# Configurable Reliability in Multicore Operating Systems

Jianwei Liao, Taku Shimosawa and Yutaka Ishikawa
Graduate School of Information Science and Technology
The University of Tokyo
{liaojianwei@il., shimosawa@, ishikawa@}is.s.u-tokyo.ac.jp

*Abstract*— **This paper presents a new multicore operating system with a fault tolerance mechanism called Shimos2, which runs an instance of the kernel on each CPU core. It allows administrators to designate applications as requiring "high availability and reliability," without any modifications to an application's source code. Shimos2 contains a checkpoint/restart module, and it saves the running status of the designated applications periodically to the kernel's private memory area. Furthermore, a timer daemon and a monitor daemon are employed to detect kernels that are not working as a result of the host kernel being dead or hanging due to transient hardware faults. Once the host kernel is not working anymore, the stopped processes can be restarted on an idle kernel by automatically reloading the checkpointed image from where the last checkpoint was set. We have conducted experiments to evaluate Shimos2 from various aspects, including runtime overhead. The experimental results show that compared with Shimos2 without its fault tolerance mechanism, Shimos2 imposes less than 1.1% extra overhead on the designated applications themselves, and less than 1.2% extra overhead on other applications which have not been designated as "highly available." Compared to restarting the stopped processes on another virtual machine in KVM by using BLCR, Shimos2 can save around 18% of service downtime.**

## 1. INTRODUCTION

Having many cores available on a single CPU die is fast becoming an industry standard [1]. The resulting computing power and capability provides great opportunities. It also presents lots of challenges to the design and implementation of operating systems for these multicore systems. In practice, due to the growing bounty of cores, there will soon be a time where the number of cores in the system exceeds the number of active processes [2].

On the other hand, modern microprocessors and devices are susceptible to a variety of transient hardware failures due to myriad reasons, such as increasing number of transistors, decreasing feature sizes, reduced chip voltages and noise margins, etc. [3]. Unlike the bugs in software applications, these transient hardware faults may cripple the operating system, with a high probability of making the whole system crash [4]. A technical report presentation from Microsoft Corporation shows that transient hardware failures brought on about 8% of all system crashes and 9% of all unscheduled reboots [5]. Therefore, a fault tolerance mechanism in modern operating systems designed for multicore use is becoming an essential requirement [2].

In this paper, we propose a reliable multicore OS called Shimos2. The motivation behind the design and implementation of Shimos2 is grounded on two main trends. First, more and more cores are available on a single CPU die, but a large number of those cores are idle, usually because the number of cores exceeds the number of active tasks. Second, as software systems become more pervasive, the impact of the consequences of their failures becomes more significant. Work stoppages caused by the failure of a highly available application bring about keenly felt lost benefits. A white paper published by a solutions provider [6] shows that the average penalty for an hour of downtime for a brokerage service is more than 6.4 million US dollars. Therefore, these two trends offer a significant opportunity and also point out the direction fault tolerance solutions should take as an important factor in designing and implementing new operating systems for multicore systems.

On modern machines, many processes belonging to different applications are running at the same time. Among them, only a few processes, such as online e-commerce service and HTTP servers, need especially high reliability. Therefore, it is not necessary to set the same reliability level for all processes, as this would entail a great deal of extra overhead. Shimos2 adopts running the kernel on each and every CPU core. It allows an application which requires high availability and reliability to be configured without any source code modification or re-compiling. Then, during the execution phrase, Shimos2 is able to set checkpoints for the application, and save the checkpointed image to a private memory area of the host kernel. At the same time, Shimos2 employs its own timer daemon and monitor daemon to monitor the status of each kernel (i.e. to detect which kernels are dead, in an infinite loop, or hanging, etc.). Whenever a kernel becomes nonoperational, stopped processes that need high availability will be migrated to an idle kernel. The processes will be resumed nearly immediately with the latest checkpoint being set automatically.

In order to show that Shimos2 is applicable in practice, we evaluated the Shimos2 kernel from two aspects. The first is the runtime overhead of the fault tolerance mechanism. Our experimental results show that compared with Shimos2 without a fault tolerance mechanism, Shimos2 with the fault tolerance mechanism adds less than 1.2% to outgoing and incoming overhead when the checkpoint interval is 15 seconds. The second aspect is service downtime. We measured service downtime by using different techniques to

978-0-7695-4477-9/11 $26.00 © 2011 IEEE
DOI 10.1109/CSE.2011.176

resume the stopped computing task, and compared it to restarting another virtual machine in KVM using BLCR. Based on big memory application benchmarks, Shimos2 can save around 18% on service downtime. Needless to say, compared with reinitializing the applications or restarting it by reloading a checkpointed image saved on the disk after a cold reboot, Shimos2 can greatly reduce the amount of service downtime.

## 2. Design and Implementation of Shimos2

Fault tolerance is a critical concern in the design and implementation of modern operating systems. Various techniques have been proposed to cope with different causes of system failures. Obviously, there is no single technique that solves all problems. For example, SUD [7] runs device drivers at user-level, allowing it to tolerate the faults caused by malicious device drivers. However, this cannot prevent the kernel from going to into panic or hanging as a result of transient hardware faults. In general, a machine or device component will be rebooted after such system failures, and the applications will be re-initialized or reloaded [8]. In order to shorten the reboot time, various optimized reboot techniques have been proposed, such as the microreboot technique [9].

As mentioned previously, diagnosing the problem that caused the kernel to fail, rebooting the machine, and then reloading the kernel takes too much time. In order to reduce the downtime of applications requiring high availability and reliability that must be re-launched from system crashes in multicore systems, we propose a fault-tolerant multicore operating system called Shimos2. The aim of Shimos2 is to provide high, continuous-serving reliability for some special applications (as designated by administrators), such as a stock exchange system, or an e-commerce server, running on multicore architecture. Shimos2 makes such applications survivable by restarting them on another active kernel whenever transient hardware faults cause the host kernel to crash or hang, thus reducing the service downtime as much as possible. After the administrator designates certain applications as in need of high availability and reliability, Shimos2 sets periodic checkpoints for them. It saves their current status in the host kernel's private memory area. If the host kernel dies or hangs, the stopped computing tasks can then be automatically restarted on an idle kernel, without any intervention from administrators.

### *2.1 Shimos2's Architecture*

We use a quad-core CPU to demonstrate the architecture of Shimos2 (as shown in Figure 1). Shimos2 runs one independent kernel on each CPU core. Each kernel has its own physical memory area called the private memory area. In general, the Shimos2 kernel and applications can only use the allocated private memory area for that kernel. In contrast, a shared memory area is dedicated to inter-kernel communication and to storing the shared data structures.

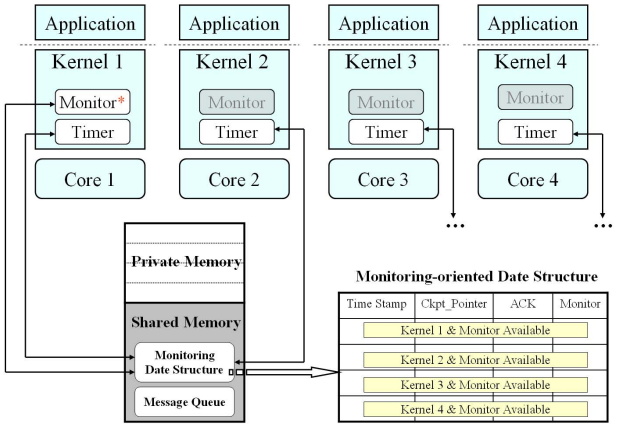


Figure 1. The Architecture of Shimos2

The checkpoint/restart module is compiled in the Shimos2 kernel. The checkpoint mechanism works only after the administrator designates which applications need both high availability and high reliability. Periodic checkpoints are then set for these applications. Section 2.3 provides detailed information about the checkpoint and restart mechanisms. Each kernel has two daemons, a timer daemon and a monitor daemon, to detect which kernel is not working (i.e. the kernel has crashed or is hanging). The monitor daemon will issue an instruction to restart the stopped computing process on an active kernel whenever needed. Section 2.4 presents the details on these two daemons.

### *2.2 Checkpoint and Restart Mechanisms*

Shimos2 saves and restores the opened files, pipes, sockets, and signals belonging to the target processes. In order to reduce the overhead caused by setting checkpoints and restarting the checkpointed processes, Shimos2 saves the checkpointed image in a reserved section of the kernel's private memory area, instead of on the disk. If the source kernel is dead or is hanging, the target kernel maps the reserved memory (in which the state of the checkpointed processes is stored) belonging to the source kernel. It then dynamically adds that area to its own virtual address space by a memory mapping technique called *ioremap*. After receiving the restart command sent by the monitor daemon, the target kernel reads the checkpointed image and restarts the stopped computing process. Finally, the dead source kernel and its private memory area will be re-initialized. Figure 2 clearly shows the process of setting checkpoints and restarting.

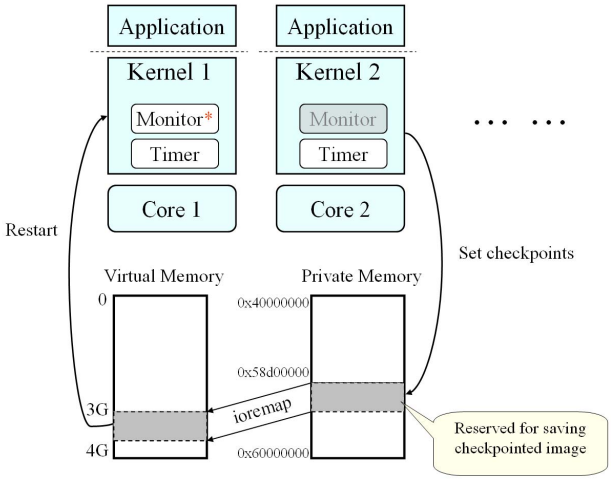


Figure 2. Checkpoint/restart Mechanism in Shimos2

### *2.3 Monitoring-oriented Data Structures*

There are two daemons in each kernel, i.e., a timer daemon and a monitor daemon. However, for all kernels, the timer daemons are active all the while the kernels are active, but only one monitor daemon is active at all times. These two types of daemon are used to detect the kernels' work status, set checkpoints, and issue the restart command to the target kernel.

A monitoring-oriented data structure has been introduced in Figure 1 to help the daemons work correctly. There are four main fields: the latest timestamp, the pointer to the reserved memory area where the checkpointed image is stored, an acknowledged timestamp, and the active monitor daemon's ID. The first two fields are used to detect whether the kernel is operational or not, while the last two fields are used to detect whether the current monitor daemon is still active. If not, a new monitor daemon will be elected. Each timer daemon can read and write only its own data entries, whilst the monitor daemon can operate on all data entries.

#### *2.3.1 Timer Daemon*

The timer daemon periodically writes the latest timestamp in the first field of the corresponding data entry in the monitoring-oriented data structure. In other words, the timer daemon shows the monitor daemon that its host kernel is still alive. In addition, the timer daemon also originates checkpoint commands.

#### *2.3.2 Monitor Daemon*

The monitor daemon checks data entries written by all of the timer daemons in sequence to detect whether there are any dead or hanging kernels. It checks the latest timestamp field and fills the acknowledge timestamp field whenever the checked timestamp is not the expected one. In other words, it evaluates the possibility that the corresponding timer daemon has not filled the latest one with the corresponding date entry. Thus, the monitor daemon deems that the kernel that contains the timer daemon is not working anymore. Next, the monitor daemon forms an inter-kernel message which includes the restart command. It then selects another active kernel and sends the message to it. Finally, it reboots and re-initializes the dead kernel after the stopped computing process has been resumed on another active kernel.

#### *2.3.3 Monitor Election*

As we mentioned before, the monitor daemon plays an important role in Shimos2. At any time, only one monitor daemon is active, and the others are designated as shadow monitor daemons. If a kernel's active monitor daemon is dead or is hanging, a new monitor daemon should be elected to take over control. In fact, whenever a monitor daemon checks the data entries, it needs to record the current timestamp in the acknowledge timestamp field, which was initialized to 0 when the timer daemon wrote the timestamp entry.

The timer daemon checks the acknowledge timestamp field of the corresponding data entry before it writes the new timestamp entry. If the value of this field is equal to 0, that means the monitor daemon has not checked this data entry since this entry has been updated. In other words, the kernel that has the active monitor daemon is most likely non-operational. Therefore, another active monitor daemon should be elected. Figure 3 shows the algorithm for a monitor election. After the crash of the current monitor daemon, the other active kernels try to nominate their own shadow monitors as the new one; finally, only one of the shadow monitor daemons can be elected as the active monitor daemon and take over the responsibilities of the crashed monitor daemon.

```
static int shimos_elect_monitor(int nominated_id, int monitor_id)
{
    …
   shimos_spin_lock_irqsave(&info->lock, flags);
    master = get_shimos_master();
    if (master->monitor == monitor_id)
    {
          master->monitor = nominated_id;
          ret = nominated_id;
    }else
          ret = master->monitor;
    shimos_spin_unlock_irqrestore(&info->lock, flags);
    return ret;
}
```

Figure 3. The Algorithm for monitor election

### *2.4 Message Mechanism*

Shimos2 provides an inter-kernel communication (IKC) mechanism to effect communication among the running kernels, such as sending restart commands from the monitor daemon to a selected active kernel, and managing the virtual device requests described in the next sub-section. The mechanism is composed of IKC packets, packet queues, and inter-kernel notification messages. An IKC packet is allocated from a shared memory area, and its structure varies based on its purpose. For instance, each kernel has multiple packet queues for console requests, network device requests, and daemon messages. The queue information is stored in a master table in

the shared memory visible to all kernels. A kernel selects the proper queue for the destination kernel from these queues. Upon enqueuing a packet, the kernel notifies the destination kernel. This is done with an inter-processor interrupt (IPI), which is standard equipment on shared-memory multiprocessor hardware.

### *2.5 Virtual Devices*

With support from the message mechanism, Shimos2 provides virtual device support to allow all cores to share hardware devices. For instance, with the support from a virtual network device, network-related applications work well, even after being resumed on another kernel. Virtual devices are abstracted to three types, just as in UNIX-like environments. These consist of character, block, and network devices. These virtual devices pass device requests and transform the IKC packets in conjunction with the kernel that manages the corresponding physical device.

### *2.6 Workflow of Shimos2*

Again using Figure 1, we show how Shimos2 works:

1) First, assume the administrator has configured an application that requires high availability on *kernel 2*. We will refer to it as *app1*. In addition, assume the active monitor daemon in the system is the monitor daemon of *kernel 1*..

2) While *app1* is running, the timer daemon of *kernel 2* writes the latest timestamps and sets checkpoints periodically.

3) On the other hand, the sole active monitor daemon checks the timestamps written by all timer daemons to make sure these kernels are still working.

4) Assume *kernel 2* starts to panic or hangs due to a transient hardware fault. At that time, the monitor daemon can detect that *kernel 2* is not working any more since its timer daemon could not write the latest timestamps.

5) The monitor daemon (please note that at any time, only one monitor daemon is active) chooses an idle kernel (for example, *kernel 1* in Figure 1), and sends an inter-kernel message to *kernel 1* to instruct it to begin a restart of the stopped *app1*.

6) After receiving the inter-kernel message, *kernel 1* restarts the stopped computing process by reloading the checkpointed image saved by *kernel 2*, and *app1* continues to run on *kernel 1* from the last checkpoint.

7) Finally, the monitor daemon reboots and re-initializes *kernel 2*.

### *2.7 Implementation*

We have implemented Shimos2 with a fault tolerance mechanism based on Shimos [12], which is a Linux kernel 2.6.31-based multicore operating system. In the Shimos2 kernel, there are more than 20 new source code files written in C, including the checkpoint/restart module[18], the timer daemon, the monitor daemon, the inter-kernel messaging module, the virtual network device, functions focused on the monitoring-oriented data structure, and so on. In addition, in order to allow the administrator to communicate with the Shimos2 kernel from the user level, we have also implemented a character device and a user level daemon. With the presence of these implementations, the administrator can designate which applications need high availability and reliability by resorting to the user level daemon dynamically from the user space.

## 3. EVALUATION

We have evaluated Shimos2 from two aspects. First, we measure the performance overhead brought on by Shimos2. The checkpoint/restart module and the daemons consume some computing resources, even though they are running in the background. Second, we also measure the service downtime once the processes are stopped due to kernel crashes or hangs. Then we measure the service interruption time of the application until it runs again. Before we present the experimental results, we will first introduce the experimental platform and the benchmarks used in our evaluation experiments.

### *3.1 Experimental Environment*

#### *3.1.1 Experimental Platform*

The experimental platform is a DELL poweredge R410, which is a dual-core, dual-processor SMP machine (four cores in total). The details of our experimental platform and the default memory configuration in Shimos2 are shown in Table I.

TABLE I. EXPERIMENTAL ENVIRONMENTS

| **Hardware Environment** | |
|---|---|
| **CPU** | Intel Xeon L5520 (2.26GHz, 4 cores)<br>Hyper-Threading and Turbo Boost enabled |
| **Memory** | 3GB |
| **Network** | NetXtreme II BCM5716 x 2 |
| **Shimos2 Configuration** | |
| **Private Memory** | 512MB |
| **Network** | 4MB |

#### *3.1.2 Benchmarks*

We selected two groups of benchmarks to be used to evaluate the Shimos2 kernel:

One of the benchmark suites is SPEC CPU2000 [13]. It is designed to test CPU performance by measuring the run time of several programs such as the gzip parser and the chess program, crafty. We use this benchmark suite to measure the runtime overhead of Shimos2 running in the failure-free mode.

The second group includes real-world application benchmarks. It is used to measure the service downtime or restart time for, for example, Thttpd, a single process HTTP server, which is a simple, small, portable, fast, and secure HTTP server [14]; the Apache HTTP Server, a well-known multiprocess HTTP web server [15]; Matrix multiplication (MAT), with double precision floating-point as the data type;

Subversion, a widely used version control software application; and Jftpgw, a proxy server for the FTP protocol[16]. Table II shows the application scenarios used.

TABLE II. APPLICATION SCENARIOS

| | |
|---|---|
| **thttpd** | Thttpd Version 2.25b with 1 process<br>1 http connection from an external client |
| **Apache** | Apache Version 2.0.63 with 7 processes,<br>2 http connections from an external client |
| **MAT** | MAT The size of matrix is 2048 * 2048<br>Element is double(64 bit) |
| **subversion** | Subversion Subversion Version 1.6.6 with 1 process<br>A local client is checking out a 1GB file |
| **jftpgw** | Jftpgw Version 0.14.4 with 1 process |

### *3.2 Runtime Overhead*

SPEC CPU2000 is used as our benchmark, and is executed on several OS kernels, including Linux kernel 2.6.31. The Shimos2 kernel used in the experiments has an active time daemon and an active monitor daemon. For all experiments in Section 3.2, we assume that all the different kernels are running in the failure-free mode.

We have conducted two kinds of experiments to show the runtime overhead incurred by Shimos2. The first one is designed to measure the degree of the influence of the fault-tolerance mechanism on the applications. These applications have not been designated as requiring high availability. This kind of overhead is defined as outgoing influence. The second experiment aims to disclose to what extent Shimos2 influences applications that have been designated as requiring high availability. Similarly, this is defined as incoming influence.

#### *3.2.1 Outgoing Influence*

In the experiments, a real-world application, the Apache HTTP server, is configured as an application requiring high availability. Accordingly, Shimos2 sets checkpoints for it every 15 seconds while it is running. The intervals for the timer daemon and the monitor daemon are set to 5 seconds, each. At the same time, SPEC CPU2000 is running on the same core.

In Figure 4, the label Shimos-ft-ck represents the Shimos2 with the fault-tolerance mechanism and the Apache HTTP server has been designated as requiring high availability. Shimos2 needs to set checkpoints for it periodically. The label Shimos-ft stands for Shimos2 with the fault-tolerance mechanism but no applications that have been designated as high availability ones. This means the daemons are active, but there is no checkpointing. The label Shimos-no means Shimos2 without the fault-tolerance mechanism. The vertical axis with the term "Base Ration" represents one of the performance parameters in SPEC CPU2000. It may be considered as a score, the higher the better.

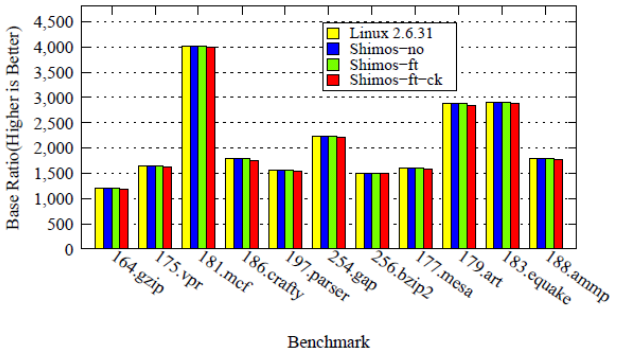


Figure 4. Outgoing Influence: The Base Ration of SPEC CPU2000

Figure 4 shows that compared with Shimos2 without the fault-tolerance mechanism, Shimos2 only slows down the SPEC CPU2000 benchmark at a rate of no more than 1.2%.

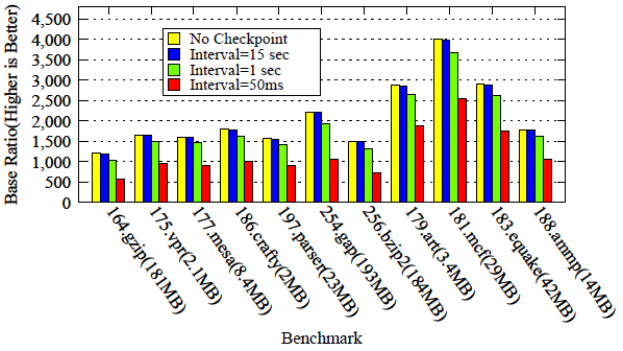


Figure 5. Incoming Influence: The Base Ration of SPEC CPU2000

#### *3.2.2 Incoming Influence*

We also used SPEC CPU2000 to measure the influence on the applications themselves. These applications have been designated as requiring high reliability.

Since different high availability applications need different checkpoint intervals. Figure 5 illustrates that in contrast to Shimos2 without the fault-tolerance mechanism, Shimos2 can provide fault-tolerance functionality with around a 1%, 10% or 45% overhead penalty at intervals of 15 seconds, 1 second, and 50 milliseconds, respectively. Although a 45% performance decrease is nontrivial, it is not necessary to set checkpoints for all applications at a level of 50 milliseconds.

### *3.3 Fault Injection*

In order to evaluate fault detection and fault recovery, we use a tool to emulate memory bit-flips by writing a modified bit to the memory area after mapping the memory device. In the automatic fault injection test, the result is nondeterministic, since the result is sensitive to the location where the fault is injected and the timing etc. We repeated the fault injection 1000 times, and there were only 7 cases requiring restarting of the target processes on another active kernel due to crashes or infinite loops on the source kernel.

TABLE III. APPLICATION SCENARIOS

| Faut Types | Activity Description |
|---|---|
| **NULL Pointer** | Unable to handle kernel NULL pointer |
| **PANIC()** | Call PANIC() marcor |
| **Infinite Loop** | In while(1) loop |
| **IRQ Deadlock** | Make deadlock with irq_save |

In addition to using the automatic fault injection tool, we also did the experiments by manually injecting faults to generate deterministic behavior and more crash cases. Several fault types involving manual injection are listed in Table III. Without doubt, these faults are most likely to lead the kernel to crash or to reach a non-operational state. As a result, the stopped computing process was required to be migrated to another active kernel automatically after the source kernel performed incorrectly.

### *3.4 Service Interruption*

We chose two comparison counterparts. One is a restart by reloading the checkpoint image from disk, even though it is unfair. Generally, on a single machine, the traditional checkpoint mechanism saves the status of the running application on nonvolatile storage, and the restart mechanism reloads the status of the checkpointed processes from this nonvolatile storage to resume them, because there is no other choice.

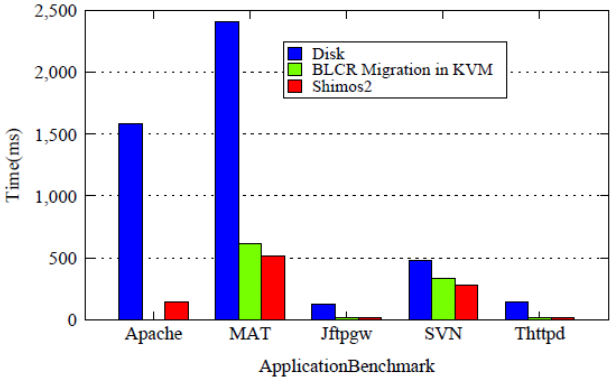

Figure 6. Restart Time

Migrating the processes to another virtual machine running on the same physical machine is another comparison counterpart. This counterpart seems pretty fair because the stopped computing process can be resumed to run on another virtual machine without a cold reboot. We make an assumption that the hypervisor of KVM never crashes, since virtual machines cannot work without the support from the KVM hypervisor. We conducted the process migration to another virtual machine in KVM by employing the unmodified version of BLCR, 0.8.2 [17], compiled with the Linux Kernel, version 2.6.28. BLCR is used to transfer the status of the checkpointed processes from the source virtual machine to the target virtual machine (the target virtual machine is determined by the source virtual machine before setting the checkpoint), using the TCP protocol. Since BLCR cannot save and restore the socket connections of the checkpointed processes, after the restart, the socket connections are closed for some application benchmarks which were preserved before setting the checkpoint.

Figure 7 shows the elapsed time to restart of the stopped process (es) with the three given restart mechanisms. All experiments are conducted with a cold cache. Because BLCR could not restart the Apache Http Server correctly in our experiments, the restart time using BLCR to resume the Apache Http Server benchmark is not shown.

It is clearly shown in Figure 7 that, in contrast to restarting a stopped computing process by reading the checkpointed image saved on the disk, Shimos2 uses much less time to resume a stopped computing process. Moreover, Figure 7 also shows that compared with Shimos2, migrating the process to another virtual machine in KVM using BLCR performs worse (around 18% worse) than Shimos2 does with the Matrix Multiplication and Subversion benchmarks, respectively. We noticed however, for the Jftpgw benchmark, Shimos2 takes 20 milliseconds. On the other hand, BLCR takes only around 16 milliseconds. Perhaps due to the small size of the checkpointed image, the time needed to transfer it via the TCP protocol to the target virtual machine requires less time than Shimos2 needs when it maps the reserved memory area and reloads the checkpoint image.

## 4. RELATED WORK

A. Depoutovitch and M. Stumm[10] have proposed a mechanism called Otherworld, which can restart applications from where the system crashed by resorting to a solution called a KDump, without the intervention of an administrator. However, these techniques take too much time to re-launch the applications when the system goes dead or hangs due to transient hardware faults. B. Gerofi, et al.[11] proposed a live migration mechanism for a cluster environment. They employ a backup node for each active node: the active node periodically transfers incremental checkpoints to the backup node for the processes running on it. The latest image of the running processes is held by the backup node. Therefore, once the active node has been detected as in a nonoperational state, the backup node can take the control and run the latest image of the running processes. The backup node will be in place within a very short time.

In addition, A. Kadav, M.J. Renzelmann and M.M. Swift presented a software solution for tolerating device transient faults[19]. Carburizer analyzes the device driver source code and to find the sensitive locations(please note that these locations are not actual bugs) where the driver may result the OS kernel to crash due to the unusual hardware device behave; then treats these locations as bugs and tries to fix them. F. David and D. Chen[20] proposed a technique for recovering from operating system crashes caused by bit-flips of kernel data and faulty hardware. This technique assumes that after a system crash, user program state and most operating system state is still in memory and never be corrupted, then it resets the processor, re-initialize the kernel and reloading the user

program state and useful operating state. However, the fault models of this technique are quite limited, for instance, the kernel task list and the state of other kernel threads should never be contaminated.

## 5. Conclusion and Future Work

In this paper, Shimos2, a new multicore operating system with fault tolerance capabilities has been proposed, implemented, and has had its performance evaluated. Shimos2 uses a checkpoint/restart module to set periodic checkpoints for certain processes requiring high availability. It employs two daemons, called timer daemon and monitor daemon, respectively, to detect which kernels are not operational. In the case of failure, another active kernel will be activated to restart the stopped processes (if the system is configured properly). The target kernel will map (using the ioremap mapping technique) the reserved memory area (where the state of the stopped processes is saved) in the dead kernel's private memory area to its own virtual address space, and reload the status of the checkpointed processes. Finally, the stopped processes will be resumed, starting to run on the target kernel.

From the experimental results showed in Section 3, we can see that Shimos2 can provide a fault-tolerance mechanism for some applications requiring high availability and reliability with reasonable outgoing and incoming overhead. For example, for the SPEC CPU2000 benchmark, when the time interval is 15 seconds, both overheads are less than 1.2%. For most cases of restarting process(es), especially for big memory applications, Shimos2 performs better than migrating the process to another virtual machine in KVM using BLCR. In other words, Shimos2 incurs little extra performance overhead to ensure an application's availability. It is able to improve system reliability to a certain degree.

We should point out that there are still some situations where Shimos2 cannot restart a stopped computing process correctly. For instance, when the crash of the local kernel has damaged important data structures belonging to other kernels, which might lead other kernel to crash as well. Because the private memory areas belonging to different kernels in Shimos2 are protected from software view, rather than hardware view, the above-mentioned situations may occur. We will investigate methods required to protect such important data structures in future work. Also, the current implementation of Shimos2 does periodic full check pointing every time, implementing an incremental check pointing feature will be better in order to reduce the overhead of setting checkpoints.

## Acknowledgments

This work was partially supported by the CREST project of the Japan Science and Technology Agency (JST). We would also like to thank Balazs Gerofi for helping us to conduct the experiments using BLCR.

## References

[1] D. Geer. Industry trends: Chip makers turn to multicore processors. Computer, 38(5):11–13, 2005

[2] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing, pages 3–14, New York, NY, USA, 2010. ACM.

[3] P. M. Wells, K. Chakraborty, and G. S. Sohi. Adapting to intermittent faults in multicore systems. In ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, pages 255–264, NY, USA, 2008. ACM

[4] D. Chen, S. Dharmaraja, D. Chen, L. Li, K. S. Trivedi, R. R. Some, and A. P. Nikora. Reliability and availability analysis for the jpl remote exploration and experimentation system. In DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks, pages 337–344, Washington, DC, USA, 2002. IEEE Computer Society.

[5] S. Arthur. Fault resilient drivers for longhorn server. Technical report, Microsoft Corporation.

[6] A. Arnold. Assessing the financial impact of downtime. 2010.

[7] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In USENIX ATC 2010, June 2010.

[8] A. Adya, W. J. Bolosky, M. Castro, G. et al.. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation, pages 1–14, New York, NY, USA, 2002. ACM.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot-a technique for cheap recovery. In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, pages 31–44, Berkeley, CA, USA, 2004. USENIX Association.

[10] A. Depoutovitch and M. Stumm. Otherworld: giving applications a chance to survive os kernel crashes. In EuroSys '10: Proceedings of the 5th European conference on Computer systems, pages 181–194, New York, NY, USA, 2010.ACM.

[11] G. Balazs, F. Hajime, and Y. Ishikawa. An efficient process live migration mechanism for load balanced distributed virtual environments. In CLUSTER '10: Proceedings of the 2010 IEEE International Conference on Cluster Computing, Washington, DC, USA, 2010. IEEE Computer Society

[12] T. Shimosawa, H. Matsuba, and Y. Ishikawa. Logical partitioning without architectural supports. In COMPSAC '08: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference, pages 355–364, Washington, DC, USA, 2008. IEEE Computer Society.

[13] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. Computer, 33(7):28–35, 2000.

[14] thttpd - tiny/turbo/throttling HTTP server. http://www.acme.com/software/thttpd/.

[15] Apache- Http Server Project. http://httpd.apache.org/.

[16] FTP roxy. www.mcknight.de/jftpgw/.

[17] J. C. Hargrove, Paul H.; Duell. Berkeley lab checkpoint/ restart (blcr) for linux clusters. Journal of Physics: Conference Series, 46(1):494–499, 2006.

[18] Liao, J. and Ishikawa, Y.: A new concurrent checkpoint mechanism for real-time and interactive processes, COMPSAC2010: Proceedings of 34th Computer Software and Applications Conference, Washington, DC, USA, IEEE Computer Society (2010).

[19] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 59–72, New York, NY, USA, 2009. ACM.

[20] Recovering from Operating System Crashes. http://choices.cs.uiuc.edu/crashrecovery.pdf.