Flexible Page-level Memory Access Monitoring Based on Virtualization Hardware

Kai Lu Wenzhe Zhang Xiaoping Wang

Science and Technology on Parallel and Distributed Laboratory, State Key Laboratory of High Performance Computing, State Key Laboratory of High-end Server & Storage Technology, College of Computer, National University of Defense Technology, Changsha, PR China

{kailu, zhangwenzhe, xiaopingwang}@nudt.edu.cn

Mikel Luján Andy Nisbet

School of Computer Science, The University of Manchester, Manchester, UK

{mikel.lujan, andy.nisbet}@cs.man.ac.uk

Abstract

Page protection is often used to achieve memory access monitoring in many applications, dealing with program-analysis, checkpoint-based failure recovery, and garbage collection in managed runtime systems. Typically, low overhead access monitoring is limited by the relatively large page-level granularity of memory management unit hardware support for virtual memory protection. In this paper, we improve upon traditional page-level mechanisms by additionally using hardware support for virtualization in order to achieve fine and flexible granularities that can be smaller than a page. We first introduce a memory allocator based on page protection that can achieve fine-grained monitoring. Second, we explain how virtualization hardware support can be used to achieve dynamic adjustment of the monitoring granularity. In all, we propose a process-level virtual machine to achieve dynamic and fine-grained monitoring. Any application can run on our process-level virtual machine without modification. Experimental results for an incremental checkpoint tool provide a use-case to demonstrate our work. Comparing with traditional page-based checkpoint, our work can effectively reduce the amount of checkpoint data and improve performance.

CCS Concepts • **Software and its engineering → Virtual memory**; **Allocation / deallocation strategies**

Keywords Memory access monitoring, Dynamic, Fine-grained, Virtualization hardware

1. Introduction

Virtual memory page protection is often used to achieve memory access monitoring in application scenarios concerning program-analysis, and checkpoint-based failure recovery, etc. Current monitoring mechanisms can only efficiently monitor modifications at page-level granularities that are directly supported by memory management unit hardware. In this paper, we introduce an improved page-level monitoring mechanism that additionally exploits virtualization hardware support, in order to achieve fine (sub-page), and flexible memory access monitoring.

We introduce a mechanism that exploits both i), hardware support for page-level protection, and ii), virtualization, in order to improve on traditional page-level memory access monitoring. The main work and contributions of this paper are listed below:

- First, we introduce a memory allocator that spreads objects sparsely in different virtual pages that are abundant on modern 64-bit CPUs. Based on virtual page-level protection, we can achieve fine-grained object-level access monitoring. Physical memory usage is reduced by mapping multiple virtual pages to one physical page. As a result, multiple objects from different virtual pages may reside in the same physical page and be monitored independently.
- Second, we rely on the additional page table (in this paper we refer to it as the *extended page table* (EPT) (Intel 2013)) provided by hardware support for virtualization, to dynamically adjust the granularity of monitoring. In this scenario, any address used by a virtualized application goes through two levels of address translation (guest virtual address → guest physical address → machine address). The two levels of address translation provides the flexibility to dynamically move two or more objects (in different virtual pages) together into the same page, and vice versa, to divide and separate them in a transparent way. It is this functionality that enables dynamic adjustment of access monitoring granularity.

- Last, we introduce a process-level virtual machine and a runtime memory allocator based on Linux. The process-level virtual machine is a Linux kernel module that manages the extended page table. Any application can run on our process-level virtual machine without change. To the best of our knowledge, this is the first work to use virtualization hardware (two level address translation) to implement flexible fine-grained sub-page memory access monitoring for native (C or C++) programs.

Our work can easily be adopted to better support existing tools (such as program analysis, and incremental checkpoint) that rely on a traditional page-level monitoring mechanism. To test and demonstrate our work, we developed an incremental checkpoint system. Compared with traditional page-based incremental checkpoint, we can greatly reduce the volume of checkpoint data and improve performance.

The rest of this paper is organized as follows: we introduce background information on paging/virtualization, the design, and the implementation of our system in sections 2, 3 and 4 respectively. We discuss the use-case of incremental checkpoint in Section 5. The experimental results are presented in Section 6. Section 7 discusses related work, and section 8 presents our conclusions.

2. Page Level Memory Access Monitoring & Virtualization Hardware

In this section we introduce background information to help place our work in context. First, we introduce current mechanisms for achieving memory access monitoring based on page protection. Then we give a brief introduction to virtualization hardware (Uhlig et al. 2005) (AMD 2005) that we exploit to achieve fine and flexible granularity.

2.1 Page Level Memory Access Monitoring

CPU memory management units provide hardware support for virtual memory (Appel and Li 1991). That is, applications are coded and executed with virtual addresses and the virtual addresses are translated by a memory management unit (using a page table) to physical addresses that are finally used to access physical memory locations. Currently this translation is done at the granularity of page. Page-level protection achieves memory access monitoring by changing the protection bit in the page table entry of monitored pages. Accessing protected pages will then trigger a page fault. It is fast to do this because the paging mechanism is supported by hardware. However, page protection can only detect modification at page-level granularity which is its core limitation. For example, the deterministic multi-threading runtime systems Dthreads (Liu et al. 2011) and RFDet (Lu et al. 2014) use page-level protection and page mapping to monitor and redirect memory accesses but they introduce considerable overhead on deciding which part of a page is modified. For example, additional software and storage overheads are incurred in order to compute diffs for each modified monitored page.

To summarize, page-level monitoring mechanisms can only efficiently determine which addresses are modified at page-level granularity.

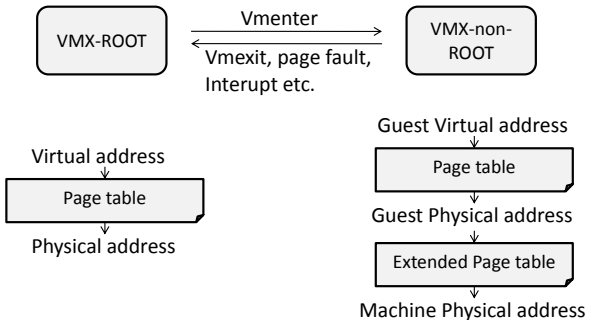

Figure 1. Hardware support for virtualization.

2.2 Hardware Support for Virtualization

Pure software virtualization for x86 is expensive, especially for memory virtualization. Intel and AMD have both developed hardware support for x86 virtualization (VT-x from Intel (Uhlig et al. 2005) and SVM from AMD (AMD 2005)). We focus on describing Intel's VT-x support for memory virtualization in this section, as it is directly used by our implementation.

As shown in Figure 1, VT-x introduces two new CPU modes: *Vmx-root* and *Vmx-non-root*. The Vmx-root mode is designed to run a native system and has access to the full instruction set architecture, whilst the Vmx-non-root mode is designed for virtualized execution. In Vmx-root mode, the CPU behaves as usual and it can execute certain new instructions to manage the virtual machine and enter virtualized execution (*vm-enter*). Any virtual address used by Vmx-root mode applications will be translated by the page table into a physical address as usual. Meanwhile, in Vmx-non-root mode, the CPU is restricted from performing certain instruction set architecture behaviors, and these behaviors cause a *vm-exit* that results in a transition into Vmx-root mode. The underlying native system in Vmx-root mode is then able to have full control of the virtual machine running in Vmx-non-root mode. In Vmx-non-root mode, any address used by an upper application (referred to as a *guest virtual address*) will first be translated by the page table into a *guest physical address* and then be translated by the extended page table into a final *machine address*. The extended page table is the main hardware support offered for memory virtualization.

The *VMCS* (Virtual Machine Control Structure) is a data structure that is used to define the behavior of the virtual machine. For example, setting appropriate bits in the VMCS determines the interrupts that initiate a vm-exit.

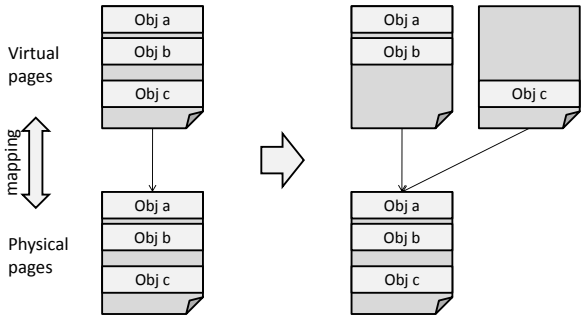

Figure 2. Fine-grained monitoring without virtualization.

3. Design

In this section, we first introduce a design without the support of virtualization hardware, that achieves static fine-grained memory access monitoring only using a memory allocator, and the normal page table. Then we present an improved design that uses the virtualization hardware's extended page table to achieve dynamic adjustment of monitoring granularity.

Note that in the design and implementation sections we discuss and show how to monitor writes by write-protecting the corresponding pages (changing the protection bit to be read-only in the page table), monitoring reads can be done in a similar manner.

3.1 Fine-grained Monitoring Based on Page Protection — No Virtualization Hardware

In order to monitor the access to objects of any size, the simplest way is to give each object a single page and then to write-protect its page. If a page is written, then we will know that the corresponding object is write-accessed. However, this may introduce two kinds of overhead: (1) we actually allocate a physical page to store each object and thus there will be a huge waste of physical memory if there are many small objects; (2) we will use a large range of virtual memory and cause many more TLB misses, leading to a significant performance overhead.

In order to solve the above problems and achieve fine-grained monitoring, we adopt the mechanism shown in Figure 2. As an example, we divide a virtual page into two parts logically: the upper half and the bottom half. Objects allocated in the upper half just reside in the same virtual page. If we find an object to be allocated in the bottom half of the virtual page, we create a new virtual page and put the object in the corresponding place of the new virtual page (the object `c` in Figure 2). In order to reduce the physical memory usage, we map the two virtual pages to the same physical page. Then we write-protect the two virtual pages. If, for example, the second virtual page (containing object `c`) is written, we will know that only the bottom half of the page content is modified. This mechanism is implemented by our memory allocator and the granularity of memory access monitoring is the page size divided by 2 for the scheme in figure 2. Similarly, we can further divide a virtual page into four parts or eight parts to reduce further the monitoring granularity. This design solves two of the overhead problems raised in the previous paragraph: (1) we map multiple virtual pages to one physical page to reduce physical memory usage. The only increase in physical memory usage is for storing more slots in the page table; (2) we do not give each object a single virtual page. Instead, we divide one virtual page into 2 or 4 parts. In this way we can limit the usage of virtual memory and thus limit the TLB misses incurred in comparison to a scheme where a single object is allocated on each virtual page.

Out-of-Bounds Access. As two objects in different virtual pages may reside in the same physical page, an object may be modified through another object's virtual page. However, we argue that this situation only happens in buggy codes which do out-of-bounds accesses. For example, an application allocates an object `a` and an object `b` (both of size 2048) through malloc(2048). Our heap returns an address 0xB0001000 for object `a` and an address 0xB0011800 for object `b`. Objects `a` and `b` are not in the same virtual page but they reside in the same physical page `p`. From the programmer's view, the address range of object `a` is [0xB0001000, 0xB0001000+2048]. Normally, an application will not access the virtual address range [0xB0001000+2048, 0xB0001000+4096] (accessing this range may change object `b`) because this address range has not been given to the application and is invisible to the program. A similar situation holds for accesses to object `b`. Above all, normally programs should just operate on the valid address ranges obtained from memory allocators (through malloc). In this paper we do not handle out-of-bounds memory access. If an upper application performs an out-of-bounds access, then our tracking system will result in a buggy state. Programs performing out-of-bounds accesses contain bugs and are likely to have unpredictable behaviour.

Objects That Cross A Divided Page Boundary. In the example above we divide each virtual page into 2 parts. There may be objects that start from the upper half of a page and cross the half boundary. In this case we just put the object in the first virtual page that represents the upper half. It crosses the half boundary but there will be no object overlapping with it in the underlying physical page.

Large Objects That Cross Multiple Pages. For large objects that cross multiple pages, our system behaves the same with traditional mechanisms that just do traditional one-to-one mapping. We argue that our system aims to achieve fine-grained monitoring and we focus on applications that use a lot of small objects. Furthermore, large objects that cross multiple pages will not benefit nor degrade with our system.

3.2 Going Flexible — with Virtualization Hardware

Our fine-grained monitoring design introduced in the previous subsection has a problem. That is, the monitoring gran-

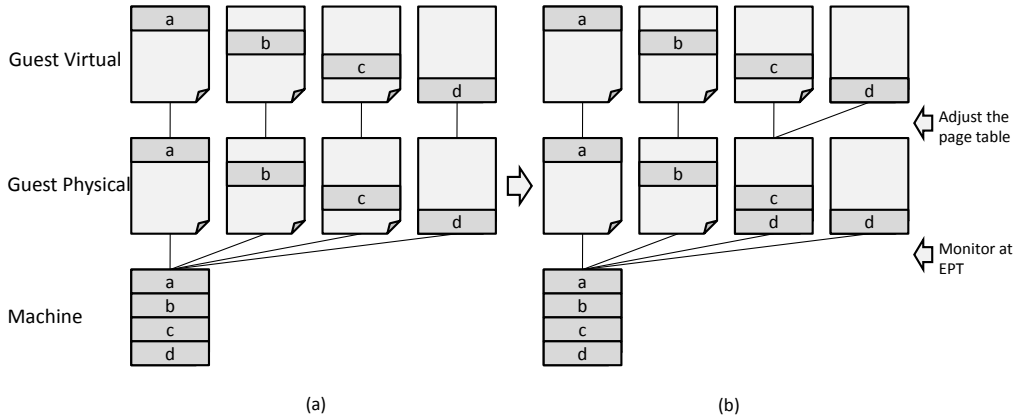

Figure 3. Fine-grained dynamic monitoring with virtualization.

ularity is determined statically at the beginning of the program. We cannot adjust the monitoring granularity during execution. For the objects, we have only one chance to determine where to put them (at object allocation time). After the *malloc*, we cannot move them because upper applications will always use the first allocated virtual addresses returned by malloc to access them.

The ability to dynamically move objects transparently is important. For example, in the example shown in Figure 2, if we find the divided two virtual pages (one containing object `a` and `b` and another one containing `c`) are always written together, a better way to reduce monitoring overhead (and hence to reduce the page faults) is to move the three objects (objects `a`, `b`, and `c`) together into one virtual page. However, we cannot achieve this in the current page-level monitoring design.

In order to move objects dynamically and transparently (thus, to achieve dynamic adjustment of monitoring granularity), we need to use virtualization hardware features where an additional level of address translation (Intel 2013) is available. Modern CPUs with virtualization support offer at least two modes, for example, Intel CPUs have: (1) Vmx-root mode and (2) Vmx-non-root mode (Intel 2013). In Vmx-non-root mode, the process address (guest virtual address) will first be translated by the page table to a guest physical address and then be translated by the extended page table to the final machine address. Figure 3 illustrates how the two levels of address translation enable dynamic adjustment of memory access monitoring granularity in our scheme. In this example, we divide a virtual page into four equal parts. At the first level of mapping (the page table level), we perform a one-to-one mapping. Then at the second level (extended page table), we map the four pages (guest physical page) into one (machine page), in order to reduce physical memory usage. In this architecture, the first level page table is used to remap guest virtual addresses to guest physical addresses, in order to adjust the granularity of memory access monitoring. At the second level of the extended page table the memory access monitoring is performed by write protecting pages. As Figure 3(a) shows, we first write-protect the four guest physical pages in the extended page table. If, for example, the guest physical page d is written, we will know only the last quarter of the page content is modified. Moreover, if we find the guest physical page c and d are always modified together, a better way is to merge them together into one page. Here we can modify the page table at the first level to achieve this. Figure 3(b) shows how to do this. We just remap the guest virtual page d to guest physical page c. Hence, now the guest physical page c actually contains two parts (c and d). If this page is further modified and the extended page table fault is triggered, we will know that the bottom half of the page content (object c and d) were modified. By doing this we can dynamically adjust the monitoring granularity during execution.

There are two benefits of this design: (1) when we adjust the monitoring granularity, we do not actually move any parts or objects from one page to another. We just change page table entries to perform remapping. There is no object copy overhead; (2) we can expose the first level of the page table to user programs. Thus, user programs can modify the page table directly to do remapping without calling into the system kernel. There is no context switch overhead. The kernel can use the extended page table to prevent upper guest applications from accessing unauthorized addresses. This second benefit is innovated by extending previous implementation work in Dune (Belay et al. 2012).

3.3 Dynamic Strategy

We have shown how to achieve flexible granularity with the support of virtualization hardware. We offer the ability to transparently and dynamically adjust granularity at run time. This is the core mechanism of this paper. For when, and how to adjust granularity, we argue that this dynamic adjustment strategy should be determined on a case-by-case basis, and should be determined by the upper application use-case scenarios.

For example, in the incremental checkpoint use-case, our dynamic strategy is history-based. We undertake periodic sampling, by first write protecting all guest physical pages, and then recording the set of modified guest physical pages using the page fault mechanism. Then after expiration of a timer, we analyze all records and write protect all the guest physical pages again. When we find any guest physical pages that are frequently modified, (such as when they are modified in several previous sampling rounds), we try to merge them together. An additional issue is to determine when to separate merged pages, as after merging we will only be able to know which merged large address part is modified (for example the merged part object c and d shown in Figure 3(b)). We cannot know which single small part (for example the merged address parts of an object c or d, as shown in Figure 3(b)) is modified. Our strategy for this is to separate pages completely after a certain period and then to merge them again gradually. We will discuss this strategy later in the experimental section. The goal of such a strategy is to enable the fine-granularity mapping to dynamically adjust to different program execution phases where the locality of write access behaviour may change over time.

4. Implementation

Our implementation is based on Linux kernel 3.16 and contains two parts: (1) a kernel module for managing the extended page table and supporting virtualization execution at process-level and (2) a runtime library for memory allocation and managing the normal page table exposed to user applications (as we have two levels of address translation, we can expose the first level of the page table to upper guest applications and at the same time guarantee safety; see Section 2.2).

4.1 Kernel Module for Process-Level Virtualization

We adopt the implementation strategies deployed in Kvm (Kivity et al. 2007) and Dune (Belay et al. 2012) to manage the virtualization hardware and related data structures (such as VMCS (Intel 2013)). Dune is a process-level virtual machine that exposes privileged information and instructions to guest applications. Dune exploits Intel processors' priviledged levels exposed as rings, where virtualization support is used to guarantee safety whilst allowing applications to run at the most priviledged level in ring 0, where they can access and modify the guest virtual to guest physical address page table information. Note that applications are prevented from modifying the extended page table. The execution within Dune is mainly based on a loop (as shown in Figure 4): Dune first sets up all data structures for virtualizated execution and performs a VM-enter instruction to allow the CPU to execute user code in Vmx-non-root mode. The precise circumstances when an application needs to perform a VM-exit to execute kernel code, such as on performing a

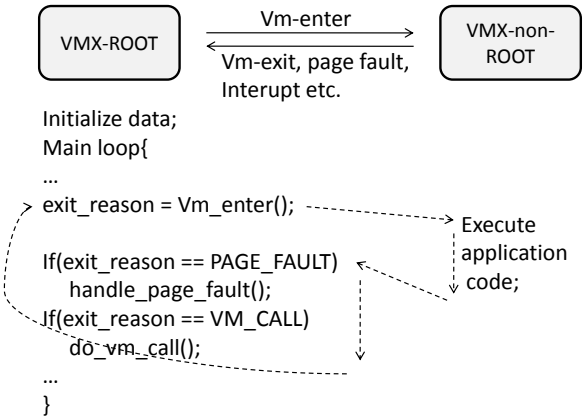

Figure 4. Control Flow of Virtualized Execution.

system call or handling an interrupt, are configured using the VMCS data structure. On a VM-exit, an application returns back to Vmx-root mode in the main loop. In the loop, Dune will handle the cause of the VM-exit (due to system call or interrupt or an EPT violation (Intel 2013)) and then go back to Vmx-non-root mode to continue running user code. All user code is executed in Vmx-non-root mode and, all kernel code is executed in Vmx-root mode. The virtualization mechanism is used to separate privileged execution modes instead of the protection ring 0 for kernel and ring 3 mechanisms typically used for the execution of non-virtualized applications. For system calls from applications, Dune replaces them with vmcalls so applications are actually issuing vmcalls for kernel services (here, a vmcall causes a VM-exit so Dune can handle it properly and securely in the main loop.).

Our implementation, like Dune, permits applications to run at ring 0, so that they can modify the (normal) guest page table (the first level of address translation) directly. At the Vmx-root mode, we additionally implemented features related to managing the extended page table for the enhanced functionality of our system, along with corresponding interface functions (vmcalls) to our upper guest application library. The main vmcalls we implemented are described as follows (their usage will be discussed later in our library implementation):

- (I) *ept_map_to(addr1, addr2, len)*: This vmcall maps two guest physical regions together into the same machine address region.
- (II) *ept_mprotect(addr, len, flag)*: Like mprotect(), this vmcall changes the protection mode of a guest physical region in the extended page table.

Moreover, we provide an EPT violation handler to handle EPT page faults that are triggered by accessing write-protected guest physical pages. In the handler, an array records which page is accessed and then write-protection is disabled for the faulting guest physical page. The array can then be used in our library to perform memory access

analysis and our library can use ept_mprotect() to reenable write-protection of guest physical pages again during the next periodic sampled interval of program execution.

We enable timers to be set for periodic interval sampling for our memory access monitoring by leveraging the support from virtualization hardware to instrument interrupts to Vmx-non-root mode execution (Intel 2013). Specifically, in the main loop (Figure 4) a cycle counter is implemented using the RDTSC time stamp counter instruction. If a set threshold is reached then we instrument a certain interrupt using the VMCS (Belay et al. 2012) and then VM-enter is executed to transition to Vmx-non-root mode. In the Vmx-non-root mode the virtual machine will receive the interrupt and we can handle it in our library. At this point we can trigger appropriate analysis of memory access and implement any dynamic changes to the granularity of memory access monitoring that may be required.

4.2 Runtime Library

Our runtime library mainly manages memory allocation and the normal page table (the first level of address translation shown in Figure 3). As the applications are running at ring 0, it can access the page table directly through the CR3 control register that contains the page directory base.

Generally, a memory allocator requests large blocks of virtual memory, referred to as a *superblock* from an operating system using mmap, and then allocates memory chunks from the superblock to upper guest applications (serving malloc and free). Our implementation is based on the memory allocator Hoard (Berger et al. 2000) and our implementation for these two parts is shown in Figure 5 and described as below where we assume that a virtual page is divided into two parts as described in the design section:

- (I) When our memory allocator asks for superblocks from an OS using mmap, the allocator must request multiples of the desired size based on the granularity. For example, if the granularity is set to the page size divided by 2 then we must request twice the application's requested size. After allocation, we set the normal page table to make the guest virtual address regions have a one-to-one mapping to guest physical regions (as shown in Figure 3(a)). Specifically, here we map the guest virtual address to the same guest physical address. This step will not allocate the real machine page because there is an underlying extended page table that must be set by the kernel module. Then we use the vmcall ept_map_to() to inform our kernel module that the two corresponding guest physical regions should be mapped to the same machine address region (as shown in Figure 3(a)). Finally if any guest virtual address is touched, first it will be translated to the same guest physical address, and then it will trigger an EPT violation. Our kernel module will handle the EPT violation to correctly set the corresponding EPT entry.

```
//this will be called when serving malloc
allocate_superblock(size) {
  …
  //allocate double space
  addr = mmap(0, size*2, …);
  …
  //set the one-to-one mapping in page table
  guest_virtual_to_guest_physical(addr, addr, size*2);
  …
  //map the two guest physical space to the same machine space
  ept_map_to(addr+size, addr, size);
  …
  //write-protect all the guest physical page
  ept_mprotect(addr, size*2, READ);
}

//this is for upper app to do dynamic allocation
malloc(size){
  if(no_space())
    allocate_superblock(SUPER_SIZE);  //if no enough superblock
  …
  addr = internal_heap_malloc(size);  //allocate through Hoard
  if(in_upper_half(addr))  //if the addr is in the upper half of a page
    return addr;
  else if(in_bottom_half(addr))
    return adjust(addr);  //this is to change another place
}
```

Figure 5. Code of Malloc.

- (II) When serving malloc, our memory allocator first behaves as if only one guest virtual block (not 2x) is allocated by the OS each time. Thus it will return an address for the object in the single block. Then we test if the returning address is at the bottom half of the page. If so, we will adjust the returning address to return the corresponding address in the second block (the doubled superblock). This address will finally be translated by the two levels of page table to the same machine page with the first address. In this way we spread objects sparsely in more virtual pages and cause no problems.

Second, as we discussed in Section 3.3, the dynamic strategy is up to upper applications. Here our library offers the basic support to perform sampling and dynamic adjustment of the monitoring granularity. For example, a signal handler can be registered to perform periodic sampling. The signal handler uses one slot in the *interrupt descriptor table* (IDT) (Belay et al. 2012). Time is counted in our kernel module and the interrupt is instrumented to user applications as described in the previous subsection. Every time the kernel module handles an EPT violation caused by writing a write-protected guest physical page, it keeps a record of these pages. We can do analysis and remapping based on this record. The remapping is done by simply modifying the normal page table (the first level of address translation as shown in Figure 3) as introduced in the design section. After sampling has finished, a vmcall (ept_mprotect) is issued to

write-protect these guest physical pages again and the next round of sampling begins.

4.3 Page Size

Currently we are working with 4K pages in both levels of page table for fine-grained monitoring. Future work will investigate the use of superpages.

5. Use Case — Incremental Checkpoint

Our flexible page-level memory access monitoring method can be adopted to improve previous page-based memory access monitoring tools in order to achieve finer-granularity results with lower overhead. For example, our work is likely to be of direct benefit to garbage collection (Boehm et al. 1991), improving strong atomicity of Software Transactional Memory (Abadi et al. 2009), efficient deterministic multithreading (Liu et al. 2011) (Lu et al. 2014), and checkpointing (Kannan et al. 2013). These studies all use page protection to monitor memory access. Here we implement an application-initiated (Kannan et al. 2013) incremental checkpoint application to test our work. Incremental checkpoint is chosen as an interesting test use-case because it has a natural tradeoff between copying and monitoring overheads. For example, if no monitoring is performed, then all data must be copied at checkpoint time (i.e., full-sized checkpoint), and the copying overhead will be very large. Alternatively, if fine-grained monitoring is performed using page fault and byte-by-byte comparison, then the monitoring overhead will dominate and be unacceptably large. Currently, incremental checkpointing is usually performed at page-level granularity that offers the best possible tradeoff.

Our methods can reduce the monitoring granularity, and therefore reduce the volume of checkpointed data to be stored or copied. We implement an incremental checkpoint system based on heap storage in an application-initiated way. To implement incremental checkpoint, we first write-protect all the guest physical pages using ept_mprotect() and register a timer clock handle. The EPT violation handler, triggered by accessing a protected guest physical page, records the information of that page. At checkpoint time, when the periodic timer has expired, we copy all modified data into files and perform a `fsync` to wait for the checkpoint to finish. Then, dynamic adjustment of the monitoring granularity can be performed by adjusting mapping relationships in the page table. Then the next round of the incremental checkpointing begins. The checkpoint interval and the dynamic adjustment strategy of monitoring granularity are discussed in the Experiments section.

Discussion Besides page protection, binary instrumentation (Luk et al. 2005) and compiler instrumentation (Lattner and Adve 2004) are two other common mechanisms to achieve memory access monitoring. We argue that these three common mechanisms suit different situations. Binary instrumentation (Luk et al. 2005) and compiler instrumentation (Lattner and Adve 2004) can achieve byte-level monitoring albeit at considerable overheads if applied program wide. Page-level monitoring has coarser-granularity monitoring and is often used in different applications such as in incremental checkpointing where byte-level monitoring would incur excessive overheads. This paper improves the traditional page-level monitoring mechanism by enabling dynamic adjustment of monitoring granularity at sub-page (object-level granularity). Thus in this paper we only undertake comparisons with the traditional page-level monitoring mechanism. Moreover, dirty bit scanning (Li 2003) or page modification logging (Intel 2015) could be used to improve the performance against our page protection scheme in some cases. However, these schemes merely apply different mechanisms to determine which pages are modified. Thus, the main contribution and novelty of our work (the flexible transparent tool functionality) is not affected as we can easily experiment with other mechanisms in future work.

6. Experiments

In this section we focus on demonstrating the overhead of the virtualized execution and the overhead of memory access monitoring in comparison to traditional page-level techniques. Furthermore, the checkpoint use-case demonstrate that our fine-grained memory access monitoring can effectively reduce the volume of checkpoint data compared to current page-level techniques.

6.1 Methodology

We selected two types of applications as our benchmarks: (1) *object-intensive* benchmark applications that frequently allocate many small objects. We selected four applications from the STAMP benchmark suite (Minh et al. 2008) as shown in Figure 6. They are from different application domains and they allocate many objects; (2) a database system.

According to the design of our mechanism, our tool fits for applications that allocate a lot of small objects. For this reason we choose the benchmarks from the STAMP benchmark suite. Although the applications in STAMP are mainly for testing transactional processing, they can also represent object-intensive applications. Moreover, database systems are another good candidate for our experiments. Most database systems manage a large number of small records and thus our tool will also work well in this scenario. Here we choose the Tokyo-cabinet (Hirabayashi 2010) lightweight database management system that can manage its database in-memory or in file. The main difference between Tokyo-cabinet and other database systems, is that Tokyo-cabinet uses a general memory allocator (malloc) whilst other database systems, such as memcached (Fitzpatrick 2004) manage object allocation in their own customized way. Tokyo-cabinet is directly aligned with the work in this paper as we offer a malloc interface that trans-

Benchmarks	Input	Number of Obj	Allocated Memory (GB)
bayes	-e-1 -i1 -n4 -p10 -q1 -r32768 -s1 -t1 -v32	48441744	1.34
intruder	-a10 -l16 -n1048576 -s1 -t1	14990453	0.62
vacation	-c1 -n10 -q90 -r1048576 -t524288 -u80	11487872	0.43
genome	-g65536 -n2097152 -s128 -t1	16957591	0.78

Figure 6. Object-Intensive Benchmarks.

Benchmarks	Number of Obj	Allocated Memory (GB)
10M @ 64B	10000021	1.28
5M @ 128B	5000021	0.96
1M @ 1024B	1000021	1.09

Figure 7. Tokyo-cabinet Benchmarks. (xM@yB means we insert xM records. Each record is of size yB)

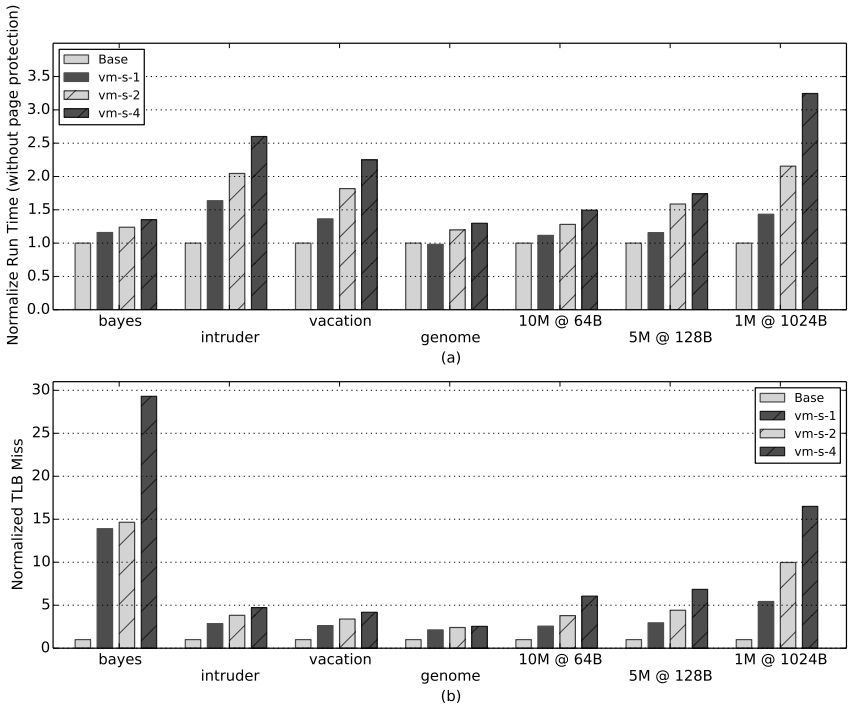

Figure 8. Overhead with virtualization execution. (Base means bare-metal execution without virtualized environment. Vm-s-n means our virtualization execution and we divided a virtual page into n parts without dynamic adjustment (static strategy).)

parently spreads objects across virtual pages. In our experiments, many records of different sizes (as shown in Figure 7) are inserted into the database and the records in memory are managed using a hash table. All the heap data is checkpointed periodically in order to test and compare our work with traditional page-based checkpoint.

First, we run the benchmarks just using virtualization with the two levels of page table. The page-protection is turned off. In this way we can show the overhead of pure virtualized execution. Second, we turn on the page-protection to show the monitoring overhead without copying any data. Finally, we then show the overhead and benefits of our approach to the incremental checkpointing use-case.

The experimental platform is an Intel server equipped with 2.2GHz 12-core CPU and 16GB of physical memory. The operating system is Linux 3.16. In our experiments we set the checkpoint interval to be 5s (unless otherwise specified) in order to demonstrate a relatively intense case for checkpointing.

6.2 Results

Note that from now on the baseline execution time is for a bare-metal process (not running in virtualized environment).

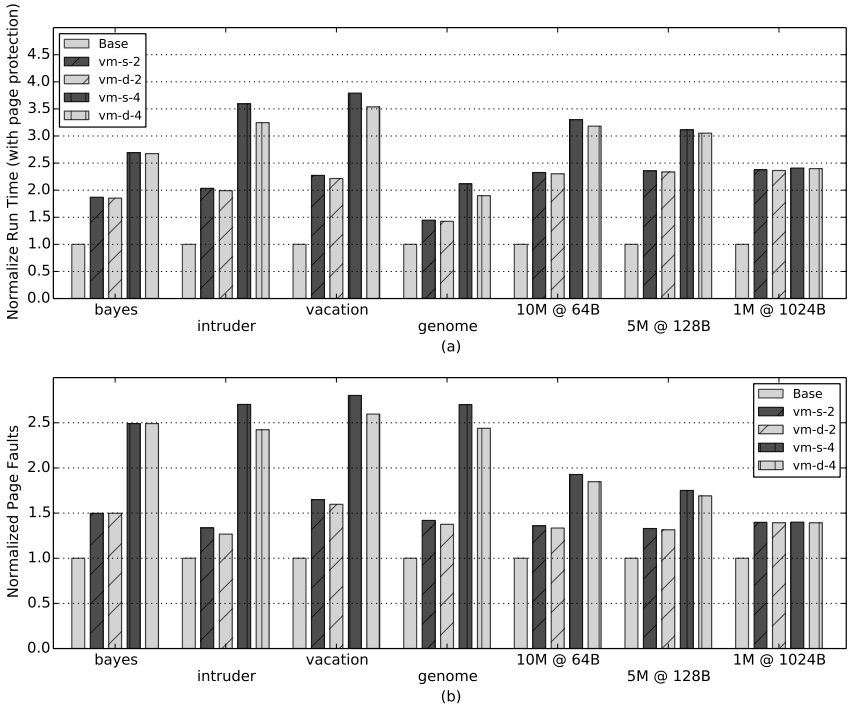

Figure 9. Overhead with page protection. (Base means bare-metal execution with traditional page protection. Vm-d-n means our virtualization execution and we divided a virtual page into n parts with dynamic adjustment. Vm-s-n means our virtualization execution and we divided a virtual page into n parts without dynamic adjustment (static strategy).)

We show the overhead of virtualization execution in Figure 8, which shows the overhead of using two-level-address-translation (more TLB misses) in a virtualized execution environment with page protection turned off. Here the non-baseline benchmarks have two levels of page tables. The baseline is a bare-metal process (not running in virtualized environment). In detail, by comparing vm-s-1 with the baseline, we will get the overhead of just running a program on our virtual machine without any many-to-one-mapping of virtual pages. For all cases shown, our system introduces a maximum of 2x overhead due to increased TLB misses and the overheads of the virtualization execution environment that are primarily due to VM enter and exit. Bayes has an unusual overhead result because the application's own memory access pattern already has very poor cache locality, even without any additional memory access monitoring overheads, thus the relatively large increase in TLB misses for this benchmark under virtualized execution do not dominate performance.

Second, we turn on the page-protection to show the monitoring overhead without copying any data. Figure 9 shows the monitoring overhead with page protection but no copying for checkpoint data. The baseline is bare-metal process (not running in virtualized environment) with traditional page protection. The page fault overhead dominates in this scenario. Here, we can see that the 2 parts design (vm-x-2) is normally faster than the 4 parts design (vm-x-4), because the 2 parts design introduces fewer page faults. Moreover, we can see our dynamic strategy introduces fewer page faults, and is generally faster. Here our dynamic strategy is: for x parts design, we merge possible pages at every checkpoint and entirely separate virtual pages after every x checkpoints. This is because by doing the merge we can effectively reduce the page faults to reduce the execution time and thus do less checkpointing. By following such a strategy we have inherently placed a priority on merging. Note, we do not expect such a strategy to be optimal, moreover it is only intended to illustrate the potential benefits of dynamic granularity adaptation.

Finally, we perform an incremental checkpoint with monitoring and data copying. The overhead of our incremental checkpoint compared with traditional page-level checkpoint is shown in Figure 10. The baseline is a bare-metal process (not running in a virtualized environment) with traditional page protection. Generally the 4 parts design (vm-x-4) reduces the volume of checkpoint data, more than the 2 parts design (vm-x-2), and our dynamic strategy (vm-d-x) reduces the volume of checkpoint data, more than the static strategy (vm-s-x). As discussed before, although the dynamic design may copy more data by merging pages together, it reduces software overhead (as shown in Figure 9) and makes programs faster. Thus it actually reduces the number of checkpoint rounds and reduces the volume of copied data. Generally, our dynamic method can improve performance up

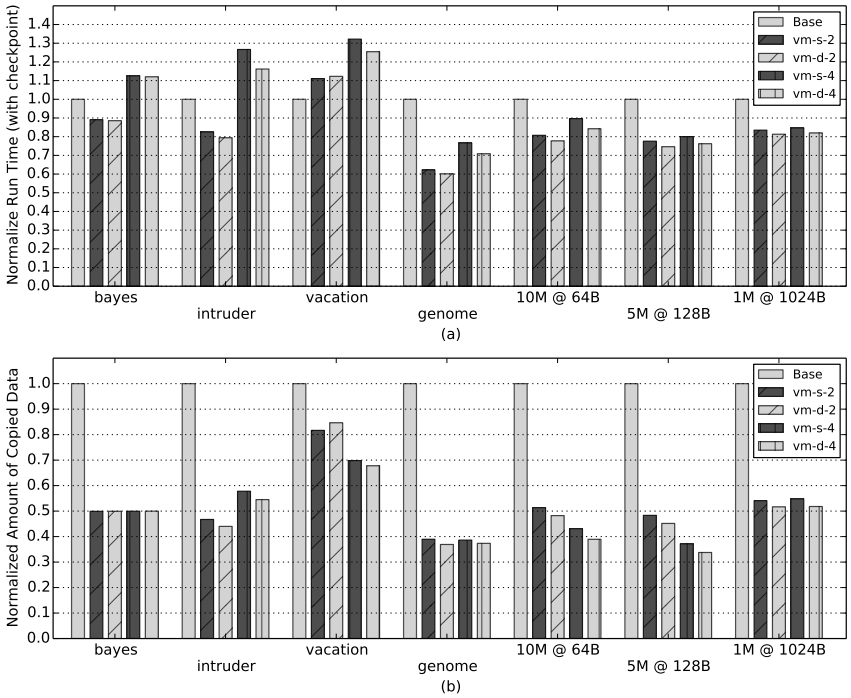

Figure 10. Overhead with checkpoint. (Base means bare-metal execution with traditional page protection.)

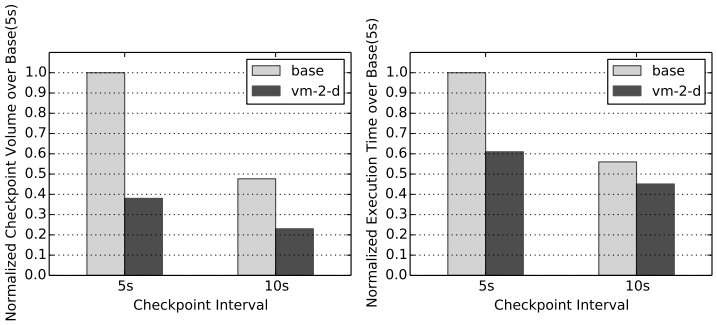

Figure 11. Detailed comparison with baseline on the benchmark genome. (Base means bare-metal execution with traditional page protection)

to 10% over a statically determined access monitoring. For some benchmarks like vacation, the total amount of reduced checkpoint data is not sufficient to give a performance benefit over the runtime overhead we introduce. However, by enlarging the problem scale or shortening the checkpoint interval, we may still get benefit.

When we are varying the checkpoint interval, the results are shown in Figure 11. Generally our fine-grained monitoring mechanism reduced the volume of checkpoint data and obtained better performance. Moreover, as we can see from Figure 11, if we increase the checkpoint interval, the total copied amount decreases, closing the performance gap between our work and the normal page-level checkpoint. Thus we argue that our work is suitable for intense monitoring situations.

Above all, the experiments show our flexible page-level monitoring mechanism has great potential to do monitoring precisely and reduce the checkpoint data. This could improve the performance and the life time of non-volatile storage. Moreover, the experiments also show the flexibility of our system to change the monitoring granularity based on traditional page protection mechanism. Further our system is likely to be able to meet the demands of different applications as it is possible to adjust its monitoring granularity dynamically and transparently.

Discussion The main contribution of this paper is the exploitation of virtualization hardware in order to provide a flexible memory access monitoring mechanism that is transparent to unmodified upper (guest) applications. We have presented the design and implementation of a fundamental

tool that supports transparent and dynamic adjustment of access monitoring granularity. The virtualization overheads, and the potential capabilities of our tool are described. The incremental checkpoint use case is designed to showcase the features and capabilities of our tool. We do not seek to claim that the use-case is state-of-the-art. In this paper, we demonstrate that even a very simple dynamic strategy achieves a modest performance benefit over static schemes. From the results we can see the benefit is not very impressive. However, we argue that our goal is to provide the fundamental function (flexibility for adjustment of monitoring granularity), and outlining specifically the limits of performance benefit is not the core focus of our contribution. Users of our tool can develop various dynamic adjustment strategies tailored to reflect the memory access patterns of their applications. Our work in this paper is to demonstrate we can achieve this because our tool is sufficiently flexible. Therefore, in this paper we focus mainly on describing the tool, and intend to include other more in-depth use-cases and evaluations in future work. Last, the checkpoint phase could be overlapped with the compute phase in order to further optimize performance of the checkpoint tool. We argue that this is totally complementary to our work. The use-cases are designed to clearly showcase the features of our tool, and not to deliver state-of-the art exemplars of use-cases.

7. Related Work

Many-to-one mapping with page-protection is used in some previous work. Dhurjati and Adve (Dhurjati and Adve 2006) deploys it to efficiently detect all dangling pointer uses at run time. Abadi et al (Abadi et al. 2009) adopts it to efficiently achieve strong atomicity in Software Transactional Memory (STM) systems. However, in both these studies, only a single level of address translation is deployed, and they cannot dynamically move objects. Our work has more flexibility compared to previous work. Also we limit the virtual space overhead by dividing a page into several predefined parts to limit TLB misses.

In comparing with previous work (Dhurjati and Adve 2006), our work here differs mainly in two points: (1) the previous work (Dhurjati and Adve 2006) sees virtual space as a limitless resource and gives each object a virtual page. This will lead to an explosion in the usage of virtual space and put huge pressure on the TLB. In our work we limit the usage of virtual space by only dividing a virtual page into several parts. This will achieve much better performance for applications that allocate a lot of objects; (2) we rely on EPT to enable the dynamic and transparent movement of objects. These two points are both core design aspects that differentiate our work.

Modern 64-bit CPUs have an abundant virtual address space. Archipelago (Lvin et al. 2008), DieHard (Berger and Zorn 2006), and DieHarder (Novark and Berger 2010) leverage the large address space to spread objects sparsely in order to achieve fault tolerance. In our work, we use it to achieve fine-grained memory access monitoring by overlapping virtual pages. Also the abundant virtual address space of modern CPUs offers further opportunities for innovation in operating system design implementations (Chase et al. 1994).

Other methods for memory access monitoring include dynamic binary (Probst 2002), and compiler inserted instrumentation (Lattner and Adve 2004). Dynamic binary instrumentation tools, such as Pin (Luk et al. 2005), Valgrind (Nethercote and Seward 2007), and Lightweight Memory Tracing (Payer et al. 2013), introduce overhead when dynamically translating applications' codes. Compiler instrumentation also introduces overhead by the insertion, and association of a function call, with every memory access. We argue that these common mechanisms suit different situations. Binary instrumentation and compiler instrumentation can achieve byte-level monitoring and they are mutually replaceable in many scenarios. Page-level monitoring is more coarse-grained and is often used in different applications, such as making incremental checkpoint. Sampling (Arnold and Ryder 2001), chooses to ignore some accesses in order to get better performance, but this is not comparable with our mechanism and is also not suitable for some use-case scenarios such as for checkpointing.

Dirty bit scanning (Li 2003) or page modification logging (Intel 2015) could be used to improve the performance against our page protection scheme in some cases. However, these schemes merely apply different mechanisms to determine which pages are modified. Thus, the main contribution and novelty of our work (the flexible transparent tool functionality) is not affected as we can easily experiment with other mechanisms in future work. Our work is also appropriate for the wide range of tools that can benefit from efficient memory access monitoring such as data-race detection (Savage et al. 1997), checkpointing (Dong et al. 2011), and deterministic processing (Devietti et al. 2009) (Bergan et al. 2010) (Liu et al. 2011).

Dune (Belay et al. 2012) is a process-level virtual machine that exposes privileged features to applications whilst relying on virtualization hardware to guarantee safety. Kvm (Kivity et al. 2007) is a Linux kernel module that implements a traditional virtual machine monitor. Dune shares some code with kvm in order to manage the details concerning low-level data structures for virtulization.

Hoard (Berger et al. 2000) is an efficient memory allocator for multi-threaded programs. In this paper we modified it to spread objects across virtual pages. Other memory allocators (Ghemawat and Menage 2009) (Shen et al.) could also be adopted to achieve similar functionality.

8. Conclusion

We have introduced a flexible page-level memory monitoring mechanism based on virtualization hardware. We first

introduced a memory allocator to spread objects over virtual pages and that overlaps these pages to achieve fine-grained monitoring. The two-level page tables offered by virtualization hardware were used to achieve dynamic transparent adjustment of the monitoring granularity without copying or moving objects in physical memory. The incremental checkpointing use-case demonstrates our work can effectively reduce the volume of checkpoint data compared with traditional page-based checkpoint and it can improve performance. We have demonstrated the potential of our work to be adopted into different use-case situations with different requirements for monitoring granularities. Future work is required to determine how strategies for dynamic adaptation of memory access monitoring granularity could be optimized by careful merging and splitting of virtual pages in order to reduce monitoring overheads.

Acknowledgments

The authors gratefully acknowledge the helpful suggestions of the reviewers. Thanks for the recommendation to use the new hardware features (page modification logging) to improve this work.

The work is partially supported by the The National Key Research and Development Program of China (No. 2016YFB0200401), by program for New Century Excellent Talents in University, by National Science Foundation (NSF) China 61402492, 61402486, 61379146, by the laboratory pre-research fund (9140C810106150C81001).

Additional support from UK EPSRC grants PAMELA EP/K008730/1, and DOME EP/J016330/1 is acknowledged. Luján is funded by a Royal Society University Research Fellowship.

References

M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *ACM Sigplan Notices*, volume 44, pages 185–196. ACM, 2009.

AMD. Secure virtual machine architecture reference manual. Technical report, Advanced Micro Devices, May 2005.

A. W. Appel and K. Li. Virtual memory primitives for user programs. *Acm Sigplan Notices*, 26(4):96–107, 1991.

M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. *Acm Sigplan Notices*, 36(5):168–179, 2001.

A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *OSDI*, pages 335–348, 2012.

T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 53–64. ACM, 2010.

E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Notices*, volume 41, pages 158–168. ACM, 2006.

E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(4):271–307, 1994.

J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 85–96. ACM, 2009.

D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 269–280. IEEE, 2006.

X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(2):6, 2011.

B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc. *goog-perftools. sourceforge. net/doc/tcmalloc. html*, 2009.

M. Hirabayashi. Tokyo cabinet: a modern implementation of dbm, 2010.

Intel. Intel r 64 and ia-32 architectures software developers manual. *Volume 3b: System Programming Guide (Part 2)*, pages 14–19, 2013.

Intel. Page modification logging for virtual machine monitor white paper. `http://www.intel.com/`, Jan. 2015.

S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing checkpoints using nvm as virtual memory. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 29–40. IEEE, 2013.

A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

K. I. Li. *Virtual Memory Primitives for User Programs*. ACM, 2003.

T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336. ACM, 2011.

K. Lu, X. Zhou, T. Bergan, and X. Wang. Efficient deterministic multithreading without global barriers. In *ACM SIGPLAN Notices*, volume 49, pages 287–300. ACM, 2014.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation.

In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 115–124. ACM, 2008.

C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

N. Nethercote and J. Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

M. Payer, E. Kravina, and T. R. Gross. Lightweight memory tracing. In *USENIX Annual Technical Conference*, pages 115–126, 2013.

M. Probst. Dynamic binary translation. In *UKUUG Linux Developers Conference*, volume 2002. sn, 2002.

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

J. Shen, M. Hamal, and S. Ganzenmüller. Dynamic memory allocation on real-time linux. *Architecture*, 86:32.

R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.