Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A low-cost memory remapping scheme for address bus protection

Jun Yang[a,*], Lan Gao[b], Youtao Zhang[c], Marek Chrobak[d], Hsien-Hsin S. Lee[e]

[a] Electrical and Computer Engineering Department, University of Pittsburgh, United States
[b] Vmware Corporation, Palo Alto, CA, United States
[c] Computer Science Department, University of Pittsburgh, United States
[d] Department of Computer Science and Engineering, University of California, Riverside, United States
[e] School of Electrical and Computer Engineering, Georgia Institute of Technology, United States

ARTICLE INFO

Article history:
Received 16 September 2008
Received in revised form
8 October 2009
Accepted 23 November 2009
Available online 6 February 2010

Keywords:
Address bus leakage protection
Secure processor

ABSTRACT

The address sequence on the processor-memory bus can reveal abundant information about the control flow of a program. This can lead to leakage of proprietary algorithms or critical information such as encryption keys. Addresses can be observed by side-channel attacks mounted on remote servers that run sensitive programs but are not under the physical control of the client. Two previously proposed hardware techniques tackled this problem through randomizing address patterns on the bus. In this paper, we examine these attempts and show that they impose great pressure on both the memory and the disk. We propose a lightweight solution to alleviating the pressure with equal security strength. The results show that our technique can reduce the memory traffic by a factor of 10 compared with the prior scheme, while keeping almost the same page fault rate as a baseline system with no security protection.

1. Introduction

In modern computing systems, it is common to protect processes running on the host machines from the exploits of malicious applications downloaded from unauthorized sources as they may contain malicious code such as viruses or worms. On the other hand, there is an often overlooked security vulnerability where the confidentiality of an application can be compromised by the processes running on the host machine itself. In other words, counter-exploit mechanisms are needed for the application providers to protect their intellectual properties when their applications are executed on untrusted machines. This problem is becoming more realistic and prevalent in distributed computing and embedded computing domains.

In distributed computing, for instance, application tasks are dispatched to geographically distributed machines that are often contributed by volunteers. Even though distributed security protocols authenticate and authorize resources and users, there is no protection mechanism once a job starts executing on a remote node. It is difficult to recognize if the execution was tampered with, or if any secret it carries was compromised locally. A host machine, having full access to all the local resources, can launch a background program to analyze the execution of an active job and extract confidential information such as the encryption key [23]. Also, the memory contents could be altered either through software breaches or by privileged users so that the computation time of a job is lengthened [9]. This is especially harmful in a commercial environment where resources and services are charged by hour [3].

There have been a number of proposals that address the attacks from a host machine to its guest programs. The attacks can originate from a privileged user [9,17,31], a normal user [23], or even physical accesses [12,9,17,31,35,37,30]. A common countermeasure is to encrypt memory contents to prevent secrecy violation, and to authenticate the memory at runtime to prevent misbehavior induced by memory corruption during execution. Both can be provided by a secure processor architecture such as XOM [17] or AEGIS [31].

In addition, the dynamic execution *address* sequence can be observed locally and analyzed to expose a program's critical information such as the private key in the RSA algorithm [14]. Losing a key allows an attacker to impersonate a trusted user or to delegate a victim's access right to other malicious users. Exposing dynamic address sequence can also reveal the program's control flow information to enable a commercial software developer to reconstruct a core algorithm from a competitor [37,10,11,36]. Hardware tapping devices for extracting addresses from the system bus, or even injecting traffic into the bus are readily available. For example, an FPGA-based programmable device can be attached to an SMP bus to perform on-line cache emulations taking real-time bus traffic

[*] Corresponding author.
E-mail address: juy9@pitt.edu (J. Yang).

0743-7315/$ – see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2009.11.008

from real workloads [22]. There are commercially available modchips that can be soldered onto the bus on the motherboard in a Sony Playstation or Microsoft Xbox to misguide the console to play pirated games [21].

Several techniques have been designed to combat the address information leakage: Goldreich et al. proposed three program transformation methods [10,11] to construct a data-oblivious memory access pattern. Unfortunately, these software-only methods suffer from either great performance penalty or memory explosion. Recently, two hardware-assisted schemes have been developed to trim down the overhead. The HIDE scheme [37] permutes memory subspace at certain intervals by reading them on-chip and writing them back after permutation. The Shuffle scheme [36] randomly shuffles the memory content whenever a block is read on-chip so that it will be written to a randomly different location later on. Both schemes try to randomize the address sequence on the bus to hide easily recognized memory access patterns such as loops.

We have observed that the HIDE scheme increases memory access by a factor of 12 to 32 for page sizes between 4 kB and 64 kB, due to sequential reads/writes and redundant permutations. On the other hand, the Shuffle scheme could induce a large number of page faults in high-performance systems with memory paging. Designs that significantly increase the memory or disk demand do not scale well in future machines incorporating multi-threaded or multi-core processors in which memory and disk bandwidth are both the first-order constraints. Thus, it is imperative to take those constraints into consideration while designing a secure architecture.

In this paper, we address the two main problems of the aforementioned security methods: memory access increase and page fault increase. These problems are mainly due to the following reasons: (1) excessive memory reads and writes on every permutation, (2) wasteful permutations, and (3) non-restricted block relocation. We propose a lightweight on-chip address permutation that effectively addresses all the three problems and achieves the lowest memory demand and page fault rate. Our main idea is to permute only *on-chip* cached blocks, to avoid the memory sweeps in HIDE, and launch a permutation for only those addresses that have not been remapped since they are read on-chip. Our scheme incurs only an 88% increase in memory accesses and a close-to-base page fault rate, without compromising the security strength.

The remainder of this paper is organized as follows. Section 2 describes the motivation of this work. Section 3 gives an overview of the two baseline schemes that we improve, and the in-depth analysis of them. Section 4 introduces our proposed scheme, followed by its architecture design issues in Section 5. Section 6 presents our experimental results. Section 7 discusses the related work and Section 8 concludes this paper.

2. Motivation

The address sequence recorded from the CPU-memory address bus may disclose the control flow information of a program under execution. This is true even in secure processors such as XOM [17] or AEGIS [31] in which the memory contents are all encrypted (i.e., the CPU-memory data bus transfers only ciphertext) but the addresses are left in plaintext. Such a plaintext address sequence can lead to critical information leakage and is the main problem we tackle in this paper. First of all, the sequence can be split into code and data sequences with explicit reads and writes. This is because code and data are in separate memory regions. Code regions are read-only and typically accessed sequentially at the start-up of the program execution. The obtained code sequence shows the control transfers only at a coarse granularity since most of the instruction reads are serviced by the on-chip caches. This is not difficult to circumvent as the on-chip caching can be disabled through setting proper control register bits [13], or minimized by running concurrent threads that compete for the shared cache with the victim thread [23,24].

The sequence obtained hereafter can be used to derive the control flow graph (CFG).[1] A sequence of "`abc abc abc`" clearly shows a loop with "`a`" being possibly the loop starting and "`c`" being the loop ending instruction. Whereas a sequence of "`abcd abd abcd abd`" indicates a conditional branch after "`b`" inside a loop containing "`a, b, c`" and "`d`". Most software nowadays have a high percentage of *reuse code* [19]—those that reuse pre-built libraries from the public domains or a third party. In other words, the reused portions of an application can be identified once their CFGs are constructed. This could ease the identification of the non-reused part of the code, leading to a potential intellectual property theft.

More seriously, the timing attacks to Diffie–Hellman, RSA and other security algorithms exploit the actual directions of a branch instruction inside a simple loop to reveal the private key bit-by-bit [14]. The loop iterates for a number of times equivalent to the bit width of the private key. Once the address sequence of the loop is exposed, the private key can be recovered.

Finally, addresses to data region can also expose control flow since some data are only accessed by one path of the program. Therefore, data addresses and code addresses should be protected simultaneously. Next, we will briefly review two existing algorithms (HIDE and Shuffle) for address sequence protection. For each of them, we give an example of how the scheme works. Then we analyze the two main problems they have. We focus on the memory access increase for the HIDE scheme and the page fault increase for the Shuffle scheme. We propose a scheme that has lower memory demand and lower page fault rate than HIDE and Shuffle. All the simulations were performed using the SimpleScalar Tool Set [1] with 11 SPEC2K benchmark programs, each simulating 1.1 billion instructions. If not specified, result is averaged across all the benchmarks.

3. Overview of address sequence protection

The basic idea of address sequence protection is to obliterate its correlation with the CFG of the program. For example, the address sequence of "`a a+ 4 a+ 8`" will not correspond to sequential instructions in the code. Also a sequence of "`abc abc`" will not necessarily represent a loop structure. Prior schemes (HIDE and Shuffle) incorporated randomization of the memory contents from time to time. Next, we will explain the processor-memory and hardware support model in those schemes.

3.1. Model and premises

We use the secure processor models proposed recently in [17,31,15] for our baseline, which was also used by both HIDE and the Shuffle schemes. In such a secure model, the processor is physically secure such that once the data and code are brought onto the chip, they cannot be tampered with. If any data are sent off-chip to the memory, they are always encrypted to ensure their confidentiality. Therefore, attacks can only be mounted to components external to the processor, such as the buses and the memory. Since the crypto operations are always performed between the processor and the memory, we assume a fast crypto mechanism is available on-chip to accelerate the process [29,32,34,28,4].

[1] The control flow graph is a directed graph that shows the transfer among instruction basic blocks.

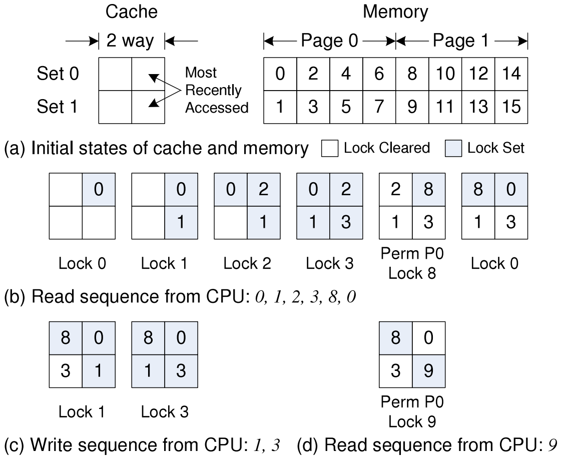


Fig. 1. Example of the HIDE scheme.

3.2. Chunk-level permutation

The basic idea of the HIDE technique [37] is to permute the address space at suitable intervals to obliterate the correlation between repeated memory addresses. Ideally, before an address recurs, a permutation is initiated to map it to a different location. In reality, it is not practical to memorize all the addresses between permutations, nor is it efficient to search the history list to detect the recurrence. Hence, HIDE proposed to augment the L2 cache replacement policy with additional locking control—*a block that is newly read from the memory or becomes dirty since its last permutation cannot be replaced after a permutation is performed*. This is because both cases could cause their addresses to recur on the bus in future reads if they were allowed to be replaced earlier. Hence, those entries are temporarily *locked* in the cache. Only blocks that are not locked could be freely replaced. Each permutation permutes a *chunk* (one to several pages) of continuous blocks, mapping each block to a different address within the chunk, so that recurring addresses on the bus may not indicate the same block. Fig. 1 shows how this scheme works.

We assume a two-set, two-way set associative cache, and a two-page memory with eight blocks in each page. In this example, one chunk is composed of one page, with even-addressed blocks mapped to set 0, and odd-addressed blocks mapped to set 1. Initially the cache is empty and no block is locked. When the CPU reads blocks 0, 1, 2, 3, 8, 0, the actual sequence on the bus is 0, 1, 2, 3, permutation traffic, 8, $\pi_1(0)$. $\pi_1(0)$ is the permuted address of block 0. Permutation traffic consists of sequential reads and writes of all blocks in page 0, and this permutation is triggered when block 8 replaces the locked block 0. After permuting page 0, all cache blocks of page 0 are cleared for their locks. Any block brought on-chip is locked afterward. Next, when the CPU writes block 1 and 3, both are locked again in the cache. The last read for block 9 causes a write-back of block 1. Since it is locked, a second permutation of page 0 is initiated, during which the locks on block 0 and 3 are cleared.

Through permuting the entire chunk, control transfers within the chunk are invisible on the bus, thus reducing the likelihood of information leakage. However, the strengthened security comes at a high cost of memory accesses as all the blocks in the chunk are swept through – read on-chip and then written off-chip – on every permutation. In Fig. 2, we categorize HIDE memory accesses into true memory requests ("true") and permutation induced accesses ("perm"). All the accesses are normalized to the first bar for 4 K byte chunk size. It is surprising to see that the percentage of true and useful memory accesses account for only 7.5% and 3% of the total for 4 kB and 64 kB chunk sizes respectively. In other words, the HIDE scheme increases the memory traffic by a factor of 12 (4 kB) or 32 (64 kB). Using larger chunk sizes has its own advantages: (1) it further reduces the control flow exposed on the bus; and (2) it reduces the frequency of permutations since the chances of clearing the locks are higher. However, the dramatic increase in memory traffic using large chunks creates a serious bottleneck in the system, which overrides its benefit in security. We break down the extra traffic into two sources:

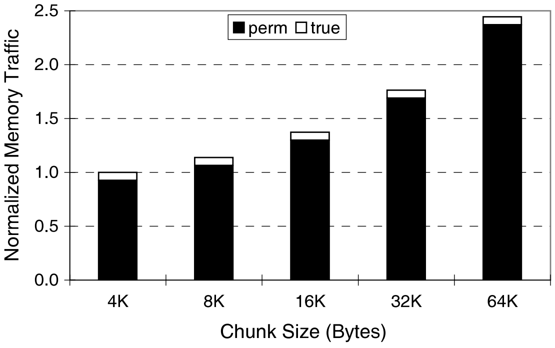


Fig. 2. Memory traffic breakdown for different chunk sizes.

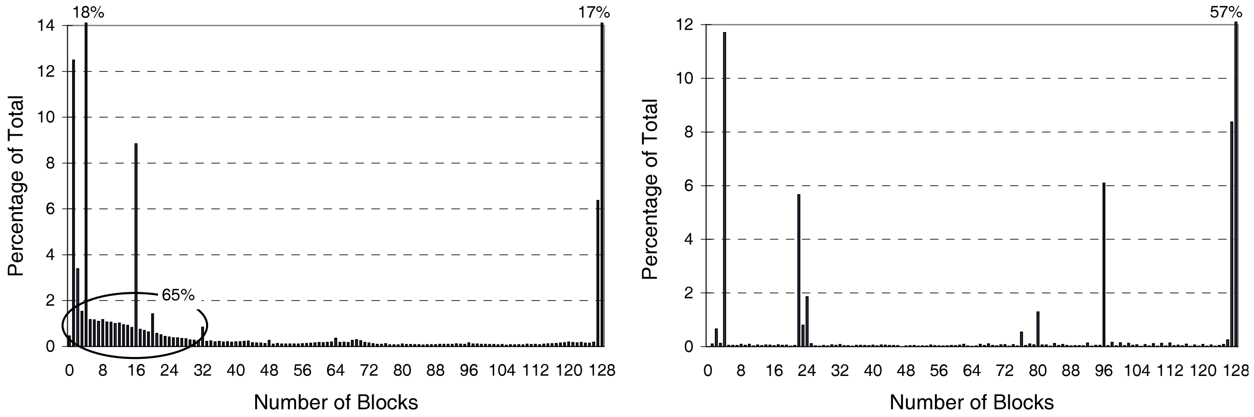
Excessive memory accesses on permutation. Fig. 2 shows that most memory accesses in HIDE are useless to the program. They are redundant accesses simply to *hide* the traces of the useful memory addresses. Here, "useful blocks" refer to those that have been accessed by the program. We observed that this quantity is fairly low between two permutations, an indication of large redundancy in HIDE.

In Fig. 1(b), only 4 blocks in Page 0 are accessed when the permutation is invoked. During the permutation, they are read on-chip again with the other half that is never touched. Then the whole page is written back, quadrupling the traffic. We studied the excessive accesses in a real system where the page size is 4 kB and each page consists of 128 32-byte blocks. Fig. 3(a) is a histogram of the pages with their numbers of accessed blocks from 0 to 128 upon a permutation. That is, when a new permutation of a page is initiated, if it accessed *i* blocks since the last permutation, we increment the *i*th bar in the histogram. We can see that only 17% of pages are fully accessed between two permutations. For those pages, efficiency is good since there is no reading of useless memory blocks. However, 65% of pages triggering a permutation are accessed less than a quarter of the blocks.

In fact, if we take a global view of the block usage in a page during the entire simulation, the percentage of pages that are used is quite high (Fig. 3(b)). More than half (57%) of the total pages are fully used, much more than that in Fig. 3(a), because some blocks inside those pages are recalled before the rest are touched. Each recall triggers a scan of the entire page and unaccessed portion of the page can be permuted multiple times due to multiple such recalls. Using the example in Fig. 1, if block 4–7 are accessed in the future, then page 0 would be fully used. Before this could happen, however, it is already permuted twice.

Redundant permutations. Due to the difficulty of tracking if an address is a repeated access, the permutation is triggered preventively. That is, before a dirty block is written back to the memory, a permutation of the hosting page is triggered, anticipating that the block will be read again in the future. This is why the second permutation in Fig. 1(d) is performed. Such permutations induced by write locks trigger multiple permutations while only one is necessary. The example below illustrates such a scenario.

Let blocks A, B, C belong to page P. They are initially read on-chip and then locked. When B has to be replaced, a permutation of P is triggered since B holds a read lock. Afterward, A, B, C are all mapped



(a) Between permutations.

(b) During the entire simulation.

Fig. 3. Histogram of pages with 0–128 blocks accessed.

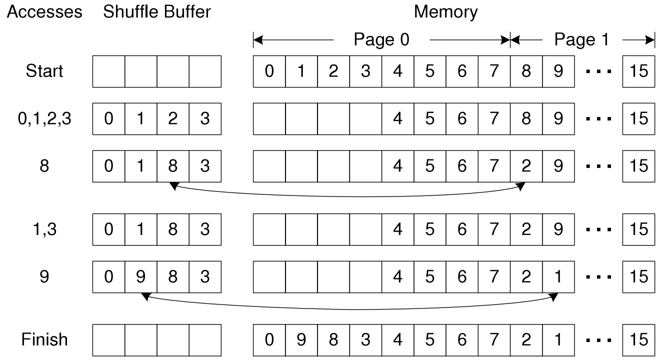


Fig. 4. Example of the Shuffle scheme.

to different addresses, B is written back to a new address while the others are still on-chip. The permutation also clears off their read locks. Suppose a later write locks A again. When A is replaced at some point, a second permutation of P is initiated, which we can see is unnecessary because A has been mapped to a new location in the first permutation. Similarly, B and C are both remapped in the second permutation, which are also redundant. Note that between permutations, no accesses to page P are necessary, indicating that a page could be permuted repeatedly without even a single access. This also explains why the 0th bar in Fig. 3(a) is positive (0.46%).

3.3. *Memory shuffle*

A simpler scheme was proposed in [36] for embedded systems, since they are more vulnerable to the side-channel attacks on the address bus. The approach is to relocate a block if it is brought on-chip so that it will be written to and read from a different memory address in the future. The new address is chosen randomly from the program's memory space. To implement it efficiently, a small portion of memory blocks is stored in an on-chip *shuffle buffer*. A random block is selected from this buffer to swap with the requested block read on-chip. Essentially, a memory read is always followed by a memory write, where the read is demanded by the processor while the write is a swap randomly picked from the buffer. The block being read still resides in the buffer (as well as the caches) so that the shuffle buffer always has a fixed number of memory blocks for swapping.

Fig. 4 (an example adapted from [36]) shows how this scheme works. We assume the on-chip shuffle buffer only stores 4 blocks. The first 4 accesses fill up the buffer. Starting from the fifth access, a random swap between the memory and the shuffle buffer (e.g., block 8 and 2) is performed. If the access hits in the buffer (e.g., block 1 and 3), no swapping is necessary. Finally, all the blocks in the shuffle buffer are written back to the very first four empty slots in the memory.

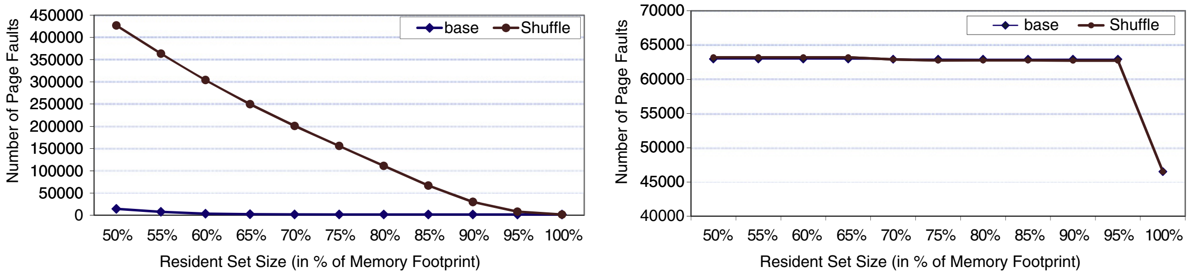
As shown, the memory traffic in the Shuffle scheme – as we will term it in this paper – is only two times of an unprotected system, because a read is always followed by a write. This overhead is significantly lower than the HIDE scheme. However, the blocks are swapped in the entire program memory space, e.g., the swapping of block 8 in page 1 and block 2 in page 0. So this scheme needs to remember the block mapping, i.e., which address a block is mapped to, using the *full-width* block address, whereas in the HIDE scheme, only the offsets in the chunk need to be remembered as the blocks are remapped only within a chunk. The storage overhead of the Shuffle scheme is therefore greater than that of the HIDE scheme (10% versus 3.5% as reported previously).

More importantly, in order to make memory accesses look "random" in its memory space, the Shuffle scheme shuffles blocks in the entire program memory. As a result, it destroys the program's locality, and blocks in hot pages may be mapped to cold pages. Eventually, in the long run, all pages are roughly equally warm. This may increase the page faults if not all pages are in the memory. If the Resident Set Size[2] (RSS) is 100%, i.e., all pages of the program reside in memory, such a randomization will not incur extra page faults, and all page faults are cold page faults. However, if RSS is 50%, i.e., at any time of the program execution, only half of the pages reside in memory, any access to the pages not in memory will incur a page fault. In this case, random swapping of blocks will increase the page faults.

Fig. 5(a) plots the trend of page faults versus the RSS for `gcc`. As we can see, the number of page faults increases by 257 × when RSS drops from 100% to 50% of the total memory pages, while the curve is almost flat for the base case. This shows that preserving locality is critical to program performance. Similar patterns can be observed in other programs as well. Here we assumed a 4 kB page size and perfect LRU memory page replacement policy.

Nonetheless, if the shuffling happens to keep or improve the program's locality, e.g., page 0 and page 1 in Fig. 4 are always accessed together, the number of page faults could remain the same or even decrease. As we can see from `gzip` in Fig. 5(b), the number of page faults in the Shuffle scheme is almost the same as that in the base case. This is because the shuffle buffer of `gzip` stores mainly those blocks within the current working set due to its special memory access pattern.

[2] The number of virtual pages resident in RAM.



(a) gcc.

(b) gzip.

Fig. 5. Page fault curves.

It is also worth noting that the Shuffle scheme can introduce redundant memory writes as well. This is because the shuffle buffer effectively holds the recently fetched memory blocks. On a new read from an address `Addr` for example, a block `B` is randomly picked from the shuffle buffer and written back to `Addr`. However, writing B back to memory might be a redundant operation as B may need to be written back again later when an actual write to B takes place. This suggests that the previous write was unnecessary. Our measurements show that such redundant writes account for ∼5% of the total memory traffic on average.

3.4. Real workloads

Apart from the simulations above, we also use real workloads to verify the impact of smaller resident set size on page faults. This serves as a supplement to our studies above, since multiple applications with small memory footprints running together could have a union of a rather large footprint. When they compete for memory resources, each application has a relatively smaller working set in the memory, leading to more page faults. Fig. 6 shows the page fault curve for Windows Media Player on a DELL Inspiron 4150 laptop with 384 MB memory running Windows XP. We use two tools in the Windows 2000 Resource Kit to perform the experiment, the page fault monitor (`pfmon`) and the memory stealer (`leakyapp`). We first use `leakyapp` to allocate and hold a fixed amount of memory, then run Windows Media Player either alone or with other applications to see how it performs when the memory is running low. Two popular applications, Microsoft Office PowerPoint and Adobe Reader, are chosen to run at the same time with the Media Player. `Pfmon` monitors the page faults generated by Windows Media Player. The result shows that when all three applications are running concurrently and the available memory is reduced to 64 MB, the number of page faults increases by almost a factor of 4. This justifies the above simulation method in which the resident set size is varied to see the page fault increase.

3.5. Summary

The HIDE scheme permutes blocks only within a chunk, practically one page, and hence has little impact on the memory paging. However, it incurs extremely high overhead of memory accesses, which makes it hard to fit into contemporary processors where memory is still a performance bottleneck and one of the most power-hungry components. The Shuffle scheme, on the other hand, introduces mild extra memory demand. Yet its demand on the disk access limits itself to embedded systems where most applications have small memory footprints. Accessing memory is usually several orders of magnitude faster than accessing the most technologically advanced hard-drives. Hence, in a demand paging system, it is important to keep the page fault rate low. In the next section we introduce our lightweight design that addresses the shortcomings of the previous systems, by reducing their memory access overhead and the excessive number of page faults.

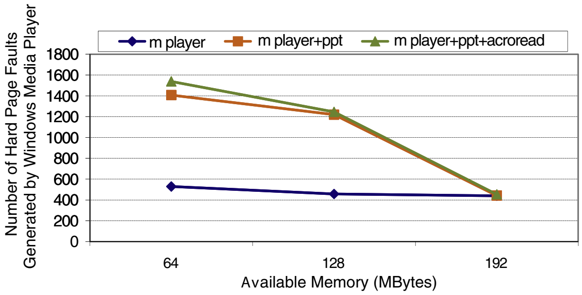


Fig. 6. Page fault curve for Windows Media Player.

4. Proposed inexpensive address permutation

Our scheme aims at achieving three goals. The first goal is to avoid wasteful memory reads and writes in each permutation. The second goal is to eliminate the wasteful permutations so as to reduce the total number of permutations. The third goal is to preserve locality and keep the page fault rate low. The approach we take is to permute selective blocks instead of a whole page, aided by a good decision of when a permutation should happen.

4.1. The permutation mechanism

The idea. We propose to perform permutations only on *on-chip* blocks. This is because their addresses have occurred once on the bus, and could recur if they are evicted and read in again in the future. Therefore, it is necessary to relocate them (i.e., permute them) before they are replaced. However, not every replacement should be preceded by a permutation because if a victim block has been permuted and mapped to a different address, it is safe to release it off-chip. It is only those blocks that are recently read on-chip (called RR blocks) but have not participated in any permutations that should be permuted. In other words, a block B of a page P is an RR block if B is fetched from memory after P's last permutation. Hence, a permutation is started only if an RR block is to be replaced. Afterward, all the RR blocks that are involved in the permutation are turned into normal blocks which can be safely released off-chip because the permutation has masked the earlier traces of accesses and their addresses have been mapped to other random locations.

Since only on-chip blocks are involved during a permutation, the need of reading chunks of blocks from memory and writing them back as in HIDE is eliminated. This is the main reason why our scheme can reduce most of the memory traffic. Another benefit of this approach is that we do not need to update the memory right away. Since the permuted blocks are on-chip, we simply

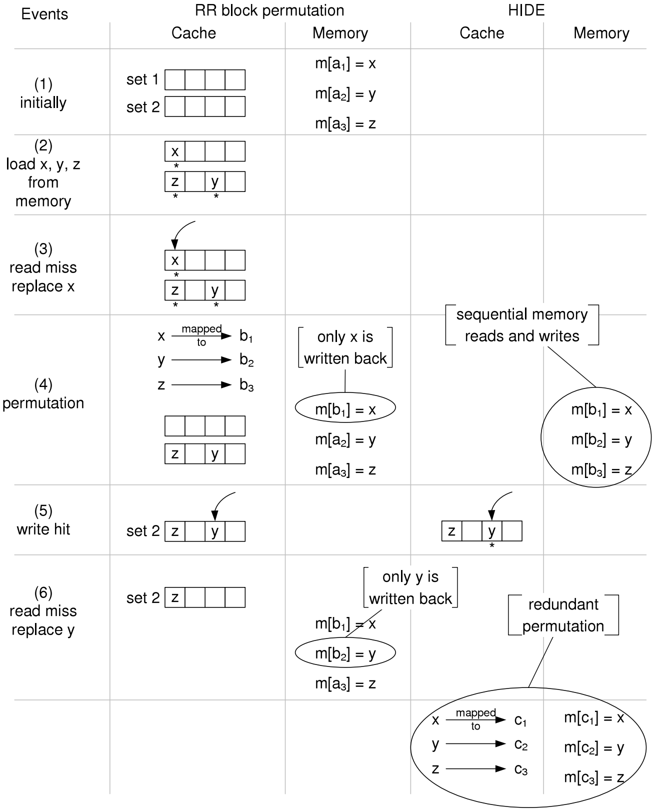


Fig. 7. Comparison of on-chip block permutation and the HIDE scheme. x, y, z are from the same memory page. x is mapped to cache set `set 1`. y and z are mapped to `set 2`. The block with a $*$ means an RR block in our scheme, and a locked block in HIDE. '$\mathtt{m[a]} = x$' means x is stored at memory address $\mathtt{a}$.

perform the permutation of their *addresses* and remember the mapping. When they need to be replaced sometime in the future, they refer to the mapping to obtain their new addresses to which they should be written. Note that we write every replaced block into the memory because it is relocated to a different address even if it is not dirty.

An example. We now walk through an example in Fig. 7 to show our permutation mechanism. Assume that initially, x, y, z are from the same page in memory with address a_1, a_2, and a_3 respectively. When x, y and z are loaded on-chip, they are marked as RR blocks. If later a read miss happens in x's set (event 3) and x is to be replaced, a permutation must be performed because x is an RR block. The permutation generates a random mapping (event 4) for all the on-chip blocks of the same page as x (and some other blocks which will be explained later) and then clears off the marks for those RR blocks being permuted. The mapping is kept on-chip. After that, x is written back to the memory at new address b_1. Note that x may not be dirty but is still copied back. y and z are not written back until they are evicted from cache. A write hit on y (event 5) does not set the RR-bit again. Finally when y is replaced (event 6), it is written back to the new address b_2 assigned during the latest permutation.

We also illustrate the actions taken by the HIDE scheme in the same figure as a comparison to our scheme. This example is along the same line as Fig. 8. The main differences are in the type and number of blocks involved in each permutation and the time a permutation is triggered. In event (4), HIDE performs sequential reads and writes to the entire page so that all the blocks in the memory are physically permuted. Note that this is wasteful because y and z are still on-chip to serve future requests from the CPU. Moreover, the write hit to y in event (5) locks y again, which triggers a second permutation upon a replacement of y in event (6). The entire block is again sequentially read and written to the memory although a remapping of all three blocks at this time is indeed unnecessary.

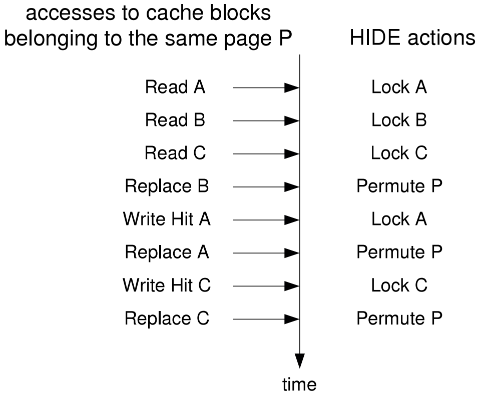


Fig. 8. An example showing redundant permutations in HIDE.

Let us mention a few subtleties here to make this example more complete. First, the initial addresses a_i are not the true physical addresses of x, y and z. They have been randomized in previous permutations or an initial permutation of all memory pages as the program started. This is a requirement in HIDE and Shuffle as well. Therefore, one cannot infer, for example "a_1 is mapped to b_1 as the permutation is triggered by their cache conflict", because what caused the conflict are not really a_1 and b_1 on-chip. They have been remapped randomly. Second, permutation on event (4) of HIDE may not happen at the same time as in our scheme because of the LRU replacement that was modified in HIDE to delay such permutation. However, it can be delayed but cannot be avoided in the future. Third, the mapping created by the two schemes may not be identical because they use different block sets. We make them equal in the example for ease of illustration and comparison only. Finally, the permutation is performed on virtual addresses. In a demand paging system, this will cause more subtleties involved in TLB and page table updates. The HIDE scheme did not address the complexity properly, while the Shuffle scheme assumes most of the embedded systems do not employ virtual memory. We will explain these problems later in Section 5.

In brief, our scheme permutes only on-chip blocks, and a permutation takes place only when an RR block is replaced. We save a significant amount of memory traffic compared to HIDE. The overhead we pay here is only the write-backs of the non-dirty blocks. Further, we eliminate permutations that are due to the write locks in HIDE scheme. The simulation result will be presented in the next section after the full scheme is described.

4.2. *Permuting sufficient number of blocks*

During each permutation, there should be enough many blocks involved to ensure the quality of the randomization. For example, in our 4 kB page setting, HIDE permutes 128 blocks every time, generating 128! possible mappings or a probability of 1/128 to guess one mapped address correctly. If there are only 32 on-chip blocks in a page and if we only permute among those blocks, then the probability is increased to 1/32, much higher than before and thus less desirable.

Therefore, we increase the number of available blocks by looking at the *chunk* level instead of page level. The HIDE scheme could not truly scale to the chunk level due to its prohibitive increase in memory demand, while we do not have this concern as we only look at the on-chip blocks, and thus our scope can be scaled up to include more pages. Our goal here is to permute the same number of blocks per permutation as the HIDE scheme. As the number of blocks on-chip is dynamic for every page, we cannot be certain

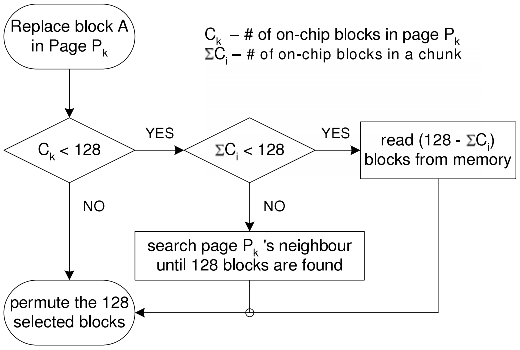

Replace block A in Page P_k
C_k – # of on-chip blocks in page P_k
ΣC_i – # of on-chip blocks in a chunk
C_k < 128
YES
ΣC_i < 128
YES
read (128 - ΣC_i) blocks from memory
NO
NO
search page P_k 's neighbour until 128 blocks are found
permute the 128 selected blocks

Fig. 9. The procedure of searching for sufficient number of blocks to permute.

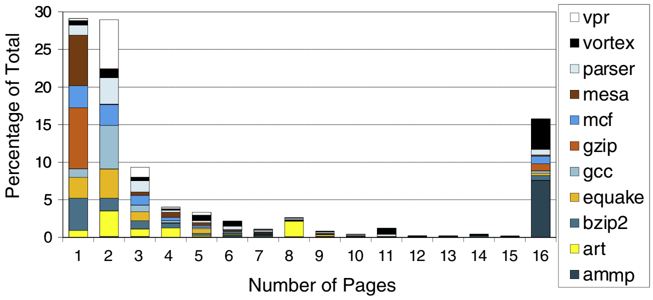


Fig. 10. Average number of pages that supply sufficient number of on-chip blocks per permutation.

how many pages should be included in order to have 128 blocks in every permutation. Therefore, we develop a simple incremental mechanism to solve this problem, as illustrated in Fig. 9.

The procedure. When a permutation is started by replacing an RR block A in page P, we first check if P has enough, e.g., 128, blocks on-chip. If so, we simply permute the 128 addresses of those blocks. Otherwise, we expand the search scope one page at a time in its neighborhood until we have enough many blocks to permute. When there are a sufficient number of blocks, giving priority to the RR blocks over the normal blocks can reduce the total number of permutations because only an RR block triggers a permutation and it becomes a normal block after the permutation. However, our search space does not grow unbounded. It is limited by the chunk size, i.e., the chunk boundary where P falls within. If we still cannot find sufficient number of blocks in the entire chunk, to ensure the same level of security, we choose to read extra blocks from the memory within the chunk as padding blocks to the permutation input. As we can see, the chunk size should be a reasonable number, as too small the chunk size leads to a HIDE-like scheme because we need to read many extra blocks, but too big the chunk size leads to a Shuffle-like scheme because we lose locality among the permuted blocks. Therefore, choosing an appropriate chunk size depends on the trade-off between the memory accesses and the incurred page faults.

Typically, the collection of sufficient number of blocks can be found in just a couple of pages. The extreme of searching in the entire chunk does not happen very often. Fig. 10 shows the average number of pages searched in order to find sufficient number of blocks on-chip. We set the chunk size to 16 pages, the choice that will be explained later. The figure is averaged across all 11 SPEC2K benchmark programs. Each bar also shows the weight contributed by each program. In most cases (∼58%), up to two pages are enough to provide 128 blocks on-chip. However, searching through the entire chunk does happen more than 15% of the time. This number is mainly contributed by programs `ammp` and `vortex`, both having poor locality and small number of blocks per chunk in the cache. Note that when the search stops at 16 pages, padding might be needed. If so, they automatically fall into the 16th bar. An anatomy of the number of pages touched per permutation for each program is given in Fig. 11. As shown, most of the programs, except for `ammp` and `vortex`, need only 4 pages for 70% of the time. This implies that the searching algorithm can terminate quickly.

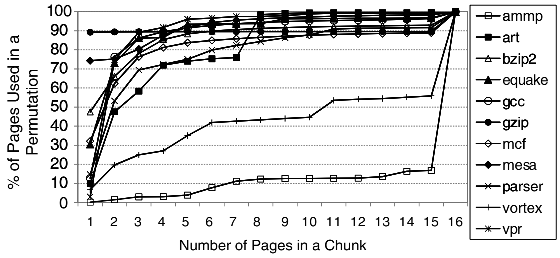


Fig. 11. Number of pages involved in each permutation.

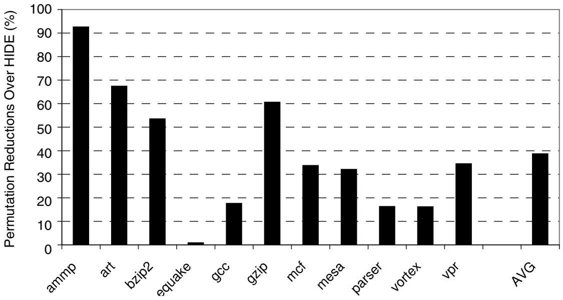


Fig. 12. Percentage of permutation reduction.

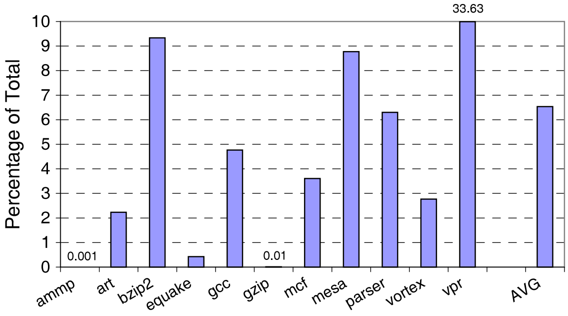


Fig. 13. The percentage of saved permutations due to write locks.

The benefit. Permuting across multiple pages on-chip helps greatly to reduce the number of permutations. For example, suppose a replacement of an RR block in page P_1 triggers a permutation, followed by a replacement of an RR block in P_2. In HIDE, both replacements cause permutations while in our scheme, P_2 could be permuted together with P_1 in the first permutation, which might clear off the RR block in P_2 and save the second permutation. Such an effect is the major contribution to the total permutation reductions shown in Fig. 12. On average, our scheme saves nearly 40% of the permutations in HIDE including those due to write locks as explained in the previous section. Specifically, Fig. 13 shows the percentage of such permutations saved by our scheme over HIDE.

We also studied how frequently permutations occur. If they happens very often, then normal memory accesses would be greatly affected. If it happens only occasionally, the costs could be amortized over a much larger number of memory accesses. In Fig. 14, we also show the density, in percentage of total memory accesses, of permutations for our chunk-level scheme and the HIDE scheme. On average, only 1.52% of the memory accesses invoke permutations, i.e., among 200 memory reads and writes, only 3 of them trigger a permutation. With efficient hardware components discussed in Section 5, the impact of our scheme on the system

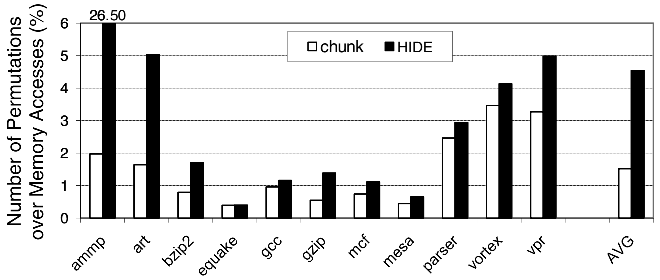


Fig. 14. Permutation frequency.

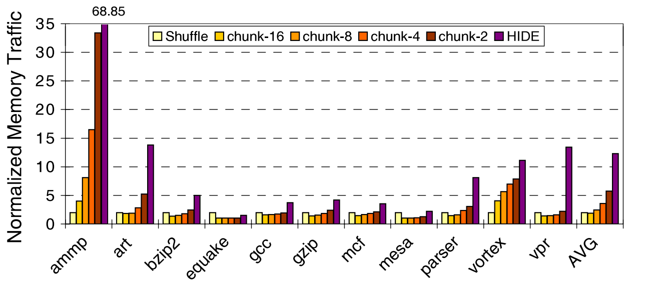


Fig. 15. Memory traffic comparison of different schemes. Chunk size is varied from 2–16 pages.

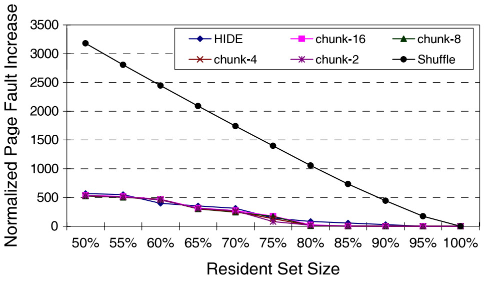


Fig. 16. Page faults comparison among different schemes averaged across all benchmarks.

is very limited. Our scheme saves the most for benchmark ammp, where only two out of 100 memory accesses trigger a permutation, whereas in HIDE, one fourth of the memory accesses trigger a permutation.

Choosing the chunk size. As mentioned earlier, a large chunk size may result in more page faults as the permutation spans over a wider address space. A small chunk size may generate more extra memory accesses since not all pages are fully cached on-chip. Hence it is important to find a good break-even point between the page faults and memory accesses. We compare the total number of memory accesses in Fig. 15 with varying chunk sizes in terms of number of pages. The results are normalized to that in a base system where no address protection is present. The Shuffle scheme doubles the memory traffic while HIDE leads to an enormous increase by a factor of 12. As expected, the larger the chunk size, the smaller the traffic. A chunk of 16 pages only brings 88% more traffic on average, even better than Shuffle, while other sizes increase the traffic by a factor of 2.46, 3.59, 5.73, respectively. Our memory traffic is lower than the Shuffle scheme because the aforementioned reason that Shuffle introduces ∼5% extra writes while our scheme can avoid them. The major portion in the 88% extra traffic comes from the write-backs of non-dirty blocks. To show the page fault increases with larger chunk sizes, Fig. 16 plots the curves with gradually decreasing memory RSS. The results are averaged across all the benchmark programs. As we can see, although a larger chunk size generates a little more page faults (less than 2% increase from chunk-2 to chunk-16), they all have far fewer page faults than the Shuffle scheme. So, overall speaking, a chunk size of 16 pages has the lowest memory traffic increase with a reasonable page fault increase.

4.3. Security analysis

To find out the address recurrence from our randomized address sequence, an observer must be able to conjecture at least one address mapping correctly between two permutations. Our scheme has many uncertainties that help reduce the probability of a successful guess. For example, there is no clear indication of when a permutation really happened from off-chip observation since most permutations do not involve off-chip accesses and only a subset of off-chip write-backs would cause a permutation. Also, even if a permutation is identified, it is hard to know which set of on-chip blocks participated in the permutation since they are determined internally. Furthermore, it is not clear what set of blocks are really on-chip soon after some initial execution as write-backs are remapped so that it is hard to know what blocks are returned to the memory.

Nevertheless, let us assume a case where an observer can make a correct guess with the highest probability. Other situations are significantly more complex than this case and the probabilities will not be higher. Suppose a page P is entirely read on-chip. Later on a block is written back to an address A in P. Since all the blocks of P were read on-chip and were initially RR blocks, any replacement would invoke a permutation inside P only. Hence, it is some block in P that is mapped and written back to A. The probability to guess A's original address correctly is $\frac{1}{128}$, assuming there are 128 blocks per page. If a second write-back to A′ in P occurs, the probability to also guess (A′)'s original address correctly is $\frac{1}{128} \times \frac{1}{127}$. In order to guess all mappings correctly, the probability is $\frac{1}{128!}$. Hence, it is best for an observer to make a guess in the first write-back after a latest permutation.

Similar to HIDE, what an observer can see on the bus is the chunk-level transition, instead of fine-grained page-level or even block-level transition that can reveal control flow easily. The HIDE technique has shown that a chunk size of 64 kB, i.e., 164 kB pages, can cover 95% of the program's control flow (some compiler-assisted layout optimization may be necessary to achieve this number). However, the HIDE scheme cannot afford to operate on real 64 kB chunk, while our scheme performs the best with this size.

By permuting mostly the on-chip blocks with occasionally off-chip padding blocks, our scheme makes full use of the on-chip cache. The more blocks stored on chip, the fewer addresses are transferred on the bus, and the more secure it is. If an attacker maliciously runs a simultaneously executed thread to inject well crafted cache accesses in order to push some cache blocks off-chip, the program under attack would create more memory accesses as fewer blocks are on-chip, leading to a more HIDE-like effect. Essentially, the address protection scheme plays against the cache-centric attack until the program runs at very low speed. Now it is probably good to raise an alarm indicating a possible attack.

However, the previous discussion is based on the assumption that there is no pattern in the block replacement. If the block replacement is very regular, an adversary might have higher probabilities to guess a correct mapping. For example, if the cache is direct-mapped, and a processor reads only one address A from a chunk and immediately evicts it, there will be 127 reads to bring enough many blocks on-chip for shuffling. If A is evicted by itself, i.e., the 127 padding blocks are evicted together without A, then an adversary can correlate the single read with the single write to find out the mapping for A. Another extreme example is that in a fully associative cache with FIFO replacement policy, where an adversary is able to tell which block is to be evicted next, it is not hard to correlate a previous read with a future write to find out the block mappings.

To defeat these kinds of attacks, we take advantage of our prefetch buffer to delay all block evictions by a random amount of time, i.e., when a block is evicted, we hold it in the prefetch buffer for some time, and evict another victim instead. The prefetch buffer is where we store the padding blocks as discussed in the previous section. Now that which block is evicted at what time is unpredictable, the adversary cannot correlate the two addresses before and after the swap as easily as before.

Another security issue we need to discuss is how much information an adversary can obtain from the side-effects of TLB and block mapping accesses. Since it is impossible to store all the address mappings on-chip, we designed a Chunk Information Table (CIT), introduced in Section 5.1, to cache partial mappings on-chip. Similar to data cache misses, TLB misses and CIT misses on the bus can reveal page and block access patterns. Let us first look at how much information an adversary can obtain. For a 4 kB page size, one page can cover 1024 32-bit instructions. For a 64 kB chunk size and a 32 B L2 block size, each block index needs 11 bits, so a 32 B CIT block can store the indices for 23 ($\lfloor 32 \times 8/11 \rfloor$) L2 blocks, which contain 184 instructions. Compared with a 32 B L2 block that have only 8 32-bit instructions, much less information can be extracted from the TLB and CIT misses.

To achieve even better security, we can build a hierarchical protection scheme as mentioned in both [37,36]. Any CIT miss will cause another look up in the CIT cache again to fetch its own mapping. Now that we need to store multi-level mapping table, and fetch multi-level such mappings just to get one L2 data block, the performance and storage overhead is even bigger. There is always a trade-off between security and performance. In our work, we choose to use only one-level mapping table to build an affordable security system.

5. Implementation issues and architecture design

5.1. The permutation architecture

To put everything together, we need the following hardware components to assist the address protection scheme. First of all, we need a permutation unit that can generate a random permutation for 128 blocks. Such a permutation unit can leverage the hardware random number generator that utilizes the complex physical phenomena [18]. Hence, no special procedure for seed generation is necessary. A shuffle algorithm performing 128 swaps would suffice as long as a hardware true random number generator is present. Notice that, in most cases, we only need to permute the addresses of the on-chip blocks. Occasionally we also need to read some padding blocks from memory to participate in the permutation. Thus, a buffer for temporarily storing them is necessary. Its size is 127 blocks as there must be at least one block on-chip in the chunk. Also, storing the 128 block offsets for permutation requires 128×7 bits in total.

Next, we need an information table for every chunk that remembers the block address mappings (in BAT) and the block status. All address mappings are stored in the chunk information table. Each chunk has an entry, consisting of two bit vectors and a virtual address BAT. One bit vector remembers if the block is on-chip, and another remembers if it is an RR block. The BAT stores the virtual-to-virtual address mappings (more in Section 5.3) for each block, with each mapping used only once before being relocated again. The record size is small compared with the size of a chunk. If we assume the block size is 32 bytes, and each page is 4 kB, then for a 64 kB chunk, the storage overhead is only 5%, since there are only two status bits per block, and 11 bits for the mapped block offset within the chunk. Hence, there is approximately the same overhead as in HIDE with the same chunk size. When compared with the Shuffle scheme where there is a 10% storage overhead for the BAT, we reduced the storage overhead by half.

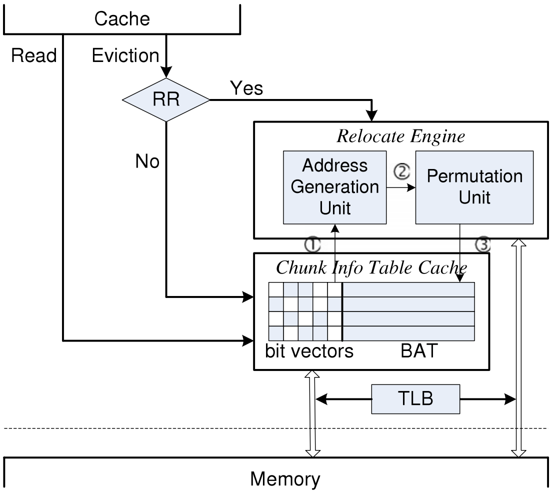


Fig. 17. Architecture of the permutation unit.

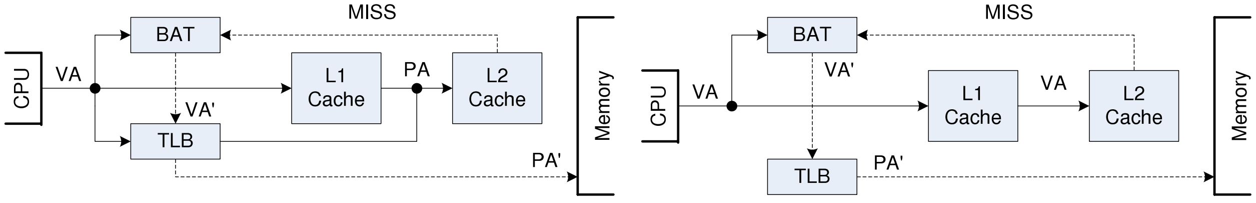
Finally, we need an address generation unit that can quickly find 128 on-chip addresses to permute. This can be easily performed through wide shift registers and an adder. We can first AND two status bit vectors of a page into the shift register, and then shift the register one bit at a time. The adder increments the block offset on each shift, and outputs the offset to the permutation unit if the bit is set, i.e., the block address should be used for permutation. Once a vector for a page is finished, the unit starts with the next page. It might take two iterations to do so as we may not have sufficient number of RR blocks in one round. In the second round, we shift only the on-chip status bit vectors. The starting point should be randomly selected to inject some randomness in selecting the on-chip cache blocks. Since the adder only operates on block offsets, it is only 7-bit wide. Hence, a conventional 32-bit adder can perform four additions in parallel, producing 4× the throughput.

The architecture design of the hardware components and the datapath are illustrated in Fig. 17. When a cache miss happens, the BAT is searched for the mapped address, followed by a TLB access to obtain the physical address. When a write-back happens, if the victim is not an RR block, the write-back just needs to go through address mapping and TLB translation as with the cache miss. If the victim is an RR block, a permutation must be initiated before it can go off-chip. The permutation starts by sending the status bit vectors to the address generation unit which then emits 128 addresses for the permutation unit. When a new mapping is ready, it is then updated into the BAT table. The victim can now be sent off-chip with a new mapping.

We assume the entire permutation takes 300 cycles on a 1 GHz processor, 128 for the address generation unit, 128 for the permutation, and the rest for address updates (BAT can be multiported). Since the number of permutations in our scheme is low, and the permutation is done on the write path which is non-critical, such a latency does not bring forth noticeable performance penalty.

5.2. Virtual addresses versus physical addresses

As with the HIDE address mapping scheme, our permutation should be performed on virtual addresses (VA), although we aim to prevent physically tapping the addresses bus. This is because if permutation was performed in the physical address space, it would



(a) Address translation in a physically addressed L2 cache.

(b) Address translation in a virtually addressed L2 cache.

Fig. 18. Double and single TLB lookups in a physically and virtually addressed L2 cache.

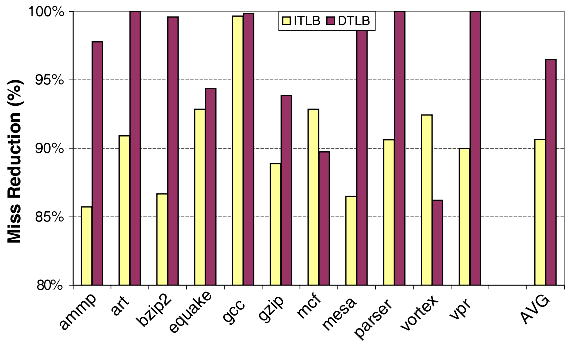


Fig. 19. TLB miss reduction with a 64 kB chunk.

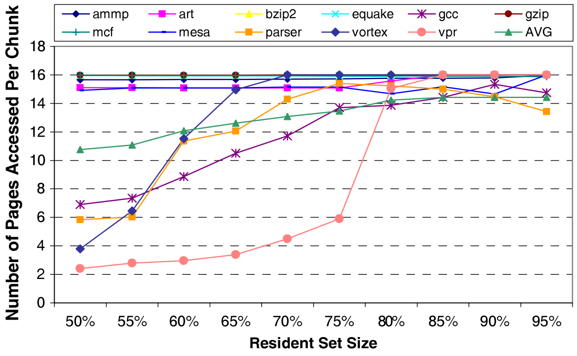


Fig. 20. Number of pages accessed per chunk when swapping out.

be interfered by the virtual–physical address mapping managed by the OS. For example, suppose that a permutation spans across two consecutive physical pages, P and P+1, and that P's association with a virtual page is later changed by OS. The permutation mapping maintained on-chip for P and P+1 now becomes invalid because P and P+1 originally contained blocks of each other, but now the contents of P are swapped out to the disk due to its change in virtual address association. In contrast, if the permutation is performed in the virtual address space, any VA will be mapped to another VA first, then translated to a PA by the TLB. The VA–PA association changes remain transparent to the hardware permutation unit. This is the main reason why we decide to permute the blocks in the virtual address space.

5.3. *Virtual cache versus physical cache*

Our address remapping process is carried beneath the last level cache, since we only need to protect the addresses that occur on the processor-memory bus. Therefore, the implementation of the permutation is different for a virtual L2 cache from that for a physical L2 cache. In this section, we discuss the advantages and disadvantages of using virtual cache versus physical cache.

Physical cache implementation. Since physically indexed, physically tagged L2 cache is the predominant cache type in ×86 CPUs, let us first discuss our implementation in a system featuring a physical L2 cache. In such systems, the MMU first translates the VA into a PA and uses it to access the cache, as shown in solid line path in Fig. 18(a). If it is a cache hit, the data is returned to the CPU and L1 and no further action is needed. If a miss happens, however, the PA should be sent to memory to fetch the data. In a memory address protected system such as HIDE and ours, the original VA is remapped to a VA′, and hence the current PA should not be used for memory access. Instead, we should first look up the new mapping for the original VA to find VA′ in the Block Address Table (BAT) [36], and then translate the VA′ to a PA′. Only after this, can we have the true physical address to access memory.

Compared with the data path in a system without address permutation, a *second* TLB lookup is necessary on an L2 miss to translate the mapped address VA′. This is because VA′ and VA are translated independently by OS to different physical addresses, and there is no correlation between PA and PA′. This procedure is illustrated in the dotted path in Fig. 18(a). The second TLB lookup might be a miss and trigger a memory access for the proper page table entries, which penalizes the performance. Our experiments show that on average, the TLB accesses increase by 5%, while the TLB misses increase by more than 12%. The increased TLB misses partly come from the increased TLB accesses, partly from the permutation induced variance.

It is possible to avoid the second address translation on an L2 miss by mapping the chunk to contiguous physical memory. Now that each chunk is contiguous in both the virtual address space and the physical address space, only one TLB entry is necessary for the entire chunk. When there is an L2 miss, PA′ can be calculated based on VA, VA′ and PA without accessing the TLB again. This essentially increases the page size, forcing the physical pages belonging to the same chunk to be swapped in and out together. In a system with plenty of memory, large benefits are gained from increasing the TLB coverage. Fig. 19 presents the best-case TLB miss reduction for each benchmark. This is obtained when the benchmarks are always given the contiguous memory regions they need. For some benchmarks, the data TLB misses are almost eliminated. However, when the system later runs out of contiguous physical memory, higher paging traffic will cause extra memory pressure because some pages are swapped without real accesses to them. Fig. 20 shows the average number of pages accessed in a chunk before it is swapped out when the available memory decreases from 100% to only half of the program's working set size. For vpr, only 2 out of 16 pages in a chunk are actually accessed when RSS drops to 50%. These I/O costs can easily outweigh the benefits from reduced TLB misses.

If we lessen the condition above, i.e., a chunk could be non-contiguous in physical space as mentioned before, then the memory pressure is greatly reduced. Fig. 21 shows that, when the RSS drops to 50%, a contiguous chunk generates around 8 times more page faults than a non-contiguous chunk, averaged across all the benchmark programs.

Virtual cache implementation. It is possible to eliminate the extra TLB accesses by using a virtual L2. In a virtual L2, TLB lookups precede the L2 misses. Hence, with the address permutation interface, the missed VA is first searched in the BAT for its mapping,

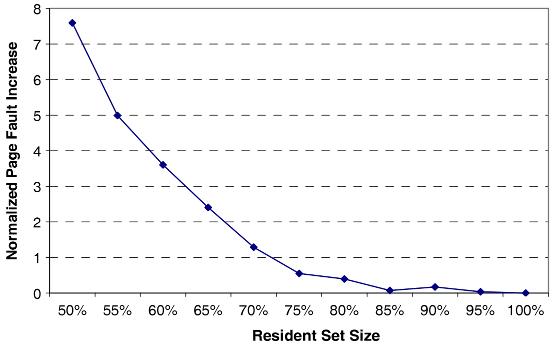


Fig. 21. Page fault increase over non-contiguous chunk in a physical cache.

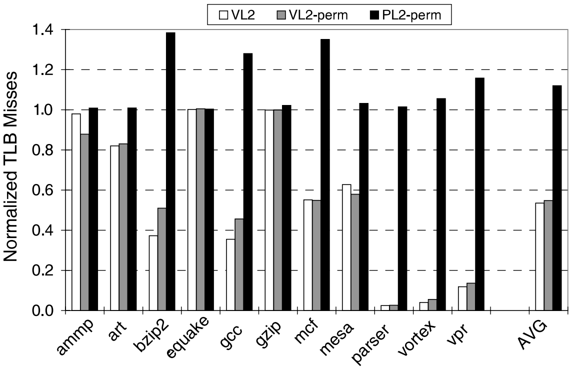


Fig. 22. Comparison of TLB misses for virtual L2, virtual L2 with permutation (our scheme), and physical L2 cache with permutation. Results are averaged across all benchmarks.

and then translated through the TLB to get the true physical address. Only one TLB lookup is necessary, as shown in Fig. 18(b). Putting the TLB below a virtual L2 cache has the advantage that its coverage on page table entry is better than the TLB in a physical L2 cache, since the former holds page entries for L2 misses [25]. Therefore, the TLB for a virtual L2 cache has much less misses than a physical L2 cache. To see this, we measured the TLB misses in Fig. 22 for a virtual L2 cache ("VL2"), our address permutation on a virtual L2 cache("VL2-perm"), and on a physical L2 cache ("PL2-perm"), normalized to the TLB misses in a baseline with a physical L2 cache.

The results show that using a virtual L2 instead of a physical L2 cache can reduce the TLB misses by half on average. When enhanced with address permutation in a virtual L2 cache, TLB misses increase only by 1% on average. However, if permutation is added to a physical L2 cache, the TLB misses are increased by 12%. These data show that a virtual L2 cache is especially useful in our scheme. Using a virtual L2 cache has some issues such as the synonyms and exception handling, so it is not widely accepted. However, those problems have been well addressed in the literature with certain hardware-assisted schemes [25–27]. In particular, a Synonym Lookaside Buffer (SLB) was proposed in [25] to dynamically translate synonyms to its corresponding main address. The main address acts as a unique identifier for all pages in a synonym set, thus preventing synonyms in the cache.

Where do we fit? In conclusion, our scheme can be implemented in both physical and virtual caches. If the infrastructure of virtual caches is already developed, our scheme is best fit in virtual caches. Because the address translation is done deeper in the memory hierarchy, the frequency of TLB accesses is reduced, and the TLB coverage is better than a physical cache. Even if we add permutation on top of it, no second TLB access is necessary, and there is only modest increase in TLB misses. In a system with physical caches, we could also achieve lower TLB misses and avoid the second address translation, as long as there is plenty of free, non-fragmented memory to support a physically contiguous chunk. Even if the chunk is not contiguous, our scheme only incurs around 12% more TLB misses, while generating much less page faults when the system is heavily loaded with very limited free memory.

5.4. Compatibility with multi-core architectures

In this paper, our design mainly targets single-core chip architectures. As technology advances to nanoscale dimensions, chip multiprocessors, a.k.a. multi-core processors start to become the norm of future processor architecture. The memory address traces exposed on the memory bus between a multi-core processor and the memory contains information from multiple cores. This makes it more difficult for an attacker to infer the information of the code on a particular core. However, such complexity is not computationally high. Therefore, it is still not safe to leave the address traces in clear text.

The technique we propose for single-core processors can be adapted to multi-core processors as well. There might be multiple memory controllers for a multi-core processor, each corresponding to a distinct memory address space. Each memory controller should contain one permutation unit as shown in Fig. 17. Hence, each permutation unit is responsible for permuting the addresses within the subspace of the memory controller to achieve protection. The bit vectors and BAT in different controllers store different information now, but the algorithm carried is the same as in the single-core scenario. Hence, although the studies and experiments performed in this paper focus only on single-core processors, the design and principle can be applied to future multi-core architectures as well.

6. Evaluation

For all the data shown earlier in the paper, we used the SimpleScalar Tool set 3.0 [1] to run 11 SPEC2K benchmark programs. All programs were simulated for 1.1 billion instructions. The parameters used are listed in Table 1.

We have shown the reductions of both the memory traffic and the page faults earlier in Figs. 15 and 16. This is mainly due to the removal of memory page sweeps during each permutation, and the reduction in the total number of permutations due to chunk-level permutations shown in Fig. 12. In this section, we focus on the performance results and the memory energy consumption. Since SimpleScalar is not a full-system simulator, we cannot model the page fault handling accurately. We assume the working set of a program fits entirely in the memory, so the results only reflect the differences of memory traffic increase.

We implemented our chunk-level permutation scheme, the HIDE scheme, and the Shuffle scheme. In Sections 6.1 and 6.2, we assumed perfect auxiliary on-chip storage for the chunk info table in our scheme, the page info record in the HIDE scheme, and the block address table in the Shuffle scheme. In Section 6.3, we implemented the chunk info table cache that only has limited on-chip storage, and compared it against the one with a perfect on-chip storage.

6.1. Memory energy consumption

We compare the memory energy consumptions for different schemes to show the impact of memory access increase on its total energy. Note that the total energy may not be proportional to the total access numbers because burst reads and writes consume

Table 1
Simulation parameters.

Clock freq.	1 GHz	Unified L2	1 MB, 4way, 32 B, 12 cycle
Decode/issue/commit width	8/8/8	L1 I- and D-cache	8 kB, 32 B direct-map, 1 cycle
RUU/LSQ size	128/64	Memory latency	80 (critical), 5 (inter) cycles
Fetch queue	32 entries	Memory bus	200 MHz, 8-byte wide
TLB miss	30 cycles	Chunk size	4 kB–64 kB

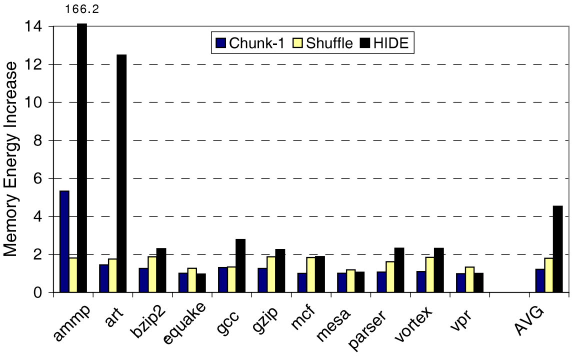


Fig. 23. Memory energy consumption increase normalized to the base.

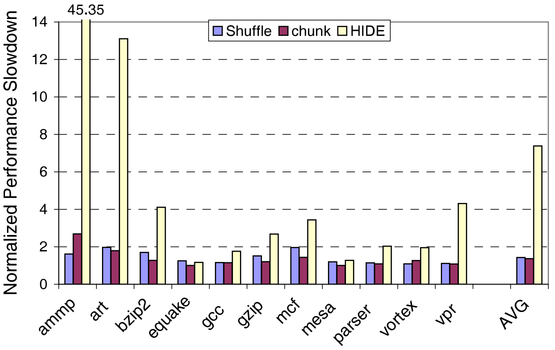


Fig. 24. Performance comparison assuming sequential memory accesses.

less energy than sparse accesses. The energy is also a function of internal banking and row buffering. We used a detailed trace driven DRAM simulator [7,6] that implemented a power model for SDRAM, DDRSDRAM and DDR2. Due to the simulation speed, the traces were generated over 100 million instructions after fast-forwarding one billion instructions and the chunk size is set to one page (Chunk-1). The 16-page chunk size (Chunk-16) we choose has even smaller memory traffic and therefore, consumes less energy than Chunk-1. We simulated our benchmarks using the DDR2 specification [20] for a 1 Gbit memory with 1 rank, each rank having 5 chips with the 5th being the ECC, 32-bit interface with a total bandwidth of 2.67 GB/s running at 667 MHz on a 4 GHz processor. Hence, our memory model projects into future processor architectures that will have high bandwidth requirement.

As expected, HIDE and Shuffle consume more energy on average than our scheme. As shown in Fig. 23, on average, HIDE and Shuffle increase energy consumption level by a factor of 4.53 and 1.79 respectively. In contrast, our scheme shows a modest increase in energy consumption of 1.21× the baseline. In the case of benchmark ammp, our scheme incurs higher memory traffic than Shuffle when running only over 100 M instructions, thus resulting in an increase of memory energy consumption. However, in the same benchmark HIDE increases energy by a factor of 166. This is because the trace for ammp during the collected interval shows much higher memory demand than the overall results (68.85× in Fig. 15) for 1.1 billion instructions.

6.2. *Performance discussion*

The evaluation of the performance benefits of our scheme varies with several parameters. Since the differences of the three schemes mainly lie in the bus traffic and the memory accesses, the report of the total execution time of a program depends on how well those components are modeled in the simulation tool. For example, the SimpleScalar 3.0 tool set adopts an unlimited write buffer, an ideal bus with unlimited outstanding requests and an overly simplified memory latency estimation. Those may result in a more optimistic evaluation as queuing effects, memory bank scheduling, bus transaction control overhead, etc. are all left out. To see different ways of how those aspects can impact the final performance reports, we experimented with three different settings: (1) using the same experimental methodology as in HIDE: the default SimpleScalar parameters with a *critical block first* policy, i.e., the critical block can always precede the other memory requests and come back in time; (2) using the same setting as in (1) except for a memory latency of 300 cycles; and (3) using a serialized atomic bus which performs atomic reads for blocks and each block is a transaction.

We found that in setting (1) the average performance slowdowns for our scheme, the Shuffle scheme, and the HIDE scheme are 0.51%, 0.5% and 0.82% respectively. There are hardly any differences among the three schemes despite the fact that their memory demands are dramatically different. This is because the assumption of critical-block-first makes all memory reads return in about the same time, leading to very close performance results. In setting (2), the average slowdowns are 2.59%, 3.07% and 2.93% for ours, Shuffle and HIDE respectively. As we can see that even each scheme itself is slowed down because of the longer memory latency, across different schemes the differences are still small due to the same reason as in setting (1). In setting (3), we see significant variations as plotted in Fig. 24.

On average, the Shuffle and our chunk scheme show a 1.43× and 1.36× slowdown respectively. However, the HIDE scheme suffers a 7.38 fold degradation. The main reason for this set of data to be very different from previous ones is that all memory and bus requests are serialized, and critical blocks are not reordered to precede any requests. Allowing the critical blocks to always precede other accesses requires that all the queues along the path have the ability to perform reordering and dependence checking. Not allowing them to bypass others puts a rather constrained limit on the performance. As we can see, this is the dominant performance factor in HIDE as many memory reads hit permutation traffic.

We want to stress that the above data only serve the purpose of showing that system variations can lead to vastly different performance results. Thus, making a conclusion on a single setting may not be entirely appropriate. Nevertheless, a design that generates low memory traffic and page faults is always desirable.

6.3. *Performance discussion in a more realistic setting*

In this section, we changed some of the simulation parameters to model a more realistic machine. We also assume there is limited

Table 2
More realistic simulation parameters.

Decode/issue/commit width	4/4/4	Memory latency	300 cycles
RUU/LSQ size	64/32	Bit vector cache	1 kB, fully assoc, 32 B
Fetch queue	16 entries	BAT cache	8 kB, fully assoc, 32 B

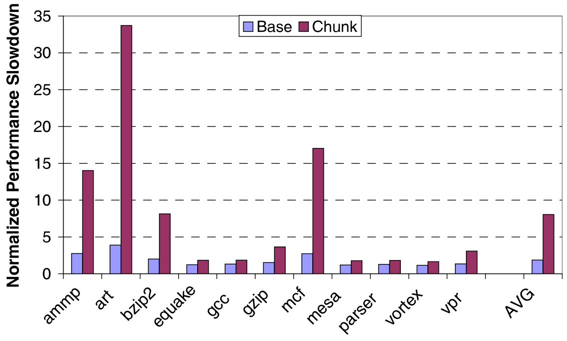


Fig. 25. Performance comparison with a more realistic machine model.

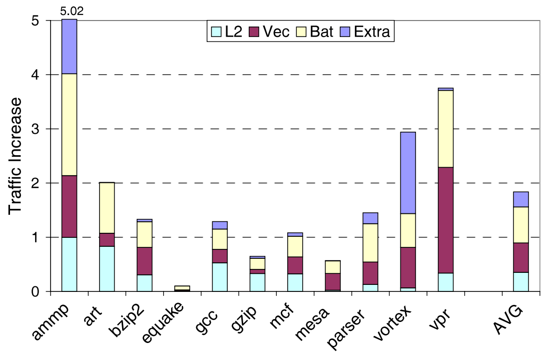


Fig. 26. Breakdown of memory traffic increase factor.

on-chip storage for the chunk information table. Table 2 lists the parameters that are different from those listed in Table 1.

As explained in Section 5.1, the storage overhead for the Chunk Information Table (CIT) is about 5% of the memory. Hence, it is more practical to cache CIT instead of keeping the entire table on-chip. The complete CIT and BAT should be stored in memory with encryption. Previously proposed encryption and authentication techniques [9,17,31,29,32,34] can be used for data exchanges between the on-chip cached part and off-chip complete tables. In our settings, the L2 cache block size is 32 bytes. For a 64 kB chunk of 16 pages, each block needs two bits for the status, and eleven bits for the mapped block address, so we use a Bit vector cache of 1 KB and a BAT cache of 8 kB to store these information. The CIT is split into two separate caches so that the bit vectors and the block mappings can be accessed independently by the address generation unit and the permutation unit as shown in Fig. 17.

Fig. 25 compares our chunk scheme with the base scheme where there is no protection mechanism. Both bars are normalized to the base scheme using parameters from Table 1. As expected, both schemes suffer more from a realistic machine model because of fewer resources and longer memory access latency. The performance overhead increased to a factor of 1.86 for the base scheme and 8.05 for our scheme on average. However, under the same realistic setting, our scheme is only 3.27 times slower than the baseline.

Fig. 26 breaks down the increased memory traffic to better show the sources of performance slowdown. We categorize them into four groups and compare each group against the original memory traffic which includes both L2 read misses and L2 write-backs. In Fig. 26, “L2” stands for the write-backs of non-dirty blocks. In order to shuffle the blocks around, we need to write-back all blocks, even if they are not dirty. How much additional traffic we get depends on the replacement rate of the L2 cache. For example, “ammp” has a rather high replacement rate of 86.81%. Because of such bad locality, blocks are constantly read on-chip and replaced, so the write-backs of non-dirty blocks almost double the traffic, and the ratio of “L2” in the first bar almost hit 1 (100%). While for benchmarks that have good locality, such as “equake”, “mesa” and “vortex”, the traffic increase due to L2 cache block replacements is less than 10%. On average, “L2” brings 35% more traffic.

The next two groups are “Vec” and “Bat”, which stand for the additional Bit vector/Bat cache read misses and write-backs. They bring about 54% and 66% more traffic due to the frequency of accesses and their limited size. The higher the hit ratio, the lower the extra traffic. For example, “equake” has a hit ratio of 99% in the Bit vector cache and 97% in the Bat cache, so we can barely see the additional traffic in the chart. In our settings, one Vec block of 32 bytes covers bit vectors for 128 L2 cache blocks, while one Bat block of 32 bytes covers block mappings for 23 L2 blocks. The traffic increase is determined by the memory footprint of the benchmark and the size of our auxiliary caches. If the memory footprint is large but the Bit vector and Bat caches are really small, then there will be a considerable increase of memory traffic.

The last group “Extra” stands for extra traffic due to padding blocks. As discussed in Section 4.2, there should be sufficient number of blocks involved in each permutation to ensure the quality of the randomization. The traffic increase relies on the spatial locality of the program. We have seen in Fig. 11 that it is hard to satisfy the permutation need for “ammp” and “vortex”, so Fig. 26 shows a lot of extra traffic for these two benchmarks. However on average, only 28% more traffic are brought by the extra padding blocks.

In total, our scheme brings about 2 times more traffic, which leads to the performance overhead of a factor of 3.27. For all of our experiments, we assume a 300-cycle permutation latency due to accesses to the on-chip CIT upon each permutation.

7. Related work

Apart from the address sequence randomization techniques [37,10,11,36], there are some other techniques that aim at protecting the control flow graph (CFG) of a program as well. Earlier attempts have taken a software obfuscation approach to transform a code into a form that is harder to reverse engineer [5]. However, it is theoretically uncertain whether a generated transformation can ensure a required level of protection, as studied and proved in [2,33]. In the world of embedded computing, protecting the program runtime execution trace has been adopted in commercial products since many embedded processors are used in financial applications in which secrecy is highly required (e.g., DS5000 and DS5002FP by Dallas Semiconductor [8]). The protection is done through encrypting the off-chip address and data buses, which unfortunately creates a fixed mapping for addresses that fails to obfuscate the CFG of a program [16].

8. Conclusion

In this paper, we propose an efficient address permutation scheme for protecting the information leakage from the address bus under physical tampering. Our technique addresses two main issues of the previously proposed HIDE scheme: the excessive memory accesses per permutation and redundant permutations. We also avoid the large number of page faults that incurred in the Shuffle scheme. On average, our scheme reduces the memory traffic in HIDE from 12× to 1.88×, and brings the memory energy consumption from 4.53× down to 1.21×. At the same time, we reduced the number of pages faults to only 1/6 of the Shuffle scheme even when the resident set size drops to 50%.

Acknowledgments

The first author was supported in part by NSF CAREER 0747242, CCF-0734339, CNS-0720595, and Intel. The third author was supported in part by NSF CAREER 0641177 and CNS-0720595. The fourth author was supported in part by NSF grant CCF-0729071 and the fifth author was supported in part by DOE Early CAREER PI Award and NSF CAREER 0644096.

References

[1] T.M. Austin, The SimpleScalar toolset, 2003. http://www.simplescalar.com.
[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S.P. Vadhan, K. Yang, On the (im)possibility of obfuscating programs, in: CRYPTO'01: The 21st Annual International Cryptology Conference on Advances in Cryptology, 2001, pp. 1–18.
[3] R. Buyya, 2002. http://www.buyya.com/ecogrid.
[4] S. Chhabra, B. Rogers, Y. Solihin, M. Prvulovic, Making secure processors os- and performance-friendly, ACM Transactions on Architecture and Code Optimization 5 (4) (2009) 111–144.
[5] C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations, Tech. Rep. 148, University of Auckland, 1997.
[6] V. Cuppu, B. Jacob, B. Davis, T. Mudge, A performance comparison of contemporary DRAM architectures, in: ISCA '99: The 26th Annual International Symposium on Computer Architecture, 1999, pp. 222–233.
[7] V. Cuppu, B. Jacob, B. Davis, T. Mudge, High-performance DRAMs in workstation environments, IEEE Transactions on Computers 50 (11) (2001) 1133–1153.
[8] Dallas semiconductor, DS5002FP secure microprocessor chip, 2006. http://datasheets.maxim-ic.com/en/ds/DS5002FP.pdf.
[9] B. Gassend, G.E. Suh, D. Clarke, M. van Dijk, S. Devadas, Caches and hash trees for efficient memory integrity verification, in: HPCA '03: The 9th International Symposium on High-Performance Computer Architecture, 2003, pp. 295–306.
[10] O. Goldreich, Towards a theory of software protection and simulation by oblivious RAMs, in: STOC'87: The 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 182–194.
[11] O. Goldreich, R. Ostrovsky, Software protection and simulation on oblivious RAMs, Journal of the ACM 43 (3) (1996) 431–473.
[12] A. Huang, Hacking the Xbox, No Starch Press, 2003.
[13] Intel Corporation, Intel 64 and IA-32 architectures software developer's manual, vol. 3a, November 2006. http://www.intel.com/design/processor/manuals/253668.pdf.
[14] P. Kocher, Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems, in: CRYPTO'96: The 16th Annual International Cryptology Conference on Advances in Cryptology, 1996, pp. 104–113.
[15] M. Kuhn, The TrustNo1 cryptoprocessor concept, Tech. Rep. CS555 Report, Purdue University, 1997.
[16] M.G. Kuhn, Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP, IEEE Transactions on Computers 47 (10) (1998) 1153–1157.
[17] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, Architectural support for copy and tamper resistant software, in: ASPLOS-IX: The 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000, pp. 168–177.
[18] D. Lim, J.W. Lee, B. Gassend, G.E. Suh, M. van Dijk, S. Devadas, Extracting secret keys from integrated circuits, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 13 (10) (2005) 1200–1205.
[19] C. McClure, Software Reuse Techniques: Adding Reuse to the System Development Process, Prentice-Hall, Inc., Upper Saddle River, NJ, 1997.
[20] Micron technology, 2007. http://www.micron.com.
[21] modchip.com, 2007. http://www.modchip.com.
[22] A. Nanda, K.-K. Mak, K. Sugavanam, R.K. Sahoo, V. Soundararajan, T.B. Smith, MemorIES: A programmable, real-time hardware emulation tool for multiprocessor server design, in: ASPLOS-IX: The 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000, pp. 37–48.
[23] D.A. Osvik, A. Shamir, E. Tromer, Cache attacks and countermeasures: The case of AES, in: CT-RSA'06: RSA Conference 2006, Cryptographers' Track, 2006, pp. 1–20.
[24] C. Percival, Cache missing for fun and profit, in: BSDCan 2005, 2005. http://www.daemonology.net/papers/htt.pdf.
[25] X. Qiu, M. Dubois, Towards virtually-addressed memory hierarchies, in: HPCA'01: The 7th International Symposium on High-Performance Computer Architecture, 2001, pp. 51–62.
[26] X. Qiu, M. Dubois, Tolerating late memory traps in dynamically scheduled processors, IEEE Transactions on Computers 53 (6) (2004) 732–743.
[27] X. Qiu, M. Dubois, Moving address translation closer to memory in distributed shard-memory multiprocessors, IEEE Transactions on Parallel and Distributed Systems 16 (7) (2005) 612–623.
[28] W. Shi, H.-H.S. Lee, Accelerating memory decryption and authentication with frequent value prediction, in: ACM International Conference of Computing Frontiers, 2007, pp. 35–46.
[29] W. Shi, H.-H.S. Lee, M. Ghosh, C. Lu, A. Boldyreva, High efficiency counter mode security architecture via prediction and precomputation, in: ISCA'05: The 32nd Annual International Symposium on Computer Architecture, 2005, pp. 14–24.
[30] W. Shi, H.-H.S. Lee, C. Lu, M. Ghosh, Towards the issues in architectural support for protection of software execution, SIGARCH Computer Architecture News 33 (1) (2005) 6–15.
[31] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, S. Devadas, AEGIS: Architecture for tamper-evident and tamper-resistant processing, in: ICS'03: The 17th International Conference on Supercomputing, 2003, pp. 160–171.
[32] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, S. Devadas, Efficient memory integrity verification and encryption for secure processors, in: MICRO-36: The 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003, pp. 339–350.
[33] C. Wang, A security architecture for survivability mechanism, Ph.D. Thesis, University of Virginia, October 2000.
[34] J. Yang, Y. Zhang, L. Gao, Fast secure processor for inhibiting software piracy and tampering, in: MICRO-36: The 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003, pp. 351–360.
[35] Y. Zhang, L. Gao, J. Yang, X. Zhang, R. Gupta, SENSS: Security enhancement to symmetric shared memory multiprocessors, in: HPCA'05: The 11th International Symposium on High-Performance Computer Architecture, 2005, pp. 352–362.
[36] X. Zhuang, T. Zhang, H.-H. Lee, S. Pande, Hardware assisted control flow obfuscation for embedded processors, in: CASES'04: The 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, 2004, pp. 292–302.
[37] X. Zhuang, T. Zhang, S. Pande, HIDE: An infrastructure for efficiently protecting information leakage on the address bus, in: ASPLOS-XI: The 11th International Conference on Architectural Support for Programming Languages and Operating Systems, 2004, pp. 72–84.

Jun Yang received the Ph.D. degree in Computer Science from the University of Arizona, 2002. She is an associate professor of Electrical and Computer Engineering at University of Pittsburgh. Jun Yang's research interests include power and thermal management of microprocessors, three-dimensional processor stacking technology, non-volatile memory technologies, network-on-chip, reliability improvement, and cache management in chip multiprocessors. Dr. Yang has co-authored a paper that won the Best Paper Award in ICCD-2007, and a paper nominated for Best Paper Award in HPCA 2009. She is a recipient of NSF CAREER award in 2008. She is a member of ACM and IEEE.

Lan Gao received the Ph.D. degree in Computer Science from University of California, Riverside, in 2007. She is a Member of the Technical Staff at VMware, Inc. She worked on VMware vCenter Converter in the past, and is now working on VMware High Availability (HA). Her research interests include architectural support for security and trusted computing, virtualization, high availability computing.

Youtao Zhang received the Ph.D. degree in computer science from the University of Arizona in 2002. He is an assistant professor in Computer Science Department, University Pittsburgh. His research interests are in the areas of computer architecture, compilers, and system security. He is the recipient of US NSF CAREER Award in 2005, the distinguished paper award of ICSE'2003, the most original paper award of ICPP'2003. He is a member of the ACM and the IEEE.

Marek Chrobak is a Professor of Computer Science and Engineering at the University of California at Riverside. He was born and studied in Poland, obtaining his M.S. and Ph.D. degrees in Computer Science from Warsaw University in 1985. His current research and teaching interests include design and analysis of algorithms, combinatorial optimization, on-line computation, job scheduling, and bioinformatics.

Hsien-Hsin S. Lee is an Associate Professor of the School of Electrical and Computer Engineering at Georgia Tech. He received his Ph.D. degree in Computer Science and Engineering from the University of Michigan, Ann Arbor. His research interests include computer architecture, 3D IC, low-power VLSI, and cyber-security. Previously, he was a processor architect at Intel Corporation and later the architecture manager of StarCore DSP Technology Center of Agere systems and Motorola. Dr. Lee received DOE Early CAREER PI Award and NSF CAREER Award. He holds 4 US patents and is a senior member of the IEEE and the ACM.