Efficient Memory Shadowing for 64-bit Architectures

Qin Zhao
CSAIL
Massachusetts Institute of Technology
Cambridge, MA, USA
qin_zhao@csail.mit.edu

Derek Bruening
VMware, Inc.
bruening@vmware.com

Saman Amarasinghe
CSAIL
Massachusetts Institute of Technology
Cambridge, MA, USA
saman@csail.mit.edu

Abstract

Shadow memory is used by dynamic program analysis tools to store metadata for tracking properties of application memory. The efficiency of mapping between application memory and shadow memory has substantial impact on the overall performance of such analysis tools. However, traditional memory mapping schemes that work well on 32-bit architectures cannot easily port to 64-bit architectures due to the much larger 64-bit address space.

This paper presents EMS64, an efficient memory shadowing scheme for 64-bit architectures. By taking advantage of application reference locality and unused regions in the 64-bit address space, EMS64 provides a fast and flexible memory mapping scheme without relying on any underlying platform features or requiring any specific shadow memory size. Our experiments show that EMS64 is able to reduce the runtime shadow memory translation overhead to 81% on average, which almost halves the overhead of the fastest 64-bit shadow memory system we are aware of.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Run-time environments, Memory management

General Terms Algorithms, Performance

Keywords Shadow Memory, Dynamic Optimization

1. Introduction

Dynamic program analysis tools often use *shadow memory* to store metadata that tracks properties of application memory. These tools shadow every application data location for a wide variety of purposes, including detecting memory usage errors [21, 24], dynamic information flow tracking [5, 18, 20], detecting race conditions [9, 12, 22, 23], and many others [3, 14, 15, 28].

1.1 Shadow Memory Mapping Schemes

The shadow memory conceptually resides in a different address space from the application address space, and is updated simultaneously as the application executes. In practice, shadow memory shares the same address space as the application memory, and is updated by instrumentation code inserted into the application instruction stream by dynamic program analysis tools.

ISMM'10, June 5–6, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0054-4/10/06...$10.00

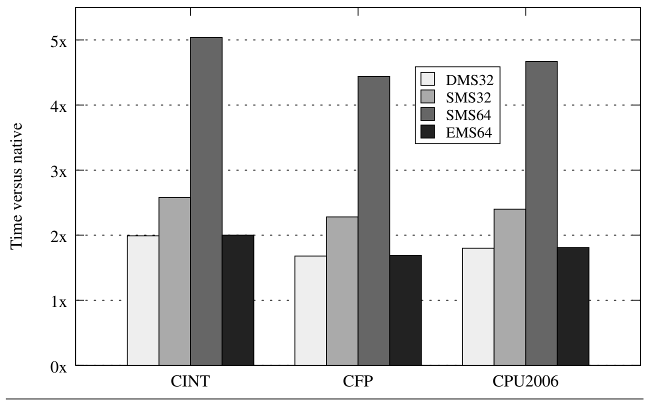

Figure 1. Performance comparison of different memory mapping schemes for a one-shadow-byte-per-application-byte mapping. DMS32 and SMS32 are DMS and SMS on a 32-bit architecture, while SMS64 is SMS using a multi-level table on a 64-bit architecture. Our novel scheme, EMS64, achieves similar performance to DMS32 while supporting the full 64-bit address space.

For 32-bit architectures, there are two shadow memory mapping schemes commonly used to translate an application memory address $addr_A$ to its corresponding shadow memory address $addr_S$: direct mapping and segmented mapping.

A *direct mapping scheme (DMS)* reserves a single contiguous memory region for shadowing the entire address space of the process. The address translation then becomes a simple displacement with a scaling factor that depends on the relative sizes of the spaces. However, the large reserved address space often conflicts with the memory layout requirements imposed by operating systems or applications, which constrains the deployment of DMS.

A *segmented mapping scheme (SMS)* segregates the application address space into segments, and allocates shadow memory for each segment separately when necessary. When translating an address $addr_A$, SMS must first locate which segment $addr_A$ belongs to, typically via a table lookup, and then apply an offset and scale appropriate to that segment. Though it gains the flexibility to resolve address conflicts by allocating shadow memory segments in variable locations, SMS pays for this freedom with the overhead of a segment lookup on every address translation.

Both DMS and SMS have problems when scaling to 64-bit architectures. Current hardware implementations do not use the whole 64-bit virtual address space, but only 48 bits in a canonical form: `[0x0000000000000000, 0x0000800000000000)` and `[0xffff800000000000, 0x10000000000000000)`. In addition, operating systems may impose their own restrictions on memory layout. For example, applications can only allocate memory from certain address ranges, and system modules can only be located in other specific areas. Table 1 lists an example memory layout of a simple HelloWorld program. It shows that the higher canonical half is reserved for kernel modules, and the user program can only allocate memory from the lower canonical half *and* usually only from memory below the stack. The executable is located near the start of the address space while the libraries, stack, and vdso (virtual dynamic shared object for system call entry/exit) are located near the top of the user address space.

Module	Memory Range
User Space	`0000000000000000-0000800000000000`
`a.out`	`0000000000400000-0000000000601000`
`libc`	`00007fa890116000-00007fa890469000`
`ld`	`00007fa890469000-00007fa890686000`
`[stack]`	`00007fffef71e000-00007fffef733000`
`[vdso]`	`00007fffef7ff000-00007fffef800000`
Non-canonical	`0000800000000000-ffff800000000000`
Kernel Space	`ffff800000000000-10000000000000000`
`[vsyscall]`	`ffffffffff600000-ffffffffff601000`

Table 1. Application memory modules for a simple 64-bit application HelloWorld.

It is clear that with such memory layout constraints we cannot find a single contiguous shadow memory region to represent the entire application address space so that a simple equation can be used for address translation. If we are not able to relocate the application memory modules, which is typically the case, DMS cannot be used.

SMS also does not work well on a 64-bit architecture. It is impractical to have a single table to represent the whole address space. SMS typically uses a multi-level table instead, which is similar to what the operating system does for its page table. However, this solution adds significant runtime overhead for retrieving the shadow segment from the multiple levels of table, which makes the already bad SMS performance even worse. MemCheck [16, 24] avoids a multi-level table by using a single table to represent the first 32GB of the address space. If the $addr_A$ is in the first 32GB, the shadow lookup is the same as SMS on a 32-bit architecture. Otherwise, translation falls to a slow path to find the correct segment. In order to improve performance, MemCheck also intercepts the memory allocation system calls and attempts to allocate all application memory from the first 32GB of the address space. Still, MemCheck suffers from a 389% translation-only overhead which excludes the runtime overhead of the underlying code cache system and any shadow metadata update overhead.

In this paper, we present a novel, efficient, and flexible shadow memory mapping scheme called EMS64 (Efficient Memory Shadowing for 64-bit architectures). The key insight of EMS64 is the observation that the 64-bit address space is very large and mostly empty. Thus, we can speculatively use a simple displacement with no table lookup by arranging the shadow memory in such a way that if we use the wrong displacement the mapping will fall into unused space, generating a page fault. In the common case this is as fast as DMS, and clever memory arrangement ensures catching the rare incorrect cases via page fault handling. In addition, to minimize the number of page faults for better performance, we take advantage of application reference locality and dynamic instrumentation: adding displacement lookup code to application references that frequently incur page faults. How to allocate the shadow memory is the key challenge for EMS64. We have solved the allocation problem in a provably correct and efficient way. Figure 1 compares EMS64 with existing mapping schemes. EMS64 achieves similar performance to 32-bit DMS, and is 2.5× faster than a 64-bit multi-level-table SMS implementation, while maintaining the flexibility of SMS.

1.2 Contributions

The following are the key contributions of this paper:

- We propose EMS64, a novel shadow memory mapping scheme for 64-bit architectures that is simple, flexible, and efficient: it out-performs all 64-bit mapping schemes known to us.
- We prove the feasibility conditions for finding a valid EMS64 memory mapping. In a 64-bit architecture, EMS64 can always accommodate at least 256GB application memory in theory, and much more in practice.
- We derive the computational complexity of calculating a valid memory layout for EMS64, which is $O(k^5)$ where k is the number of application address space units.
- We present an efficient randomized algorithm to find suitable shadow memory locations for EMS64 that lowers the running time to $O(k^3)$ in theory, and much faster in practice.
- We implement EMS64 on top of Umbra [29] and show that its average overhead is 81%, which nearly halves the overhead of the fastest 64-bit shadow memory system we are aware of and is much lower than MemCheck's 389% overhead.

1.3 Paper Layout

The rest of the paper is organized as follows: Section 2 describes the EMS64 memory mapping scheme. Section 3 explains the shadow memory allocation model of EMS64, and Section 4 presents optimizations to improve the performance of the shadow memory allocation algorithm. Section 5 shows our implementation of EMS64; Section 6 evaluates EMS64's performance. Section 7 discusses related work and Section 8 concludes the paper.

2. EMS64: Efficient Memory Shadowing

In this section we present EMS64, our efficient memory shadowing scheme. We begin by noting that the address translation process of DMS can be summarized as Equation 1.

$$addr_S = addr_A \times scale + disp \tag{1}$$

We observe that the translation process of SMS involves first finding the segment and then scaling the address and applying the correct displacement value. In other words, the SMS process can also be expressed as Equation 1, with an appropriate $disp$.

This observation allows us to further optimize the SMS translation process, which is the key idea of EMS64: instead of looking up the correct $disp$ value for address translation, we speculatively use a $disp$ without checking its correctness. If the $disp$ is correct, the program continues execution without any problem. If the $disp$ is not correct, we rely on being notified by an implicit check, a memory access violation fault, and can then find the correct $disp$ to use.

Figure 2 shows an example memory layout explaining how EMS64 arranges the shadow memory. The whole address space is split into 16 units of equal size. The last four units, i.e., 12-15, are unavailable to the user as they may be reserved for the kernel. However, since a memory reference to these pages will generate a fault, they can used by EMS64 as reserved units. Units 0, 7, and 11 host application memory modules A_0, A_1, and A_2. There is no single offset that can be used to place shadow memory for all the application units as units 12-15 are unavailable; thus, DMS cannot be used. EMS64 allocates units 2, 6, and 10 as S_0, S_1, and S_2 for hosting shadow memory. There are two valid displacement values: 2 for translating from A_0 to S_0, and -1 for translating from A_1 to S_1 or from A_2 to S_2. There is also a list of units that are reserved by EMS64 and cannot be used for either application memory or shadow memory. By arranging memory units this way, we can

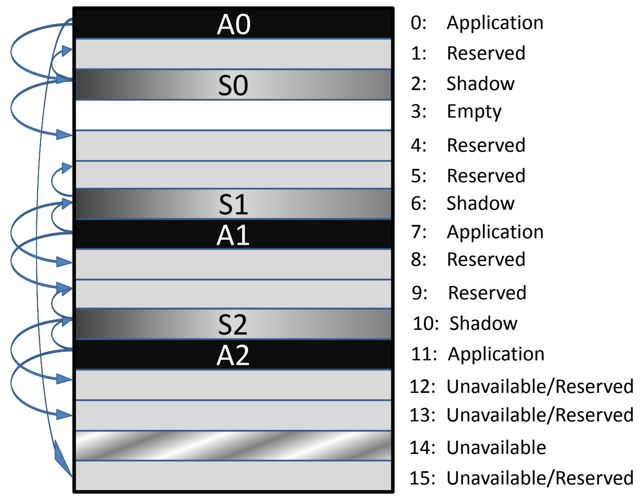

Figure 2. An example EMS64 memory layout for a one-shadow-byte-per-application-byte mapping. For each allocated unit, the translations resulting from applying the two valid displacements of 2 and -1 are shown by arrows. These displacements map each application unit to its corresponding shadow unit. As can be seen, an incorrect displacement from an application unit, or from a wild address in any other unit, always originates from or maps to an invalid unit.

use either 2 or -1 for translating each memory address without checking whether it is the right displacement, because EMS64 guarantees that any address calculated from application memory with an incorrect displacement value will land in a reserved unit. For example, A_0 with displacement value -1 results in unit 15. A_1 and A_2 with displacement value 2 result in unit 9 and 13, respectively. Moreover, even a wild address will cause at least one memory access fault: as we can see, any unit that can be calculated from an S unit with a valid displacement value is reserved by EMS64, including units 1, 4, 5, 8, 9, 12. This prevents EMS64 from suppressing an application page fault.

There are two major challenges to the success of this approach:

- We need a guarantee that if a shadow memory address $addr_S$ is calculated from an $addr_A$ that was not allocated by the application or from an incorrectly speculated $disp$ then a hardware fault will occur.
- Because handling the memory access violation fault is expensive, the frequency of an incorrect $disp$ must be low enough to result in negligible overhead. Since an un-allocated $addr_A$ results in a fault when the application is run natively, its frequency does not concern us.

To solve the first challenge, EMS64 makes use of the largely empty 64-bit address space and carefully allocates shadow memory in such a way that any shadow address $addr_S$ calculated from a non-application-address $addr_A$ or incorrect $disp$ is an invalid memory address. Although the memory allocation is controlled by EMS64, we fully respect the application or system's requirements for memory layout. For example, if the application requests memory from a specific address that conflicts with existing shadow memory, we relocate the shadow memory to accommodate the new application memory. The second challenge is solved by taking advantage of application reference locality and dynamic instrumentation, and will be discussed in Section 5.

EMS64 assumes that metadata does not need to be maintained for memory that is not currently allocated in the address space, other than knowing that the memory is in fact not allocated. This is true for all shadow value tools we are aware of.

To apply EMS64, we virtually split the available address space into fixed-size (e.g., 4GB) address space units. Each unit has four possible states:

A is a unit that hosts application memory only.

S is used for storing shadow memory only.

R is a unit that is reserved either by the operating system or EMS64, and cannot be used for application or shadow memory.

E is an empty unit that is not currently in use.

Table 2 shows an example of how the application and shadow memory modules are clustered into different units in a HelloWorld program for a one-shadow-byte-per-application-byte mapping. There are four application units, and four corresponding shadow units. Two valid $disp$ values 0x1000000000 and -0x500000000000 are used for address translation between units. If the program accesses module a.out or [vsyscall], EMS64 should use 0x1000000000 as the $disp$ to locate the corresponding shadow memory address. If the address $addr_A$ is in module libc, ld, [stack], or [vdso], -0x500000000000 should be used for address translation.

Module	Memory Range
User Space	`0000000000000000-0000800000000000`
A: `a.out`	`0000000000400000-0000000000601000`
	($disp$ = 0x1000000000)
S: `[vsyscall]`	`0000000fff600000-0000000fff601000`
	($disp$ = 0x1000000000)
S: `a.out`	`0000001000400000-0000001000601000`
	($disp$ = -0x500000000000)
S: `libc`	`00002fa890116000-00002fa890469000`
S: `ld`	`00002fa890469000-00002fa890686000`
	($disp$ = -0x500000000000)
S: `[stack]`	`00002fffef71e000-00002fffef733000`
S: `[vdso]`	`00002fffef7ff000-00002fffef800000`
A: `libc`	`00007fa890116000-00007fa890469000`
A: `ld`	`00007fa890469000-00007fa890686000`
A: `[stack]`	`00007fffef71e000-00007fffef733000`
A: `[vdso]`	`00007fffef7ff000-00007fffef800000`
Non-canonical	`0000800000000000-ffff800000000000`
Kernel Space	`ffff800000000000-10000000000000000`
A:`[vsyscall]`	`ffffffffff600000-ffffffffff601000`

Table 2. EMS64 memory layout of shadow memory and application memory for a simple HelloWorld application, with one shadow byte per application byte.

EMS64 maintains a simple translation table that maps each A unit to its corresponding S units, whose number depends on the relative mapping size between application memory and shadow memory. If two A units are adjacent to each other, their corresponding S units must also be allocated contiguously in order to handle an access that spans the two A units. For example, the shadow units corresponding to the application units hosting a.out and [vsyscall] are reserved contiguously. Because of the large unit size (4GB), the total number of used units is small, and thus the cost of querying and maintaining the table is low.

During program execution, EMS64 intercepts every memory allocation system call to monitor its parameters and results. If the system returns memory from existing S or R units, and the application has no specific address requirement, we will request memory again from existing A units or E units instead. If the operating system returns memory from existing A units or from E units that then become A units, we allocate corresponding shadow memory from existing S units or from E units that then become S units based on the $disp$ for those A units. If an application allocation requests a specific address that conflicts with an S or R unit, we can relocate the S and R units and update the translation table accordingly. In addition to the system reserved R units, we also reserve some R units so that any shadow mapping from an incorrect $disp$ will hit an R unit.

EMS64 has the flexibility of a segmented mapping scheme, supporting multiple regions with different $disp$ to accommodate the system and application's memory layout requirements. In addition, it avoids the runtime overhead of $disp$ lookup and achieves similar efficiency to a direct mapping scheme.

In the next section, we prove the possibility of this approach. We then discuss the algorithm design and implementation.

3. Shadow Memory Allocation Model

Although the 64-bit address space is large, EMS64 may fail to find a suitable address space unit to host new shadow memory due to the restrictions on shadow memory placement. In this section, we prove the feasibility of EMS64's mapping scheme and determine limits within which the scheme can successfully operate.

The key to success of the scheme is the displacement $disp$ selection and the corresponding shadow memory allocation. We must answer the question: given an application memory layout, can we find a set of $disp$s to guarantee that any translation using an incorrect $disp$ or an $addr_A$ that was not allocated by the application will result in an invalid address?

To study whether this is possible, we consider the equivalent shadow slot finding problem. The application's memory address space is divided into n equally-sized address space unit slots. Each slot can be labeled as A (application), S (shadow), R (reserved), or E (empty).

Assuming one byte of application memory is mapped to one byte of shadow memory, given an application memory layout of n slots including k A-slots, m R-slots, and x E-slots, can we find k S-slots taken from the E-slots that satisfy the following properties:

1. For each slot A_i, there is one associated slot S_i with displacement d_i where $d_i = S_i - A_i$.
2. For each slot A_i and each existing displacement d_j where $d_i \neq d_j$, slot $((A_i + d_j) \bmod n)$ is an R-slot or an E-slot.
3. For each slot S_i and any existing valid displacement d_j, slot $((S_i + d_j) \bmod n)$ is an R-slot or an E-slot.

A_0	A_1	E_0	E_1	E_2	E_3	E_4	R_0

Figure 3. A sample application memory layout. For this layout we can use E_0 and E_1 as S_0 and S_1 and satisfy properties 1, 2, and 3.

The answer depends on the layout of the slots and on how many E-slots we have. Figure 3 shows a sample layout, for which we can use E_0 and E_1 as S_0 and S_1 and satisfy properties 1, 2, and 3.

A_0	E_0	A_1	E_1	E_2	E_3	E_4	R_0

Figure 4. Another sample application memory layout. There is no assignment of shadow slots that will satisfy properties 2 and 3 for this layout, as shown in Table 3.

However, there exist layouts for which there is no set of S-slots that satisfy properties 2 and 3. Figure 4 shows an example of such a layout. To map the S-slots we check each E-slot in turn to see whether it can be used as an S-slot. Table 3 summarizes the conflicts for each selection. The first column is the E_i slot to be checked, and the second and fourth column show whether E_i can be used as S_j, and what the $disp$ value would have. The third and fifth columns show the reason that E_i cannot be used as S_j. For example, E_0 cannot be S_0 for A_0, because the $disp$ value of 1 causes $E_0 + 1$ to conflict with A_1, which violates the third property. E_2 cannot be S_1 for A_1 because the $disp$ value of 2 results in A_0 being translated into A_1, which violates the second property. From the table we can see that there is no solution for this layout, since only one slot E_1 can be used as an S-slot, even though there are five empty slots.

E-slots	S_0 $(disp)$	Conflict	S_1 $(disp)$	Conflict
E_0	× (1)	$E_0 + 1 => A_1$	× (7)	$E_0 + 7 => A_0$
E_1	√ (3)		√ (1)	
E_2	× (4)	$E_2 + 4 => A_0$	× (2)	$A_0 + 2 => A_1$
E_3	× (5)	$E_3 + 5 => A_1$	× (3)	$E_3 + 3 => A_0$
E_4	× (6)	$A_1 + 6 => A_0$	× (4)	$E_4 + 4 => A_1$

Table 3. Possible shadow slot assignments for the application memory layout of Figure 4, the displacement $disp$ for each, and the conflicts, if any, of each assignment.

In fact, we cannot find a solution even if we use R_0 as an S-slot. If we use E_1 as S_1 ($disp = 1$), and R_0 as S_0 ($disp = 7$), E_1 can translated into A_1 using $disp$ value 7. Similarly, if using E_1 as S_0 and R_0 as S_1, the valid $disp$ values are 3 and 5 and $E_1 + 5$ translates into A_0. Thus, there is no solution for such an application layout even with six E-slots.

We will prove that we can always find k S-slots to satisfy the first two properties, if the number of unassigned slots x is greater than or equal to $2k^2 - k$. Then we prove that $O(k^2)$ is also the bound when satisfying property 3.

Following is the informal proof, which also describes the steps to find a possible S-slot assignment. We start with two empty sets:

- The *difference set* stores the values that cannot be used as valid $disp$ assignments.
- The *translation set* stores the existing valid $disp$ values for S-slots that have already been assigned.

We perform the following steps to identify k S-slots:

1. Given k A-slots, we first calculate all pairwise differences among the A-slots and add each to the difference set. There are at most $(k-1)k$ distinct values added.
2. We pick an A-slot A_i that is not yet associated with any S-slot and try to find an E-slot U whose displacement value $d = U - A_i$ is not in the difference set.
3. We assign slot U as S-slot S_i, and add $d_i = S_i - A_i$ into the translation set.
4. In addition, we calculate the difference v_j between S_i and every A-slot A_j ($v_j = S_i - A_j$) where $i \neq j$, and add each v_j into the difference set. This is to prevent A_j from being translated to S_i using v_j. There are at most $k - 1$ distinct values added.
5. If there are still A-slots that are not associated with an S-slot, we go to step 2. Otherwise, we have found k S-slots satisfying the properties 1 and 2.

Having x E-slots, we can have x different possible displacement values from an A-slot A_i. It is clear that we can always find a slot U for an A-slot A_i at step 2 if the total number of E-slots is greater than the total number of difference values in the difference set. There are at most $(k-1)k + (k-1)k$ values in the difference set, the first $(k-1)k$ values from A-slot pairs at step 1 and the second $(k-1)k$ values from A-slot S-slot pairs at step 4. Thus, we can guarantee finding k S-slots if there are at least $2(k-1)k+k$ E-slots, i.e., if $x \geq 2k^2 - k$.

If an application inadvertently references a wild memory address due to a bug in the application, or deliberately probes unallocated memory, the target address may happen to lie in a shadow memory unit. The third property guarantees that we do not suppress what would be a fault during native execution, by ensuring that a shadow mapping fault will be raised. When a mapping fault occurs, EMS64 checks whether the application address is in a shadow unit; if so, EMS64 sends a fault to the application.

To incorporate the third property, the steps for finding shadow slots are similar to the algorithm above but with additional checks at step 2 as shown below.

1. Given k A-slots, we first calculate all pairwise differences among the A-slots and add each to the difference set. There are at most $(k-1)k$ distinct values added.
2. We pick an A-slot that is not yet associated with an S-slot and try to find an E-slot U whose displacement value is $d = U - A_i$, where
   (a) d is not in the difference set;
   (b) slot $((U+d) \bmod n)$ is neither an A-slot nor an S-slot;
   (c) for each value v in the translation set, slots $((U \pm v) \bmod n)$ are neither A-slots nor S-slots.
3. We assign slot U as S-slot S_i, and add d into the translation set.
4. We calculate and add to the difference set:
   (a) the difference v_j between S_i and every A-slot A_j except value $v_j = S_i - A_i$, which has been stored in the translation set already.
   (b) the difference u_j between S_i and every S-slot S_j except $u_j = S_i - S_i$.
5. If there are still A-slots that are not associated with an S-slot, we go to step 2. Otherwise, we are done.

The key question is whether we can find a slot U in step 2. There are no more than $4k^2 - 3k$ values in the difference set: all $4k^2$ pairs from the $2k$ A-slots and S-slots combined, minus (A_i, A_i), (S_i, S_i), and (A_i, S_i). There are at most $2k$ displacements d that may cause $((U+d) \bmod n)$ to conflict with existing S-slots or A-slots. And there are at most $4k^2$ (k different v with $2k$ different S-slots or A-slots) possible slots U that may cause slots $((U \pm v) \bmod n)$ to conflict with A-slots or S-slots. Thus, if x is greater than or equal to $4k^2 - 3k + 2k + 4k^2 + k$, or $8k^2$, we can always find k E-slots for S-slots in step 2.

For a 64-bit architecture, if using 4GB as the slot size, there are 2^{15} slots in the 47-bit user-allocatable address space. We can guarantee to support around 64 slots (256GB) regardless of the A-slot layout.

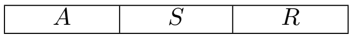

Figure 5. An optimal memory layout in EMS64. For this layout application, shadow, and reserved memory each occupy 1/3 of the overall address space.

The $8k^2$ result is a very loose bound. The actual space depends on the A-slot distribution and how many different displacement values are used. In an optimal case, application memory can use up to 1/3 of the overall available address space, as shown in Figure 5. Assuming that y different displacement values are used, each A-slot will reserve y slots as either S-slot or R-slots, and each S-slot will reserve y R-slots, resulting in $((2y+1) \times k)$ total slots used. This is a much lower value than the $8k^2$ bound. This is only an estimate as different A-slots or S-slots may reserve the same R-slot, and some slots are reserved for other reasons. In practice, we can always find a feasible set of S-slots with a small number of displacement values for a given layout with much fewer E-slots available than the upper bound (e.g., Figure 3).

The proof steps above also describe an algorithm to find suitable shadow address space units. To find an S-slot for a given A-slot, we may have to check $O(k^2)$ E-slots before finding a valid S-slot location. The process of checking each E-slot incurs $O(k^2)$ operations, including:

- Step 2(a): check if the new d is in the difference set, which is an $O(k^2)$ operation.
- Step 2(b): check if $((U+d) \bmod n)$ conflicts with any existing A-slot or S-slot, which is an $O(k)$ operation.
- Step 2(c): check for all v in the translation set whether slots $((U \pm v) \bmod n)$ conflict with any existing A-slot or S-slot, which is an $O(k^2)$ operation.

Since there are k A-slots, the overall runtime is $O(k^5)$ $(= k \times O(k^2) \times O(k^2))$.

The proof and algorithm for application memory mapping to different sized shadow memory is similar, and the bound is also $O(k^2)$ with a larger leading constant.

4. Efficient Shadow Slot Finding Algorithm

There are several ways to improve on the algorithm from Section 3. We focus on reducing the overhead at step 2.

First, instead of linearly scanning the available E-slots to check whether each can be used as an S-slot, we randomly select a slot and check whether it is an E-slot and can be used as an S-slot. Because of the large number of available E-slots (2^{15}) compared to the number of unsuitable slots $O(k^2)$, we expect to find a suitable one in a small number of attempts. The expected number of E-slots we must check is $O(1)$.

Second, we use an $O(k^2)$-size hashtable to store the $O(k^2)$ values in the difference set, and use a linked list to resolve hash collisions. The expected running time for step 2(a) of checking whether a value is in the difference set is reduced to $O(1)$. Similarly, we use hashtables of size $O(k^2)$ to store the A-slots, S-slots, and the R-slots that are reserved by EMS64. We use the base address of each slot as the key. The running time of checking whether a slot conflicts with any existing slots is reduced to $O(1)$. Thus, the running time of step 2(b) is reduced to $O(1)$. The running time of step 2(c) is reduced to $O(k)$ for checking the $O(k)$ valid values in the translation set.

The third optimization is to reuse existing valid displacements $disp$ in the translation set whenever possible. As shown in Table 2, there are two $disp$s for 6 modules, or 4 different application address space units. Ideally, if we can find one $disp$ for every possible application unit, every speculative $disp$ is correct, and the overall performance would be the same as the direct mapping scheme. Even if we cannot find a single one to fit all units, the fewer $disp$, the better. If a memory reference dynamically varies and accesses two different units that have the same $disp$, it can reference the correct shadow memory without raising any fault. In addition, a smaller number of $disp$s in the translation set also reduces the possibility of conflicts at step 2(c), which makes it easier to accommodate more S-slots. Before looking for a suitable unit S for application unit A, we first try all existing $disp$ values and check whether the unit calculated from A and $disp$ could be a valid S-slot. If so, we use the existing $disp$ value. Since the running time to check whether slot $((A_i + disp) \bmod n)$ is valid is $O(k)$, the running time to check

```
Shadow-Slot-Search($A_i$)
$A$ : $A$-slot set
$S$ : $S$-slot set
$R$ : $R$-slot set
$D$ : difference set
$T$ : translation set
 1. For each value $d \in T$, $U = ((A_i + d) \bmod n)$
    (a) if $U \in (A \cup S \cup R)$, try next $d$ at step 1.
    (b) if $((U + d) \bmod n) \in (A \cup S)$, try next $d$ at step 1.
    (c) if $((U \pm v) \bmod n) \in (A \cup S)$ for any $v \in T$ and $v \neq d$,
        try next $d$ at step 1.
    (d) add $((U \pm v) \bmod n)$ except $A_i = ((U - d) \bmod n)$ into
        $R$, for all $v \in T$.
    (e) add value $U - A_j$, $A_j - U$, $U - S_j$, and $S_j - U$ except
        $d = U - A_i$ into $D$ for all $A_j \in A$, $S_j \in S$.
    (f) assign $U$ as $S_i$ and return.
 2. Randomly select a slot $U$, $d = U - A_i$.
    (a) if $d \in D$, jump to step 2.
    (b) if $U \in (A \cup S \cup R)$, jump to step 2.
    (c) if $((U + d) \bmod n) \in (A \cup S)$, jump to step 2.
    (d) if $((U \pm v) \bmod n) \in (A \cup S)$ for any $v \in T$, jump to step
        2.
    (e) add $d$ into $T$.
    (f) add $((U \pm v) \bmod n)$ except $A_i = ((U - d) \bmod n)$ into
        $R$, for all $v \in T$.
    (g) add value $U - A_j$, $A_j - U$, $U - S_j$, and $S_j - U$ except
        $d = U - A_i$ into $D$ for all $A_j \in A$, $S_j \in S$.
    (h) assign $U$ as $S_i$ and return.
```

Figure 6. Shadow slot finding algorithm for each A-slot A_i.

every valid $disp$ is $O(k^2)$. In practice, however, there are only 2 to 3 displacement values in the translation set.

Given an A-slot, our final algorithm to locate a valid S-slot is shown in Figure 6, whose running time is $O(k^2) + (O(1) \times O(k)) = O(k^2)$. Thus, the overall running time to find k S-slots is $O(k^3)$. If a scale is needed when mapping application memory due to differently-sized shadow memory, the algorithm is the same except $((U + d) \bmod n)$ is replaced with $((U \times scale + d) \bmod n)$ for the slot calculation.

5. Implementation

To demonstrate the feasibility of EMS64, we have built a prototype using Umbra [29].

5.1 Umbra

Umbra is an efficient and scalable memory shadowing tool built on top of DynamoRIO [1, 2], which is a state-of-the-art runtime code manipulation system. Using the APIs provided by DynamoRIO, Umbra inserts code into the application's runtime instruction stream to perform memory address translation from application memory to shadow memory. Umbra also provides an interface allowing users to write an Umbra client to insert code for updating shadow memory metadata without needing to know the memory translation details.

Umbra uses a segmented mapping scheme. Instead of representing the whole address space, Umbra uses a small mapping table to store information only about each allocated application memory module and its corresponding shadow memory module. This allows Umbra to easily scale to the full 64-bit address space. However, Umbra must walk the whole table to find the correct segment. Umbra uses several optimizations to avoid this table walk. One of the key optimizations is the *reference cache*. A reference cache is a software data structure that stores the last translated application address unit tag and displacement (offset) of the corresponding shadow address:

```
struct ref_cache {
  void *tag;
  ptrdiff_t offset;
}
```

Umbra associates each static application memory reference with a reference cache, and takes advantage of application locality to avoid table walking. If the instruction accesses the same application unit again, Umbra will use the cached displacement for the translation directly. If it is different, Umbra will execute the slow path to find the right unit and displacement and update the reference cache. Experiments show that the hit ratio of the reference cache is 99.97% on average, with most address translation taking the fast path and avoiding a table walk.

The high hit rate (99.97%) of the reference cache in Umbra suggests that most reference cache checks are redundant. However, Umbra cannot hardcode the offset as in DMS, and has to check to make sure that it is correct. This is because Umbra must handle the 0.03% cache misses and provide correct translation for the case that an application instruction references a different module from its last execution.

Figure 7 lists the instrumented instructions for the reference cache check and address translation on the fast path for the case of a reference cache hit. Even with a high cache hit rate, Umbra is still unable to match the performance of 32-bit DMS. Umbra achieves a 2.5× slowdown compared to native execution, which is mainly attributable to redundant checks and code expansion.

```
     lea [addr]            => %r1
     and %r1, TAG_MASK     => %r1
     cmp %r1, rc->tag
     je  label_hit
     ...
label_hit:
     lea [addr]            => %r1
     add %r1, rc->offset   => %r1
```

Figure 7. Umbra's instrumented code for the reference cache check and address translation on the fast path for a reference cache hit. The code for context switching is omitted.

5.2 EMS64

Implementing EMS64 involved three major modifications to Umbra:

- adding our shadow slot finding algorithm to Umbra's shadow memory manager,
- adding signal handling to handle page faults,
- changing Umbra's instrumenter to selectively insert reference check code for application references that frequently cause page faults.

We implemented Section 4's shadow slot finding algorithm in Umbra's shadow memory manager. We also changed Umbra's instrumentation. The reference cache of Umbra has a natural role under EMS64. The most straightforward implementation is to simply assign a valid $disp$ value to the reference cache's `offset` field on initialization, and remove all reference cache check code inserted by Umbra. The instrumented code is shown in Figure 8, which is the same code as the address translation on a reference cache hit in Umbra. If an $addr_S$ is calculated from an unmatched $addr_A$ and

```
lea [addr]           => %r1
add %r1, rc->offset => %r1
```

Figure 8. EMS64 instrumented code for address translation. The code for context switching is omitted.

$disp$, the resulting memory reference for accessing shadow memory will raise a fault. Our fault handler will catch the fault, find the correct $disp$ value, re-calculate the $addr_S$, update the reference cache, and re-execute the instruction that raised the fault.

In most cases, a single memory reference instruction only ever references one address space unit. However, there are some exceptions. For example, the C library function `memcpy` is often called by applications to perform a memory copy from source to destination. The memory reference in such a function may access different address space units in different invocations, and thus often have an incorrect $disp$. For such instructions, we should use an inline check to find $disp$ instead of paying the cost of frequent notification by a fault. The question is how to detect such memory references.

DynamoRIO's basic block cache and trace cache duality provides us a ready answer. DynamoRIO first copies application code into its basic block cache and executes from there. It upgrades frequently executed sequences of basic blocks, or *traces*, into its trace cache. We add a counter for each memory reference to record the number of reference cache misses. For basic blocks emitted into the basic block cache, we insert the reference cache checks just as Umbra does. In addition, we insert code to update the reference cache miss counter on the slow path when the inline reference cache check misses. When hot code is to promoted into the trace cache, we check the reference cache miss counter, and remove the inline check code if the counter value is below a threshold (2 in our implementation).

It is still possible that a memory reference in a trace accesses different address space units. Our fault handler addresses that problem. In addition to finding $disp$ and recalculating the $addr_S$, the fault handler marks the trace to be removed from the trace cache after the current execution instance. If the code is executed again later, it will be emitted into the basic block code cache with reference cache check and profiling. Thus, memory references are re-profiled when their behavior changes.

In this way, we can identify memory references that often change their access targets and add or remove the reference cache checks adaptively. Because basic blocks only host code that is infrequently executed, the profiling overhead in the basic block cache incurs negligible overhead.

6. Experimental Results

We conducted several experiments to evaluate the performance of EMS64.

6.1 Experimental Setup

We have implemented EMS64 on top of Umbra for Linux. We used the SPEC CPU2006 benchmark suite [25] [1] with reference input sets for evaluating EMS64. All benchmarks were compiled as 64-bit using gcc 4.3.2 with -O2 optimization. The hardware platform is a dual-die quad-core Intel Xeon processor with 3.16GHz clock rate, 12MB L2 cache on each die, and 8GB total RAM. The operating system is 64-bit Debian GNU/Linux 5.0. We configured 4GB as the address space unit size.

[1] 400.perlbmk, 464.h264ref, and 481.wrf are excluded because they fail to run correctly natively.

6.2 Performance Evaluation

We first evaluate the runtime overhead of Umbra and our EMS64 translation scheme with and without a shadow memory update. Figure 9 shows the performance results. The Umbra-null and EMS64-null datasets show the performance of translation without any shadow memory update. Umbra-AD and EMS64-AD add an access detection tool that uses shadow memory to detect whether the corresponding application data has been accessed by the application after allocation. This involves additional instrumentation to update shadow memory after translation.

With translation only, Umbra incurs a 149% overhead while EMS64 only has an 81% overhead, which is comparable to the 80% runtime overhead of the direct mapping scheme (DMS) in Figure 1. This shows that EMS64's profiling overhead from reference cache checks in the basic block cache has negligible performance impact. Note that in this comparison the performance of EMS64 is the idealized performance. Because it performs translation without touching the shadow memory, it will not incur any faults even if a wrong $disp$ is used.

The results for Umbra-AD and EMS64-AD show the performance overhead of a sample shadow memory tool for memory access detection. Umbra-AD's runtime overhead is 626%, while that for EMS64-AD is only 448%. The difference of 178% is significantly larger than the 68% difference between Umbra-null and EMS64-null. Although the shadow memory update code is identical for the two tools, EMS64-AD has less code for address translation, which leads to a much smaller code cache size. The code cache size difference has more impact when shadow memory update code is included, where the cache pressure is higher.

6.3 Statistics Collection

To better understand the impact of EMS64, we also collect statistics on Umbra and EMS64. In Table 4, the *check-rate* statistic shows the ratio of the number of reference cache checks versus the total number of memory references executed. Umbra uses static analysis to cluster memory references to avoid redundant reference cache checks, such as when two memory references access different members of the same data object. In contrast, EMS64-AD performs significantly fewer reference cache checks, only $1/20$ of the number of checks for Umbra-AD. The *hit-rate* is the reference cache hit ratio, which is decreased from 99.97% in Umbra-AD to 97.83% in EMS64-AD, due to the removal of a large number of redundant reference checks that always hit in Umbra-AD.

	Umbra-AD			**EMS64-AD**		
Metric	**CINT**	**CFP**	**All**	**CINT**	**CFP**	**All**
check-rate	62.29%	50.73%	57.87%	2.09%	3.42%	2.91%
hit-rate	99.98%	99.96%	99.97%	98.81%	97.21%	97.83%

Table 4. Statistics on the effectiveness of Umbra and EMS64. The *check-rate* data shows the ratio of the number of reference cache checks versus the total number of memory references executed. The *hit-rate* is the reference cache hit ratio.

Table 5 lists the number of reference cache checks for each benchmark for both Umbra-AD and EMS64-AD. The *BB* column shows the number of checks executed in the basic block cache, while *Trace* shows the number of checks executed in the trace cache. The *FAULT* column displays the total number of access violation faults raised. The table shows that the number of checks executed in traces with EMS64-AD is much smaller than the number of checks with Umbra-AD. On average, the number of checks in the trace cache for EMS64-AD is only 7% of the number of checks in the trace cache for Umbra-AD. This means that EMS64-AD has removed a significant number of redundant checks. In contrast,

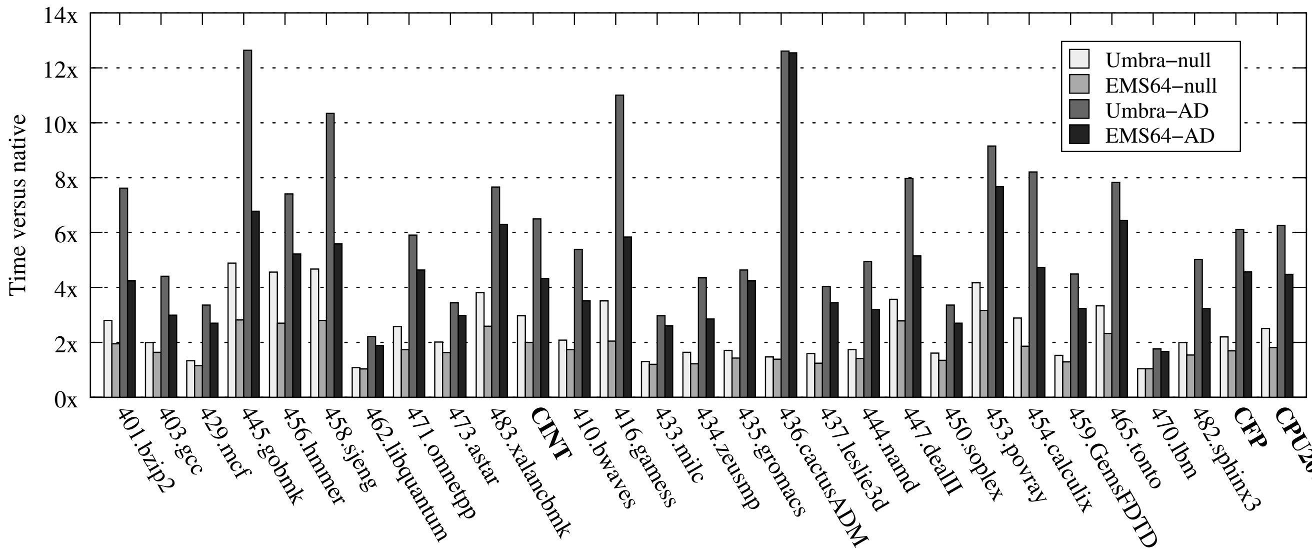

Figure 9. The performance of EMS64 compared to Umbra on the SPEC CPU2006 benchmarks, without (Umbra-null and EMS64-null) and with (Umbra-AD and EMS64-AD) shadow memory updates. The shadow memory updates implement an access detection tool that detects whether application data has been accessed after allocation. Both Umbra and EMS64 are configured to use 1 shadow byte per application byte.

Umbra-AD has slightly fewer checks executed in basic blocks. This is because some memory references in EMS64-AD will fault and subsequently be removed from the trace cache and profiled again in the basic block cache.

The number of checks reduced depends on the characteristics of the application. The better reference locality the application has, the fewer number of checks are kept in traces and the fewer faults are raised, resulting in better performance. For example, the benchmark 470.lbm has excellent reference cache locality. There is only one fault raised, and the number of checks in traces is reduced from 6.12×10^{10} to merely 6.72×10^{3}. It only incurs a 66.7% runtime overhead under EMS64-AD. Benchmarks 429.mcf, 444.namd, 459.GemsFDTD, 462.libquantum have similar properties. A particularly interesting result is that 429.mcf also shows good locality. 429.mcf is a benchmark that causes a significant amount of hardware cache misses, but in the coarse-grained address space unit level, most of its accesses are to the same address space unit. In contrast, benchmarks like 465.tonto do not have a good unit-level locality: it contains more than 600 references that cause access violations and result in a 6.44× slowdown. Similar benchmarks include 436.cactusADM and 447.dealII. There are a few anomalies. For example, 403.gcc causes many faults (360) but has only moderate slowdown (2.99×). One cause may be gcc's time spent on disk i/o overshadowing this overhead. 458.sjeng has only 7 faults, and EMS64 eliminates 99% of its checks, but it still has a 5.58× slowdown. This might be because the code cache overhead due to instrumented code dominates the overall runtime overhead. For 458.sjeng, Umbra-AD has 10.34× slowdown, which implies that the code cache size has a significant impact on this benchmark.

	Umbra-AD		EMS64-AD		
Benchmark	**BB**	**Trace**	**BB**	**Trace**	**Faults**
401.bzip2	$7.12e5$	$6.99e11$	$8.77e5$	$1.79e10$	53
403.gcc	$2.50e7$	$2.56e11$	$4.05e7$	$2.42e10$	360
429.mcf	$5.50e4$	$7.89e10$	$5.51e4$	$1.26e06$	13
445.gobmk	$5.05e6$	$4.09e11$	$5.25e6$	$2.79e09$	134
456.hmmer	$2.38e5$	$1.34e12$	$2.94e5$	$3.34e08$	38
458.sjeng	$1.57e5$	$4.72e11$	$1.59e5$	$2.37e09$	7
462.libquantum	$3.37e4$	$2.43e11$	$4.20e4$	$1.28e08$	17
471.omnetpp	$5.73e5$	$1.74e11$	$6.51e5$	$2.56e10$	134
473.astar	$1.88e5$	$3.39e11$	$2.74e5$	$8.64e09$	109
483.xalancbmk	$1.02e6$	$2.99e11$	$1.14e6$	$6.59e09$	202
410.bwaves	$1.66e5$	$1.14e12$	$1.82e5$	$2.62e11$	93
416.gamess	$2.53e6$	$1.52e12$	$2.66e6$	$2.01e09$	57
433.milc	$1.50e5$	$2.04e11$	$2.12e5$	$6.42e10$	86
434.zeusmp	$4.81e5$	$3.86e11$	$4.98e5$	$2.77e06$	13
435.gromacs	$2.48e5$	$2.20e11$	$3.01e5$	$7.25e09$	56
436.cactusADM	$2.51e5$	$1.98e11$	$2.95e5$	$1.19e11$	187
437.leslie3d	$2.08e5$	$6.32e11$	$2.35e5$	$2.30e10$	13
444.namd	$2.43e5$	$2.56e11$	$2.49e5$	$4.89e07$	10
447.dealII	$1.01e6$	$5.13e11$	$1.22e6$	$5.42e10$	256
450.soplex	$6.73e5$	$2.12e11$	$7.85e5$	$1.04e09$	88
453.povray	$4.25e5$	$1.91e11$	$5.05e5$	$7.24e10$	102
454.calculix	$8.05e5$	$1.77e12$	$9.20e5$	$1.92e10$	213
459.GemsFDTD	$4.70e5$	$4.85e11$	$4.83e5$	$8.30e08$	37
465.tonto	$1.36e6$	$5.63e11$	$1.87e6$	$6.34e10$	619
470.lbm	$3.64e4$	$6.12e10$	$3.64e4$	$6.72e03$	1
482.sphinx3	$2.84e5$	$8.58e11$	$3.33e5$	$7.07e08$	119

Table 5. Number of reference cache checks performed in the basic block cache (BB) and trace cache (Trace) by Umbra-AD and EMS64-AD, and the number of faults raised in EMS64-AD.

7. Related Work

There are two common types of shadow memory mapping for 32-bit architectures used by dynamic program analysis tools: direct mapping schemes (DMS) and segmented mapping schemes (SMS). A direct mapping scheme maps the full user address space into a single shadow address space. This simplifies translation, requiring only an offset and potentially a scale if the shadow memory size does not match its corresponding application size. However, using a single shadow region sacrifices robustness, as it requires stealing a large chunk of space from the application.

LIFT [20] uses a direct mapping scheme to shadow each application byte with only one shadow bit. Consequently its mapping uses both a scale and an offset, and its shadow region only requires one-eighth of the user address space.

TaintTrace [5], Hobbes [3], and Eraser [23] all use direct mapping schemes as well, but with one shadow byte per application byte. They assume a 3GB 32-bit user address space and take 1.5GB for shadow memory. Their shadow memory mapping involves a simple offset and incurs little overhead. However, claiming a full half of the address space gives up flexibility and presents problems supporting applications that make assumptions about their address space layout. Such a design is problematic on operating systems that force various structures to live in certain parts of the address space or use different address space splits for kernel versus user space.

A segmented mapping scheme splits the user address space into segments, and allocates corresponding shadow memory only when necessary. The segmented mapping scheme has the flexibility to avoid address conflicts, and allows shadow memory to be larger than application memory. However, it sacrifices efficiency for this flexibility. A segmented mapping scheme must first locate which memory segment an application address lies in, and then apply the correct address translation. To achieve better performance, a tool using a segmented mapping scheme often uses a page-table-like data structure to maintain the segment information for the full address space, and uses the most significant bits of each memory address as an index to quickly retrieve the segment information from the table. A number of shadow value tools use segmented mapping schemes for flexibility and robustness. Using segment mapping schemes gives up some performance but provides support for a wider range of applications and platforms.

MemCheck [24] employs a segmented mapping scheme [16]. MemCheck's scheme was designed for a 32-bit address space. It splits the space into 64K regions of 64KB each. A first-level table points at the shadow memory for the 64KB region containing the address in question. MemCheck originally kept all of its shadow memory in a single contiguous region but was forced to split it up in order to support a wider range of applications and platforms, due to the limitations discussed earlier with claiming too large of a contiguous fraction of the application address space.

MemCheck extends its scheme to 64-bit address spaces with a larger first-level table that supports the bottom 32GB of the address space. It uses a slower translation path for addresses above 32GB, and attempts to keep as much memory as possible in the lower 32GB. The MemCheck authors report problems with their approach on other platforms and suggest it may need improvement [16]: "It is unclear how this shadow memory scheme can best be scaled to 64-bit address spaces, so this remains an open research question for the future."

MemCheck uses several optimizations to reduce overhead, but most of them are specific to MemCheck's particular metadata semantics. It saves memory and time by pointing shadow memory regions that are filled with a single metadata value to a shared shadow memory structure. For aligned memory accesses it processes all bytes in a word simultaneously. And it maintains bit-level shadowing granularity without requiring shadow bits for every application bit by compressing the shadow metadata to only use such granularity when byte-level granularity is not sufficient.

The TaintCheck [18], Helgrind [12], and Redux [15] tools are all built on the same Valgrind [17] dynamic binary instrumentation platform as MemCheck. They all use the same segmented mapping scheme as MemCheck.

pinSel [14] uses a segmented mapping scheme similar to MemCheck's, but with 4KB shadow units rather than 64KB units. VisualThreads [9] uses 16MB units in its two-level approach.

EDDI [28] shadows each memory page with a shadow page that stores for each application byte whether a data watchpoint has been set. It uses a similar approach to pinSel. A table is used to locate the shadow page for each memory page. Unlike most segmented mapping schemes, EDDI stores the displacement instead of a pointer in the table for translation, which enables several optimizations for better performance.

DRD [22] uses a nine-level table to hold its shadow memory, which shadows memory accessed during each unit of time.

Commercial shadow value tools include Purify [21], Intel Parallel Inspector [11], Insure++ [19], and Third Degree [10]. Unfortunately, their shadow translation details are not published.

MemTracker [27] and HARD [30] propose using additional hardware to provide low-overhead shadow memory translation and memory access monitoring (but not propagation) for MemTracker, and data race detection for HARD. The introduced hardware is targeted to a specific tool in each case.

Metadata management and propagation directly in hardware [7, 8, 26] imposes limitations on the metadata format stored in shadow memory but can reduce overheads significantly for tools that can use the supported formats. Other hardware proposals support a wider range of dynamic analysis tools using shadow memory by targeting the costs of dynamic binary instrumentation [6, 31] or providing metadata support independently of the metadata structure [4].

Although it uses a segmented mapping scheme, unlike other tools that map the entire address space uniformly Umbra [29] maps regions based on application memory allocation. This key difference allows Umbra to scale up to 64-bit architectures, but also forces Umbra to walk the entire table to locate a segment. Umbra uses several optimizations to avoid the slow table walk and provide efficient mapping.

EMS64 improves on Umbra. It takes advantage of the large 64-bit address space and application reference locality to provide a fast and flexible mapping scheme for 64-bit architectures based on insights from the similarities between the DMS and SMS address translation process.

EMS64 is implemented entirely in software on top of Umbra using the DynamoRIO [2] dynamic binary instrumentation system. It could be implemented using other binary instrumentation systems such as Pin [13] or Valgrind [17].

8. Conclusion

In this paper we present EMS64, an efficient memory shadowing scheme for 64-bit architectures. It achieves comparable efficiency to a direct mapping scheme while providing the flexibility of a segmented mapping scheme. EMS64 does not rely on idiosyncrasies of the operating system or underlying architecture and is not limited to specific shadow metadata sizes or semantics.

We proved the feasibility of EMS64 and proposed an efficient algorithm to find suitable shadow memory locations. We implemented EMS64 on top of Umbra and evaluated its performance. EMS64 shows substantial performance improvements over Umbra and over all existing 64-bit shadow mapping schemes known to us. We hope that the EMS64 approach can be applied to other mapping problems beyond mapping application memory to shadow memory.

References

[1] DynamoRIO dynamic instrumentation tool platform, February 2009. `http://dynamorio.org/`.

[2] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T., September 2004.

[3] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. Run-time type checking for binary programs. In *Proc. of the*

12th International Conference on Compiler Construction (CC '03), pages 90–105, 2003.

[4] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proc. of the 35th International Symposium on Computer Architecture (ISCA '08)*, pages 377–388, 2008.

[5] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proc. of the Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC '06)*, pages 749–754, 2006.

[6] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Dise: a programmable macro engine for customizing applications. In *Proc. of the 30th International Symposium on Computer Architecture (ISCA '03)*, pages 362–373, 2003.

[7] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. of the 37th International Symposium on Microarchitecture (MICRO 37)*, pages 221–232, 2004.

[8] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proc. of the 34th International Symposium on Computer architecture (ISCA '07)*, pages 482–493, 2007.

[9] Jerry J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proc. of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 331–342, 2000.

[10] Hewlett-Packard. Third Degree. `http://h30097.www3.hp.com/developerstoolkit/tools.html`.

[11] Intel. Intel Parallel Inspector. `http://software.intel.com/en-us/intel-parallel-inspector/`.

[12] OpenWorks LLP. Helgrind: A data race detector, 2007. `http://valgrind.org/docs/manual/hg-manual.html/`.

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200, June 2005.

[14] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proc. of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06/Performance '06)*, pages 216–227, 2006.

[15] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

[16] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proc. of the 3rd International Conference on Virtual Execution Environments (VEE '07)*, pages 65–74, June 2007.

[17] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[18] James Newsome. dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[19] Parasoft. Insure++. `http://www.parasoft.com/jsp/products/insure.jsp?itemId=63`.

[20] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. of the 39th International Symposium on Microarchitecture (MICRO 39)*, pages 135–148, 2006.

[21] Rational Software. Purify: Fast detection of memory leaks and access errors, 2000. `http://www.rationalsoftware.com/products/whitepapers/319.jsp`.

[22] Michiel Ronsse, Bastiaan Stougie, Jonas Maebe, Frank Cornelis, and Koen De Bosschere. An efficient data race detector backend for diota. In *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, volume 13, pages 39–46. Elsevier, 2 2004.

[23] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[24] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the USENIX Annual Technical Conference*, pages 2–2, 2005.

[25] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmark suite, 2006. `http://www.spec.org/osg/cpu2006/`.

[26] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*, pages 85–96, 2004.

[27] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proc. of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pages 273–284, 2007.

[28] Qin Zhao, Rodric M. Rabbah, Saman P. Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proc. of the 17th International Conference on Compiler Construction (CC '08)*, pages 147–162, 2008.

[29] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Proc. of the International Symposium on Code Generation and Optimization (CGO '10)*, April 2010.

[30] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proc. of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pages 121–132, 2007.

[31] Yuanyuan Zhou, Pin Zhou, Feng Qin, Wei Liu, and Josep Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):3–33, 2005.