FREM: A Fast Restart Mechanism for General Checkpoint/Restart

Yawei Li, *Member*, *IEEE*, and Zhiling Lan, *Member*, *IEEE Computer Society*

Abstract—As failure rate keeps on increasing in large systems, applications running atop restart more frequently than ever. Existing research on checkpoint/restart mainly focuses on optimizing checkpoint operation, without paying much attention to the restart operation. As a result, application restart latency maybe substantial, which greatly threatens system dependability and performance. To attack the restart latency problem, in this paper, we present FREM, a fast restart mechanism for general checkpoint/restart protocols. By dynamically tracking the process data accesses after each checkpoint, FREM masks restart latency by overlapping application recovery with the retrieval of its checkpoint image. We have implemented FREM as a prototype system and tested it under Linux environments. Extensive experiments with real applications demonstrate that it can effectively reduce restart latency by over 50 percent on average, as compared to the conventional restart mechanisms.

Index Terms—Fast restart, operating system, Linux, fault tolerance, high performance computing.

1 Introduction

Checkpoint/restart (C/R) is a long-standing fault tolerance technique to alleviate the impact of system failure. It saves the state of a running application to a stable storage, and in case of failure the application can restart from the latest checkpoint image, rather than from scratch. Further, it provides a great flexibility for networked computing by allowing a failed application to restart on an alternative machine rather than waiting for the repair of the failed machine, e.g., via a mounted NFS file system. Nevertheless, existing checkpoint-based restart involves nontrivial restart latency. Here, *restart latency* is the amount of time that elapses from the initiation of checkpoint retrieval to the reexecution of the failed application. Existing restart protocols typically require the entire checkpoint image to be available on the destination machine before reexecuting the failed application. A substantial amount of time is required for network transmission and disk I/O of the checkpoint image. Moreover, the fast-growing memory consumption of modern applications further deteriorates failure restart [10], as the memory footprint is a major contributor to the checkpoint image.

Traditional C/R research mainly focuses on optimizing checkpoint frequency [1], [4], [18], [28], [39], [42] or reducing checkpoint overhead [26], [27], [32], with little attention paid to restart latency. While failure recovery has been previously studied in various fields including operating systems, databases, and internet services [2], [24], [25], these studies are either specific to a particular problem domain or hardly applicable to improve checkpoint-based restart.

With the rapid development of networked systems, reducing restart latency is becoming critical and needs to be treated as a first-class citizen, just like optimization of checkpoint operation itself. There are two major reasons for this. First, due to the increasing failure rates, failure restarts are becoming as a norm rather than an exception on large-scale networked systems. Recent studies have pointed out that the mean-time-between-failure (MTBF) of teraflop and petaflop machines are only on the order of 10-100 hours [6], [14], [22], [33]. As a result, applications running on these systems are forced to restart more frequently than ever. Second, due to the increasing system scale and application size, restart latency grows rapidly. Experiments have shown that restart latency of a typical large-scale application could be as high as over 10 minutes [1]. Hence, in this study, we focus on optimizing checkpoint-based restart.

1.1 Key Observations

Delving into the current restart mechanisms, we have made three important observations:

1. *They fail to explore parallel opportunities provided by modern hardware devices.* Modern computer peripheral devices, such as network cards and disk controllers, can offload heavy I/O duties from CPUs, thereby providing abundant opportunities for overlapping the retrieval of the checkpoint image with the reexecution of the failed application.
2. *Not all checkpoint data are immediately needed for application restart.* With existing restart protocols, the application must restore its entire address space from the checkpoint image before moving forward. This is not necessary since most applications only require a small portion of its checkpoint data for restarting during a short period of time. Existing restarting mechanisms do not explore such a temporal locality for application restart.
3. *Data access patterns of an application are tractable after its resurrection.* Upon failure restart, the application

- *Y. Li is with Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043. E-mail: yaweili@google.com.*
- *Z. Lan is with the Department of Computer Science, Illinois Institute of Technology, 10 W. 31st Street, Chicago, IL 60616. E-mail: lan@iit.edu.*

Manuscript received 5 Mar. 2009; revised 12 Feb. 2010; accepted 25 Apr. 2010; published online 7 June 2010.
Recommended for acceptance by M. Eltoweissy.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-03-0105.
Digital Object Identifier no. 10.1109/TC.2010.129.

rolls back to the last checkpoint and repeats its execution done before the failure. During this replay, we can precisely know which data are immediately needed for restart. This information could be utilized to reduce restart latency.

1.2 Paper Contributions

Motivated by the above observations, we propose *FREM, a Fast REstart Mechanism* to enhance general C/R protocols by reducing their restart latency. *The core idea* is to enable quick restart on a partial checkpoint image based on the knowledge of process data accesses after each checkpoint. More specifically, after each checkpoint, a user-transparent system support is provided to record the data accessed within a time interval (denoted as *the tracking window*). The recorded datum is called *the touched set*, which is defined as the intersection of the process working set during the tracking window and the latest checkpoint. The description of the touched set is stored to the stable storage, along with the checkpoint image. Upon failure recovery, rather than retrieving the entire checkpoint image for restart, FREM first retrieves the touched set and restarts the application on this partial checkpoint image. Meanwhile FREM spawns a dedicated thread to simultaneously load the remaining of the checkpoint data to the process address space. By doing so, FREM *hides restart latency* by overlapping application restart with checkpoint retrieval.

While the idea is straightforward, the design and implementation of FREM is challenging, especially given the complexities of modern computer hardware and software. In this paper, we present a suite of techniques for the design of FREM. These include *a tracking mechanism* to precisely identify the touched set, *an adaptive mechanism* to determine the tracking window, and *a partial image loading mechanism* to coordinate checkpoint retrieval with process recovery. Together, they form a transparent runtime support for fast recovery of user applications in networked systems.

We have implemented FREM in Linux and evaluated its effectiveness under a variety of networked environments. The implementation is built on a well-known checkpoint tool called Berkeley Lab Checkpoint/Restart (BLCR) tool [10]. Our experiments with the SPEC CPU2006 [31] benchmark suite demonstrate that FREM can effectively reduce restart latency by over 50 percent on average, as compared to generic checkpoint/restart protocols. In addition, this substantial performance improvement comes with a modest tracking overhead. To the best of our knowledge, this is the first effort to reduce restart latency in general C/R protocols.

While FREM is built on our previous work [15], the FREM system presented in this paper has made two crucial improvements. First, we employ an adaptive approach to estimate the tracking window, which will greatly reduce restart latency and improve the reliability of FREM. Second, we redesign the entire restart part by allowing on-demand remote page fault handling and efficient data loading. Together, not only the correctness of data loading is enhanced, but also the overhead of data loading is reduced.

1.3 Paper Organization

The remainder of this paper is organized as follows: Section 2 briefly discusses the related work. Section 3 gives an overview of FREM, followed by a detailed description of its postcheckpoint tracking in Section 4 and its fast restarting mechanism in Section 5. Section 6 presents our evaluation of FREM with a number of real-world applications under various networked configurations. Finally, Section 7 summarizes the paper and points out future directions.

2 Related Work

The idea of fast restart is not new, and has been studied in several fields. For example, Baker and Sullivan have discussed the use of a "recovery box" (a protected area of nonvolatile memory) in the Sprite system to store crucial process state needed for fast recovery [2]. In database systems, quickly resuming transaction process is the focus. For example, the Oracle systems have applied "on-demand rollback" to allow new transactions to execute while the rollback is still being performed [24]. Recently, more attention has been paid to fast recovery for Internet services. A representative work is the ROC project from Berkeley and Stanford [25]. It focuses on providing a holistic solution for postfailure recovery of Internet services by using fine-grained system partitioning and recursive restart. Rao et al. have proposed a class of hybrid protocols to enable the failure-free performance of sender-based protocols while approaching the performance of receiver-based protocols during recovery [29]. *FREM is fundamentally different from these studies in that it emphasizes the reduction of restart latency for general C/R-based applications.*

Existing studies on C/R mainly focus on checkpoint optimization. One major direction is to determine an optimal checkpoint frequency. Young has derived a simple first order approximation of the optimal checkpoint interval, based on the assumption of Poisson failure arrivals [39]. By considering failures during checkpointing or recovery, Daly has proposed a higher order interval approximation model by extending Young's work [4]. Vaidya has developed an improved checkpoint interval by differentiating checkpoint latency and overhead [39]. Plank and Thomason have investigated the optimal checkpoint interval for parallel applications [28]. Additionally, there are numerous papers on dynamic checkpoint scheduling, such as aperiodic checkpointing [18] and cooperative checkpointing [22]. The other major direction is to reduce checkpoint overhead, especially the disk I/O time. Latency hiding and memory exclusion are two key techniques [26]. The studies in this category include copy-on-write [17], incremental checkpointing [32], and diskless checkpointing [19], [27], [43]. More checkpointing optimization techniques can be found in [26].

While checkpoint optimization has attracted most research attention over the past, there are only a few studies focusing on the recovery component in C/R protocols. Plank and Thomason have considered allocating extra compute nodes to reduce application downtime, assuming the checkpoint is stored on a shared remote storage [28]. Some efforts have been done to optimize the recovery time by using improved message logging protocols. Rao et al. have proposed a hybrid logging protocol to reduce log retrieval and roll forward time during the recovery of message passing applications [29]. Gupta et al. have expedited failure recovery in communication-induced checkpoint through the

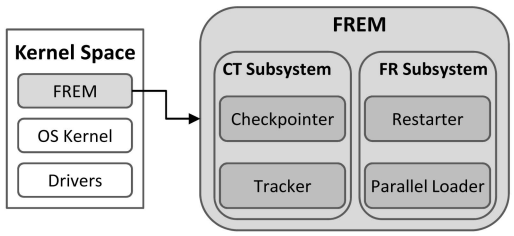


Fig. 1. Overview of FREM.

elimination of unnecessary comparisons when determining the consistent global state [9]. In some C/R protocols, the reduction of recovery time is achieved as a byproduct of checkpointing optimization. For example, in diskless checkpointing protocols [19], [27], the recovery of a process takes less time since the checkpoint image is stored in the RAM of spare nodes. Similarly, the double-in-memory checkpointing technique used in FTC-Charm++ helps reduce recovery time [43]. Despite the technical differences, all these C/R protocols share a common feature, that is, an application cannot restart only until its entire checkpoint data are restored, either from a disk or RAM. *Instead, FREM aims to reduce restart latency by overlapping checkpoint retrieval with application execution. It provides a set of fast recovery enhancements for general C/R tools.*

Utilizing paging mechanism to optimize system performance is a common approach in system research. For example, demand paging is widely used in modern operation systems [37]. It allows a process to begin execution with portion of its pages available in the physical memory. Read-ahead is another popular technique, which prefetches data pages so that the amortized I/O latencies can be decreased when locality is expected [40]. Similarly, in distributed systems, many process migration protocols leverage page-level data access patterns to achieve fast process restart on the destination machine [20]. The Choices system allows the minimal state of a process to be sent to the remote node for execution while the remaining data are being transferred in parallel [30]. Clark et al. have applied precopy to migrate live Xen virtual machine by iteratively transferring dirty pages, for the purpose of reducing application freeze time [3]. *While FREM also exploits paging mechanism, it distinguishes itself from the above studies at two aspects. First, it is able to track the precise pages that will be immediately needed for process reexecution. Second, unlike existing migration protocols focusing on efficient page transferring between the source and the destination machines, FREM intends to reduce restart latency. Further, it does not require any live copy of the application from the source machine.*

3 System Overview

As shown in Fig. 1, FREM is a kernel module residing in the OS kernel to provide transparent C/R support. It consists of *a postcheckpoint tracking (CT)* subsystem and *a fast restarting (FR)* subsystem. The CT subsystem contains two components: the checkpointer and the tracker.

Upon each checkpoint, the checkpointer dumps the application state to a remote storage, and the tracker tracks and collects the touched set after each checkpoint. The

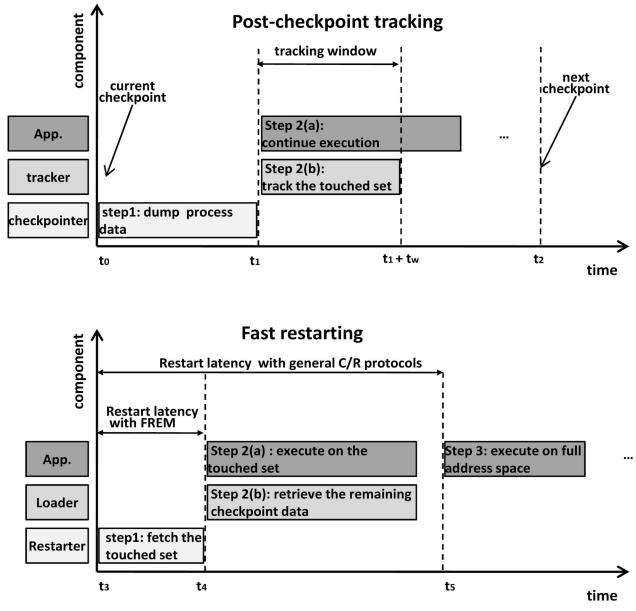


Fig. 2. Workflows of FREM Main Components.

FR subsystem contains two components: the restarter and the parallel loader. Upon failure recovery, the restarter retrieves the touched set and reruns the application. The parallel loader loads the remaining bulky checkpoint data concurrently with application execution.

To illustrate how these components collaborate with each other, Fig. 2 presents the steps of a typical FREM workflow in detail.

The *CT* subsystem takes the following steps during *prefailure execution*:

- *Step 1.* At time t_0, the checkpointer is triggered by an external signal, such as a checkpoint request. The checkpointer first stops all application threads and dumps their state to a stable remote storage.
- *Step 2a.* At time t_1 when the checkpointer finishes its operation, the application execution is resumed.
- *Step 2b.* At the same time, the tracker starts tracking the touched set during the time period $[t_1, t_1 + t_w]$. The tracker uses the access bit of each page table entry (PTE) to track which pages have been accessed during the tracking window t_w (i.e., the touched set). The information of the touched set is saved as a descriptor file to the stable storage, along with the checkpoint image. The above checkpointing and tracking steps repeat upon each checkpoint cycle. In case that a failure occurs during the tracking window, the application resumes to the conventional restart approach to recover the process.

The *FR* subsystem takes the following steps during *postfailure recovery*:

- *Step 1.* At recovery time t_3, the restarter starts to read in the touched set descriptor from the remote storage. According to the descriptor, the restarter retrieves the touched set pages and other necessary process information such as registers, signals, and file descriptors from the checkpoint file to the destination machine.

- *Step 2a.* At time t_4 when the touched set is loaded, the application process overlays the restarter process and starts its execution.
- *Step 2b.* At the mean time, the parallel loader starts as a dedicated thread to retrieve the remaining bulky checkpoint data. The time range of [t_4, t_5] is called *the overlapping time*. During the overlapping time, the application execution and data retrieval are performed concurrently.
- *Step 3.* When the loader thread finishes the checkpoint retrieval at time t_5, the application starts to execute on its full address space.

The rationale of FREM is to achieve the overlapping between process execution and checkpoint retrieval. Consequently, the restart latency is reduced to the retrieval time of a small portion of its checkpoint data, e.g., ($t_4 - t_3$) shown in Fig. 2.

Our design of FREM intends to follow three principles:

1. *Efficiency*. FREM should not only reduce restart latency, but also keep its runtime overhead low.
2. *Generality*. FREM should be independent of specific characteristics of the underlying hardware platforms and checkpoint protocols. It should be easily integrated with any general checkpointing tool in various environments.
3. *Transparency*. FREM is intended to be transparent to user applications by residing in the OS kernel. It should be plugged into the kernel in a nonintrusive way such that the OS code can accommodate FREM without any modification.

Because of the increasing complexities of computer hardware and software, the design and implementation of FREM is challenging. In the following two sections, we will describe a suite of techniques to address these challenges, i.e., *how to accurately identify the touched set*, *how to set the tracking window*, and *how to effectively and correctly load data during the overlapping time*.

4 Design and Implementation of Postcheckpoint Tracking

The CT subsystem consists of two components: the checkpointer and the tracker. Since the checkpointer follows general checkpointing protocols, for brevity, we focus on the design of the tracker here. The tracker addresses the first two challenges listed in Section 3, namely how to accurately identify the touched set and how to appropriately set the tracking window size.

4.1 Identification of the Touched Set

FREM uses the access bits to track the touched set. However, to merely scan the access bits at the end of the tracking window is not sufficient. There are two types of potential inaccuracies in the identification of the touched set: 1) *false positives* where pages not of interest are included in the touched set and 2) *false negatives* where pages of interest are missing from the touched set. These errors have serious performance implications. False positives increase restart latency as unnecessary checkpoint data are delivered for quick process restart. False negatives cause extra execution overhead. As some data immediately needed by the resumed process are not available on the destination machine, the application must be interrupted to handle cross-network remote page faults during execution. A naïve mark-and-scan algorithm cannot accurately capture the touched set.

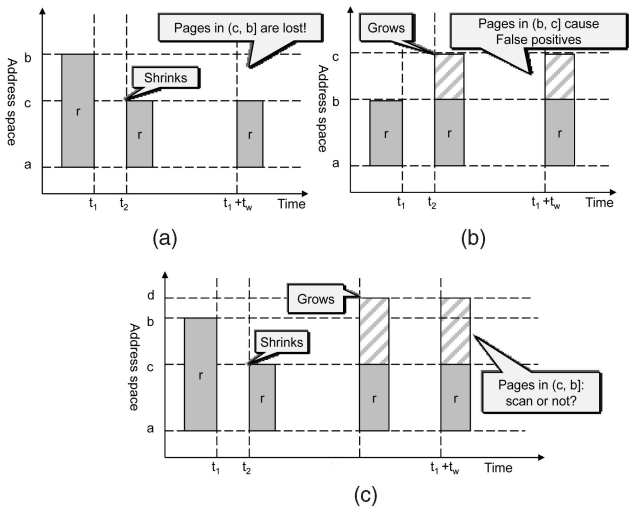


Fig. 3. Pitfalls caused by dynamic memory usages.

In essence, all potential inaccuracies stem from the interferences on the access bits due to the hardware and software complexity in modern computer systems. Through a systematic analysis, we classify the interferences from three system layers in a top-down manner: application dynamic memory usages, OS intervention, and hardware bypassing.

4.1.1 Application Dynamic Memory Usage

For a typical application, its process addresses constantly change and thus introduce complications in the identification of the touched set. As shown in Fig. 3, we identify three types of representative pitfalls stemming from dynamic memory usage.

In Fig. 3a, at time t_1, the memory region r (the region of [a, b] in address space) is saved on stable storage as part of the checkpoint image. At time t_2, a deallocation operation shrinks r to [a, c] and frees all the pages in (c, b]. Therefore, when FREM scans for the touched set at time $t_1 + t_w$, a false negative error may occur—the pages in (c, b] accessed during time (t_1, t_2) are lost. In Fig. 2b, the memory region r is checkpointed at time t_1. At time t_2, an allocation operation extends r to [a, c]. At the scan time $t_1 + t_w$, the pages in (b, c] accessed during time (t_2, $t_1 + t_w$) should not be counted in the touched set; otherwise a false positive error is introduced. Recall that the touched set is defined as the intersection of the process address space saved in the checkpoint image and its working set during the tracking window. Although the pages in (b, c] were accessed during time (t_2, $t_1 + t_w$), they are not part of the checkpoint image, indicating they do not need to be retrieved during the restart phase. In Fig. 3c, the memory region r is checkpointed at time t_1. At time t_2, a deallocation operation shrinks r to [a, c]. Then at time t_3, an allocation operation extends it to [a, d]. The question is whether we should scan the pages in (c, b] or not? The answer is twofold. At time t_2, just before their

deallocation, the pages in $(c, b]$ should be tracked because they are part of the checkpoint image; otherwise a false negative error is introduced. At time $t_1 + t_w$, the pages in the same range $(c, b]$ are actually newly allocated and should not be counted in the touched set; otherwise a false positive error is introduced.

The above analysis indicates that recording memory deallocation is critical since the touched set is always a subset of the memory regions at the checkpoint time, which monotonically decreases during the tracking window. Based on this key observation, we develop a simple yet effective algorithm to track the touched set. At the beginning of the tracking window, FREM records the memory regions saved by the checkpointer (denoted as *the candidate range*). Next, FREM intercepts the standard POSIX system calls brk and mmap to track memory deallocation. Whenever a memory region is freed, FREM scans the intersection between this region and the candidate range, and then update the candidate range by removing this intersection. The algorithm can eliminate the false positives and false negatives shown in Fig. 3.

4.1.2 OS Intervention

OS intervention is another contributing factor to the identification errors. Most modern operation systems adopt some variant of clock algorithm that periodically sweeps the page table to collect page usage information [12]. This process cleans the access bits. As a result, false negatives will be produced when the sweeping activity takes place inside the tracking window. To address this issue, FREM monitors such sweeping activities and scans the access bits before they are cleaned by the OS.

4.1.3 Hardware Bypassing

Two types of hardware optimizations will cause memory accesses bypass the access bits, thereby leading to false negatives: the Translation Lookaside Buffer (TLB) cache in processors and DMA operations in peripheral devices.

TLB translates logic addresses to physical addresses so that the processor could bypass the PTE and directly address the RAM. In other words, TLB hits may introduce false negatives. For instance, suppose that an access bit of a PTE is marked as unaccessed while its TLB entry remains valid at the beginning of the tracking window. In the subsequent access of a datum in this page, the CPU directly read the address translation information from the TLB entry, thereby leaving its access bit in the PTE stale. When the tracker walks the page table at the end of the tracking window, this page would be excluded from the touched set, thereby resulting in a false negative.

To address this issue, at the beginning of the tracking window, FREM not only clears the access bit in the PTE, but also invalidates the corresponding TLB entry. By doing so, FREM enforces a TLB miss upon the first access of a page to update the corresponding access bit in the PTE.

In addition, a direct memory access (DMA) operation on user-space memory like direct I/O also bypass the CPU, thereby causing false negatives. We suggest to instrument the corresponding device driver to appropriately set the access bits of PTEs whenever such a DMA transfer is initiated.

4.2 Estimation of the Tracking Window

The tracking window size t_w plays an important role in the identification of the touched set. A short window may yield an inadequate touched set and consequently cause a significant amount of remote page faults during recovery. Here, *a remote page fault* refers to the case when the restarting application accesses a page that has been not retrieved yet. On the other hand, a long tracking window may produce a large touched set and thus slow down the restart process. Furthermore, a large window size makes FREM more vulnerable to failure as the likelihood of failure grows with time.

Ideally, the tracking window should be set such that when the recovered application first accesses a datum outside the touched set, the retrieval of the remaining checkpoint image just finishes at the very moment. In other words, the size of the tracking window, t_w, should be equal to the time for FREM to retrieve the remaining image:

$$t_w = \frac{checkpoint\_size - touched\_set\_size}{data\_transfer\_rate}. \quad (1)$$

Here, *data_transfer_rate* refers to the data transfer rate from the remote storage to the local RAM, which can be measured through benchmarking [41].

Based on the above rationale, an adaptive method is adopted to estimate the tracking window. More specifically, initially we set the window size t_w^0 to the retrieval time of the entire checkpoint image; during each checkpoint cycle i, at the end of the tracking window, FREM records the size of the touched set and calculates its generation rate (i.e., $touched\_set\_size^i/t_w^i$); during the next checkpoint cycle $(i+1)$, FREM applies the LAST prediction method [5], and the touched set generation rate obtained from cycle i will be used as an estimate for the generation rate of cycle $(i+1)$. Thus,

$$\begin{aligned} t_w^{i+1} &= (checkpoint\_size^{i+1} - touched\_set\_size^{i+1})/ \\ &\quad (data\_transfer\_rate) \\ &= (checkpoint\_size^{i+1} - t_w^{i+1} \\ &\quad \times touched\_set\_generation\_rate^{i+1})/ \\ &\quad (data\_transfer\_rate) \\ &\approx (checkpoint\_size^{i+1} - t_w^{i+1} \\ &\quad \times touched\_set\_generation\_rate^{i})/(data\_transfer\_rate) \\ &= \left(checkpoint\_size^{i+1} - t_w^{i+1} \times \left(touched\_set\_size^i/t_w^i\right)\right)/ \\ &\quad (data\_transfer\_rate). \end{aligned}$$

After simple mathematic transformation, we obtain the following equation:

$$t_w^{i+1} = \frac{checkpoint\_size^{i+1}}{date\_transfer\_rate + \frac{touched\_set\_size^i}{t_w^i}}. \quad (2)$$

Compared with our previous work [15] that employs a conservative method, the above adaptive strategy allows FREM to capture the recent behavior of the application and thus to make more accurate runtime estimation.

The value of t_w calculated by (2) could be very large when the data transfer rate is low, e.g., when data are transferred across a wide area network. This may increase

the failure possibility of FREM during the tracking window. To prevent this case, FREM adopts an upper bound threshold to cap t_w, usually in a couple of minutes in practice. By adaptively tuning the tracking window, FREM is capable of adapting to dynamic application behaviors.

4.3 CT Implementation in Linux

We have implemented the CT subsystem in Linux as a kernel module. We summarize our specific implementation as follows:

1. To invalidate a TLB entry, the architecture-independent kernel function flush_tlb_page is used.
2. In Linux, both brk and mmap system calls reply on the kernel function do_munmap. Hence, for convenience FREM directly intercepts this function to track memory deallocation.
3. To monitor page table sweeping from OS, FREM instruments the kernel daemon thread kswapd so that the access bits are scanned by FREM before they are cleared by the kernel.
4. To ensure the efficiency of the search and insertion operations, the double linked list and red-black tree data structures are used to store the information of the touched set and the candidate ranges.

We shall point out that the design principles of the CT subsystem are generic and do not depend on any specific architecture or system. It is feasible to port the above implementation to other systems like FreeBSD [38], OpenSolaris [23], etc.

5 Design and Implementation of Fast Restarting

The FR subsystem consists of two components: the restarter and the parallel loader. The restarter is invoked first to restore the geometry of the address space of the failed application and retrieve the touched set. It, then, is overlaid by the application thread which resumes the application execution immediately. In the meantime, the parallel loader thread is spawn to retrieve the remaining checkpoint data. As the design of the restarter is relatively simple and straightforward, in this section, we mainly focus on the design of the parallel loader. The parallel loader addresses the third challenge listed in Section 3, that is, how to effectively and correctly load the partial image.

Fig. 4 presents the main tasks performed by the parallel loader. With only the touched set pages loaded by the restarter, the address space contains a number of separated *memory holes*. FREM records and organizes them into *a hole list*. Fig. 4a illustrates the iterative hole populating mechanism to fill in memory holes from the checkpoint image, and Fig. 4b presents the detailed steps of remote page fault handling used in FREM.

Directly handling a remote page fault in the same thread context where it is triggered can substantially increase synchronization cost, as well as complicate the design. Hence, FREM adopts *a quasi on-demand remote page fault handling* approach. When a remote page fault occurs, the application thread is suspended and the remote page fault request is delegated to the loader thread. The loader always serves any outstanding remote page fault request before

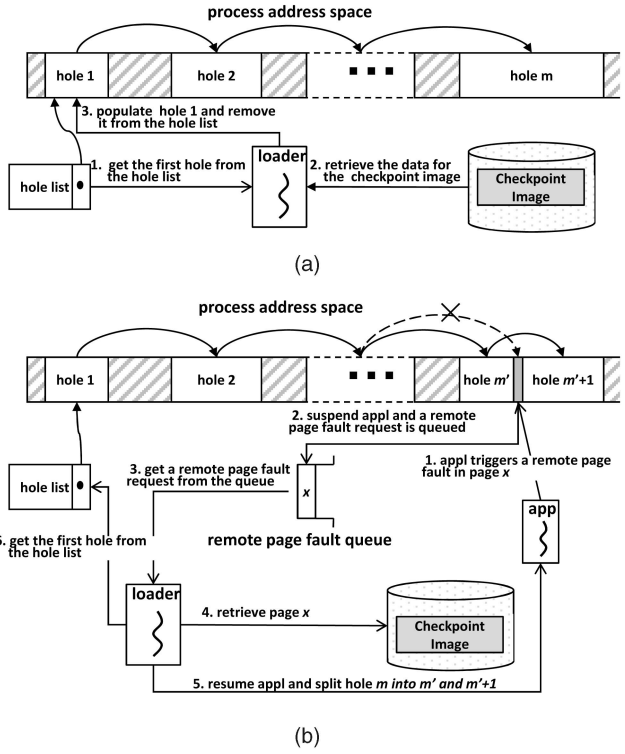


Fig. 4. The parallel loader. It includes (a) an iterative hole populating mechanism to fill in the process address space and (b) a remote page fault handling mechanism to deal with remote page faults during fast recovery.

populating memory holes from the hole list. The detailed steps of remote page fault handling are as follows:

- *Step 1.* The application execution triggers a remote page fault in the page x, which is in the memory hole m.
- *Step 2.* The application thread is suspended and a request for page x is inserted to a remote page fault queue maintained by FREM.
- *Step 3.* Before loading a memory hole, the loader first checks whether there are any outstanding remote page fault requests in the queue.
- *Step 4.* In case that the queue is not empty, the loader retrieves and loads page x from the checkpoint image into the address space.
- *Step 5.* The loader resumes the suspended application thread and updates the hole list. As an example, the hole m of page x will be split into two smaller holes m' and $m' + 1$.
- *Step 6.* Once the remote page fault queue is empty, the loader continues to populate memory holes iteratively as presented in Fig. 4a.

The loader needs to guarantee both the correctness and the efficiency of data loading. In the following two sections, we present how FREM achieves these goals.

5.1 Correct Data Accessing

During the overlapping time, the execution of the loader thread and the application thread share the same address space. They need to be synchronized properly; otherwise race condition occurs and consequently leads to incorrect data accesses.

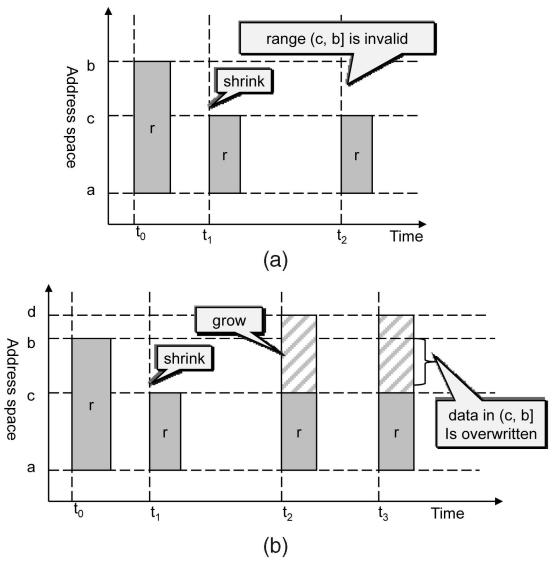


Fig. 5. Erroneous data access. (a) invalid access, and (b) data overwrite.

Dynamic memory management of the application thread may cause two types of data access errors: *invalid access* and *data overwrite* (see Fig. 5). Fig. 5a illustrates the case of invalid access. Suppose at time t_0, the memory region $r(=[a, b])$ is a hole and will be populated by the loader thread sometime later. At time t_1 before r is filled, the application thread executes a deallocation operation and shrinks r to [a, c]. Hence, at time t_2 when the loader tries to populate r with the data from the checkpoint image, segment fault occurs since (c, b] is *no longer within the valid address range.*

Fig. 5b illustrates the case of data overwrite, which is more subtle. At time t_1, the application thread executes a deallocation operation and shrinks the hole r from [a, b] to [a, c] and then at time t_2, the application reallocates the memory region [c, d], which will be gradually filled with user data written by the application threads. The loader, however, does not have the knowledge of this change. Hence, at time t_3 when the loader tries to populate region [a, b] according to the hole list, the useful data in [c, b] is already overwritten.

To avoid the above data access errors, we need to synchronize the application thread with the loader thread in such a way that the hole list can be updated timely to reflect the latest address space changes made by the application thread. Similar to the approach described in Section 4.1.2, FREM monitors every memory deallocation operation during the overlapping time. When the application thread shrinks a memory region, FREM intercepts the Linux *do_munmap* call and updates the hole list. It first identifies the intersections of the memory region to be freed and the hole list, and then excludes these intersections from the hole list. By doing so, FREM ensures that the loader always works on the updated geometry of the memory holes, thereby avoiding the above data access errors.

5.2 Efficient Image Loading

As indicated in Fig. 4, the loader is responsible for handling remote page faults and populating memory holes. These tasks may involve both fine-grained I/O (e.g., reading one page at a time) and bulky I/O operations (e.g., reading tens of thousands of remote pages at a time). The mix of these I/O operations can cause two efficiency issues:

1. *Application freeze*. Application freeze refers to the phenomenon when the application thread is stalled for a long period of time due to a remote page fault. When a remote page fault request is sent to the page fault queue, the loader maybe in the middle of populating a large hole requiring bulky I/O operations. Thus, the remote page fault request must wait for the loader to complete filling this hole, and the application has to wait, which is undesirable especially for interactive applications.
2. *Fragmental loading*. Fragmental loading is caused by excessive fine-grained remote I/O operations. There are two sources for fragmental loading. First, each remote page fault incurs a single-page size remote I/O operation. Second, as shown in Fig. 4b, serving a remote page fault page may split an existing hole into two smaller ones. Eventually, a large amount of small holes maybe created, leading to fragmental I/O operations. Fragmental loading can decrease the overall I/O throughput. Another side effect of fragmental loading is that as the number of holes is large enough, the search and update operations required to maintain the hole list also become slow.

To mitigate application freeze and fragmental loading, the parallel loader enforces an appropriate granularity for data loading by using a threshold called *CHUNK_SIZE*. Upon populating a hole or serving a remote page fault, the loader attempts to read in CHUNK_SIZE pages. In case of hole-populating, either the entire hole is loaded if its size is less than CHUNK_SIZE or the first CHUNK_SIZE pages in the hole are loaded. The handling of remote page faults is similar, except that the loader attempts to read in CHUNK_SIZE pages centering at the fault address. This is done to exploit spatial locality of an application, with the goal to minimize the number of remote page faults.

Algorithm 1 presents the pseudocode of the parallel loader. In terms of implementation, *a red-black tree* is built to construct the hole list, for the purpose of expediting the search and update operations. To guard the hole list against concurrent access, a mutex synchronization primitive is used. To ensure nonintrusiveness, the remote fault handling logic is transparently hooked to the kernel via the *no_fault callback* interface and the interception of the *do_munmap* call is done through the *jprobe* instrument interface. Note that these do not need any change in the kernel code.

Algorithm 1. FREM Image Loading

```
1: main procedure of the loader thread {
2:   while hole list not empty do
3:     while remote page fault queue is not empty do
         dequeue a remote page fault request and service it
4:         by retrieving CHUNK_SIZE pages around the
           faulted address
5:     wake up the app thread blocked on the request
6:     split the hole list
7:   end while
8:   retrieve the first CHUNK_SIZE pages in hole list
9:   update the hole list
```

TABLE 1
Application Cases from SPEC CPU2006

Case	Name	Workload	Description
1	bzip2	input.source	compression tool
2	Cactus	ref	physics/general relativity
3	dealII	default	finite Element Analysis
4	Games	cyclone	quantum chemistry
5	Gcc	expr2	c compiler
6	Mcf	ref	combination optimization
7	Perl	checkspam	perl programming language
8	Soplex	ref	linear programming

```
10:   end while
11: }
12: upon a page fault {
13: if the fault address falls into the hole list then
14:   insert the fault address to the remote fault queue
15:     suspend current thread
16: else
17:   turn to the default page fault handling
18: }
19: upon each memory deallocation {
20:   search out the intersections between the hole list and
        the memory region to be freed
21:   if the intersections exist, update the hole list by
        excluding those intersections
22: }
```

5.3 FR Implementation in Linux

The FR subsystem is implemented as a Linux kernel module using similar techniques mentioned in Section 4.3. In addition, the customized fault handler is injected via the no_fault interface provided by Linux kernel. Again, the design of FR subsystem is generic and the implementation can be ported to other systems as long as their kernel sources are available.

6 Experiments

We have implemented FREM in Linux 2.6.23 kernel currently, built upon the BLCR tool [10]. In the experiments, we compare *FREM-enhanced BLCR* as against *the native BLCR*. To make the comparison general to other checkpoint tools like EPCKPT [7] and TICK [32], we disable the BLCR-specific optimizations. We set the upper bound of the tracking window to 120 seconds and the CHUNK_SIZE to 100 pages.

Our testbed consists of a PC machine equipped with a Intel Core2Duo 2.4 GHz processor and 2 GB RAM, an NFS file server to provide remote storage, and an interconnected network to connect the PC machine to the NFS server. To evaluate FREM under different networking environments, two network configurations are tested: 1) **FAST**, which denotes a local area network (LAN) with the data transfer rate ranging from 8.5 to 12.0 MB/s and 2) **SLOW**, which represents a wide area network (WAN) with the data transfer rate ranging from 1.2 to 3.5 MB/s.

The benchmark suite SPEC CPU2006 is tested in our experiments [31]. The suite contains a variety of programs from the scientific and engineering domains. As FREM targets applications with large memory demands, we choose the applications whose memory footprints are greater than 150 MB. Totally, we have tested 27 application cases.

TABLE 2
Measurement of Checkpoint and Touched Set

Appl	CKP Size (MB)	CKP Overhead (s)		Tracking Window (s)		Touched Set (MB & Percentage)	
		FAST	SLOW	FAST	SLOW	FAST	SLOW
1	850.0	147.4	244.4	54.0	196.1	332.9 (39.41%)	348.1 (40.95%)
2	998.6	172.9	442.8	62.0	120.0	411.0 (41.18%)	411.0 (41.18%)
3	239.2	48.7	114.5	18.1	91.6	90.0 (37.66%)	97.3 (40.66%)
4	629.3	102.3	295.8	120.0	120.0	1.6 (0.25%)	2.8 (0.44%)
5	629.5	90.1	247.8	26.0	120.0	80.0 (12.66%)	354.1 (56.25%)
6	838.4	134.6	387.7	12.0	15.0	718.0 (85.64%)	822.0 (98.04%)
7	158.0	41.9	69.6	13.2	45.3	31.0 (19.62%)	75.6 (47.84%)
8	489.6	51.4	188.0	31.2	120.0	188.9 (38.58%)	192.4 (39.29%)

The numbers in the tracking window columns are the average values. The parenthesized numbers in the table are the ratios of the touched sets to the checkpoint image sizes (in percentage).

Based on the size ratio of the touched set to the checkpoint image measured in the FAST network, we categorize all the 27 applications into three groups: 1) *small group*, containing 10 test cases whose ratios are less than 33 percent; 2) *medium group* containing 14 test cases whose ratios are between 33.3 and 66.7 percent; and 3) *large group* containing four test cases whose ratios are greater than 66.7 percent. We randomly select eight applications from these groups (see Table 1) and present their experimental results in the following sections. Among them, the applications 4, 5, and 7 are from the *small* group, the applications 1-3 and 8 are from the *medium* group, and the application 6 is from the *large* group.

6.1 Restart Latency Improvement

In this set of experiments, we compare application restart latencies by using FREM-enhanced BLCR as against the native BLCR.

6.1.1 Measurement of Checkpoint and Touched Set

Table 2 lists our measured data, including application checkpoint image size, checkpointing overhead, tracking window, and the touched set. For the touched set, we not only list its size in MB, but also present its ratio to the checkpoint image. The results from both FAST and SLOW network configurations are presented here. In the experiment, for the applications with stable memory requirements, we randomly trigger a checkpointing during their lifetimes; for the applications whose memory requirements vary during their lifetimes, we randomly trigger a checkpointing during the period when their memory footprint size is moderate.

By comparing the checkpoint size and the touched set size, we can observe that *generally only a small portion of the*

TABLE 3
Restart Latencies by Using FREM and BLCR

Appl	FAST		SLOW	
	BLCR (s)	FREM(s)	BLCR(s)	FREM(s)
1	87.5	37.3	721.3	287.4
2	94.0	55.6	433.0	229.0
3	23.0	8.6	215.3	84.5
4	76.6	0.4	377.0	1.8
5	59.6	8.1	598.4	320.0
6	88.3	77.9	642.3	626.7
7	33.9	8.8	166.9	90.5
8	48.3	20.5	356.4	198.6

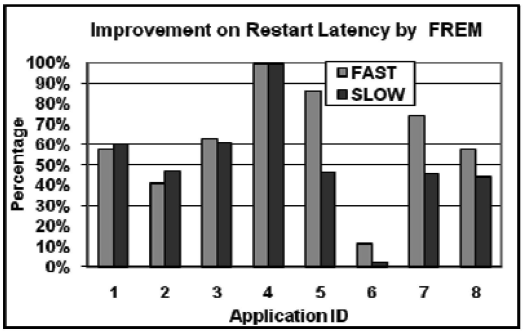


Fig. 6. Relative improvement on restart latency by FREM.

checkpoint image is immediately needed for application restart. These also justify the fundamental assumption made by FREM. In the FAST network configuration, among eight test cases, seven applications yield the touched sets less than half of their checkpoint footprints, with the ratio ranging from 0.25 to 41.18 percent and with the average at 34.35 percent. The reason for the small touched sets is due to good data locality and/or dynamic memory usage. The only exception is the application #6 showing extremely poor locality, for which the touched set is 85.64 percent of its checkpoint image. With respect to the SLOW network configuration, except for the application #6, the touched sets generally oscillate between 0.44 and 56.25 percent of their checkpoint images, with the average at 45.58 percent. Due to the slow data transfer rate in the SLOW configuration, for the application #6, the predefined tracking window size of 120.0 seconds is occasionally used.

As shown in Table 2, generally the larger a checkpoint image, the longer its checkpointing overhead. There is one exception in the SLOW configuration where the application #1 has an unexpectedly shorter checkpoint overhead while its checkpoint size is relatively large. We attribute this to the volatility and instability of the WAN networking connection.

6.1.2 Measurement of Restart Latency

Table 3 lists the raw restart latencies by using FREM and BLCR under both the FAST and SLOW configurations. Fig. 6 illustrates relative improvements achieved by FREM over BLCR. It is clear that the performance achieved by FREM is very promising. For all the cases, FREM outperforms BLCR with the average raw reduction of 36.75 seconds (61.26 percent) under the FAST configuration and 209.00 seconds (50.82 percent) under the SLOW configuration. The improvement on restart latency mainly stems from data locality and existence of useless "dead data" in most applications.

Further, we have two interesting observations. First, network performance has a substantial impact on FREM's performance. In general, under the FAST environment, FREM achieves more relative improvement as compared to the SLOW environment. In a low-performance networked environment like SLOW, a longer tracking window is needed, thereby resulting in a larger touched set and a lower relative improvement on restart latency. We shall also point out that the raw gain on restart latency, however, grows when data transfer takes longer time. In other words, FREM is able to provide different improvements under different network configurations: in a fast networking environment, it can effectively improve relative gain on restart latency, whereas in a slow networking environment it can substantially reduce the raw restart latency.

Second, checkpoint image transmission time is positively correlated with, but not strictly proportional to, its size. By examining the latency reduction ratio presented in Fig. 6 and the proportion of the checkpoint data immediate needed for restart listed in Table 2 (i.e., the ratio between the touched set and the checkpoint image), we find that the former is always modestly smaller than the latter. While the tracking window calculated in (1) provides us a good estimation, this observation indicates that our estimation is relatively conservative and there is a room to further improve it.

6.2 FREM Overhead

The use of FREM introduces two types of overhead: the postcheckpoint tracking overhead O_{CT} and the fast restart overhead O_{FR}. The former is introduced after each checkpoint, and the latter is only triggered upon a failure occurrence.

6.2.1 Postcheckpoint Tracking Overhead

The postcheckpoint tracking overhead can be broken down into three parts: 1) the page table scanning time to go through the entire process address space at the end of the tracking window (denoted as *PTE Scan*), 2) the monitoring cost to identify the touched set pages (denoted as *Touched Set Monitoring*), and 3) the I/O time to store the touched set descriptor (denoted as *Touched Set I/O*). Moreover, the touched set monitoring cost is influenced by two types of operations: the *search* operations to calculate the intersection of the candidate ranges and the memory region to be freed, and the *update* operations (e.g., scans and insertions) to insert the touched set pages into the touched set descriptor tree (See Section 4.1.2).

Tables 4 and 5 present postcheckpoint tracking overheads, which are divided into different components, collected in both FAST and SLOW environments.

Generally speaking, the overall tracking overhead is trivial, ranging from 32.32 to 411.56 milliseconds. They are typically several orders of magnitude less than the gains achieved by FREM on restart latency (Table 3). The PTE scan cost dominates in the overall overhead. This is due to

TABLE 4
Postcheckpoint Tracking Overhead in the FAST Configuration

Appl	Touched Set Monitoring			PTE Scan (ms)	Touched Set I/O (ms)	Total (ms)
	No. of Search	No. of Update	Cost (ms)			
1	6	4	2.03	44.25	0.06	46.34
2	78	3	8.82	41.01	0.12	49.94
3	34	8	4.97	29.73	0.09	34.79
4	0	0	0.00	53.95	0.13	54.08
5	83493	6	246.91	34.41	0.14	281.46
6	0	0	0.00	57.23	0.05	57.28
7	0	0	0.00	37.33	0.11	37.44
8	0	0	0.00	47.83	1.89	49.72

It includes three parts: 1) PTE Scan to go through the entire process address space at the end of the tracking window, 2) Touched Set Monitoring to identify the touched set pages, and 3) Touched Set I/O to store the touched set descriptor. The last column lists the aggregated cost.

the large memory demands of the applications and the costs of capturing dynamic pages. The touched set I/O cost is normally less than 10.96 milliseconds. This low cost is achieved due to the use of red-black trees in FREM.

We also note an exception with the application #5. For this application, high numbers of touched set monitoring operations (i.e., 83,499 and 123,943 in the FAST and SLOW settings, respectively) are observed. As a result, its tracking overhead is larger than other cases.

With respect to the overhead of TLB invalidation, a benchmark is conducted to show that a single TLB miss leads to 17.1 nanoseconds. So with 256 TLB entries in the processor, the upper bound of the TLB invalidation per checkpoint is about 4.4 microseconds. It is negligible compared to the performance gain at the macrosystem level. This is because such TLB invalidations don't entail cache miss or context switch and are handled by hardware efficiently.

6.2.2 Fast Restarting Overhead

When using FREM, the restart of the process is overlapped with the image retrieval until the remaining image is delivered to the destination machine. This overlapping inevitably incurs some overhead to the program execution due to resource contention. From the perspective of process execution, there are three sources of restarting overhead: 1) the cost to retrieve remote page faults (denoted as *Page Fault Retrieval*), 2) the cost to maintain the hole list (denoted as *Hole List Monitoring*), and 3) the concurrency overhead (e.g., memory contention and context switch) introduced by the simultaneous execution of the parallel loader thread and the applications thread (denoted as *Concurrency Cost*). The page fault retrieval cost is dependent on the number of remote page faults, whereas the hole list monitoring cost is influenced by the numbers of *search* and *update* operations to maintain the hole list.

Tables 6 and 7 list fast restarting overheads, which are divided into different components, collected in both FAST and SLOW environments. The overall overhead ranges from 0.50 to 16.35 seconds in the FAST environment and from 1.61 to 74.99 seconds in the SLOW environment. This may seem daunting at the first glance. However, we shall point out that restart overhead is only triggered upon a failure occurrence. By comparing with the data listed in

TABLE 5
Postcheckpoint Tracking Overhead in the SLOW Configuration

Appl	Touched Set Monitoring			PTE Scan (ms)	Touched Set I/O (ms)	Total (ms)
	No. of Search	No. of Update	Cost (ms)			
1	7	4	0.52	101.53	0.32	102.38
2	213	3	5.56	132.83	0.65	139.03
3	105	12	6.15	25.62	0.55	32.32
4	0	0	0.00	38.74	0.71	39.45
5	123934	9	307.27	103.04	1.25	411.56
6	0	0	0.00	60.19	0.29	60.48
7	0	0	0.00	43.17	0.69	43.86
8	0	0	0.00	42.36	10.96	53.32

TABLE 6
Fast Restarting Overhead in the FAST Environment

Appl	Page Fault Retrieval		Hole List Monitoring			Concurrency Cost (s)	Total (s)
	No. of Faults	Cost (ms)	No. of Search	No. of Update	Cost (ms)		
1	4	45.09	4	0	0.11	6.42	6.47
2	32	873.43	42	0	0.11	4.72	5.59
3	7	40.65	4	2	0.38	4.12	4.16
4	19	318.81	0	0	0.00	16.03	16.35
5	8	118.47	8	6	0.42	11.46	11.58
6	2	10.55	0	0	0.00	1.27	1.28
7	43	400.73	0	0	0.00	0.10	0.50
8	747	9638.30	0	0	0.00	2.49	12.12

It includes three parts: 1) Page Fault Retrieval to obtain the missing pages from the remote storage, 2) Hole List Monitoring to update the hole list when the address space shrinks, and 3) Concurrency cost introduced by the simultaneous execution of multiple threads. The last column gives the aggregated cost.

TABLE 7
Fast Restarting Overhead in the SLOW Environment

TABLE 8
Normalized Restart Overhead (Ratio to the Base Case) with Different Reload Chunk Sizes in FAST Network

CHUNK_SIZE (page number)	Case 1	Case 2	Case 5
10	1.3	0.8	0.2
20	1.2	0.7	0.4
50	1.1	0.6	0.7
100	1.0	1.0	1.0
200	0.8	2.3	1.6
500	1.0	4.7	2.6
1000	1.2	10.6	3.8
2000	1.2	22.7	4.9

TABLE 9
Adaptive Window versus Conservative Window in FAST Network

Appl	Window Size Reduction(%)	Restart Latency Reduction(%)	Execution time reduction (s)
1	28.4%	26.7%	9.1
2	39.7%	16.4%	8.5
3	43.0%	22.4%	2.3
4	0.0%	0.0%	0.0
5	37.8%	9.5%	0.9
6	69.2%	15.2%	11.5
7	15.1%	8.7%	0.9
8	36.5%	1.5%	0.9

Table 3, we can see that this overhead is easily offset by the improvement on restart latency.

Concurrency cost is generally the most significant contributor in restarting overhead, which ranges from 0.10 to 16.03 seconds in the FAST environment and from 0.33 to 31.57 seconds in the SLOW environment. It is mainly determined by the amount of remaining image in the remote storage.

The data also indicate that page fault retrieval cost is nontrivial, varying from 10.55 to 9,638.30 milliseconds in the FAST environment and from 1,023.71 to 57,506.46 milliseconds in the SLOW environment. This cost is directly influenced by the number of remote page faults, which is usually less than 60 for most of the test cases. The exception is for the application #8, where an exceptionally large number of remote page faults occurs and thus introduces a high overhead to handle these faults. By analyzing the application, we find out that this is caused by a specific stride data pattern of the application. The pattern disperses the remote page fault addresses in different parts of the hole list, thereby making the prefetching strategy ineffective as currently we prefetch the pages within the same hole node at a time.

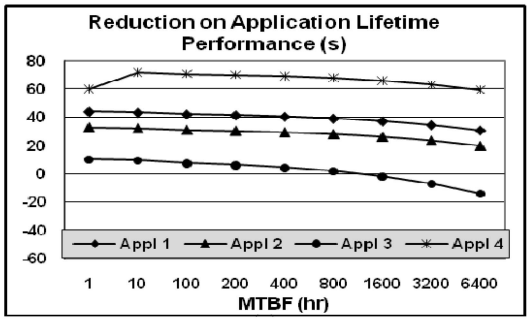
The hole list monitoring cost is relatively small, up to 4.26 milliseconds. This indicates that the hold list maintenance algorithm presented in Section 5.2 not only guarantees the correctness of the restart protocol, but also achieves good performance by excluding unused data (i.e., the FR subsystem will not retrieve the checkpoint data whose memory space is freed by the application during the overlapping time). This data exclusion could be substantial when the unused data are large and the network is slow.

To investigate the impact of the reloading chunk size, we vary the parameter CHUNK_SIZE from 10 to 2,000 pages and measure the restarting overhead of three representative applications: test case 1, 2, and 5 (see Table 8). Test case 1 represents the applications whose touched sets increase rapidly with time; test case 2 represents the applications whose touched sets are relative stable; and test case 3 represents the applications whose touched sets are highly dynamic due to memory allocation/deallocation. The normalized restart overheads (ratio to the base case with $\mathrm{CHUNK\_SIZE} = 100$) are presented in Table 8. For test case 1, the restart overhead goes down when CHUNK_SIZE increases from 10 to 200. This is because a small reload chunk size cannot satisfy the aggressive data demand from the application, thereby leading to large remote page fault overhead. When CHUNK_SIZE increases beyond 200, the overhead grows. A large reloading chunk size causes a long freeze of the application since it must wait for the completion of each chunk loading. For test case 2, the overhead monotonically increases with CHUNK_SIZE. As test case 2 has a stable and constant working set, FREM can accurately capture the touched set. Consequently, remote page faults rarely occur for these applications. For these applications, application freeze time caused by chunk reloading becomes the dominant overhead factor. For test case 3, a similar phenomenon is observed as test case 2. Furthermore, by using large CHUNK_SIZE, FREM may end up with loading more unused data, thereby entailing unnecessary overhead. Based on this experiment, we believe a modest CHUNK_SIZE like the base case is a balanced choice which can mitigate both remote page cost and application freeze time.

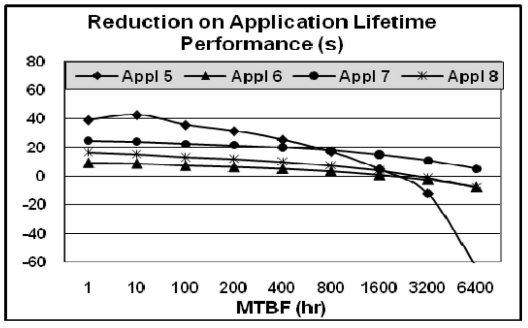
To illustrate the improvement brought by adaptive window estimation, we compare it with the conservative estimation presented in our previous work [15]. In Table 9, we list window size reduction, restart latency reduction, and the reduction of application execution time after restart. As we can see, the adaptive method can significantly reduce the window size by 33.7 percent and the restart latency by 12.5 percent on average. A smaller window size means less risk of FREM failure. The adaptive method dynamically captures the trend of application data accesses, thereby resulting in more accurate touched set and lower restart latency. With respect to application execution time after restart, the adaptive method still outperforms the conservative method, but with a less significant margin. Because the adaptive method produces smaller touched sets, the overlap time to reload the remaining image becomes longer. Therefore, concurrent execution overhead becomes larger, which offsets the gain brought by reducing restart latency.

6.3 Application Lifetime Performance Analysis

The results shown so far indicate that FREM can significantly reduce restart latency, but also introduces some runtime overheads. Given that checkpoint frequency is usually greater than that of recovery, a key question is *whether FREM is capable of producing positive performance gain in the long run*. To answer this question, in this section, we

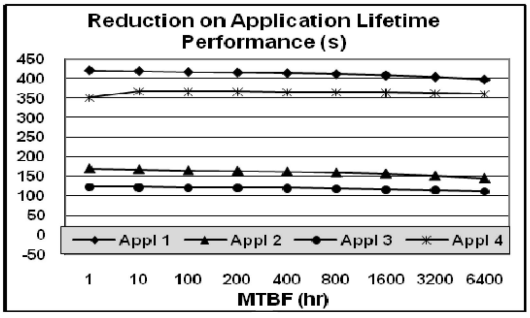


(a)

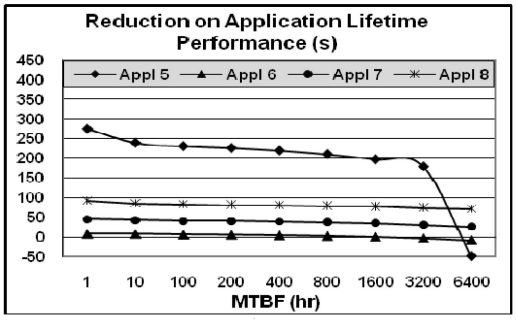


(b)

Fig. 7. Improvement of application lifetime performance in the FAST environment.



(a)



(b)

Fig. 8. Improvement of application lifetime performance in the SLOW environment.

conduct statistical studies to examine application lifetime performance. Here, "lifetime" means that we statistically evaluate application performance during each failure cycle, i.e., between two restarts.

In the experiments, we simulate Poisson failure arrivals with the MTBF varying from 1 to 6,400 hours. This range covers the realistic MTBF values that we have observed in the failure traces collected from production systems [14], [16], [33]. The checkpoint interval is set according to Young's first order approximation [42].

Two evaluation metrics are used to measure the overall performance of FREM: 1) E_{gain}, the expected *restart improvement* achieved by FREM between two restarts and 2) $E_{overhead}$, the expected runtime overhead introduced by FREM between two restarts. They are calculated as follows:

$$\begin{aligned} E_{gain} &= Restart\_Latency\_Gain \times (1-f), \\ E_{overhead} &= O_{CT} \times N_{ckp} + O_{FR} \times (1-f). \end{aligned} \tag{3}$$

Here, f is the failure probability during the tracking window and N_{ckp} is the number of checkpoints per failure cycle. Note that O_{CT} is the postcheckpoint tracking overhead and O_{FR} is the fast restart overhead. The improvement on application lifetime performance is the difference between E_{gain} and $E_{overhead}$.

Figs. 7 and 8 present the improvements on application lifetime performance achieved by FREM in the FAST and SLOW environments, respectively. Overall speaking, the results are impressive: the average improvement is 22.18 seconds in the FAST environment with the maximum gain of 71.5 seconds achieved by the application #4; the average improvement is 171.6 seconds in the SLOW environment with the maximum gain of 419.9 seconds achieved by the application #1. The larger improvement achieved in the SLOW environment is attributed to the larger reduction on restart latency.

By examining the curves plotted in the figures, we can see that the performance gain achieved by FREM is higher when the MTBF is smaller. When failure rate is high, the fast restart mechanism is invoked more frequently, thus leading to more reduction on restart latency. When the MTBF is extremely large, e.g., 6,400 hours, we notice that the performance gain achieved by FREM may become negative for some cases. The reason is that a larger MTBF leads to more checkpoints per failure cycle. As a result, the accumulated tracking cost could overshadow the gain on restart latency. This is especially true for the application #5, which has a significant tracking overhead due to a huge amount of memory deallocation calls (see Tables 4 and 5).

6.4 Discussion

In summary, the above experimental studies have provided the following key points regarding FREM:

1. For the 27 applications from the SPEC CPU2006 suite, FREM has demonstrated a pronounced improvement in terms of reducing restart latency over by 50 percent on average and improving application lifetime performance by up to 419.9 seconds.
2. Our prototype implementation has also demonstrated a successful example to exploit hardware parallelism at the system level for better application performance. This is becoming increasingly important with the prevailing use of multicore architectures.
3. In addition to the touched set, additional knowledge about process data patterns is also needed to further

improve the efficiency of FREM. This can help to reduce remote page faults.

4. For the applications exhibiting poor data locality, FREM provides less impressive benefit. There are two possible solutions to address the issue. One is to rely on users to provide some guidance on application locality for a better use of FREM. The other is to design an adaptive mechanism to dynamically (de)activate FREM at runtime based on online monitoring of application locality.

The current implementation of FREM has several limitations. First is related to optimization for applications with large amount of dynamic memory activities. Two out of 27 test cases in the SPEC CPU2006 suite (one of them is the application #5) have such characteristics. For these applications, the tracking overhead is quite high and may outweigh the performance gain achieved by FREM. Our analysis shows that exploiting an in-depth understanding of application access patterns can help to address the issue. For example, with regard to the application #5, despite its high volume of deallocation calls, only a few of them overlap with the candidate ranges, and consequently the actual necessary update operations are much less. Furthermore, these update operations are highly clustered together right after the checkpoint. Hence, we can optimize FREM by only tracking the beginning tens of deallocation calls to reduce the overhead without losing too many touched set pages. Our preliminary test shows that with this optimization, for these applications, FREM can reduce the tracking cost by a thousand times. How to automatically detect and exploit in-depth access patterns of applications in a systematic way remains as one of our future tasks.

Another potential drawback of FREM is that when the MTBF value gets very large, e.g., greater than 6,400 hours, the overall performance gain on application lifetime becomes marginal or even negative. In these circumstances, we suggest turning off FREM if application execution time is the primary concern. Nevertheless, empirical studies of various production systems have shown that in realistic environments systemwide MTBFs are generally far less than 6,400 hours [14], [19], [22], [33]. As the size and complexity of computer systems continue to grow, unexpected failures could frequently occur. Hence, effective failure recovery tools are indispensable and we believe that FREM can greatly mitigate failure impact on user applications by effectively reducing their restart cost.

7 Conclusions

In this paper, we have presented the design and implementation of FREM, a kernel-level fast restart mechanism to tackle the restart latency problem of general checkpoint/restart protocols in networked environments. It complements the state-of-the-art fault tolerance research by improving failure restart. Through user-transparent system support, FREM hides restart latency by parallelizing the application restart with checkpoint image retrieval. Our extensive experiments with the SPEC CPU2006 suite have indicated that FREM can greatly reduce application restart cost by over 50 percent on average, with the maximum raw reduction of 433.9 seconds. Our prototype implement is currently built upon the BLCR tool, under Linux 2.6.23 kernel. The principle of FREM should be easily ported to other operating systems or checkpoint/restart tools.

There are lots of rooms for improvement on FREM. We are currently investigating a systematic approach to extract in-depth data patterns of user applications, with the goal to further improve the efficiency of postcheckpoint tracking for FREM. Further, an adaptive triggering design is underway to make runtime decisions on whether and when to invoke FREM by considering system and application characteristics. We also plan to investigate a systematic way for the choice of system parameters used in FREM. Finally, our ultimate goal is to integrate FREM with prefailure fault tolerance tools including our own work [14], [16] as a compound solution for fault management.

Acknowledgments

Zhiling Lan is supported in part by the US National Science Foundation (NSF) grants CNS-0834514, CNS-0720549, and CCF-0702737. Some preliminary results of this work were presented in [15]. This work was performed while Yawei Li was a student at Illinois Institute of Technology.

References

[1] S. Arunagiri, J. Daly, P. Teller, S. Seelam, R. Oldfield, M. Varela, and R. Riesen, "Opportunistic Checkpoint Intervals to Improve System Performance," Technical Report UTEP-CS-08-24, 2008.

[2] M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment," *Proc. Summer USENIX Technical Conf.*, 1992.

[3] C. Clark, K. Fraser, H. Steven, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," *Proc. ACM/USENIX Symp. Networked Systems Design and Implementation*, 2005.

[4] J. Daly, "A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps," *Proc. Int'l Conf. Computational Science*, 2003.

[5] P. Dinda and D. O'Hallaron, "An Evaluation of Linear Models for Host Load Prediction," *Proc. IEEE Int'l Symp. High Performance Distributed Computing*, 1999.

[6] E. Elnozahy and J. Plank, "Checkpointing for Peta-Scale Systems: A Look Into the Future of Practical Rollback-Recovery," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 2, pp. 97-108, Apr.-June 2004.

[7] EPCKPT: A Checkpoint Utility for Linux Kernel, http://www.research.rutgers.edu/~edpin/epckpt, 2010.

[8] S. Feldman and C. Brown, "IGOR: A System for Program Debugging via Reversible Execution," *Proc. ACM SIGPLAN and SIGOPS Workshop Parallel and Distributed Debugging*, 1989.

[9] B. Gupta, Z. Liu, and Z. Liang, "On Designing Direct Dependency-Based Fast Recovery Algorithms for Distributed Systems," *ACM SIGOPS Operating Systems Rev.*, vol. 38, no. 1, pp. 58-73, 2004.

[10] P. Hargrove and J. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," *Proc. Scientific Discovery through Advanced Computing (SciDAC)*, 2006.

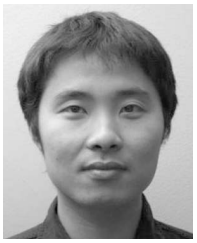
[11] J. Henning, "SPEC CPU2000 Memory Footprint," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 84-89, 2007.

[12] B. Jacob and T. Mudge, "Virtual Memory: Issues of Implementation," *Computer*, vol. 31, no. 6, pp. 33-43, June 1998.

[13] O. Laadan and J. Nieh, "Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems," *Proc. USENIX Ann. Technical Conf.*, 2007.

[14] Z. Lan and Y. Li, "Adaptive Fault Management of Parallel Applications for High Performance Computing," *IEEE Trans. Computers*, vol. 57, no. 12, pp. 1647-1660, Dec. 2008.

[15] Y. Li and Z. Lan, "A Fast Recovery Mechanism for Checkpointing in Networked Environments," *Proc. Int'l Conf. Dependable Systems and Networks (DSN)*, 2008.
[16] Y. Li, Z. Lan, P. Gujrati, and X. Sun, "Fault-Aware Runtime Strategies for High-Performance Computing," *IEEE Trans. Parallel and Distributed Systems*, vol. 20, no. 4, pp. 460-473, Apr. 2009.
[17] K. Li, J. Naughton, and J.S. Plank, "Low-Latency, Concurrent Checkpointing for Parallel Programs," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 8, pp. 874-879, Aug. 1994.
[18] Y. Ling, J. Mi, and X. Lin, "A Variational Calculus Approach to Optimal Checkpoint Placement," *IEEE Trans. Computers*, vol. 50, no. 7, pp. 699-708, July 2001.
[19] C. Lu, "Scalable Diskless Checkpointing for Large Parallel Systems," PhD thesis, Univ. of Illinois at Urbana-Champaign, 2005.
[20] D. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241-299, 2000.
[21] NCSA web site, http://teragrid.ncsa.uiuc.edu, 2009.
[22] A. Oliner, L. Rudolph, and R. Sahoo, "Cooperative Checkpointing: A Robust Approach to Large-Scale Systems Reliability," *Proc. Int'l Conf. Supercomputing*, 2006.
[23] OpenSolaris, http://hub.opensolaris.org, 2010.
[24] Oracle high availability document, http://www.oracle.com/technology/deploy/availability/htdocs/fs_on-demand_rollback.htm, 2010.
[25] D. Patterson et al., "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.
[26] J. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory Exclusion: Optimizing the Performance of Checkpointing Systems," *Software—Practice and Experience*, vol. 29, no. 2, pp. 125-142, 1999.
[27] J. Plank, K. Li, and M. Puening, "Diskless Checkpointing," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972-986, Oct. 1998.
[28] J. Plank and M.G. Thomason, "Processor Allocation and Checkpoint Interval Selection in Cluster Computing Systems," *J. Parallel and Distributed Computing*, vol. 61, no. 11, pp. 1570-1590, 2001.
[29] S. Rao, L. Alvisi, and H. Vin, "The Cost of Recovery in Message Logging Protocols," *IEEE Trans. Knowledge and Data Eng.*, vol. 12, no. 2, pp. 160-173, Mar./Apr. 2000.
[30] E. Roush and R. Campbell, "Fast Dynamic Process Migration," *Proc. Int'l Conf. Distributed Computing Systems*, 1996.
[31] SPEC CPU benchmark, http://www.spec.org/cpu2006/, 2006.
[32] J. Sancho, F. Petrini, G. Johnson, J. Fernández, and E. Frachtenberg, "On the Feasibility of Incremental Checkpointing for Scientific Computing," *Proc. Int'l Parallel and Distributed Processing Symp.*, 2004.
[33] B. Schroeder and G. Gibson, "A Large Scale Study of Failures in High-Performance-Computing Systems," *Proc. Int'l Symp. Dependable Systems and Networks*, 2006.
[34] J. Squyres and A. Lumsdaine, "A Component Architecture for LAM/MPI," *Proc. European PVM/MPI Users' Group Meeting*, 2003.
[35] L. Snyder and Z. Shen, "Managing Disruptions to Supply Chains," *The Bridge*, vol. 36, no. 4, pp. 39-45, 2006.
[36] T. Tannenbaum and M. Litzkow, "The Condor Distributed Processing System," *Dr. Dobb's J.*, vol. 227, pp. 40-48, 1995.
[37] A. Tanenbaum and A. Woodhull, *Operating Systems: Design and Implementation*, second ed., Prentice-Hall, 1997.
[38] The FreeBSD Project, http://www.freebsd.org, 2010.
[39] N. Vaidya, "Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme," *IEEE Trans. Computers*, vol. 46, no. 8, pp. 942-947, 1997.
[40] F. Wu, H. Xi, and C. Xu, "On the Design of a New Linux Readahead Framework," *ACM SIGOPS Operating Systems Rev.*, vol. 42, no.5, pp. 75-84, 2008.
[41] R. Wolski, "Dynamically Forecasting Network Performance Using the Network Weather Service," *J. Cluster Computing*, vol. 1, no.1, pp. 119-132, 1998.
[42] J. Young, "A First Order Approximation to the Optimal Checkpoint Interval," *Comm. ACM*, vol. 17, no. 9, pp. 530-531, 1974.
[43] G. Zheng, L. Shi, and L. Kale, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," *Proc. IEEE Cluster Computing*, 2004.



Yawei Li received the BS and MS degrees from the University of Electronic Science and Technology of China in 1999 and 2002, and the PhD degree in computer science from Illinois Institute of Technology in 2009. He specializes in parallel and distributed computing, in particular, adaptive fault management and checkpointing optimization. He is now with Google Inc. He is also a member of the IEEE.

Zhiling Lan received the BS degree in mathematics from Beijing Normal University, the MS degree in applied mathematics from Chinese Academy of Sciences, and the PhD degree in computer engineering from Northwestern University in 2002. She is currently an associate professor of computer science at Illinois Institute of Technology. Her main research interests include fault-tolerant computing, dynamic load balancing, and performance analysis and modeling. She is a member of the IEEE Computer Society.