# A New Approach To Real-Time Checkpointing

Antonio Cunei Jan Vitek

Department of Computer Science
Purdue University
West Lafayette, IN 47907, USA
{cunei,jv}@cs.purdue.edu

## Abstract

The progress towards programming methodologies that simplify the work of the programmer involves automating, whenever possible, activities that are secondary to the main task of designing algorithms and developing applications. Automatic memory management, using garbage collection, and automatic persistence, using checkpointing, are both examples of mechanisms that operate behind the scenes, simplifying the work of the programmer. Implementing such mechanisms in the presence of real-time constraints, however, is particularly difficult.

In this paper we review the behavior of traditional copy-on-write implementations of checkpointing in the context of real-time systems, and we show how such implementations may, in pathological cases, seriously impair the ability of the user code to meet its deadlines. We discuss the source of the problem, supply benchmarks, and discuss possible remedies. We subsequently propose a novel approach that does not rely on copy-on-write and that, while more expensive in terms of CPU time overhead, is unaffected by pathological user code. We also describe our implementation of the proposed solution, based on the Ovm RTSJ Java Virtual Machine, and we discuss our experimental results.

***Categories and Subject Descriptors*** D.4.7 [*Operating Systems*]: Organization and Design—Real-time systems and embedded systems; D.4.5 [*Operating Systems*]: Reliability—Checkpoint/restart

***General Terms*** Design

***Keywords*** Real-time, Checkpoint, Java, Virtual Machine



*VEE'06* June 14–16, 2006, Ottawa, Ontario, Canada.


## 1. Introduction

Checkpointing and rollback recovery are important mechanisms used to improve the fault-tolerance of a system. Should a transient fault occur, like a crash, a power failure, a hardware glitch, or even a transient software error, the most recently saved state can be restored and execution simply resumed. The increased reliability that is obtained is particularly appealing in the context of embedded and real-time systems, where manual intervention in case of problems might be impractical, or impossible. Within the context of a virtual machine (VM), the mechanism can be implemented at the VM level, mostly hiding the details of the checkpointing operations from the user code.

Despite the attractiveness of the technique implementing checkpointing in a real-time system is far from easy. The time-critical nature of real-time systems imposes precise constraints that the implementation must respect in terms of latency, overhead, and consumption of resources. A real-time virtual machine, for instance a system that implements the Real-Time Specification for Java (RTSJ) [5], has to deal with such constraints in a similar way.

In particular we want our checkpointing scheme to be *non-intrusive*, meaning that it should not interfere in the regular operation of the threads that run user code accessing and modifying memory as they progress, the so-called "mutators" [8]. Consequently, it must not suspend such threads for too long (it must exhibit low latency), nor cause excessive slowdowns or variations in their execution speed, making it difficult for the programmer to reason about the ability of their code to meet the required deadlines. Trivially, the checkpointing operation must *complete in a bounded time*. We want it to be available on *common hardware and operating systems*. Finally, we prefer it to *use efficiently system resources*, for example main memory.

In this paper we discuss previous attempts at defining an effective mechanism for the implementation of real-time checkpointing, and in particular the use of copy-on-write in order to save a snapshot of active memory while allowing the mutators to continue their execution. After describing existing approaches discussed in literature, we show how copy-on-write presents non-obvious problems that are in direct conflict with the desired real-time characteristics of the system. In particular we show that worst-case sequences of operations performed by the mutators while a background checkpointing is underway may cause the mutator itself to be slowed down dramatically, regardless of its priority. Deadlines may be missed as a result.

In order to avoid the problem, we introduce an alternative and novel technique that, at the price of some CPU time overhead, ensures that the mutators will never slow down unexpectedly because of a concurrent checkpointing operation, making our approach particularly suitable for real-time applications. We also show how our technique can be concretely implemented using commonly available POSIX system calls, and we describe our implementation on

the Ovm Real-Time Java Virtual Machine. The target applications of this work are safety/mission-critical embedded systems such as avionics applications where temporary power failure is a real possibility and fast restart essential. These applications usually run on moderately to fairly powerful processors and have hard real-time constraints. As multiprocessor systems are still relatively uncommon in hard real-time systems, and none of the available real-time virtual machines support multiprocessors, this work focuses on uniprocessor machines.

## 2. Implementation Strategies

The simplest possible approach to checkpointing is suspending the execution of all the mutators, saving the required state (typically a block of memory), and then resuming execution. Such a stop-the-world checkpointing trivially prevents the mutators from progressing for the whole duration of the operation, preventing high-priority threads from assuming control if needed. That approach is therefore not suited to a real-time context. The time required for a memory-to-memory copy, even on modern, high performance machines, can be quite high. Additionally, the time required increases with the size of the memory area that needs to be saved, therefore scalability becomes a concern.

Table 1 shows the time required to copy memory blocks using `memcpy()` on various computer systems.

| Times in ms, avg. of 20 tests | MacOS X | Linux PowerPC | Linux x86 | Linux MPC8260 |
|---|---|---|---|---|
| memcpy(), 2 MB | 1.418 | 2.378 | 1.894 | 33.961 |
| memcpy(), 4 MB | 2.950 | 5.136 | 4.060 | 67.843 |
| memcpy(), 8 MB | 6.082 | 10.355 | 8.581 | 135.886 |
| memcpy(), 16 MB | 12.068 | 20.549 | 18.120 | 271.277 |
| memcpy(), 32 MB | 24.021 | 41.195 | 37.205 | 541.829 |
| memcpy(), 64 MB | 46.059 | 81.091 | 75.379 | 1,075.948 |

"MacOS X" is a dual PowerPC G5 2.5GHz, Mac OS 10.4.1, 2GB RAM
"Linux PowerPC" is a PowerPC G5 1.8GHz, Linux kernel 2.6.9-rc3, 1GB RAM
"Linux x86" is a dual AMD Opteron 2.4GHz, Linux kernel 2.6.11.10, 6GB RAM
"Linux MPC8260" is an Embedded Planet board w/ MPC8260 300MHz, Linux kernel 2.4.22, 256MB RAM

**Table 1.** Time required to copy memory

In order to reduce the latency, and to create an implementation that is suitable for real-time applications, it makes sense to adopt a concurrent model of checkpointing. When the operation is initially started, the state of the memory at that time must be somehow preserved. While a background thread takes care of storing the saved state, execution should be free to resume normally, allowing high-priority threads to intervene when necessary.

### 2.1 Copy-On-Write

In order to preserve the state of the memory without having to copy it all at once, Li, Naughton, and Plank suggested the use of a "copy-on-write" mechanism [16, 17], by which, when a checkpoint is requested, all of the pages in the relevant memory area are copy-protected by using the Memory Management Unit. The execution of the mutators is then promptly resumed, while a background thread saves the write-protected pages. If a mutator tries to alter the memory content, a page access violation is triggered by the MMU and that single page is quickly copied in a separate buffer, allowing the mutator to continue without excessive delays.

## 3. Worst-Case Scenario

The use of copy-on-write, in principle, appears perfectly suited to real-time applications. The mutator is only suspended for the time strictly necessary to copy the block of memory imposed by the MMU granularity (usually a few kilobytes) after which the computation can resume. It can be expected, therefore, that the resulting latency is extremely modest. However, there is a fallacy in such expectation, which derives from a lack of assumptions about the memory access pattern of the mutators.

Consider, for example, the following scenario. Let us assume that the area of memory that should be saved during checkpointing is 64 megabytes and that the MMU has a granularity of memory protection of 4 kilobytes. When a checkpoint operation is requested by an application or by the system using copy-on-write, 16,384 pages are immediately made write-protected. From this moment on, each write operation performed by the mutator on a protected page will trigger a page fault, a 4KB memory copy, and a change in protection in the MMU tables.

Since no assumption is made about the memory access pattern of the mutator, the system must be able to cope with any subsequent mutator operation. In the worst case, the mutator might perform 16,384 subsequent write operations, *each in a different page!* The net result is that a task that would normally require just a handful of memory writes, requires now a full 64MB memory copy, plus 16,384 page access faults and that many MMU access changes, requiring overall a considerably longer time to complete. Regardless of the priority and the urgency of the mutator's activity, execution cannot continue until all of the mentioned operations are complete. Such large overhead can seriously affect the ability of the mutators to meet their deadlines, and is obviously detrimental to the real-time characteristics of the system.

| Times in ms, averages of twenty runs | MacOS X | Linux PowerPC | Linux x86 | Linux MPC8260 |
|---|---|---|---|---|
| Loop of 16,384 writes, one per page | 3.104 | 4.310 | 1.290 | 6.925 |
| 16,384 writes, each write causes one access violation and a change in page protection | 572.941 | 103.331 | 95.319 | 1,263.411 |
| 16,384 writes, each write causes one access violation, one page copy, and a change in page protection | 669.184 | 202.554 | 173.157 | 2,393.107 |

**Table 2.** Memory access times

A concrete measurement of the impact of copy-on-write in such a worst-case scenario is shown in Table 2. Even though the latency involved by each individual memory access fault is relatively small, the total slowdown is dramatic.

An application in which a large number of objects need sparse updating would be particularly sensitive to this issue. For example, a fast network router that keeps detailed traffic accounting information for remote nodes in a RAM-based database would need to update sparse information continuously in response to all the traffic simultaneously flowing through the device, leading to memory access patterns of the kind described.

In order to contain the overhead, and to give a precise upper bound to the overhead required by checkpointing, it is necessary to restrict the number of pages written to by the mutators whenever a checkpointing might be underway. Such a condition is extremely impractical to calculate and to enforce, especially considering that multiple threads might be in execution concurrently. Although it is reasonable to assume that the mutators exhibit the usual memory locality properties, the strict guarantees required by real-time systems cannot be fully satisfied when using copy-on-write checkpointing, unless precise assumptions are made about the memory access patterns of the mutators.

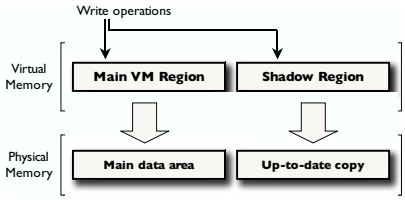



**Figure 1.** Main and shadow regions

## 4. A different approach

The troubles involved in the use of copy-on-write led us to devise a different kind of memory management technique, which avoids the problems previously described. Recalling the earlier discussion, the key to performing a concurrent checkpointing is being able to preserve, by some means, the state of memory at a given moment. In principle, nothing prevents us from maintaining a mirror copy of the main memory, constantly updated while the mutators run. When a checkpoint is encountered, the updates to the mirror copy can be suspended, and the mirror used for checkpointing while the mutators continue unimpeded. While this approach requires an auxiliary block of memory as large as the main memory, it is worth pointing out that copy-on-write has the same worst-case requirement: as we have seen, a full copy of the whole memory can be triggered in a very short time, arguably before even the first page has been fully transferred to permanent storage.

Maintaining a mirror copy of the main memory is easier if special hardware is used, but it can also be done purely in software by adding so-called write barriers, short sequences of instructions that follow write operations in the program code. We will show in the next section how, by using the Memory Management Unit in a rather unconventional way, we can not only keep the overhead required by the mirror copy down to a reasonable minimum, but we can also minimize the required latencies while keeping the execution speed of high-priority mutators easily predictable, making this approach very suitable for use in real-time systems.

### 4.1 Mirror Copy

We would like to implement a memory mirroring system purely in software. Therefore, every time something is written to memory, the same value must also be written to the mirror copy, but only if a concurrent checkpointing operation is not currently in progress. The write barrier, therefore, must involve some logic that allows the additional writes to be performed conditionally. The presence of write barriers involves some degree of cooperation by the compiler, or by the virtual machine. On the other hand, adding the required logic to an existing compiler or VM is relatively straightforward. Write barriers are used, for example, in generational garbage collectors [26] and are also usually involved in the implementation of scoped memory, as defined by the Real-Time Specification for Java (RTSJ) [5].

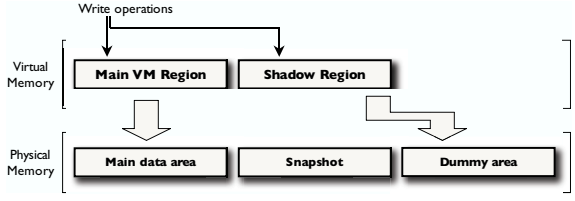



**Figure 2.** Preserving the snapshot

In our case the write barrier should detect whether a checkpointing is underway, and if not perform an additional memory write in a separate memory region. In detail, the write barrier should include a test instruction, a branch instruction (if checkpointing then do nothing), and the write operation to the mirror copy, possibly including the address calculation for the destination in the mirror copy. If some space in a machine register cannot be permanently sacrificed as a flag to indicate that a checkpointing is underway, an additional read from memory has to be added to the sequence.

Such a complex write barrier can cause a large penalty on the performance of the mutator. On the other hand, having a mirror copy available at all times would enable us to perform a checkpointing at any moment while maintaining a more predictable execution speed of the mutators. We manage to reduce to a minimum the performance impact of maintaining a mirror copy by using in an unconventional way the Memory Management Unit (MMU), as explained in the next subsections.

### 4.2 Moving Mirror

Figure 1 shows how the organization of memory would look like in a system that implements memory mirroring using write barriers. We will also refer to the mirror copy using the term “shadow”.

Since the mirrored copy is actually accessed through the MMU, an effective alternative to using an explicit test in the write barrier is to simply modify the MMU tables so that, when a checkpoint is underway, the updates to the mirror are redirected to a different area, and basically discarded. Altering the MMU mappings is a relatively quick operation that can be performed atomically. The tests can be therefore removed from the write barriers, reducing the additional code to a single additional write operation, which is always executed. When the mirror is being updated the MMU mapping will direct the additional writes to the copy. Conversely, when the checkpointing is being executed the additional writes will be redirected elsewhere, as shown in Figure 2. Using a third area

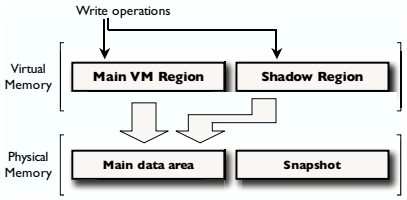



**Figure 3.** Writing twice the main data area

of real physical memory would obviously be quite wasteful, especially on memory-constrained embedded devices. A possible approach would be altering the mapping following the scheme shown in Figure 3, in which both the main VM region and the shadow are redirected onto the main physical memory area. In practice, this approach can still cause problems. In many microprocessor families it is important to explicitly reschedule machine instructions in order to keep the execution pipelines within the microprocessor full. Rescheduling the memory writes belonging to the write barriers, however, would cause unexpected effects, even leading to incorrect computations. Consider, for example, the code fragment shown in Figure 4, on the next page, written in a pseudo-assembler style.

The deferred execution of the write barriers can cause incorrect executions unless the compiler is made aware of the possible multiple mappings of the same physical memory area. That involves complicating the alias analysis stage, taking into account the additional aliasings and the fact that they may be different depending on whether certain memory areas are subject to the MMU remapping or not. In the case of the RTSJ, where memory scopes are created

and used dynamically and where it is not possible to detect statically whether memory writes take place in a specific scope or in the heap, such additional complexity might be undesirable.

```
read a -> R0
R0 * 2 -> R0
write R0 -> a
write R0 -> a+offs // will be moved
read a -> R1
R1 * 2 -> R1
write R1 -> a
write R1 -> a+offs // will be moved
read a -> R2
// R2 contains R0 * 4
```

a) before rescheduling

```
read a -> R0
R0 * 2 -> R0
write R0 -> a
read a -> R1
R1 * 2 -> R1
write R1 -> a
write R0 -> a+offs // a is overwritten !!!
read a -> R2
write R1 -> a+offs //
// R2 contains R0 * 2
```

b) after rescheduling

**Figure 4.** Interaction between instruction rescheduling and mirror writes

### 4.3 Multiple Mappings

In order to solve that problem, while at the same time using as little memory as possible, our solution is presented in Figure 5. During checkpointing, the new configuration of the MMU is such that every individual page in the addressing space corresponding to the shadow region is repeatedly mapped over a single, "dummy" page. Using this unusual but effective arrangement, all of the writes caused by the execution of the write barriers will end up in the same page, the eventual content of which will simply be ignored. Since we are not actually interested in saving the effect of those writes, the solution is perfectly acceptable. Furthermore, this approach is also likely to be more efficient than using a full-size additional memory area, since less commits from the cache to main memory will be needed (assuming write-back caching, as opposed to write-through caching).

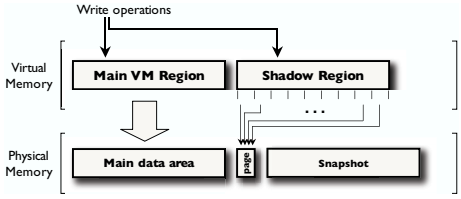
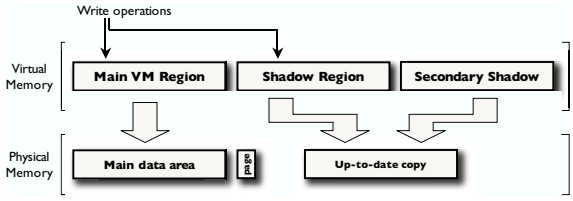


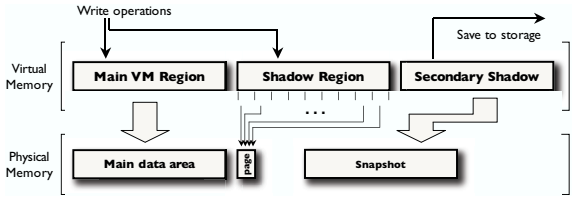

**Figure 5.** A single dummy page

The only remaining piece of the jigsaw is enabling access to the physical memory which contains the snapshot while the shadow memory region is mapped over the dummy page. We can easily do so by establishing a further permanent mapping of the snapshot area onto an additional region of virtual memory, which we will call "secondary shadow". This last region can be mapped anywhere in the addressing space, and will be the interface between the snapshot data and the checkpointing thread. The complete and final configurations of the MMU mappings during regular operation and during checkpointing are therefore represented in Figures 6(a) and 6(b). It is worth noting that, regardless of the complexity of the MMU mappings, switching from one to the other is just a matter of altering a few machine registers and it can be done in a very small number of instructions, with negligible impact on the resulting latency.

The actual background checkpointing can be performed as appropriate for the specific application, possibly making use of a number of techniques widely known in literature in order to minimize the amount of data actually transferred to the storage unit (recording differences with respect to the previous checkpoint, using "dirty pages" information, performing static program analysis, and so on). The specific strategy used to save data is completely orthogonal to our solution, and any mechanism will do as long as it respects the real-time constraints required by the system in terms of latency, and in terms of the total time allotted to the background checkpointing thread to complete.




6(a) During regular operation




6(b) During checkpointing

**Figure 6.** The complete memory configurations

### 4.4 Resynchronization

We have talked so far about maintaining an up-to-date shadow copy of the relevant memory region, and about detaching said copy in order to obtain a snapshot of the memory state at a given point in time. Once the background checkpointing is complete, it is necessary to resynchronize the state of the shadow with the main memory region, so that the cycle can start again.

Performing the resynchronization is particularly easy. To begin with, the MMU mapping is restored to the non-checkpointing configuration. From this moment on, all the write barriers will write again on both the main memory area and on the physical memory that contained the snapshot. In order to realign the two regions, it is now sufficient to scan linearly the main memory area and perform a simple copy of its content over the shadow. If the system offers information about so-called "dirty pages", identifying the pages that have been written to since the snapshot was last detached, the copy can be sped up by only copying those pages. The copy takes place at a lower priority than more urgent threads, and the synchronization can be preempted. It is necessary to make sure that no interference between the background copy and the regular mutators takes place. That is easy to ascertain in virtue of the following considerations.

To begin with, we should notice that we cannot allow a context switch to take place between the memory write and the corresponding write barrier. If a different thread were to write to the same memory location, the content of the shadow would become stale. If the write operation and the corresponding write barrier are executed atomically, it is sufficient to make sure that no context switch takes place between a read and the matching write during the resynchronization copy. Both conditions can be easily satisfied if, as is frequently the case in uniprocessor systems, yield/safe points are used for thread switching.

In the case of preemptive scheduling, the requirement of having atomicity of write and write barrier is no more stringent than what

is normally required by other very popular memory management schemes that involve write barriers, as for instance generational garbage collectors. The issue was discussed in some detail by Stichnoth et al. [24].

The atomic section in the background copy can be at a minimum literally just two machine instructions long, enclosing one read and one write, and therefore the operation has negligible impact, once again, on the overall latency.

## 5. A Concrete Implementation

We have implemented our solution in the context of the Ovm Real-Time Java framework, developed at Purdue University. Our implementation relies on common POSIX mechanisms, and uses some facilities offered by the Linux kernel. We will now shortly describe how the mechanisms introduced in the previous section have been implemented, and the following section will present our results.

### 5.1 MMU Mappings

There are two main classes of POSIX library calls that can be used to alter the mapping of virtual memory: `mmap()` utilities and shared memory calls. The call `mmap()` is normally used to support memory-mapped files, but it can also be used to reserve and map anonymous memory without a corresponding backing file. However, it cannot be used to map the same block of anonymous memory twice. Shared memory is more flexible in this respect, and it was used in our implementation.

A minor drawback of the shared memory mechanism is that there is no single call available in the standard that can be used to switch from one set of MMU mappings to another one atomically, as we would need. Switching between the different configurations, therefore, implies performing atomically a group of multiple `shmat()` and `shmdt()` calls.

In order to limit the number of operations, we allocate a dummy area equal in size to a given fraction of the size of the main memory area. If the snapshot size is one megabyte, for instance, we map sixteen times a unique 64KB dummy area. This strategy allows us to limit the total number of operations executed in the atomic section while still using a very modest amount of additional memory.

The remapping performed by `shmat()` and `shmdt()` may be time-consuming on some systems. As memory remapping is a relatively infrequent operation in most applications, the time used by some implementations may depend on the number of page mappings that need to be altered.

In our case we neatly sidestep the issue by taking advantage of the ability offered by modern microprocessors to use multiple page sizes simultaneously. By using the "Huge TLB" feature available in the Linux kernel, we can use pages large several megabytes rather than a few kilobytes. As the number of page mappings stored in the MMU tables is greatly reduced, the execution time of `shmat()` and `shmdt()` drops dramatically. On our test system, detailed later, switching the mapping of 16MB of memory requires just a few tens of microseconds.

The Real-Time Specification for Java defines three kinds of memory areas used for data storage: heap, scoped regions, and immortal memory [6]. In our test implementation we have decided to checkpoint immortal memory only, on the grounds that information stored in immortal memory is by definition long-lived, and more likely to deserve preservation across system crashes. Saving the entire heap would have been equally possible. Since, in the context of the RTSJ, it is quite difficult to determine statically in which kind of memory a given memory access will be performed, we must add a write barrier to all memory writes, regardless of whether the access is performed in the heap, in scopes, or in immortal memory.

### 5.2 Copying Data

Once a snapshot is frozen, the data can be stored on permanent storage. This phase requires particular attention in order to avoid introducing unwanted latency. As the Ovm system implements multithreading using "green threads" (user-lever thread scheduling), it was particularly important for us not to block on I/O operations. Support for fully asynchronous I/O had to be included, therefore, in the checkpointing mechanism. On particularly sensitive systems it may be advantageous to control the raw block device corresponding to the storage unit directly, bypassing the file system altogether. In our case, a high transfer rate of the checkpointed data was not a pressing requirement, while low latency was instead the defining aspect. Consequently we used the Asynchronous Input/Output (AIO) library calls, as defined in the POSIX.1b real-time extensions. Such calls are now available, with varying implementation efficiency, on many operating systems including Linux, Mac OS X, Solaris, and others. In order to remove possible latency originating from the `open()` operation, we set up the checkpoint file so that it is opened just once, at startup time, while only asynchronous calls are used afterwards.

### 5.3 Resynchronization

After all the data has been copied, the current state of the immortal memory is resynchronized, so that a new checkpoint can be started. As previously seen, it is important to avoid interference between the background copy and the mutators. In Ovm yield points never occur between the main write and the write barrier, nor between reads and writes during the copy, therefore the background copy is inherently safe. To perform the operation efficiently without impacting too much on latency, we implemented the synchronization copy using plain `memcpy()` calls on small chunks of memory, so that control can be promptly relinquished if high-priority threads need to run.

Summarizing, when a checkpointing is requested the implementation swaps atomically the MMU mappings using POSIX shared memory primitives and spawns a low-priority thread that takes care of the remaining stages of the checkpoint. The checkpointing thread enqueues the necessary asynchronous I/O operations using the AIO library, scheduling the transfer of the memory snapshot (using the secondary shadow address range) to disk. It then waits for the operating system to complete the asynchronous operations. Once all I/O is terminated, and all the I/O buffers have been properly flushed to disk, the checkpointing thread swaps MMU mappings again and proceeds with the synchronization, after which the procedure is complete, leaving room for further checkpointing operations.

## 6. Measurements

We measured the performance of our test bed implementation, with particular attention to the overhead of the write barriers and to the latencies of our checkpointing scheme. In particular, achieving low levels of latency is crucial in a real-time setting. The data obtained by our testing shows that the impact on the performance levels of the write barriers, while sensible, is overall contained thanks to our use of the MMU facilities. The latencies introduced by our mechanism are, as expected, quite small.

| Test | original | w/barrier | overhead |
|---|---|---|---|
| _201_compress | 9.118 | 10.095 | 10.72% |
| _202_jess | 4.376 | 4.928 | 12.61% |
| _209_db | 13.952 | 16.814 | 20.52% |
| _213_javac | 7.607 | 8.931 | 17.41% |
| _222_mpegaudio | 11.691 | 12.117 | 3.64% |
| _227_mtrt | 3.765 | 4.398 | 16.80% |
| _228_jack | 7.432 | 8.306 | 11.75% |

**Table 3.** Write barriers overhead

Table 3 shows a summary of the execution times, in seconds, of a number of SPECjvm benchmarks. The time required to run the tests on the stock Ovm are compared with the execution times on our modified version. The ahead-of-time, fully optimizing compiler of Ovm was used to run all the tests. The system used for the tests was a dual AMD Opteron 2.4GHz, Linux kernel 2.6.11.10, 6GB of RAM, using a 256MB heap, and the SPECjvm problem size was set to 100. The times are averages of 20 runs.

The overheads shown in the table may appear rather small, considering that the technique involves the duplication of all memory writes. However, several factors should be taken into account. First of all, Ovm implements the region-based memory management scheme required by the Real-time Specification for Java (RTSJ) [5], which requires some run-time checks to be performed during execution. The resulting overhead may contribute, to an extent, to mask the cost of the additional write barrier.

Other factors should also be considered. While in our implementation all writes induced by the program are duplicated, so that immortal memory is duplicated as well, the memory writes performed by the garbage collector are not replicated, as immortal memory is not affected by garbage collection in any case. That allows us to avoid some duplication overhead. Finally, although it might be expected that the processor cache efficiency is reduced by the additional memory accesses, it may be useful to point out that only writes are actually duplicated, while the shadow copy is never read from during the normal program execution. In our implementation, additionally, only the immortal memory is fully duplicated, while the heap is shadowed by repeatedly mapping a single memory block. As a consequence, the pressure on the cache is actually rather lower than what might be perceived thinking about a complete duplication of the memory areas.

It should be noted that such overhead is rather easy to predict for the programmer, being fundamentally associated with the execution of write operations. That is in stark contrast with the behavior of copy-on-write checkpointing, where a large and sudden slowdown might arise depending on conditions that are much more difficult to control, as discussed in Section 3.

In order to determine the all-important latency introduced by our checkpointing mechanism, we have conducted tests using the periodic real-time thread facility offered by the RTSJ. By measuring the time difference between the expected beginning of each period and the actual time when the thread resumes execution we can evaluate the latency introduced by lower-priority threads.

In detail, in our setup a low-priority thread initially allocates a large array in immortal memory. Subsequently, in an infinite loop, it performs the following actions:

- it sets up the array by storing in it some arbitrary data, overwriting one word every 4KB of memory
- it then sends a request to a checkpointing daemon, initiating the background checkpointing operation
- as the checkpointing proceeds, in the background, it quickly modifies the previous content of the array. We expect the checkpointing daemon to save the previous content of the array, as it was when the operation started, rather than the new one.
- the low-priority thread then waits for the background checkpointing to complete.

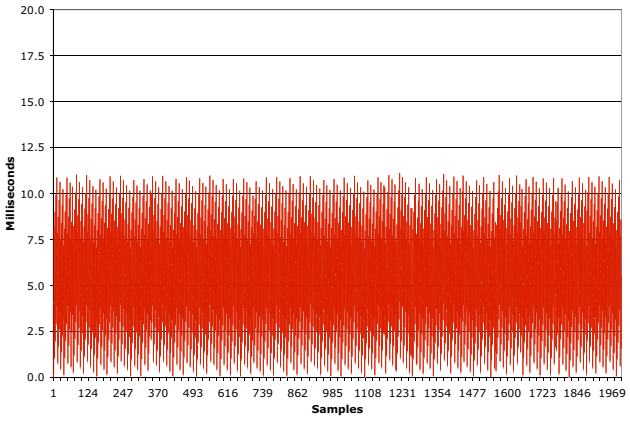



**Figure 7.** Response time of a periodical RTSJ real-time thread, period is 150ms with 10ms precision, standard Ovm

While all that happens, a high-priority thread is created, started, and suspended. A timer is prepared so that it triggers the high-priority thread at specified intervals. Such thread keeps the cpu busy for a short while and then suspends itself again, waiting for the following timer event. As our scheduling scheme uses priority preemption, we expect the high priority thread to execute immediately, preempting both the low-priority thread and the checkpointing daemon.

That allows us to give a first estimate of the latency imposed by the background checkpointing. Each time the high-priority thread is started, the current time is checked against the expected occurrence of the timer event. If some background activity prevents the high-priority thread from promptly resuming execution, such delay is detectable as additional latency, measured as the time from the expected timer event to the actual moment in which execution of the high-priority thread resumes.

The timer precision offered by the Linux kernel is 10 ms, which means that each timer event can occur up to about 10 ms after the programmed time. As we used a periodic timer, with a 150 ms period, our events are fired up to 10 ms after the beginning of each period.

As opposed to the timers, the clock has a much higher precision, allowing us to measure time differences with a resolution of 1 $\mu$s. We use that clock to measure the delay from the start of each period to the beginning of the activity of the high-priority thread. A delay of up to 10 ms is normal and due to the timer precision. Any delay significantly greater than 10 ms would indicate the presence of a non-interruptible section in the checkpointing daemon, offering us a first estimate of the latencies introduced by the background checkpointing.

We sampled the latency as described above 2000 times, using a period of 150 ms, comparing the standard Ovm configuration, with just the high-priority thread running, against our customized version of Ovm, with the checkpointing daemon and the low-priority thread in addition. The latter was used to trigger continuously and without pauses repeated background checkpoints of a 24MB array allocated in immortal memory (plus about 2MB of additional immortal space used by Ovm).

Figure 7 shows the graph obtained using the standard Ovm configuration, with no background threads. Since the declared timer

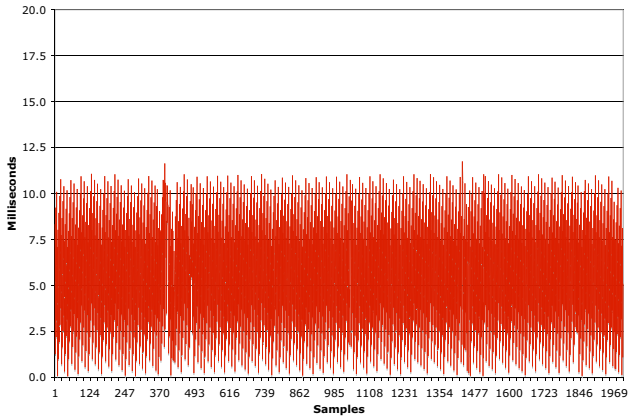



**Figure 8.** Response time of a periodical RTSJ real-time thread, period is 150ms with 10ms precision, background checkpointing

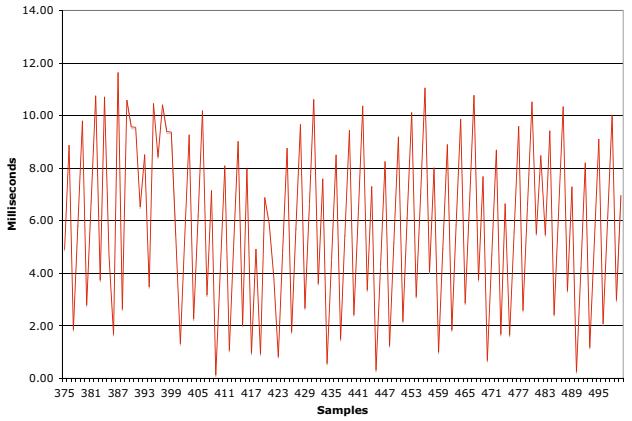



**Figure 9.** Response time of a periodical RTSJ real-time thread, period is 150ms with 10ms precision, background checkpointing, detail

resolution is 10 ms, the variation in the response time is overall normal and within the expected limits.

Conversely, the graph shown in Figure 8 shows the response times of our complete configuration, in which data is constantly mirrored and the low-priority threads repeatedly save the content of the immortal memory. During the time represented in the graph (5 minutes) the checkpointing daemon saved the 24MB area about 350 times. The delay introduced by the background checkpointing is almost unnoticeable, showing up in the graph only in a couple of points. Figure 9 offers a detail of the same test, pointing to a latency due to the checkpointing daemon of about 0.2 milliseconds.

In order to validate that measurement we instrumented the system, measuring the maximum time used by the non-interruptible sections within the time-critical checkpointing daemon. All measurements were taken using the same clock as before, with a 1 $\mu$s resolution. As a stress test, we ran a long-running test twice, measuring 40,000 samples, using a period of 60 ms, while saving more than 24MB of immortal memory to disk in the background approximately 2800 times. The two tests ran for a total of 1 hour and 20 minutes. The maximum latency was still just over 0.2 milliseconds.

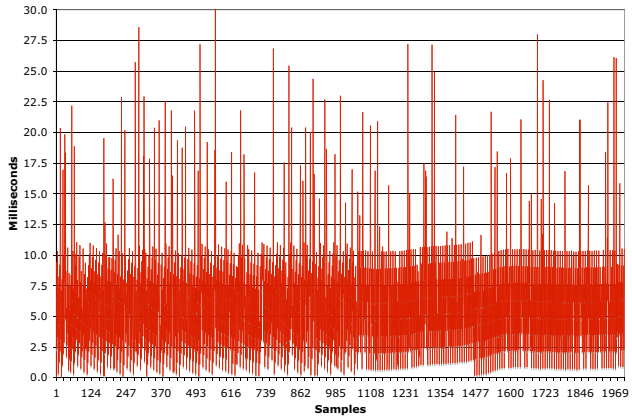



**Figure 10.** Response time of a periodical RTSJ real-time thread, period is 150ms with 10ms precision, validation test with 20ms non-interruptible pauses

Figure 10 shows the result of a further test that we used to validate our earlier test. Here, the part of code that saves data to disk in the checkpointing daemon was replaced with a routine that occasionally, at random, enters a 20 ms non-preemptible busy loop. The resulting delays clearly appear in the graph, confirming the correctness of our methodology.

## 7. Comparison with Copy-On-Write

In order to make a more direct comparison, we also implemented a copy-on-write checkpointing scheme in Ovm, and compared the behavior of the two implementations in terms of latency and impact on high-priority mutators. In this implementation, copy-on-write has at its disposal a buffer as large as the one we used for the previous technique, and the mechanisms used to save the checkpointing file and to spawn the background checkpointing daemon are identical. The scheme that we implemented works as follows. When a checkpointing is requested, the entire memory area that needs to be checkpointed (the RTSJ immortal memory in our case) is write protected, and control returns to the mutator. The low-priority checkpointing daemon begins to copy the memory pages in the background from the write-protected memory area to the side buffer. As each page is copied, its protection status is restored to read/write. If a mutator needs to write to a page that has not yet been copied, an illegal access signal is generated and the corresponding page is immediately copied to the buffer. A flag is kept for each page in the buffer, so that the checkpointing daemon can detect as it progresses that certain pages have already been copied, and will not copy them again.

When the background copy is complete, the side buffer contains a complete copy of the state that must be checkpointed and, similarly to our main technique, asynchronous I/O POSIX operations are used to save its content to disk. Once that operation is complete, the checkpointing daemon simply suspends itself, waiting for new requests. We used copy-on-write and our double-write approach to conduct a comparison, that we now describe in more detail.

In this test, a low-priority thread allocates a 24MB array in immortal memory. It then repeatedly fills it in with some random value, and requests a checkpoint, which is performed according to one of the two mechanisms earlier described. The low-priority thread then waits for the checkpointing to be complete, after which the sequence repeats. At the same time, a high-priority thread is triggered every 150 ms, similarly to our previous setting.

Each time the high-priority thread is triggered, it checks the current time, writes in the 24MB array some random value (one word every 4KB), then checks the current time again. The first measured time, when compared with the expected beginning of the period, offers us an estimate of the same kind of latency measured in the previous experiment. The difference between the two measurements, instead, tells us how long it took for the high-priority thread to complete its loop.

The measured preemption latency was similar in the two configuration, which is unsurprising considering that the same asynchronous operations were used to save the data. However, the time required by the high-priority thread to complete its task was considerably different, as shown in Figures 11 and 12. Using our scheme, the high-priority thread completed its task consistently in under 0.6 ms. Using copy-on-write the time varied widely, reaching values as high as 17.2 ms. This direct comparison confirms that adverse memory access patterns of the mutators impact heavily on copy-on-write, while our technique is not affected.

## 8. Additional considerations

The proposed technique offers better real-time guarantees than copy-on-write whenever, as is usually the case, the memory access patterns of the mutators cannot be precisely determined. A possible criticism is that the buffer region needs to be as large as the memory region that we want to checkpoint. While it is true that two copies of the memory region are required, it should be noted that copy-on-write has, in the worst case, the same limitation, as we previously mentioned. The technique suggested by Li et al. [16, 17] uses a fixed size buffer, but that buffer would be quickly filled up by a rapid sequence of page faults. As a consequence checkpointing would block, not just because of the page copies but because the buffers must be flushed to disk before it can be used again to copy new pages. In order to avoid pauses, the copy-on-write buffer has to be, once again, as large as the main memory region.

The use of the MMU is part of the novelty of this work, and its use allows us to reduce the write barrier to just one additional write instruction. Some variations on our approach could also be applied, however, to simpler embedded processors that do not use a memory management unit. A simple approach, previously mentioned, is testing explicitly a flag and performing the write operation conditionally. A possible alternative is performing the second write at an address obtained by summing the original address to an index register dedicated to that purpose, if one is available. By modifying the index register, all mirror writes would end up in a different memory range that does not refer to any physical memory. The memory controller, however, might have to be properly configured in order to acknowledge the fictitious writes as if physical memory were mapped to that address. Finally, since the MMU-based remapping described in the paper is relatively straightforward, it could also be implemented on systems without a MMU by adding some minimal external logic in hardware, implementing a memory multiplexing mechanism.

## 9. Related Work

While the literature on checkpointing is very rich and extensive, the work done specifically on real-time checkpointing is surprisingly modest. Li, Naughton, and Plank were the first to describe the application of copy-on-write in checkpointing [16]. Their work was then further refined, and a following paper by Plank and Li [17] presented a “low-latency, concurrent checkpointing” algorithm. Their low latency was defined as shorter than 0.1 seconds.

The natural complement to this and other papers with implementation details is the analysis of the schedulability characteristics of systems in the presence of checkpointing. Punnekat et al. [20, 22] analyze the schedulability issues involved in real-time checkpointing. Other related analyses are made by Shin et al. [23], Kwak et al. [13, 14], Zhang and Chakrabarty [32], Ranganathan and Upadhyaya [21] and others. Vaidya [28, 27] describes the correlation between overhead and latency in checkpointing.

The bibliography on checkpointing in general is extremely vast. The papers mentioned above, and the ones listed in the references section of this paper, can be an initial starting point for the interested reader.

## 10. Conclusions

Checkpointing is an attractive technique, useful to provide full or partial recovery capabilities after a system failure or during normal run-time, as well as for debugging purposes by preserving information after a system crash. The strict time constraints typical of real-time systems, however, make the implementation of checkpointing in such contexts particularly difficult. Previous work available in literature used copy-on-write techniques to implement real-time checkpointing. Our study, however, indicates that copy-on-write is

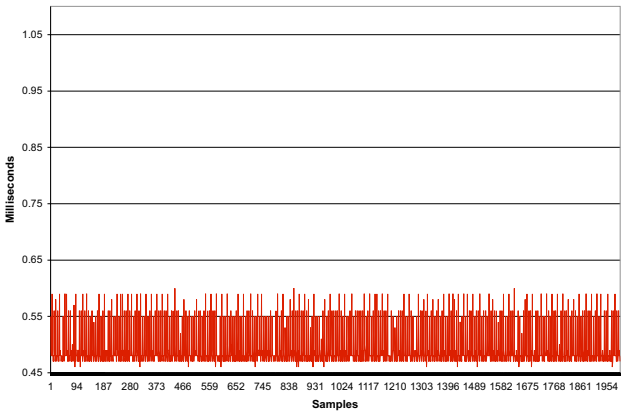



**Figure 11.** Double writes/MMU approach, execution time of a sample task in a high-priority thread, 2000 samples shown.

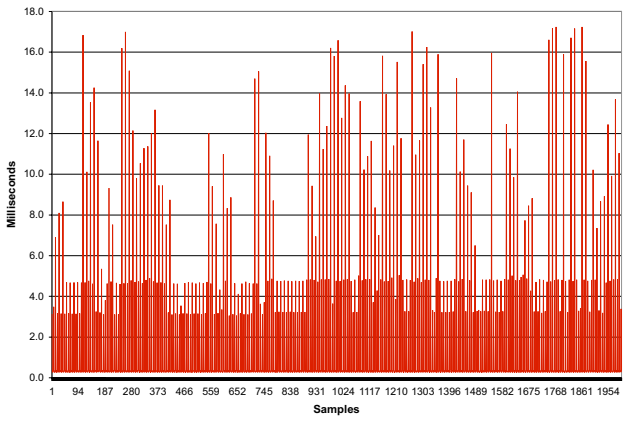



**Figure 12.** Copy-on-write, execution time of a sample task in a high-priority thread, 2000 samples shown.

not, in general, completely suitable as a core technique for the implementation of real-time checkpointing.

In this paper, we argue that copy-on-write can impose a sudden and unexpected overhead on the operation of mutators if no assumptions are made on their memory access patterns. The difficulties of formalizing and enforcing such assumptions led us to the formulation of an alternative technique that is not affected by the memory access pattern followed by mutators.

Our use of a mirror copy, while more expensive in terms of CPU time overhead, allows the mutators to progress in a way that can be controlled more easily by the programmer. The novel use of the MMU allows the required write barrier to be reduced to a bare minimum. Our experimental implementation, based on the Ovm RTSJ Java Virtual Machine, confirms our assumptions, exhibiting low latencies and avoiding the problems associated with copy-on-write. The latency measured in our implementation was 0.2 milliseconds.

While the CPU time overhead of our approach can be sensible, depending on the application, the only alternative would be to use copy-on-write while strictly enforcing the memory access pattern of all mutators, for every possible scheduling, which can be extremely difficult in practice. If real-time checkpointing is desired, and the overhead is deemed to be acceptable, our technique offers the certainty that no mutator will be unexpectedly slowed down as a side effect of the concurrent background checkpointing activity.

We would like to thank the anonymous reviewers for their insightful comments and suggestions. This work was partially supported by NSF Grants HDCCSR-0341304 and CAREER-0093282.

## References

[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 96–107, New York, NY, 1991. ACM Press.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.

[3] R. Bettati, N. Bowen, and J. Chung. Checkpointing imprecise computation. In *IEEE Workshop on Imprecise and Approximate Computation*, pages 45–49, Phoenix, AZ, Dec. 1992.

[4] G. Bollella, T. Canham, V. Carson, V. Champlin, D. Dvorak, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. Programming with non-heap memory in the Real-Time Specification for Java. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 361–369, New York, NY, USA, 2003. ACM Press.

[5] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[6] G. Bollella, J. Gosling, B. Brosgol, J. Gosling, P. Dibble, S. Furr, M. Turnbull, T. J. Bergin, and R. G. Gibson. *The Real-Time Specification for Java*. Addison-Wesley, New York, NY, 2000.

[7] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: a soft-state system case study. *Perform. Eval.*, 56(1-4):213–248, 2004.

[8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[9] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Symposium on Reliable Distributed Systems*, pages 39–47, 1992.

[10] R. Geist, R. Reynolds, and J. Westall. Selection of a checkpoint interval in a critical-task environment. *IEEE Trans. Reliability*, 37(4):395–400, 1988.

[11] V. Grassi, L. Donatiello, and S. Tucci. On the optimal checkpointing of critical tasks and transaction-oriented systems. *IEEE Trans. Softw. Eng.*, 18(1):72–77, 1992.

[12] C. M. Krishna, Y.-H. Lee, and K. G. Shin. Optimization criteria for checkpoint placement. *Commun. ACM*, 27(10):1008–1012, 1984.

[13] S. Kwak, B. Choi, and B. Kim. An optimal checkpointing-strategy for real-time control systems under transient faults. *IEEE Transactions on Reliability*, 50(3):293–301, September 2001.

[14] S. W. Kwak, B.-J. Choi, and B. K. Kim. Checkpointing strategy for multiple real-time tasks. In *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000)*, pages 12–14, Cheju Island,South Korea, Dec. 2000.

[15] H. Lee, H. Shin, and S. L. Min. Worst case timing requirement of real-time tasks with time redundancy. In *6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA '99)*, pages 410–413, Hong Kong, China, Dec. 1999. IEEE Computer Society.

[16] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. *SIGPLAN Not.*, 25(3):79–88, 1990.

[17] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):874–879, 1994.

[18] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[19] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[20] S. Punnekkat and A. Burns. Analysis of checkpointing for schedulability of real-time systems. In *4th International Workshop on Real-Time Computing Systems and Applications (RTCSA '97)*, pages 198–205. IEEE Computer Society, Oct. 1997.

[21] A. Ranganathan and S. Upadhyaya. Simulation analysis of a dynamic checkpointing strategy for real-time systems. In *27th Annual Simulation Symposium*, pages 181–187, La Jolla, CA, Apr. 1994. IEEE Computer Society.

[22] A. B. S. Punnekkat and R. Davis. Analysis of checkpointing for real-time systems. *Real-Time Systems Journal*, 20(1):83–102, Jan 2001.

[23] K. G. Shin, T.-H. Lin, and Y.-H. Lee. Optimal checkpointing of real-time tasks. *IEEE Trans. Comput.*, 36(11):1328–1341, 1987.

[24] J. M. Stichnoth, G.-Y. Lueh, and M. Cierniak. Support for garbage collection at every instruction in a Java compiler. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1–4, 1999.

[25] A. N. Tantawi and M. Ruschitzka. Performance analysis of checkpointing strategies. *ACM Trans. Comput. Syst.*, 2(2):123–144, 1984.

[26] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Software Development Environments (SDE)*, pages 157–167, 1984.

[27] N. H. Vaidya. On checkpoint latency. Technical report, College Station, TX, USA, 1995.

[28] N. H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Trans. Comput.*, 46(8):942–947, 1997.

[29] N. H. Vaidya. Staggered consistent checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):694–702, 1999.

[30] J. C. Wu and S. A. Brandt. Storage access support for soft real-time

applications. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, Toronto, Canada, May 2004.

[31] Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003)*, pages 10918–10925, Munich, Germany, Mar. 2003. IEEE Computer Society.

[32] Y. Zhang and K. Chakrabarty. Fault recovery based on checkpointing for hard real-time embedded systems. In *18th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2003)*, pages 320–327, Boston, MA, Nov. 2003.

[33] Y. Zhang and K. Chakrabarty. Dynamic adaptation for fault tolerance and power management in embedded real-time systems. *Trans. on Embedded Computing Sys.*, 3(2):336–360, 2004.