A Fast Restart Mechanism for Checkpoint/Recovery Protocols in Networked Environments

Yawei Li and Zhiling Lan
Department of Computer Science
Illinois Institute of Technology, IL 60626, USA
{liyawei,lan}@iit.edu

Abstract

Checkpoint/recovery has been studied extensively, and various optimization techniques have been presented for its improvement. Regardless of the considerable research efforts, little work has been done on improving its restart latency. The time spent on retrieving and loading the checkpoint image during a recovery is non-trivial, especially in networked environments. With the ever-increasing application memory footprint and system failure rate, it is becoming more of an issue. In this paper, we present a Fast REstart Mechanism called FREM. It allows fast restart of a failed process without requiring the availability of the entire checkpoint image. By dynamically tracking the process data accesses after each checkpoint, FREM masks restart latency by overlapping the computation of the resumed process with the retrieval of its checkpoint image. We have implemented FREM with the BLCR checkpointing tool in Linux systems. Our experiments with the SPEC benchmarks indicate that it can effectively reduce restart latency by 61.96% on average in networked environments.

1. Introduction

Checkpoint/recovery (C/R) has been widely used for fault tolerance in networked computing environments, such as parallel and distributed systems [4, 9, 18]. It periodically stores a snapshot of the running program, including CPU registers, signals, file caches, and process address space, on stable storage and uses it to restart execution in case of failure. A networked system is generally composed of abundant resources, thereby making it possible to restart the crashed program on an alternative resource from the checkpoint image, rather than waiting for the repair of the failed resource. As a matter of fact, such a remote-restart mechanism is common in Grid computing [24] as well as in high performance computing [20].

Existing research on C/R has mainly focused on reducing checkpoint overhead, whereas little work has been done on reducing its restart latency. Here, *restart latency* refers to the time that elapses between the initiation of the checkpoint image retrieval and the restart of the failed process. In the current C/R practice, a restart requires the checkpoint image to be completely available on the destination machine before it can proceed. In networked environments where the checkpoint image is accessed via interconnected networks, restart latency can be substantial. This is especially problematic in the field of high performance computing where applications typically are memory demanding. Research has determined that the memory footprint is a major contributor to the checkpoint image size [7, 20]. Further, due to the ever-increasing system size and complexity [4], failures occur more frequently than before, thereby making restart latency a critical concern in networked environments.

The recovery problem has been previously studied in various fields including operating systems, databases, and internet services [1, 14, 15]. However, existing solutions are either specific to particular problem domains or hardly applicable to improve checkpoint based restart. As mentioned earlier, the research on C/R mainly focuses on runtime optimization of checkpointing, with little attention to process recovery. Therefore, reducing restart latency for general C/R protocols remains an open problem.

In this paper, we present FREM, a Fast REstart Mechanism, to enhance general C/R protocols by concentrating on reducing restart latency. The core idea of FREM is to enable quick restart on a partial checkpoint image by recording the process data accesses after each checkpoint. More specifically, at runtime, through a user-transparent system support, it tracks the memory access information of the process (denoted as *the touch set*) following each checkpoint within a specific time period (denoted as *the tracking window*). At recovery time, rather than retrieving the entire checkpoint image for restart, FREM only requires the touch set on the destination machine for quick restart. The remainder of the checkpoint image is then transferred after the process is restarted on the destination machine. By doing so, FREM intends to overlap application execution with the retrieval of the

checkpoint data which is not immediately needed, thereby reducing restart latency.

While the idea may be straightforward, the design and implementation of FREM is challenging. The key issues include how to accurately identify the touch set, how to appropriately set the tracking window, and how to effectively load the partial image on the destination machine. To address these challenges, in this paper we propose:

- **A post-checkpoint tracking method for capturing the touch set.** Hardware and software complexities in real systems introduce numerous complications to the identification of the touch set. The proposed method monitors the memory access pattern of the process during the tracking window by considering the underlying hardware and software features, and records the precise access pattern as the touch set along with the checkpoint image. More importantly, such support is provided through a user-transparent system implementation.
- **A heuristic method for estimating the tracking window.** The tracking window, which determines the size of the touch set, plays a crucial role in FREM. The ideal scenario is such that the execution time of the resumed process on the touch set exactly matches the retrieval time of the remaining checkpoint image. We present an upper bound heuristic to estimate the window size, which intends to make a balanced tradeoff between performance and design simplicity.
- **A revised page fault handling mechanism for partial image loading.** To restart the process with its partial address space available, the kernel page fault handler is modified to coordinate the regular kernel paging mechanism with the special page fault handling required by FREM.

We have implemented FREM with the BLCR [6] checkpointing tool in Linux systems. Our experiments with the SPEC CPU2006 benchmarks [21] show that the average improvement achieved by FREM is *61.96*% in terms of reducing restart latency. To the best our knowledge, we are among the first to exploit runtime data access information to achieve fast process restart in networked environments. FREM complements existing studies on checkpoint/restart by enhancing the recovery process. As an example, FREM can be integrated with MPICH-V [2] and LAM-MPI [22] to enhance fault management for high performance computing [8].

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces the main idea of FREM, followed by a description of the detailed methods in Section 4. Section 5 presents our experimental results. Finally, Section 6 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 non-volatile memory) in the Sprite system to store crucial process state needed for fast recovery [1]. In database systems, quickly resuming transaction processing is the focus. For example, the Oracle systems have used the "on-demand rollback" technique to allow new transactions to execute while the rollbacks are still being performed [14]. Recently, more attention has been paid to fast recovery for Internet services. A representative work is the ROC project from Berkeley and Stanford [15]. It focuses on providing a holistic solution for post-failure recovery of Internet services by using fine-grained system partitioning and recursive restart. Rao et al. have proposed a class of hybrid protocols to maintain the failure-free performance of sender-based protocols while approaching the performance of receiver-based protocols during recovery [19]. *FREM is fundamentally different from these works in that it emphasizes the reduction of restart latency for general C/R 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 [26]. To allow failures during checkpointing or recovery, Dali has proposed a higher order interval approximation model by extending Young's work [3]. Vaidya has developed an improved interval by differentiating checkpoint latency and overhead [25]. Plank and Thomason have investigated the optimal checkpoint interval for parallel applications [18]. Additionally, there are numerous papers on dynamic checkpoint scheduling, such as aperiodic checkpointing [10] and cooperative checkpointing [13]. The other major direction is to reduce checkpoint overhead, especially the disk I/O time. Latency hiding and memory exclusion are two key techniques [16]. The studies in this category include copy-on-write [9], diskless checkpointing [17], and incremental checkpointing [5, 20]. Despite these runtime optimizations, no dedicated attention is paid to reducing restart latency during recovery. *Complementing the above studies on checkpoint optimization, our proposed FREM emphasizes the*

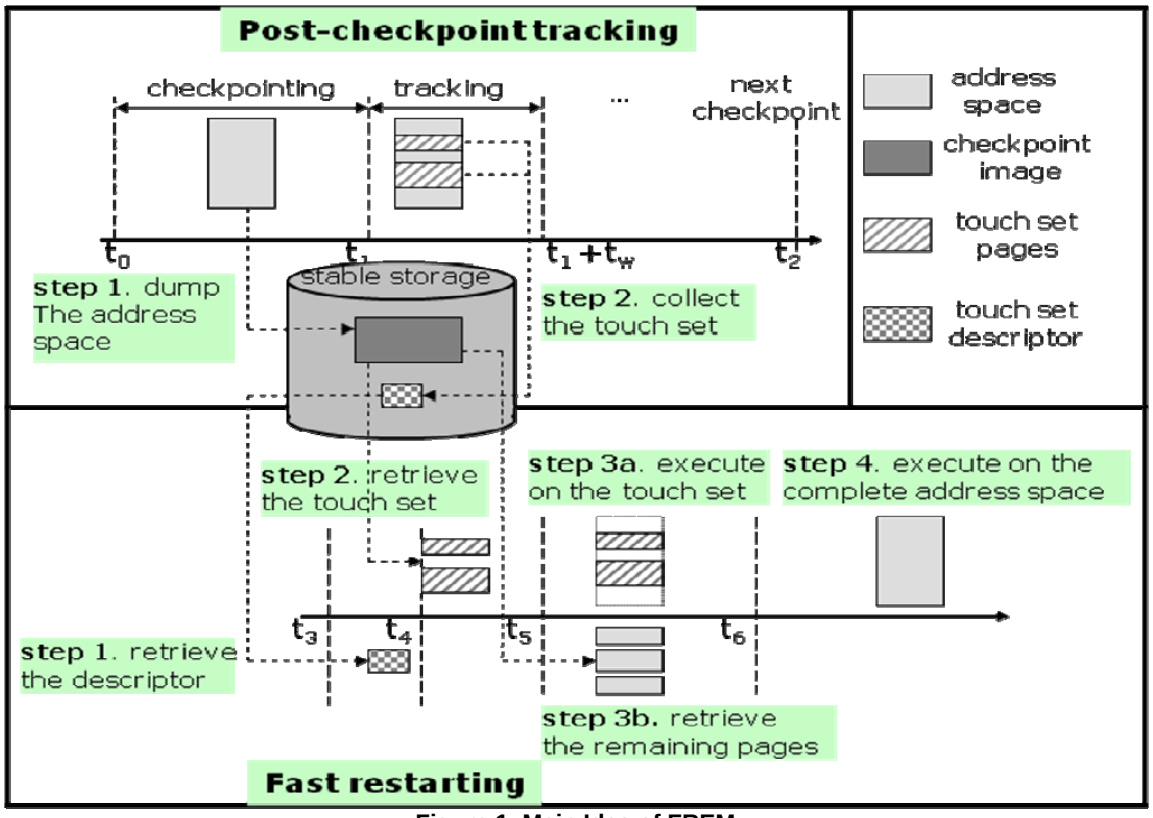

Figure 1. Main Idea of FREM

reduction of restart latency. Further, the principle of FREM is applicable to general C/R protocols.

There also exist several optimization techniques that utilize memory paging mechanisms to achieve fast process execution. For example, demand paging is a well-known technique, which allows a process to begin execution with part of its pages available in the physical memory [23]. *Unlike demand paging, FREM selectively restores the pages that will be immediately needed for fast recovery by tracking the pages used after the checkpoint.*

Similarly in the field of process migration, paging mechanisms are also incorporated to achieve fast process restart on the destination machine [11]. *While these migration methods focus on optimally transferring process state between the source and destination processes, FREM targets reducing restart latency and does not require a live copy of the process on the source machine.*

3. Main Idea

The main idea of FREM is illustrated in Figure 1. There are two phases in FREM: (1) the *post-checkpoint tracking* phase at runtime and (2) the *fast restarting* phase during recovery.

The post-checkpoint tracking phase is composed of two steps:

- At time t_0 the checkpointing tool is invoked to dump the process state onto stable storage, just as any regular checkpoint mechanism does.
- Upon completion of the checkpoint at time t_1, FREM starts to track the page-level memory accesses of the process between t_1 and (t_1+t_w) where t_w is *the tracking window size*. The memory access information, called "*the touch set*" in this paper, is formally defined as the intersection of the process address space saved in a checkpoint and its working set during the following tracking window. The goal of this step is to capture the touch set and store its information remotely on stable storage along with the regular checkpoint image at the end of the tracking window. FREM takes advantage of the paging mechanism supported by modern computer systems to monitor the page access: it first clears the access bit of each page table entry (PTE) at t_1, which will be set by the CPU when the corresponding page is accessed; at the end of the tracking window, FREM collects the pages touched by the process by scanning the status of the access bit of each PTE. The touch set information consists of a set of page address ranges accessed by the process during the

tracking window (denoted as *the touch set descriptor*).

During recovery, FREM takes four steps to achieve fast restart on the destination machine:

- At recovery time t_3, FREM retrieves the touch set descriptor.
- At time t_4, based on the descriptor, FREM retrieves the touch set as well as other necessary process state, such as register contents and process signals from the checkpoint image.
- Upon completion of retrieving the touch set at time t_5, the process is restarted on the touch set. Meanwhile, FREM forks another thread to simultaneously retrieve the remaining pages from the image file.
- At time t_6, when all the remaining pages are retrieved and loaded on the destination machine, the process continues running on the complete address space.

The rationale of FREM is that the touch set captures the precise data access of the process during process recovery. We exploit this feature to optimize the restart procedure by overlapping the computation with the communication and disk I/O as shown in Figure 1 (Step 3a -3b). The effectiveness of FREM requires that the process only access a relatively small portion of its address space within a given time window after a checkpoint. This assumption is justified by two facts in practice: (1) many applications demonstrate good temporal locality in data accesses, and (2) applications using dynamic memory allocation may have a large amount of unused or dead data in their checkpoint image files [16].

4. Methodology

In this section, we elaborate our research methods. They are developed to address the key challenges listed in Section 1, namely how to accurately identify the touch set, how to appropriately set the tracking window size and how to effectively load the partial image on the destination machine.

4.1. Identification of the Touch Set

Precisely identifying the touch set is crucial in the design of FREM. There are two types of possible errors: (1) *false positives* where pages not of interest are included in the touch set and (2) *false negatives* where pages of interest are missing from the touch set. These errors stem from the complicated features of hardware and software design, which include hardware bypassing, page swapping and dynamic memory management.

4.1.1. Hardware bypassing. Although the access bit of the PTE is often used to track page-level data accesses, not every single memory access updates the access bit in the PTE [27]. For example, a Translation Lookaside Buffer (TLB) hit can cause the memory access to bypass the PTEs. When a TLB hit occurs, the process directly reads the address translation information from the TLB, rather than going through the page table maintained by OS. Hardware bypassing can introduce false negatives in the identification of the touch set. In the architectures that support software-managed TLBs, FREM can directly look into the TLB to obtain the correct status of the access bit, thereby solving the issue.

But in the architectures without such support, such as our target platform x86, using a hardware-managed TLB, TLB peeking is forbidden. To address the issue, FREM must ensure the consistency between the TLB and the PTE entries. In our current design, 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 guarantees the first access of each page will cause a TLB miss and consequently set the access bit of the PTE. While such TLB invalidation introduces some overhead from the perspective of the micro-architecture, the overhead is generally several orders of magnitude smaller than the reduction of restart latency at the macro-system level. Clearing the access bits of the PTEs could disturb the kernel page replacement, but in Linux, page replacement is based on a separate PG_REFERENCE bit of each physical frame. So to guarantee the original replacement algorithm, FREM turns on the corresponding PG_REFERENCE bit when it clears an access bit which has been previously set.

In addition, a DMA (Direct Memory Access) operation also bypasses the CPU, thereby causing false negatives. We suggest instrumenting the corresponding device driver to set the access bits of PTEs whenever a DMA transfer is initiated. In the current design, we adopt a simple strategy which includes all the mapped DMA pages in the touch set. Typically the amount is much smaller than the entire process address space on the x86 platforms.

4.1.2. Page swapping. Page swapping, which clears the PTE access bits, may also cause false negatives in the identification of the touch set. Hence, FREM must track the access bits upon page swapping. To solve the problem, we instrument the Linux kernel swap thread **kswapd** to ensure that whenever a page swap occurs during the tracking window, the access bits of the PTEs are first scanned by FREM before they are cleared by the kernel.

4.1.3. Dynamic memory management. Dynamic memory allocation and deallocation operations change the process address space. Without a careful analysis, they may cause identification errors. As shown in Figure 2, we identify three types of pitfalls stemming from dynamic memory usage.

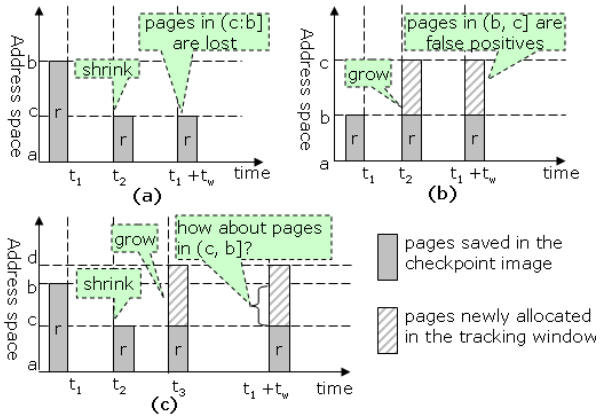

Figure 2. Pitfalls in the identification of the touch set caused by dynamic memory management

In Figure 2(a), at time t_1 the memory region r (the region of $[a,b]$) is saved on stable storage as the checkpoint image. At time t_2, a deallocation operation shrinks r to $[a,c]$ and releases all the pages in $(c,b]$. When FREM scans for the touch set at time $(t_1 + t_w)$ (the end of the tracking window), a false negative error may occur - the pages in $(c,b]$ accessed during time (t_1, t_2) are lost.

In Figure 2(b), 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 touch set; otherwise a false positive error is introduced. Recall that the touch 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 Figure 2(c), the memory region r is checkpointed at time t_1. At time t_2, a deallocation operation shrinks r to $[a,c]$. Then later 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 two-fold. 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 touch set; otherwise a false positive error is introduced.

The above analysis indicates that the touch set is always a subset of the checkpoint image, which monotonically decreases during the tracking window. Based on this key observation, we develop a simple yet effective algorithm to track the touch set: upon the completion of a checkpoint, the address information of the pages saved in the checkpoint image is stored by FREM (denoted as *the candidate pages*); whenever a memory deallocation takes place, FREM checks the intersection between the candidate pages and the pages to be released for the identification of the touch set; after that, FREM updates the candidate pages by excluding the intersection. The algorithm can eliminate the potential false positives and false negatives as illustrated in Figure 2. Figure 3 summarizes our algorithm to identify the touch set.

Dumping the checkpoint image {
  Step 1. Invoke BLCR to save the checkpoint image;
  Step 2. Record the pages saved in the checkpoint image as the candidate pages;
  Step 3. Initialize the touch set descriptor;
  Step 4. Invalidate the TLB entries if necessary;
}
Tracking the touch set {
  Upon each memory deallocation {
    Step 1. Check the intersection between the pages to be released and the candidate pages;
    Step 2. Identify the accessed pages (in the intersection) as part of the touch set;
    Step 3. Update the candidate pages by excluding the intersection;
  }
  Upon each page swap {
    Step 1. Check the intersection between the pages to be traversed and the candidate pages;
    Step 2. Identify the accessed pages (in the intersection) as part of the touch set;
  }
  Upon the completion of tracking {
    Step 1. Check the intersection between the current memory region and the candidate pages;
    Step 2. Identify the accessed pages (in the intersection) as part of the touch set;
    Step 3. Store the touch set descriptor, along with the checkpoint image;
  }
}

Figure 3. The touch set identification algorithm

In our implementation, to ensure the efficiency of the search and insertion operations, we use a double linked list and a red-black tree to store the touch set descriptor and the candidate pages respectively. In addition, to monitor memory space deallocation, the Linux kernel function **do_munmap** is instrumented.

The modified kernel only impacts the target process within the tracking window, thereby minimizing its disturbance to other processes in the system.

4.2. Estimation of Tracking Window

The tracking window size t_w is equally important in the FREM design and is dynamically determined at the beginning of the post-checkpoint tracking. How to set an optimal window size is challenging. A small window size reduces the time duration of overlapping computation and image retrieval during recovery and also incurs numerous remote page faults. On the other hand, a large window size leads to a large touch set, thereby increasing recovery latency and also increasing the risk of failures. An ideal t_w should yield a perfect touch set such that the resumed process first accesses a page not in the touch set just as the transmission of the remaining checkpoint image finishes, e.g., the time t_6 in Figure 1.

In our current design, a heuristic method is adopted to estimate the window size: once we know the checkpoint image size *W* at the completion of a checkpoint, we set t_w to the retrieval time of the checkpoint image (denoted as *retrieval(W)*). It can be simply calculated as the sum of the disk I/O and network transfer time:

$$retrieval(W) = W/_{disk\ BW} + W/_{network\ BW} + network\ latency$$

Here, the parameters like latency and bandwidth can be obtained according to the hardware specifications or through benchmark tools. Note that *retrieval(W)* is a conservative estimate and can be used as an upper bound. In practice, the actual image retrieval time should be less as the disk I/O may be overlapped with the network transfer. The rationale is to ensure the availability of the entire checkpoint image before the process completes its execution on the touch set. It is possible to set t_w to a smaller value. Currently we did not do this and leave it as future work.

Further, the tracking window size can be set to zero so as to disable FREM under two conditions:

- When *W* is smaller than a pre-defined threshold *TH*, t_w is set to zero. For the applications with small memory footprints, FREM is not used. The value of *TH* can be set by system administrators or users based on their tolerable restart latency.
- If *retrieval(W)* is larger than the checkpoint interval, we also set t_w to zero. Based on our experience, this violation is rare in practice since the retrieval time *retrieval(W)* is usually much less than the checkpoint interval.

4.3. Partial Image Loading

To enable the process restoration on the touch set, FREM coordinates its partial image loading with the Linux demand paging mechanism:

- Once the touch set is retrieved, FREM restores the structure of the process address space via the **memmap** function call, and then loads in the touch set. Afterward the process is restarted and another kernel thread is forked to simultaneously retrieve the remaining image to the destination machine.
- During the overlapped execution, FREM provides special page fault handling for the process by implementing the **no_page** callback function as a memory map driver. (1) If a page fault address belongs to the touch set (this is possible due to dynamic memory allocation), the default page fault handling is used. (2) If a page fault address falls out of the touch set, FREM first checks whether the requested data is already available in the local image file; if yes, the requested page will be loaded on-demand; otherwise a remote page fault occurs. A simple strategy is employed to deal with remote page faults in our current design, which stops the application until the requested page is retrieved. Due to the conservative estimation of the tracking window size, the probability of remote page faults is rare. As soon as the entire checkpoint image is available and loaded, FREM unhooks this driver from the memory management subsystem to restore its normal operations. A more sophisticated mechanism like on-demand remote data retrieval will be investigated in our future work.

5. Experiments

To evaluate FREM, we have implemented a prototype system with the BLCR checkpointing tool [6] in Linux 2.6.22 systems. Our testbed consists of two x86 machines, one used as the source machine and the other as the destination machine. Each machine is equipped with a 2.8GHz Pentium 4 processor, 512KB cache, 1 GB RAM and an 80GB 7200RPM Maxtor disk. Two network configurations at the National Center for Supercomputing Applications (NCSA) are tested: (1) **FAST,** which denotes a fast Myrinet2000 network deployed in the NCSA Mercury cluster and (2) **SLOW,** which represents a relatively slow Ethernet connection deployed between the Mercury and Tungsten clusters [12]. Table 1 lists the measured data retrieval parameters.

Table 1. Parameters of image retrieval cost

Network	Network latency	Network. bandwidth	Disk bandwidth
SLOW	200ms	7MB/s	7MB/s
FAST	70ms	32MB/s	7MB/s

The benchmark suite SPEC CPU2006 is tested in our experiments [21]. Since FREM targets the applications with large memory consumptions, we choose the applications whose memory footprints are greater than 150 MB. Among these applications, we randomly select twelve applications and present their results in the following.

5.1. Application Restart Latency

In this set of experiments, we compare application restart latencies by using the regular BLCR and FREM-enhanced BLCR. Table 2 lists our measured results, including application checkpoint image size, size of the touch sets, and restart latencies using BLCR and FREM. As we can see from the table, for most applications, the touch sets are substantially smaller than the checkpoint images. The use of FREM can significantly reduce application restart latency. The performance achieved by using FREM is very promising: the average reductions on restart latency are 72.43% and 61.96% in the **FAST** and **SLOW** networks, respectively.

Table 2. Restart latency (*RL*) by using BLCR and FREM with SPEC CPU2006 applications. The parenthesized numbers in the last two columns are relative improvements (in percentage) achieved by FREM.

Application (input set)	*W* (MB)	Touch set size (MB)		*RL* with BLCR (s)		*RL* with FREM (s)	
		FAST	SLOW	FAST	SLOW	FAST	SLOW
1: astar (1)	280	34	44	49.4	80.1	6.0 (87.88%)	12.6 (84.26%)
2: bzip2 (5)	847	45	152	161.3	254.1	8.6 (94.64%)	45.6 (82.04%)
3: bzip2 (6)	609	64	244	109.3	176.0	11.5 (89.49%)	70.5 (59.97%)
4: dealII	239	12	28	31.0	57.2	1.6 (94.80%)	6.8 (88.13%)
5: gamess (1)	629	5	9	112.0	180.9	0.8 (99.28%)	2.6 (98.57%)
6: gcc (4)	311	48	82	53.4	87.6	11.6 (78.20%)	23.0 (73.75%)
7: gcc (6)	771	216	211	136.7	221.2	38.2 (72.05%)	60.6 (72.62%)
8: lbm	409	402	402	73.6	118.5	72.3 (1.80%)	116.3 (1.80%)
9: mcf	839	394	827	151.0	242.8	70.9 (53.03%)	239.5 (1.37%)
10: perl (1)	171	31	50	24.7	43.5	4.4 (82.16%)	12.8 (70.66%)
11: soplex (2)	490	186	191	89.3	142.9	33.9 (62.05%)	55.7 (61.06%)
12: wrf	685	37	346	117.8	192.9	54.5 (53.78%)	97.9 (49.25%)

Figures 4 and 5 show, respectively, the raw improvement and the relative improvement on restart latency achieved by FREM over BLCR. As we can see from Figure 4, the reduction ranges from a couple of seconds to a couple of hundred seconds. The highest reduction is 152.6 seconds in the **FAST** network and 208.5 seconds in the **SLOW** network. According to Figure 5, except for applications 8 and 9, the relative improvement is more than 53.78% in the **SLOW** network and more than 49.25% in the **FAST** network. The trivial improvements on applications 8 and 9 are attributed to their low temporal data locality. For instance, for application 8, its touch set is 402 MB, which is very close to the checkpoint image of 409 MB; for application 9, the improvement achieved by FREM drops sharply when the network performance is changed from **FAST** to **SLOW**. This is also caused by the rapid growth of the touch set when the network performance is low. However, we shall point out that even in a slow network, the raw restart latency is still reduced by at least a couple of seconds.

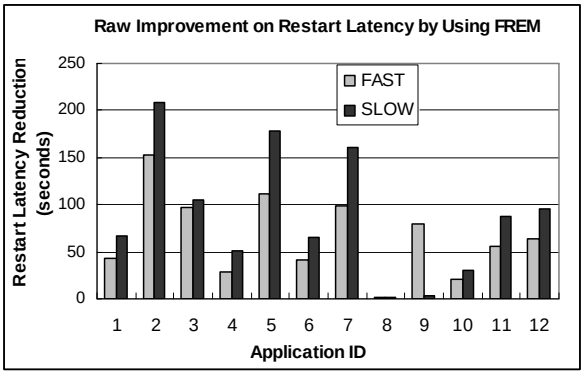

Figure 4. Raw improvement on restart latency by using FREM over BLCR

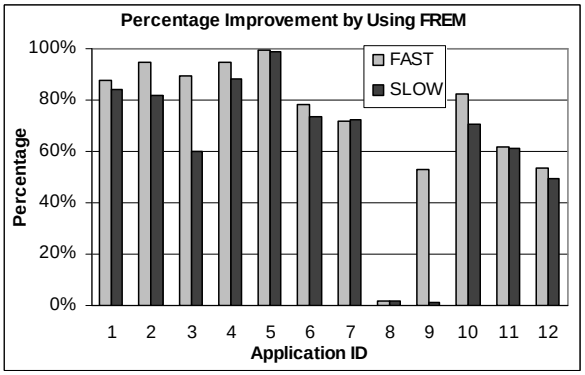

Figure 5. Percentage improvement on restart latency by using FREM over BLCR

For the *gcc* test cases (applications 6 and 7), the relatively small touch set is largely attributed to the dynamic memory deallocation of the applications. For

these applications, although the checkpoint image is large, it has many pages that will never be used again and will soon be freed. This observation indicates that dynamic memory management can provide more optimization opportunities for FREM.

5.2. Runtime Overhead

The use of FREM introduces two types of runtime overhead: (1) the post-checkpoint tracking overhead and (2) the fast restart overhead.

Table 3. Post-checkpoint tracking overhead (in milliseconds)

Application (input set)	Scan time	Search and insertion time	Descriptor I/O time	Tracking overhead
1: astar (1)	18.7	13.5	1.4	33.6
2: bzip2 (5)	48.8	0.3	0.2	49.2
3: bzip2 (6)	38.8	0.4	0.1	39.4
4: dealII	4.7	1.7	0.2	6.6
5: gamese (1)	36.7	1.7	0.3	38.7
6: gcc (4)	29.3	4.6	0.5	34.5
7: gcc (6)	43.7	15.0	1.5	60.2
8: lbm	21.8	0.5	0.1	22.4
9: mcf	33.9	0.2	0.2	34.2
10: perl (1)	18.6	13.6	1.4	33.6
11: soplex (2)	29.7	14.4	1.5	45.6
12: wrf	41.6	4.6	0.6	46.8

Table 4. Fast restart overhead (time unit: seconds)

Application (input set)	Remaining image (MB)	Duration of overlapping	Fast restart overhead
1: astar (1)	162	18.7	7.1
2: bzip2 (5)	476	55.4	13.1
3: bzip2 (6)	250	48.0	11.8
4: dealII	144	14.1	10.2
5: gamess (1)	424	59.5	21.7
6: gcc (4)	157	19.1	10.6
7: gcc (6)	560	76.9	22.7
8: lbm	5	0.8	0.1
9: mcf	8	1.4	0.3
10: perl (1)	83	12.4	6.9
11: soplex (2)	205	30.3	4.8
12: wrf	231	34.6	7.8

The post-checkpoint tracking overhead is mainly caused by three factors: the PTE scan time, the descriptor search and insertion time, and the I/O time to store the descriptor. Table 3 lists the measured post-checkpoint tracking overheads. We have observed similar results for runs in the **FAST** and **SLOW** networks. Due to space limitations, here we only present the results obtained in the **SLOW** network. It is shown that the post-checkpoint tracking overhead is generally less than 60.2 milliseconds, which is trivial compared to the performance gain achieved by FREM. The PTE scan time is the dominant contributor to the overhead. This is due to the fact that memory-demanding applications typically have huge page tables. In general, the search and insertion time is less than 15.0 milliseconds, while the descriptor I/O time is less than 1.5 milliseconds. These overheads are mainly determined by the number of entries in the touch set descriptor, which should not exceed two thousand, because of spatial data locality.

When using FREM, the restart of the process is overlapped with the image retrieval until all the remaining image is delivered to the destination machine. This overlapping inevitably incurs some overhead to the program execution due to resource contention. This is denoted as *the fast restart overhead*. Table 4 lists the sizes of remaining images to be retrieved, the durations of overlapping and the fast restart overheads in the **SLOW** network, for all applications. In general, the restart overhead is less than 22.7 seconds, which is much smaller than the reduction of restart latency achieved by FREM (see Section 5.1). Further, when the duration of overlapping is increasing, the overhead generally grows. This is caused by the fact that the number of context switches increases, thereby incurring more overhead. We believe on the emerging multi-core machines, the overhead can be reduced due to greater parallelism provided by the advanced architectures.

5.3. Statistical Performance Analysis

The results shown so far indicate that FREM can significantly reduce the restart time, but also introduces some runtime overheads. Given that checkpoint frequency is usually greater than that of recovery, a key question that may be raised is *whether FREM is capable of producing positive performance gain in the long run*. To answer this question, we conduct a set of experiments to examine application performance when using FREM in a long run. Here, the "long run" means that we statistically evaluate application performance between two restarts. In our experiments, we simulate Poisson failure arrivals of the underlying system, where the arrival rate ranges from one failure per 1000 days to 10 failures per day. The application checkpoint interval is set according to Young's approximation formula [26]. The **SLOW** network is used in this set of experiments.

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:

$$E_{gain} = RL\_improvement \times (1 - f)$$

$$E_{overhead} = tracking\_overhead \times N_{ckp} + fast\_restart\_overhead \times (1 - f)$$

Here, f is the failure probability of FREM, i.e., the chance that a failure occurs during the tracking window. N_{ckp} is the average number of checkpoints between two restarts.

Our simulations show that for all applications other than applications 8 and 9, E_{gain} surpasses $E_{overhead}$ by a significant margin ranging from 14.3 seconds to 183.3 seconds under different failure arrival rates. Due to space limitations, we only present the results for applications 1, 2 and 8 in Figure 6. For applications 1 and 2, the runtime overhead $E_{overhead}$ introduced by FREM is substantially smaller than the performance gain E_{gain} achieved regardless of failure rates. Further, it is shown that the overhead drops as the failure rate grows. When the failure rate increases, the number of checkpoints N_{ckp} decreases, thereby resulting in less post-checkpoint tracking overhead. For application 8, the benefit achieved by FREM is much less impressive. A major reason is that the application lacks temporal data locality, thereby resulting in trivial restart improvement. When the failure rate gets higher, the runtime overhead may overshadow the restart improvement. This observation suggests that data locality should be used as key guidance to determine whether to apply FREM or not.

5.4. Result Summary

In summary, the above experiments have shown that:

- For most applications, FREM can reduce restart latencies by 61.96% on average, as compared to the regular C/R mechanism. The results on the applications with good temporal data locality are more promising.
- The post-checkpoint tracking overhead incurred by FREM is around tens of milliseconds, which is trivial compared to the reduction in restart latency achieved by FREM (e.g., in the range of a couple of seconds to 208.5 seconds). The restart overhead depends on application characteristics, generally ranging from less than one second to 22.7 seconds.
- Our statistical performance analysis has shown that by using FREM, the expected application execution time between two restarts can be reduced by 14.3 seconds to 183.3 seconds.

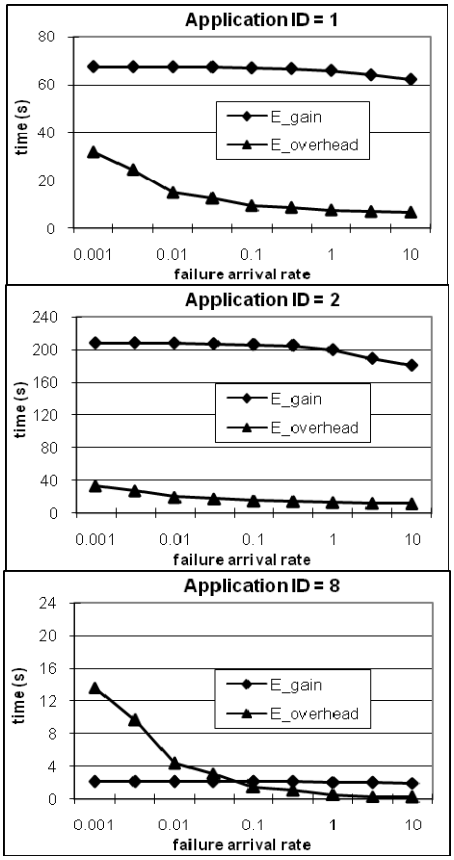

Figure 6. Statistical performance analysis of FREM

6. Conclusions

We have presented a novel mechanism called FREM to tackle the restart latency problem of general checkpoint protocols in networked environments. Through user-transparent system support, it allows fast restart on a partial checkpoint image by recording the process data access after each checkpoint. We have implemented FREM with the widely used BLCR checkpointing tool in Linux systems. Experiments on SPEC CPU2006 benchmarks have shown that FREM can effectively reduce process restart latency by 61.96% on average. In future, we will explore an aggressive way to estimate the tracking window. In addition, a more sophisticated image loading mechanism will be developed for better performance of FREM. Our ultimate goal is to integrate FREM with existing checkpointing tools for better fault management of applications.

Acknowledgement

The authors appreciate the valuable comments and suggestions from the anonymous reviewers. We would like to thank our paper shepherd, David Taylor, for his time and guidance.

References

[1] M. Baker and M. Sullivan, "The recovery box: Using fast recovery to provide high availability in the UNIX environment," in *Proceedings of Summer USENIX Technical Conference*, 1992.

[2] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, F. Cappello, "MPICH-V: A multiprotocol automatic fault tolerant MPI," *International Journal of High Performance Computing and Applications*, vol. 20(3), pp. 319-333, 2005.

[3] J. Daly, "A model for predicting the optimum checkpoint interval for restart dumps," in *Proceedings of International Conference on Computational Science*, 2003.

[4] E. Elnozahy and J. Plank, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Trans. on Dependable and Secure Computing*, vol. 1(2), pp. 97-108, 2004.

[5] S. Feldman and C. Brown, "IGOR: A system for program debugging via reversible execution," in *Proceedings of ACM SIGPLAN and SIGOPS workshop on parallel and distributed debugging*, 1989.

[6] P. Hargrove and J. Duell, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," in *Proceedings of SciDAC*, 2006.

[7] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *Proceedings of USENIX Annual Technical Conference*, 2007.

[8] Z. Lan and Y. Li, "Adaptive fault management of parallel applications for high performance computing," *IEEE Trans. on Computers*, in press.

[9] K. Li, J. Naughton and J. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Trans. Parallel and Distributed Systems*, vol. 5(8), pp. 874-879, 1994.

[10] Y. Ling, J. Mi and X. Lin, "A variational calculus approach to optimal checkpoint placement," *IEEE Trans. Computers*, vol. 50(7), pp. 699-708, 2001.

[11] D. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou, "Process migration," *ACM Comput. Surv.*, vol. 32(3), pp. 241-299, 2000.

[12] NCSA web site, *http://teragrid.ncsa.uiuc.edu.*

[13] A. Oliner, L. Rudolph and R. Sahoo, "Cooperative checkpointing: A robust approach to large-scale systems reliability," in *Proceedings of International Conference on Supercomputing*, 2006.

[14] Oracle high availability document website, *http://www.oracle.com/technology/deploy/availability/htdocs/fs_on-demand_rollback.htm*.

[15] D. Patterson et al., "Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies," *UC Berkeley Computer Science Technical Report UCB//CSD-02-1175*, 2002.

[16] J. Plank, Y. Chen and K. Li and M. Beck and G. Kingsley, "Memory exclusion: Optimizing the performance of checkpointing systems," *Software — Practice and Experience*, vol. 29(2), pp. 125-142, 1999.

[17] J. Plank, K. Li and M. Puening, "Diskless checkpointing," *IEEE Trans. Parallel and Distributed Systems*, vol. 9(10), pp. 972-986, 1998.

[18] J. Plank and M. Thomason, "Processor allocation and checkpoint interval selection in cluster computing systems," *Journal of Parallel and Distributed Computing*, vol. 61(11), pp. 1570-1590, 2001.

[19] S. Rao, L. Alvisi and H. Vin, "The cost of recovery in message logging protocols," *IEEE Trans. on Knowledge and Data Engineering*, vol. 12(2), pp. 160-173, 2000.

[20] J. Sancho, F. Petrini, G. Johnson, J. Fernandez and E. Frachtenberg, "On the feasibility of incremental checkpointing for scientific computing," in *Proceedings of International Parallel and Distributed Processing Symposium*, 2004.

[21] SPEC CPU 2006 benchmark website, *http: // www.spec.org/cpu2006/*.

[22] J. Squyres and A. Lumsdaine, "A component architecture for LAM/MPI," in *Proceedings of European PVM/MPI Users' Group Meeting*, 2003.

[23] A. Tanenbaum and A. Woodhull, *Operating Systems: Design and Implementation*, 2nd ed., New Jersey: Prentice-Hall, 1997.

[24] T. Tannenbaum and M. Litzkow, "The Condor distributed processing system," *Dr. Dobb's Journal*, vol. 227, pp. 40-48, 1995.

[25] N. Vaidya, "Impact of checkpoint latency on overhead ratio of a checkpointing scheme," *IEEE Trans. on Computers*, vol. 46(8), pp. 942-947, 1997.

[26] J. Young, "A first order approximation to the optimal checkpoint interval," *Comm. ACM*, vol. 17(9), pp. 530-531, 1974.

[27] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.