective hardware for setups with tens or even hundreds of cores on a processor chip. This twist in the processor design leads to asymmetric multi-core processing. Naturally, the scheduling of processes/threads will be influenced by

such an infrastructure and RTSJ needs to be aware of this aspects too. The issues related to asymmetric processing are presented in Section 3.

2.2 RTSJ and system software

One of the problems of operating systems using priorities (and real-time operating systems are no exception to the rule) appears when a high priority process cannot get a lock held by a lower priority process, which in turn cannot run because of its priority. This problem is known as priority inversion. On uniprocessor systems, two of the most widely used solutions that avoid priority inversion are priority inheritance and priority ceiling. RTSJ requires the implementation of priority inheritance [8]. Essentially, this solution requires to raise the priority of the lower priority thread to that of the blocked thread as long as it holds the lock so that the lower priority thread can get the chance to run and, thus, to set the lock free. The main assumption behind the corectenss of this algorithm is that only one lock contender runs at a time (an this is true on uniprocessor systems). However, as it will be presented in Section 6, this algorithm doesn't work anymore on multiprocessor architectures.

Parallelism changes also the classical view on scheduling. On uniprocessor systems, time-sharing can be used to maximize the throughput of the CPU. This view can be adopted on multiprocessors as well, with the caveat that now one needs to ensure also load balancing across the processors in the system. However, multiprocessor scheduling supports a different aproach as well, namely one can schedule cooperating threads simulteneously on a subset of the processors of the system in order to improve their performance. RTSJ doesn't currently support this form of scheduling which is discussed in Section 4.

Happenings (and event handling in general) are handled in RTSJ by schedulable entities that are most similar to threads. In fact, RTSJ implementations like Jamaica VM [2] use a thread (or a thread per event, if need be) to run the computation associated with event handlers. If a thread per event is being used, then maximum of parallelism is at hand, depending of the scheduling of these threads. The downsize of using one thread per event is greater resource consumption. However, if the events are not bound to a single thread, a need for load balancing appears, since using a single thread for handling many events may create a hot spot in the system. The issue may be further complicated if happening affinity (see the previous subsection) is being enforced.

3 Asymmetric processing

The latest trend in processor research targets asymmetric (or heterogeneous) multi-core architectures, where a processor chip may have cores with different performance. All of the cores of the same processor support the same instruction set but their performance may differ as a result of having different clock speeds or different hardware characteristics (issue width, in-order/out-of-order execution, etc.). From the user point of view, all these difference sum up to a difference in perceived performance.

The motivation behind this development can be found in the endeavor to build processors that support tens and even hundreds of cores on a single chip, as it is foreseen for the next decade [17]. Such a development needs cost-effective solutions that minimize the die area and the power consumption while delivering high performance. It has been shown that this approach is viable [4, 15, 16].

Asymmetry may arise in symmetric (or homogeneous) multi-core systems as well. For instance, for fault tolerant purposes failing cores may be switched off. Even further, some may regard hyperthreading (or simultaneous multithreading, as it is generally known) as a form of asymmetric processing. Indeed, some processor makers like Intel provide two logical processors per core that share the execution engine of the core and its bus interface but have separate local APICs (and therefore separate identity). Using logical processors improves the overall CPU throughput, but may hinder the individual performance of a given thread of execution. Indeed, Intel recommendations for system programmers point out that there are differences in performance between a core running logical processors and a core using a single logical processor because two logical processors of the same core compete for the hardware resources of the core [9]. Therefore, when the operating system runs the idle loop on a logical processor, Intel recommends switching off that logical processor to improve the performance of the other logical processor(s) of the core. Also, as a general rule of thumb, Intel recommends dispatching threads of execution first on every core of every processor chip in the system, and only afterwards start logical processors on every core, as needed [9].

Given the trends in processor development that target asymmetric architectures but also things like hyperthreading, one needs to build appropriate asymmetry-aware schedulers. Prototypes for general purpose operating systems like Linux have been proposed and implemented [17], but the real-time world still lacks its specific asymmetric scheduling algorithms. The main difficulties stem from the additional level of impredictibility introduced by the hard-to-quantify degree of performance asymmetry. Indeed, if one thinks only to hyperthreading, it is hard to predict with mathematical precision the performance of two threads of execution running on logical processors of the same core. Nevertheless, a smart scheduler may turn off hyperthreading for a while when the CPU throughput is of no concern, so that the well-known SMP paradigm can function for a

given application. Also, the operation of switching on and off logical processors has to be taken into account when doing the feasibility analysis.

Asymmetric processing raises problems for real-time systems also from the point of view of priority inversion avoidance. If the high priority thread runs on a fast core and the low-priority thread is bound to a slow core, perhaps due to affinity reasons, raising the priority of the thread holding the lock doesn't suffice. The lower priority threads needs to be scheduled on a fast processor (perhaps as fast as the processor on which the high priority thread is supposed to run) in order not to penalize too much the waiting high priority thread. This decision may imply migrating threads between processors, which is another debatable issue in the case of real-time systems which, most of the time, prefer to use a static assignment of processes/threads to processors. Some real-time systems may not tolerate the overhead of thread migration.

Enforcing interrupt affinity, another important aspect for real-time systems, may create problems to an asymmetric-aware scheduler as well. In general, any operating system attempts to minimize the interrupt handling latency. A natural consequence for an asymmetric processing scheduler using interrupt affinity would be to tie the interrupt handling to fast processors. However, due to the aforementioned costs in terms of die area and power consumption, there may not be too many such fast cores. Therefore, high priority threads that need to run shielded from interrupts on fast cores may find themselves in competition for these fast cores with the interrupt handlers. In RTSJ, the happening affinity lends itself to lazy happening handling solutions that may avoid this competition for fast cores by postponing the handling of the happening until the results of the handler are actually needed (see also Section 7).

4 Co-scheduling

In multiprocessor systems there are two possible types of scheduling: temporal and spatial scheduling. In temporal scheduling, one tries to improve the overall throughput of the processors by keeping all of them busy. In contrast, spatial scheduling targets the improvement of the performance of a given parallel application by simultaneously executing its parallel processes/threads. The idea appeared in distributed systems where it was called co-scheduling [20]. Co-scheduling attempted to optimize the execution of processes that cooperate through messages in a distributed system. In such a system, two cooperating processes that need to exchange a single message may experience a heavy penalty if they are not simultaneously scheduled for execution. Imagine the following scenario [21]: the first process gets scheduled and sends the message but the second process doesn't run, so the first process blocks waiting for the response and releases the processor. After a time slice (usually 10 ms in modern operating systems) the receiver process gets scheduled and replies. However, the initiator of the message exchange needs to be awaken and after another time slice it gets the processor and can receive the message. So the message exchange takes roughly 20 ms, which is completely unacceptable.

Co-scheduling may be appealing in multi-core real-time systems as well. For instance, two cooperating periodic threads may need to be co-scheduled to improve the behavior of the system. Even if feasibility analysis makes sure that deadlines are met regardless of co-scheduling, the use of co-scheduling may leave room for the execution of other threads on the two processors during the current period of execution, thus helping improve the overall throughput of the multiprocessor system. However, co-scheduling may conflict with priority-based scheduling. For instance, if four threads need to be co-scheduled on a four-processor system and a fifth independent thread of greater priority is runnable as well, then the fifth thread will get scheduled and the rest of the three processors will either stall or execute threads with lower priority than that of the co-scheduled threads.

This scenario underlines the main assumption behind co-scheduling: a global scheduler. However, global solutions are usually performing poorer than the local ones in parallel/distributed systems. So the design of a scheduler implementing co-scheduliong needs to be balanced between the need to share global knowledge about the scheduled threads and the overall performance of the solution. For instance, the Linux SMP kernel runs a scheduler on every processor (core), which has its own private run queue. However, the processors exchange scheduling information in order to improve the overall performance of the system (for instance, the schedulers aim to perform load balancing in order to keep all of the processors busy [1]).

The advent of asymmetric multi-core processors (see Section 3) complicates even further the issue of co-scheduling. The scheduler needs to find a subset of processors of "equal" performance for the set of co-scheduled threads to avoid potential problems. Imagine the following scenario: two co-scheduled threads run on two cores, one of which is twice as fast as the other one. If the first thread asks the second thread for a service, it will have to wait twice as long as it would have had to wait if the cores would have been equally fast. Since the cooperation is the main incentive behind co-scheduling, the first thread cannot block awaiting for the response of the second thread (see the example at the beginning of this section), instead it will most probably have to busy-wait until the response arrives. Busy-waiting in general must be kept short, but in shared memory multiprocessor systems it is further complicated by the access contention to the system bus. The penalties associated with the traditional test-and-set (TAS) method

can be alleviated by using variants of test-and-test-and-set (TATAS), but nevertheless, the longer the busy-waiting, the worse the performance. Therefore, a scheduler performing co-scheduling needs to be aware of the asymmetric performance of the processor cores.

Another source of potential problems for co-scheduling comes from the use of CPU and interrupt affinity (see Section 2). Indeed, if a thread (or a bunch of independent threads) competes with a set of co-scheduled threads for a given core (or cores), the scheduler has to decide whether to enforce the CPU affinity or to honor the co-scheduling contract. Depending on a certain policy and, in the case of real-time systems, perhaps on the priorities, the scheduler may not enforce CPU affinity at the cost of losing some performance. However, when it comes to interrupt affinity, a real-time system may pay heavier for such a decision if the interrupt gets handled on another core that presumably has declined its availability for handling interrupts, or, even worse, if the handling of the interrupt is delayed just to enforce the affinity. In RTSJ the situation is a bit easier, in the sense that interrupts (happenings in RTSJ) are handled by a schedulable entity, something equivalent to a thread [8]. Therefore, handling happening affinity in RTSJ is somewhat equivalent to handling CPU affinity (for happening affinity in RTSJ please see [22]).

Most of the current global-purpose operating systems do not support co-scheduling. As for real-time operating systems, they only now get out of the uniprocessor paradigm. We believe that this area of research should be very interesting for multi-core real-time systems.

5 Coordination between the operating system and user-level threads in multi-core systems

One of the problems traditionally found in parallel systems using a hierarchy of schedulers is the lack of coordination between them. The typical example is that of user-level threads packages that employ a user-space scheduler (Java is such an example). This scheduler is un-aware of the decisions taken by the scheduler of the operating system when one of the user-level threads executes within the kernel and fires a decision that conflicts with the expectations of the user-level scheduler. For instance, if a thread of a parallel application blocks within the operating system (either within a system call or because of a page fault), the operating system suspends the whole process even though there are other runnable threads within the application.

Notifying the user-level scheduler about the situation helps, but there are further issues. For instance, if the preempted thread holds a spin-lock, other threads of the application trying to get the lock will have to busy-wait and thus will waste processor time, because until the preempted thread doesn't make progress the other threads won't get the lock. Lock holder preemption may occur also in a different context. Think of a thread holding a spin-lock whose time slice ends. If the scheduler preempts the holder of the lock, other runnable threads trying to aquire the lock will get scheduled but they will busy-wait in vain until the holder of the lock gets back a processor (core) to run on.

From these scenarios it can be concluded that lock holder preemption is especially important for co-scheduled threads that use spin-locks. If one of the co-scheduled threads grabs a lock and then blocks within the operating system for some reason and if the user-level scheduler is not informed about the blocking, then the other threads busy-wait on the spin-lock. In an SMP, this can affect the performance of the whole system, because of the contention for the system bus. The threads try to aquire the lock writing a memory location and the coherence protocol is fired up. In the end, nothing happens because the lock is not free. So this is wasted operation. Moreover, it affects even the processors (cores) that are not involved in sychronization, because they have to compete for the bus as well when they access memory and, thus, their activity is disturbed by the sychronization traffic.

This issue of bus contention in SMPs (and multi-core processors as well) advocates for cache-aware locking. Essentially, a smart scheduler would have to schedule cooperating threads spinning on the same lock on cores that share on-chip caches (either L2 or L3 caches). The scheduler can find out whether the cores share on-chip caches by calling a special instruction of the processor, if supported (for instance, *cpuid* in the case of Intel processors [10]). Thus, when busy-waiting for a lock, the threads do not have to access the bus.

The lock holder preemption problem is relevant also for priority inversion avoidance protocols, and thus for real-time systems as well. If the lower priority thread holding the lock gets the chance to run as a result of having its priority raised, but is preempted while trying to finish its task so that it can release the lock, the higher priority thread remains blocked and pays an unwanted penalty, namely that incurred by the preemption of the lock holder.

There are several solutions for the lock holder preemption problem (some of them to be found in [3, 18]). Essentially, all these solutions rely on notifying the upper levels about the preemption. This notification can be made when the event occurs or can be sent ahead of time if the system can establish that preemption will occur in the near future (for instance, when the time quantum is about to expire). Depending on the situation, the user-level thread or scheduler takes the most appropriate action. For instance, if a warning is issued about imminent preemption, the thread being warned may give up the idea of taking a lock before preemption. A user-level scheduler informed about a lock holder preemption may choose not to schedule any-

more threads that spin on the same lock.

6 Parallel priority inversion avoidance

The priority inversion problem appears when a runnable high priority process/thread waiting for a lock cannot proceed because the lock is held by another, lower priority process/thread that doesn' t get the chance to run because of its priority level. In uniprocessor systems, the solution is to use either priority inheritance or priority ceiling [8], but both solutions are hinging on one main assumption: the system has only one processor and, therefore, only one lock contender can run at a time.

In an SMP system, this assumption doesn't hold anymore and, thus, the aforementioned solutions do not work. For instance, consider the priority inheritance algorithm. To solve the priority inversion problem, the lower priority thread "inherits" the priority of the higher priority condender of the lock as thus gets the chance to run and to set the lock free. When releasing the lock, there's a window of time when the lock is free and no one can contend for it on a uniprocessor system. Essentially, this window of time lasts at least for the duration of the context switch time. However, on a shared memory multiprocessor system no one can prevent a process/thread running on another processor to grab the lock during that window of time, because the lock is free. If this process/thread has a lower priority than the processes/threads waiting for the lock to be released, the priority inversion problem is solved inappropriately for a real-time system. A lower priority process/thread gets the lock although higher priority processes/threads are waiting. Theoretically, starvation is also possible. If every time the lock is released a new low priority process/thread gets the lock, then the high priority processes/threads never get the chance to grab the lock. A similar argument holds for priority ceiling.

To solve the problem of priority inversion in a shared memory multiprocessor, one needs to establish an order between the contenders of the lock. In a real-time system, this order is given naturally by priorities. For equal priorities, an additional rule needs to be set to break the ties. Once this order is globally enforced, the problem can be easily solved by a synchronization mechanism that does not allow getting a free lock unless there are no higher order contenders of the lock.

A first candidate solution for such a synchronization mechanism is to enhance traditional locking mechanisms with the ordering rule. For instance, when taking the lock, the contender can check the queue of the waiting lock contenders and, if there are higher order competitors in the queue, it blocks putting itself at the right place (according to its order) in the waiting queue. While simple, this solution breaks the formal definition of locks (or binary semaphores) that doesn't mention anything about checking the queue of the waiting lock contenders. The locking mechanism should be independent of the contenders. As soon as the lock is set free, all of the waiting contenders should be able to run and to compete for the lock. The locking scheme should rely only on the atomically accesible variable controlling the lock.

Another possible solution is to use a barrier. All of the lock contenders should pass a barrier before attempting to get the lock. Nevertheless, this may lead to performance penalties, because processors may have to be stalled until the other contenders get the chance to run. Moreover, even if all the contenders are competing simultaneously for the lock, global ordering is still needed to make sure that a lower priority process/thread doesn't inappropriately get the lock.

A more proper solution is to use an election algorithm. Every contender casts a ticket stamped with its priority in an auction won by the highest "bidder". If there are runnable higher bidders that are not currently running, the lower running bidder blocks itself waiting for the next auction.

7 Fairness

In general, real-time systems have little to do with fairness for obvious reasons. The task priorities, the scheduling algorithm and the feasibility analysis leave little room for complaints about fairness. Nevertheless, real-time systems are intimately tied to a source of potential unexpected fairness problems: external events signalled through interrupts. A typical scenario found in general purpose operating systems for uniproccesors is relevant for real-time systems as well, as they use more and more networking capabilities. The scenario refers to the way network interrupts are handled. For every incoming network packet, a hardware interrupt is raised and requires two stages for its handling. First, a so called hardware interrupt handler (sometimes called also *upper half*) is run in order to perform the minimal operations needed to handle the interrupt. Its processing must be kept as short as possible to avoid increasing interrupt latencies. As soon as there is enough time to finish the handling of the event, either at the end of the hardware handler or at a later time, before re-scheduling, a so called software interrupt handler (or *bottom half*) gets executed. Here are taking place the less time-critical operations. The problem with these two handlers is that they get called in the context of the process that currently runs when the interrupt is raised. As a result of that, the interrupt processing time is accounted to that process, which is unfair, because the network packet may be destined for another application. Solutions for this problem have been proposed [6, 7] but they seem to make their way slower into the industry.

For real-time multi-core systems, the situation is com-

plicated by two additional facts. First, if interrupt affinity is enabled, there's another stage added to the typical interrupt handling: the interrupt has to be routed to the processor that will actually handle it. Second, the unexpected behavior of the network has to be taken into account when doing the feasibility analysis. Even so, the fairness problem stays. Think about RTSJ in the case of the following scenario. A set of happenings corresponding to applications with different priorities are not bound to a specific thread, instead they are all run by the same thread. As a result of that, happenings for low priority applications may get executed before the applications that need their result. Thus, higher priority applications wait for lower priority computation whose results are not needed for the moment. This is not only unfair, but also inapproriate in a real-time system. There is a proposal in JSR 282 [22] to assign priorities to happenings as well. This decision might help solve the problem.

The above scenario makes the situation worse if happening affinity (as proposed in [22]) is used. Indeed, if the low priority happenings target applications that are not running on the processor the happenings are bound to, the processor is unfairly used for the needs of low-priority, "remote" applications.

To avoid these two problems, a possible solution would be to bundle the happening and the application it addresses to into a single schedulable object that would handle happenings similar to the way asychronous events are handled in Unix systems by means of signal handlers. The advantages are manifold. First, one doesn't need to assign priorities to happenings anymore. Second, the handling of the happening is directly charged to the application that is the target of the event. No unfair behavior towards other applications is possible since the happening is handled by the application thread during its own running period. Third, happening affinity cannot unfairly use the processor for "remote" threads, since the handling of the happening is done by the thread the happening is destined for. Fourth, no additional resources of the system are needed to handle happenings. Each thread handles its own events, there is no need for special threads to handle only happenings. Fifth, if the happening is handled directly by the thread it targets, there's an additional bonus for BoundAsyncEventHandlers [2, 8] that require a separate thread to handle the event. Indeed, in this special case, handling an event requires an additional context switch to run the event handler. The proposed solution avoids that additional context switch by handling the event within the context of the thread that is the target of the event.

There's also a drawback to this solution. Happening affinity ceases to be what it is supposed to be. However, this is not much of a problem, because in RTSJ happenings are anyway user-level handlers, whereas interrupt affinity concerns mostly the low-level (operating system level) handlers. And because these user-level handlers are run by threads, happening affinity becomes a form of processor affinity. The main challenge of shielding the processor from unwanted interrupts remains the task of the real-time operating system.

8 Future work

Our first aim for the upcoming months is to develop a multi-core aware benchmark for real-time Java. This benchmark will help us evaluate both the importance and the impact of the issues discussed in this paper. Currently, there are just a few real-time Java benchmarks but they are either not multiprocessor-aware [11] or not true real-time benchmarks [13] (the latter doesn't support NoHeapRealtimeThreads and AsyncEventHandlers [8]). We plan to extend one of the existent benchmarks by enhancing it with the missing features (either in terms of real-time or parallelism support) and by adding specific features that will help us conclude on the issues raised in this paper. We are looking for a synthetic benchmark but we would like also to develop a benchmark suite of relevant applications for real-time Java on multi-core architectures.

Once we have such a benchmark that can clarify the points in the previous discussions, we will have a tool that can help us evaluate the performance of a real-time Java VM running on multi-core processors. Then, we can start developing specific optimization solutions for the problematic issues.

9 Summary

In this paper, we presented some of the challenges raised by the use of multi-core architectures for real-time systems, with special interest for real-time Java. There are two orthogonal directions of development. One of them tackles the problem of dealing with multi-core architectures from a software point of view. The other one refers to the way parallelism and real-time issues influence each other in a software system.

From a software point of view, the multi-core architectures features are interesting in two ways. First, some of the hardware characteristics of the processors need to be exported to the software level so that the software can make optimal use of the hardware facilities. For instance, the software needs to be aware of asymmetric cores, hyperthreading (or simultaneous multithreading), the architecture of the on-chip caches, etc. All these aspects influence the optimality of the software run on multi-core processors, sometimes regardless whether the software is single- or multithreaded. The main areas sensitive to these aspects are scheduling, synchronization and fairness. Second, one needs to harmonize the operation the various software components of a real-time system. For instance, user-level schedulers

need to be informed about operating systems decisions on scheduling, blocking and sychronization for a more optimal use of the system resources.

Using parallelism in real-time systems complicates them in many ways. For instance, the predictibility of real-time systems is endangered because of the use of logical processors (SMT or hyperthreading), asymmetric scheduling, cache-aware scheduling, etc. Important issues for real-time systems such as priority inversion avoidance and interrupt affinity need to be revisited in the context of parallel computing as well. Perhaps other ways of scheduling such as co-scheduling need to make their way as well within the real-time world. We believe that all these issues will spark a good deal of interesting research in the years to come.

References

[1] *The Linux Cross Reference*. http://lxr.linux.no/.

[2] aicas Gmbh. *The Jamaica Virtual Machine*. http://www.aicas.com/jamaica/doc/rtsj_api.

[3] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th Symposium on Operating System Principles*, October 1991.

[4] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttle. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. *The landscape of parallel computing research: A view from Berkeley*. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[6] G. Banga, P. Druschel, and J. Mogul. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems . In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, October 1996.

[7] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems . In *Proceedings of the Third Symposium on Operating System Design and Implementation* , February 1999.

[8] Greg Bollella and James Gosling. *The Real-Time Specification for Java*, 2000. Addison-Wesley Longman Publishing Co., Inc.

[9] Intel Corporation. *Intel 64 and IA-32 Architectures, Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, 2007.

[10] Intel Corporation. *Intel Processor Identification and the CPUID Instruction*, December 2007.

[11] A. Corsaro and D. Schmidt. Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, September 2002.

[12] A. Corsaro and D. Schmidt. The Design and Performance of the jRate Real-Time Java Implementation . In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, October 2002.

[13] Brian P. Doherty. A Real-time Benchmark for Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '07)*, September 2007.

[14] J.L. Gustafson. Reevaluating Amdahl's law. In *Communications of the ACM 31(5), pp. 532-533*, 1988.

[15] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2003.

[16] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, June 2004.

[17] Tong Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC07)*, November 2007.

[18] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos Markatos. First-Class User-Level Threads. In *Proceedings of the 13th Symposium on Operating System Principles*, October 1991.

[19] Sun Microsystems. *The Sun Java Real-Time System*. http://java.sun.com/javase/technologies/realtime/index.jsp.

[20] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceeding of the 3rd IEEE International Conference on Distributed Computing Systems, pp. 22-30*, 1982.

[21] Andrew S. Tanenbaum. *Modern Operating Systems*, 1992. Prentice Hall Inc.

[22] A. J. Wellings. Multiprocessors and Real-Time Specification for Java. In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008.