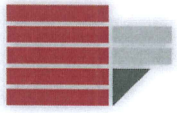
UNIVERSITÀ DELLA CALABRIA



Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica (DIMES)

Dottorato di Ricerca in

Information and Communications Technology

ciclo
XXXI

Hardening the Security of Modern Operating Systems Against Side-Channel and Rowhammer Attacks

---

Settore Scientifico Disciplinare ING-INF/05

Coordinatore:
Prof. Felice CRUPI

Supervisore:
Prof. Andrea PUGLIESE

Dottorando:
Marco OLIVERIO

Abstract

Advancements in exploitation techniques call for the need of advanced defenses. Modern operating systems have to face new sophisticate attacks that do not rely on any programming mistake, rather they exploit leaking information from computational side effects (*side-channel attacks*) or hardware glitches (*rowhammer attacks*). Mitigating these new attacks poses new challanges and involves delicate trade-offs, balancing security on one side and performance, simplicity, and compatibility on the other. In this disseration we explore the attack surface exposed by *page fusion*, a memory saving optimization in modern operating systems and, after that, a secure page fusion implementation called *VUsion* is shown. We then propose a complete and compatible software solution to rowhammer attacks called *ZebRAM*. Lastly, we show *OpenCAL*, a free and general libray for the implementation of Cellular Automata, that can be used in several security scenarios.

Acknowledgements

I would like to thank my Supervisor prof. Andrea Pugliese for the encouragement and the extremely valuable advice that put me in a fruitful and exciting research direction. A hearty acknowledgment goes to Kaveh Razavi, Cristiano Giuffrida, Herbert Bos and all the VUSec group of the Vrije Universiteit of Amsterdam. I felt at home from day one and I found myself surrounded by brilliant researchers and good friends.

Contents

To Mom and Dad...

Introduction

The Morris worm in 1988 [146], Zatko private note on buffer overflow [173] and, above all, Elias Levy's (AlephOne) phrack article on stack-based exploitation [108] published in 1996, made extremely clear that bugs were more threatening than previously thought. They showed how to abuse common memory bugs to get complete control of computer systems.

As writing complex software without making any mistake is impossible (a mistake can even hide in the specifics or in the verifier of a formally verified system), the research community focuses its efforts on preventing the exploitation of bugs, more than bugs themselves.

Non-executable memory [9, 124], bound checking [6, 102], address space layout randomization [136, 22], and control flow integrity [1] are just a few of the effective and already deployed techniques that, even if not bullet-proof, make bug exploitation way harder than it was a decade ago.

Unfortunately, since the second half of 2010s, new types of attack techniques were discovered. They use a completely different approach: they do not rely on any software bug. This means that even a (hypothetical) bug-free software is at danger and previous mitigation techniques are useless.

The first of these techniques mines confidentiality of a computer system or, in other words, allows unauthorized users to learn sensitive data. Even if breaking confidentiality looks less severe than complete system compromise, total compromise can easily follow either by login credentials disclosure or, as we will shortly see, by combining it with other techniques.

When developing complex software (or hardware) components, we can be tempted to abstract away "irrelevant" properties and to focus only on their functional behavior. Bug-free implementations guarantee that unauthorized users cannot access a component's secret state using the information exposed by its functional behavior. We should not make the mistake, though, of assuming that determined attackers will constrain themselves to look just at the functional behavior. A careful observer can, indeed, focus on overlooked aspects, such as the time, the memory, or the energy spent doing a specific computation. If a correlation exists between these aspects and the secret, then attackers can measure them and then use them to infer the secret state, breaking the confidentiality of our "bug-free" implementation. These kinds of attack are called *side-channel* attacks and can affect components at every level of the hardware and software stack.

While initially the research community focused on side-channel attacks targeting cryptographic algorithms [99], in the last decade the community discovered numerous side-channel attacks to applications [172, 111], operating systems [74, 78, 63, 64] and at the micro-architecture level [64, 74, 63], such as the now famous *meltdown*/*spectre* bug [98, 110].

Researchers discovered another threat around 2016 that was dubbed *rowhammer effect* [96]. The effect is a hardware glitch that affects common DRAM chips. The root cause is a physical electrical charge leak between DRAM memory cells that can be abused to flip memory bits. While initially considered just a mere reliability issue, today the literature gives us numerous examples of security breaches using the rowhammer effect [171, 26, 143, 67].

Hardware glitches like rowhammer share one thing with side-channel attacks: they also do not rely on any software bug. Moreover, they offer an important missing piece in bug-free software exploitation: while side-channels can be used to break confidentiality (read primitive),

hardware glitches can be used to corrupt data (write primitive). Net effect: a system can be totally compromised without a single bug.

The combination of side-channel and hardware glitches poses a severe threat to modern computer systems [143, 26]. Mitigating the resulting attacks is challenging. First of all, we need to consider the software and hardware components in all their complexity, examining every effect that can be measured by an attacker to infer secret information. Moreover, we need to put under scrutiny the interactions between different components and the sharing of resources, as they can be a source of side-channel information themselves.

We also need to preserve performance as much as possible. Software and machines employ very advanced optimizations, with the natural goal of doing computations as fast as possible. Easily optimizable instances of a computational problem can be solved faster. Unfortunately, this also means that an attacker can time the computation to infer information on the internal state of a problem instance. Another very effective optimization approach is resource sharing. While the advantages of sharing resources obviously regard optimal resource usage, it easily leads to side-channel problems. Indeed when a resource is shared between two or more distrusting parties, then they can "spy" on each other by observing side-effects on the shared resource. It is really hard to completely solve this kind of issues without sacrificing performance. Even if nowadays communities are more inclined to accept performance loss for higher security guarantees (as in linux meltdown [126] or tlbleed openbsd [131] mitigation), we should always be careful of performance issues if we want to develop practical defenses.

Mitigations also need to preserve software and hardware legacy compatibility. Indeed, while many hardware solutions were proposed to defend against side-channel and hardware glitches [96, 95, 91], many will leave an important amount of critical legacy systems undefended. Protecting them is not trivial, as often the source of the problem is in hardware, but mitigation needs to be developed in software.

Another challenge when developing effective side-channel and hardware glitches defenses is the lack of precise information on how hardware components work, as manufactures are reluctant to make this information publicly available. That means that, in order to develop effective defenses, we often need to reverse engineer that information.

Lastly, mitigation should be transparent, as software and operating systems should work unmodified, with minimal or no additional manual software modification.

In Chapter 1 of this thesis we propose *VUsion* [130], a defense against side-channels in the page fusion engines of commodity operating systems. These side-channels allow attackers to break the confidentiality of a machine and, combined with the rowhammer effect, to completely compromise a system. After a complete analysis of the attack surface, a principled approach to mitigate the issue, while preserving performance, is shown.

In Chapter 2, we propose *ZebRAM* [101]: the first complete and transparent software defense against rowhammer attacks. We build upon past research [138, 154] to reverse engineer complicated hardware mappings between physical and DRAM address spaces. With this knowledge at hand, we partition the DRAM memory in a way that makes bitflips harmless. In the process, the operating system loses the ability to directly use half of the memory, but in ZebRAM we repurpose it as a compressed RAM device used as swap. The solution has an overhead that is a function of the working set size.

As different security techniques, especially in cryptography, can profit from cellular automata[116, 58, 167], in Chapter 3 we show *OpenCAL* [57], a parallel framework to develop cellular automata in a efficient and portable way.

Chapter 1

VUsion

To reduce memory pressure, modern operating systems and hypervisors such as Linux/KVM deploy page-level memory fusion to merge physical memory pages with the same content (i.e., *page fusion*). A write to a fused memory page triggers a copy-on-write event that unmerges the page to preserve correct semantics. While page fusion is crucial in saving memory in production, recent work shows significant security weaknesses in its current implementations. Attackers can abuse timing side channels on the unmerge operation to leak sensitive data such as randomized pointers. Additionally, they can exploit the predictability of the merge operation to massage physical memory for reliable Rowhammer attacks. In this chapter, we present VUsion, a secure page fusion system. VUsion can stop all the existing and even new classes of attack, where attackers leak information by side-channeling the merge operation or massage physical memory via predictable memory reuse patterns. To mitigate information disclosure attacks, we ensure attackers can no longer distinguish between fused and non-fused pages. To mitigate memory massaging attacks, we ensure fused pages are always allocated from a high-entropy pool. Despite its secure design, our comprehensive evaluation shows that VUsion retains most of the memory saving benefits of traditional memory fusion with negligible performance overhead while maintaining compatibility with other advanced memory management features.

1.1 Introduction

On modern systems, different processes [26] or co-hosted virtualized environments [11, 35, 17] tend to have many memory pages that store similar data. To remedy this situation, operating systems and hypervisors perform *page fusion* by periodically scanning memory to find pages with the same content and merging them by keeping a single read-only copy. Writing to a shared page from either party triggers an unmerge by performing copy-on-write into a private copy. Given its memory saving benefits, page fusion has been applied to several classes of real-world systems, ranging from co-hosted virtual machines (VM) [11] and containers [86] in the cloud to desktop [26] and mobile [72] systems.

Unfortunately, existing page fusion systems are insecure. As we shall see, attackers can abuse the *unmerge* operation to build side-channel attacks to leak sensitive data, or the *merge* operation to construct reliable and deterministic Rowhammer attacks that flip bits in vulnerable memory modules. As a result, vendors either disable page fusion by default (e.g., Microsoft [48] or VMWare [161]) or sacrifice security in favor of capacity (e.g., Intel Clear Containers [86]). In contrast, we show that it is not only possible to fuse pages securely, but even retain its memory saving benefits and compatibility without sacrificing performance. This is possible due to a key observation: benefits of page fusion mostly come from idle pages in the system. Hence, we can apply heavy-weight mechanisms to secure page fusion on idle pages, while preserving performance by not fusing pages in the working set.

Attacking page fusion There are two classes of attacks that abuse page fusion: information disclosure and physical memory massaging [143] (i.e., preparing the state of physical memory for corrupting target data using a DRAM bit flip [96]). Known information disclosure attacks [18, 26, 88, 135, 153, 170] abuse the slow unmerge to detect whether another copy of a certain memory page exists in the system. Some attacks even use advanced versions of the side channel to break address space layout randomization (ASLR) in the browser from JavaScript [26] or across VMs [18]. The recent Flip Feng Shui attack [143], on the other hand, abuses the predictability of the merge operation to control where sensitive data is stored in physical memory. In cloud environments, Flip Feng Shui allows an attacker VM to compromise a victim VM by performing a Rowhammer attack on (fused) RSA public keys in memory.

With a careful security analysis of current implementations of page fusion, we realize that the attack surface is much larger than previously assumed, raising even more challenges to build secure page fusion systems. On the information disclosure side, it is possible to detect merge events by observing changes in the physical memory or in the virtual memory translation data structures, using a variety of side channels on shared system resources (e.g., last-level cache). This allows an attacker to disclose sensitive data *by just reading from memory*. On the Flip Feng Shui side, it is possible to abuse the predictability of memory reuse to perform reliable Rowhammer attacks. This allows an attacker to mount Flip Feng Shui *even when merged pages are backed by new page allocations*. We show that while the implementation of page fusion in Windows is protected against plain Flip Feng Shui since new pages are allocated during merge, the attackers can overcome this hurdle by exploiting memory reuse patterns for fused pages.

Secure page fusion Our security analysis helps us derive a number of design principles that, once respected, can protect against all existing and future attacks on page fusion. To stop an attacker from disclosing sensitive information, a secure page fusion system should enforce the same behavior for all pages in the system, whether they are merged or not. We call this principle Same Behavior (**SB**). Every time we are making a decision on whether to merge a page or not, we remove all accesses to that page. The next access to this page, regardless of its merging status, results in a page fault. To stop Flip Feng Shui, we should always allocate a random physical page for backing the page that is a candidate for merging. We call this principle Randomized Allocation (**RA**).

While **RA** can be implemented with negligible overhead, **SB** can be expensive in terms of performance due to the increased number of page faults and can reduce fusion benefits since memory pages need to be unmerged even when read. Fortunately, neither is a problem in practice: a simple working set estimation can restrict page fusion to idle pages and dramatically reduce the additional number of page faults. This strategy has a small impact on memory savings, since, as we show, most benefits of page fusion come from idle pages. As a result, VUsion, our secure page fusion system built on top of the Linux kernel, provides similar benefits in terms of saving memory with minimal performance overhead (e.g., 2.7% on SPEC CPU2006 and 0.4% on memcached) compared to the default insecure implementation in the Linux kernel. We further address the non-trivial challenge of keeping VUsion compatible with popular memory management features of the Linux kernel such as transparent huge pages (THPs) [109, 21, 73, 162, 104].

Contributions We make the following contributions:

- The first study of page fusion in recent Windows operating systems, which reveals a different design than the widely studied Kernel Same-page Merging (KSM) in Linux (§1.2).

- The first study of previously known attack vectors on page fusion complemented by new attack vectors, which we use to draw up principles for secure page fusion (§1.4 and §1.5).
- The design and implementation of VUsion, a Linux-based secure page fusion system which follows these principles (§1.6 and §1.7).
- A comprehensive evaluation of security, performance and memory savings of VUsion when compared to KSM. Our results demonstrate that VUsion's design improves the security of KSM and only marginally reduces memory savings while preserving performance (§1.9).

1.2 Page Fusion

Page fusion is often used in situations where it is not possible to directly share memory that originates from the same content. For example, while it is possible to share pages for libraries across different processes *inside* a VM, the same is not possible *across* VMs. Hence, to find memory pages with the same content, a page fusion system should periodically scan the memory.

Once pages with duplicate content are found, only one copy is kept, and all the page-table entries (PTEs) of the sharing parties are updated to point to this copy without the write permission bit (i.e., they are *fused*). The duplicates can now be returned to the system. At any point in time, one party may decide to write to this (now) fused page, resulting in a page fault. To preserve the correct behavior, the system handles this situation with copy-on-write: a new page is allocated and filled with the content from the shared copy before updating the PTE of the writing party to this new page with the write permission bit so that the write can continue.

We now study how these steps are implemented in practice using two popular implementations in the Linux and Windows operating systems.

1.2.1 Linux Kernel Same-page Merging

The Linux kernel fuses memory pages in its KSM subsystem. KSM is opt-in and user processes that want page fusion should inform KSM via an `madvise` system call. The Kernel Virtual Machine (KVM) is a prime user of KSM and co-hosted virtualized environments in the cloud are an important target.

Scanning Whenever a process registers a chunk of its virtual memory to KSM, KSM first locates all virtual memory areas (VMAs) associated to this chunk. In Linux, VMAs are contiguous areas of virtual memory and the (virtual) memory pages that belong to the same VMA share certain properties such as permissions. After finding the VMAs, KSM adds them to a list containing all candidate VMAs for fusion. KSM scans this list in a round-robin fashion. Every T milliseconds, the KSM thread wakes up and scans N virtual memory pages (belonging to one or more VMAs). T and N are configurable, for example on the Linux kernel version 4.10.10, the default values are $N = 100$ and $T = 20$, i.e., 5000 pages are scanned per second. The pages are merged in-line with the scan.

Merging To detect sharing opportunities, KSM uses two red-black trees, called *stable* and *unstable*, that use the contents of the pages to balance themselves. The stable tree contains fused pages that are made read-only and are (often) shared between multiple parties. The pages in the unstable tree, however, are not protected against writes and their contents may

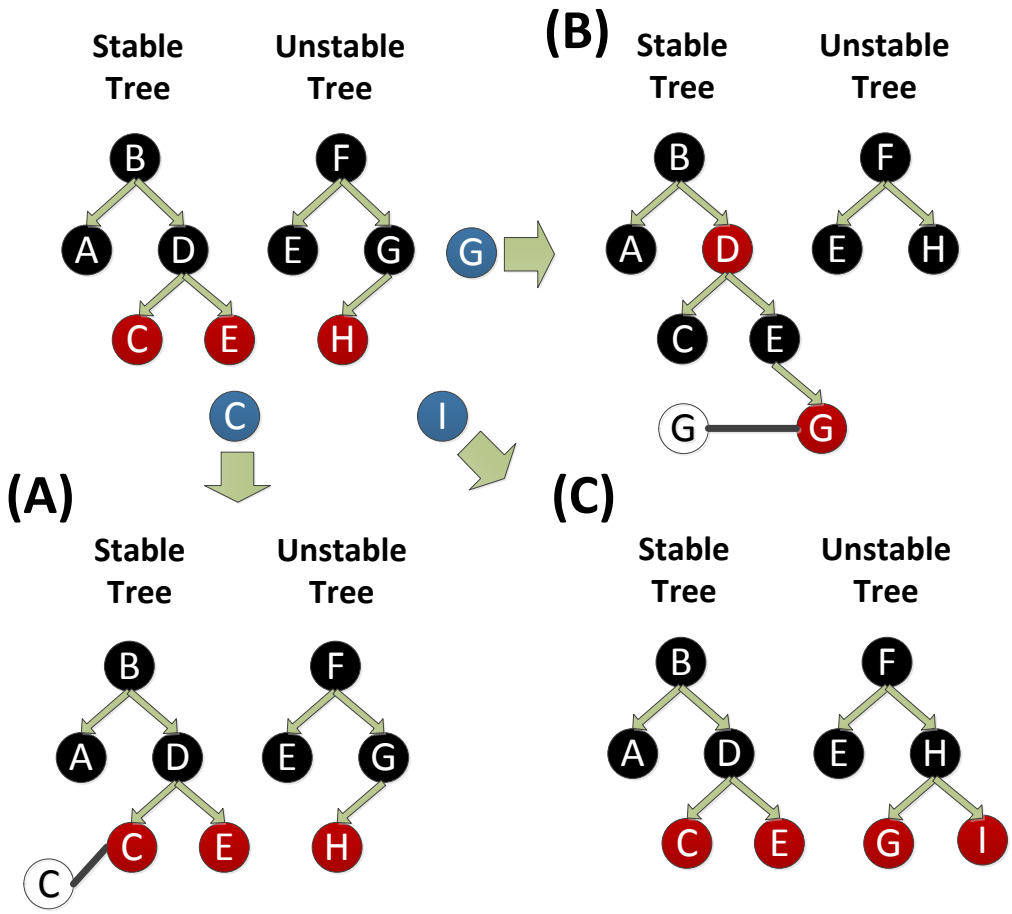


FIGURE 1.1: Modifications to the KSM red-black trees during merging. Blue circles are the pages that are being considered for fusion.

change. Hence, the tree is not always perfectly balanced, but given that every page insertion and deletion triggers rebalancing, the tree mostly maintains its balance [11].

Figure 1.1 shows how KSM finds duplicate pages in the registered VMAs. For each page that is scanned, it is first checked whether another page with the same content already exists in the stable tree. If that is the case, then the page is merged by updating the PTE of the page that is being scanned to point to the page in the stable tree without the write permission bit and releasing the page back to the system (Figure 1.1-A). If that is not the case, KSM matches the page against the unstable tree. If a match is found, it makes the page read-only and puts it in the stable tree. It also removes the match from the unstable tree and makes it point to the stable tree page, similar to the previous case (Figure 1.1-B). Finally, if KSM finds no match in either tree, it puts the page in the unstable tree (Figure 1.1-C). Note that merging implies that the physical memory of one of the pages is used for backing the shared copy between various (distrusting) parties.

Unmerging The pages in the stable tree are reference-counted (very much like any page in Linux). As soon as one party writes to a write-protected page, a copy-on-write event triggers, as mentioned earlier. This copy-on-write event drops the reference count by one. Only if the count reaches zero, the system removes the original page from the stable tree, which means that as long as there is even a single user, that page remains in the stable tree.

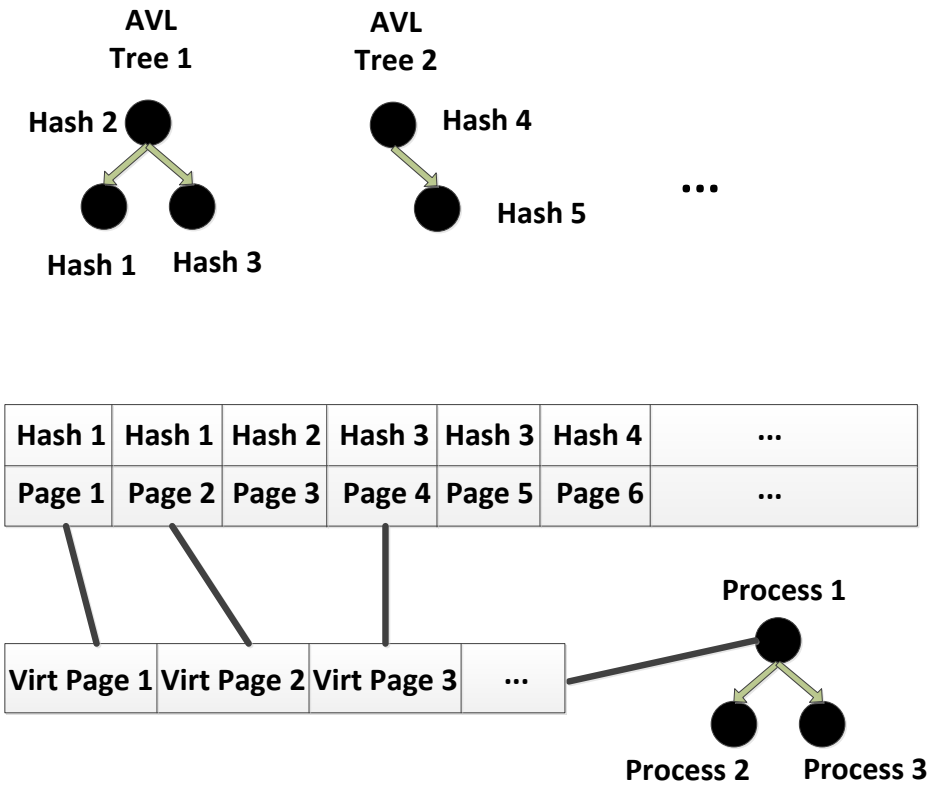


FIGURE 1.2: WPF's sorted list of hashes and processes.

1.2.2 Windows Page Fusion

Microsoft turned page fusion on by default for Windows 8.1 and later releases. Duplicate pages generated by related processes in the absence of *fork* semantics are a primary target of page fusion on Windows systems. We have reverse engineered parts of the Windows kernel to gain insight into the mechanisms that Windows Page Fusion (WPF) uses for this purpose. Note that Microsoft recently disabled WPF after the Dedup Est Machina attack [26], at the expense of memory wastage. However, it is still important to study a different (insecure) design other than KSM to derive the necessary key principles for secure memory fusion.

Scanning Compared to its Linux counterpart, WPF has no opt-in mechanism that allows user-space applications to register which memory pages can be merged. Instead, it scans all anonymous physical memory pages every 15 minutes and tries to merge as many pages as possible. WPF stores the metadata about the already merged pages in multiple AVL trees that have the same functionality as the KSM's stable tree. During each scan, WPF computes the hash of every physical page that is a candidate for merging in a list that is sorted by the hash value.

Merging With the sorted list of hashes, it is now time for WPF to start merging physical pages. For each physical page, through a reverse mapping, WPF determines the process that owns the page. Each process will have a list of these physical pages, sorted by their corresponding virtual addresses. These processes are inserted into a tree that is sorted by their memory management struct pointer (i.e., `_EPROCESS->Vm`). Figure 1.2 visualizes the relation between these structures.

WPF then performs the actual merging on a per-process basis. For each process, it first examines the content of each physical page in the process's list. If a page with the same content has been previously fused, then WPF will find it in one of the AVL trees. WPF then updates the corresponding PTE to point to the page in the AVL tree and returns the physical page back to the system. For pages that have not been previously fused, WPF checks whether there are pages with the same content. In case there are some, WPF adds a copy to one of the AVL trees, updates the relevant PTEs and returns the duplicate pages back to the system.

An important difference between KSM and WPF is that WPF allocates *new* physical pages for insertion into the AVL trees (i.e, the physical pages of the sharing parties are not used to back the fused page). Specifically, it allocates new pages using a specialized linear allocator to improve performance by not contending for the system-wide page allocator. This allocator scans the physical address space from the end and tries to reserve as many pages as necessary. If the allocator finds a page that is currently in use, it tries to steal this page from the owner. As a result, the allocated memory will be mostly contiguous, starting from the end of the physical address space. While this mechanism leads to some uncertainty in the selected page during a merge operation, we later show this design is still vulnerable to Flip Feng Shui.

Unmerging Unmerging is performed similar to KSM using copy-on-write.

1.3 Threat Model

We assume a strong threat model in line with prior work in the area [26, 143, 18], where an attacker can 1. directly interact with the page fusion system by crafting memory pages with her chosen contents, and 2. trigger bit flips using the Rowhammer vulnerability or other potentially exploitable reliability issues [94, 31]. In case page fusion is applied inside the OS, the attacker can remotely create memory pages with arbitrary contents in a malicious JavaScript application and in the case that page fusion is applied in a cloud setting, the attacker executing in a malicious VM can directly create arbitrary memory pages. The attacker pursues one of the following two goals:

- *Information disclosure*: abuse timing side channels induced by page fusion to detect fused pages in the system and disclose secrets.
- *Flip Feng Shui*: abuse physical memory massaging primitives induced by page fusion to land a target of interest into a chosen vulnerable physical page and mount Flip Feng Shui attacks.

Next, we describe how these attacks can be mounted in practice.

1.4 Known Attack Vectors

We now describe how an attacker can exploit known page fusion issues in order to achieve the goals mentioned in our threat model.

1.4.1 Information Disclosure

Existing information disclosure attacks based on page fusion are *unmerge-based*. They exploit timing side channels introduced by unmerge (or copy-on-write) events in traditional page fusion systems.

Writing to a merged page will trigger a copy-on-write event which is measurably slower than a normal write. The attacker can use this timing difference as a side channel to tell

whether a page exists in the victim. In the past, researchers used this side channel to fingerprint applications, libraries, operating systems and to build covert channels [153, 135, 88, 170].

The CAIN [18] attack brute-forces pointers of other VMs randomized by Address Space Layout Randomization (ASLR) [136] by creating many guesses for possible pointers and checking which guess gets shared with the victim VM. Brute-forcing high-entropy data in this way requires a large amount of memory and becomes noisy in a Web browser. Dedup Est Machina [26] shows that it is possible to leak high-entropy secrets in the browser by controlling the alignment of the secret, partially overwriting the secret with known data, or engaging in a birthday attack. The first two techniques allow for leaking only a part of the secret in the first fusion pass. Once a part of the secret is known, it is possible to leak another part in a subsequent fusion pass. The birthday attack relies on generating many secrets in the target process (in this case a JavaScript runtime) to increase the chance of guessing one of them.

These techniques show that page fusion can act as a weird machine [29] in the hands of an attacker able to time *unmerge* events for reading secret information from a victim process or VM.

1.4.2 Flip Feng Shui

Existing Flip Feng Shui attacks based on page fusion are *merge-based*. They exploit physical memory massaging capabilities introduced by merge events to land sensitive information in a vulnerable memory page and trigger hardware bit flips to corrupt it. Existing attacks of this kind are based on the DRAM Rowhammer hardware vulnerability. We now briefly provide some background information on how Flip Feng Shui triggers memory corruption using Rowhammer. We refer interested readers to the original Flip Feng Shui article for more information [143] .

DRAM architecture Memory is internally organized in *rows*. Depending on the DRAM architecture, each row can span a number of pages. The rows themselves consist of memory cells. Each cell is made out of a capacitor and a transistor. The transistor controls whether the cell is being read or written into and the capacitor stores the actual one bit of data. Capacitors lose charge over time and if enough charge is lost, the stored data will be lost. To avoid this, the memory chip periodically refreshes the cells in each row.

Every time the CPU needs to read data from memory, the memory chip selects the corresponding row and loads it into the *row buffer*. The row buffer acts as a cache for rows, so the CPU can keep reading from it as long as the requested addresses belong to the same row. Similar to refreshing, writing the row from the row buffer back (before opening another row) recharges the capacitors.

Rowhammer Kim et al. [96] noticed that if two rows (i.e., aggressor rows) are activated in succession many times within a refresh interval (e.g., 64 ms), some bits that are in adjacent rows (i.e., victim rows) to these aggressor rows will start to flip. This is because a small charge leaks from cells in the victim rows every time aggressor rows are loaded into the row buffer and if this happens fast enough between refresh intervals, enough charge will leak from some cells in the victim rows. This causes the value of the high bit to switch to a low bit which is observed as a bit flip from the perspective of the CPU. Reading from memory in this fashion with the aim of triggering bit flips has been dubbed as Rowhammer. A variant of Rowhammer where the aggressor rows are one row apart, called double-sided Rowhammer, is known to trigger more bit flips reliably in the victim row that is in the middle.

Attack	Issue	Abused mechanism	Attacker operation(s)	Mitigation
Copy-on-write [18, 26]	Slow write	Unmerge	Write	**SB**
Page color (new)	Physical address changes	Merge	Read or fetch	**SB**
Page sharing (new)	Sharing changes	Merge	Read or fetch	**SB**
Translation (new)	Translation changes	Merge	MMU ops	**SB**
Flip Feng Shui [143]	Predictable merge	Merge	Memory massaging	**RA**
Reuse-based Flip Feng Shui (new)	Predictable reuse	Reuse	Memory massaging	**RA**

TABLE 1.1: Summary of attacks against page fusion and how design principles used in VUsion mitigate them.

Exploitation Being able to change memory without having control over it has strong security implications. Recent studies show that it is possible to abuse these bit flips to escalate privilege by flipping bits in the PTEs [148, 159] or escape the JavaScript sandbox by flipping bits in data pointers [26]. The Flip Feng Shui attack [143] shows that it is possible to reliably circumvent the strong hardware-enforced virtual machine abstraction using Rowhammer and the merge operation of page fusion.

To mount a Flip Feng Shui attack, the attacker VM first finds memory cells that are susceptible to exploitable Rowhammer bit flips in the physical memory that backs her VM. At this point, the attacker needs to force the system to store the sensitive data of a victim VM (e.g., cryptographic keys) on the physical page with the exploitable bit flip. Page fusion makes this step extremely easy: in the case of KSM, for example, the physical memory of one the sharing parties is chosen to back the merged page. Hence, if attackers want to corrupt a page in the victim, all they need to do is write that page content on one of their own pages that is vulnerable to Rowhammer bit flips. If KSM chooses the attacker's physical page, then the victim page gets corrupted when the attacker triggers Rowhammer. The last step essentially breaks the copy-on-write semantics necessary for safe and correct behavior of page fusion.

Flip Feng Shui shows that page fusion can act as a *physical memory massaging* primitive in the hands of attackers, making it possible for them to control which physical memory pages should be used to back sensitive data through *merge events*. In turn, this allows them to reliably corrupt sensitive information from a victim process or VM.

1.5 New Attack Vectors

In this section, we describe two new classes of advanced attack vectors against page fusion. The first class targets *merge* events in page fusion to mount information disclosure attacks. The second class targets *reuse* properties of page fusion to mount Flip Feng Shui attacks. Along with the attack vectors detailed earlier, they help us derive design principles for secure page fusion that we adhered to in the implementation of VUsion.

1.5.1 Information Disclosure

Traditional page fusion is characterized by well-defined *merge* and *unmerge* events. Existing attacks exploit copy-on-write side channels associated with *unmerge* events, but we now show an attacker can also exploit several timing side channels associated with *merge* events to detect fused pages without writing to these pages.

Page color changes Page colors refer to how physical pages map on cache sets such as those of the last-level cache (LLC). For example, the Intel Xeon E3-1240 v5 processor used in our evaluation partitions its 8 MB LLC into 8192 cache sets of 16 cache lines of 64 bytes each, and each 4 KB page covers 64 cache lines in 64 cache sets. If the first cache lines of two

distinct physical pages share the same cache set, the other 63 cache lines are also guaranteed to land on the same cache sets as well. This allows us to color different physical memory pages in the system only based on their first cache line. For example, our Intel Xeon E3-1240 v5 processor has $8192/64 = 128$ different page colors.

To color a page, we first build eviction sets for all possible colors in the system. An eviction set is a sequence of memory addresses that all map to the same cache set and "covers" the cache set completely [132]. Thus, by accessing all the addresses in the eviction set, we clear out all other data from the corresponding cache set. Using a `PRIME+PROBE` attack [111, 133], it is now possible to determine the current color of a page. The attacker first primes a certain color C by accessing its eviction set. After that, the attacker reads from the target page. During the probe phase, if accessing the eviction set for C is slow, it means that the target page is of color C. By waiting for a page fusion pass to occur, the attacker can detect whether a target page has been fused if its color is no longer C. This attack assumes that a new page is allocated to back the shared copy (e.g., WPF) and is successful if the new page has a different color. That is $P_{success} = \frac{C_t - 1}{C_t}$ where C_t is the total number of colors. For example, in our testbed, $P_{success} = \frac{128-1}{128} = 0.99$. Our implementation of this attack can find eviction sets for all colors in a few minutes and detect changes in the color after a fusion pass.

Page sharing changes It is tempting to think that if we always randomize the physical location of a page considered for fusion (regardless of a merge), then allowing reads stops information disclosure while conserving performance and benefits of page fusion. Unfortunately, this design is still insecure since attackers can detect whether pages are fused over shared resources. An attack over the LLC is similar in spirit to a 1-bit version of `FLUSH+RELOAD` [172]. We first flush the target page from the cache by either executing a cache flush instruction or accessing a cache eviction set (i.e., `FLUSH`). Next, we make the victim access the secret page for which we want to check whether it is merged. Finally, we access the target page again and measure how long it takes (i.e., `RELOAD`). If the access is slow, it means that the data was not in the cache which implies that the victim did not access the page. If, on the other hand, the access is fast, it means that the victim has accessed the exact same physical location as the attack, suggesting that a merge event has occurred as a result of page fusion.

Translation changes Finally, it is also possible for an attacker to detect a merge event indirectly, by observing changes in the behavior of pages that are physically adjacent to the target page. For example, KSM breaks a huge page when merging a 4 KB page inside of it. This means that the other adjacent pages that make up the huge page now require an additional page table look up for the last translation level. As our recent AnC attack shows [74], attackers can easily observe the additional page table lookup in the LLC and detect a merge event of the target page.

1.5.2 Flip Feng Shui

At the physical memory management level, page fusion systems either *merge* two duplicate physical pages into one (and discard the other), as done on Linux, or allocate a third physical page by *reusing* memory from a dedicated pool of pages to improve performance. Existing attacks exploit physical memory massaging capabilities associated with the *merge* behavior, but we now show that an attacker can also perform memory massaging by exploiting *reuse* behavior.

Our first attempt at reproducing Flip Feng Shui on top of WPF failed. This was due to the fact that Windows allocates a new page when updating its stable AVL trees as discussed in §1.2.2. This means that memory massaging is no longer reliable. At this point, we started

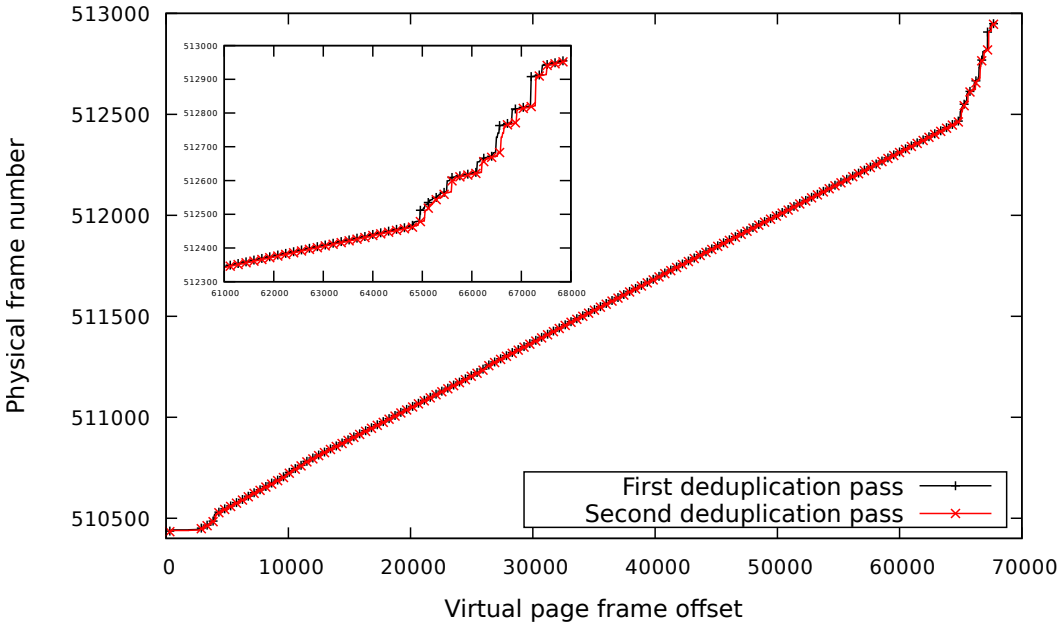


FIGURE 1.3: Ideal physical memory massaging in WPF by exploiting the deterministic behavior of page allocation during page fusion passes.

looking into memory reuse patterns during page fusion. Interestingly, we found that the *reuse* behavior of WPF provides much better predictability than that of the standard system physical page allocator, encouraging fusion-based memory massaging rather than a system-wide attack (e.g., [159]).

Given that WPF merges all possible candidates in one go, it knows how many (new) physical pages are necessary for backing fused pages. We found that, as a performance optimization, WPF calls the `MiAllocatePagesForMdl` routine with the number of physical pages it needs. Reverse engineering this routine showed that it tries to allocate contiguous physical pages from the end of memory, but allows for holes if physical pages cannot be reclaimed.

This has an interesting implication for physical memory massaging: we can get close to perfect memory reuse if we can directly use memory that is backing fused pages for mounting the attack. Furthermore, Flip Feng Shui relies on huge pages for double-sided Rowhammer but we do not always have access to huge pages on Windows (e.g., in the browser). As fused pages are, due to how `MiAllocatePagesForMdl` works, mostly contiguous we can use them as an alternative to perform a reliable double-sided Rowhammer. Hence, we can follow these steps for templating with WPF:

1. Allocate a large number of memory pages. 2. Write pair-wise duplicates into these pages. 3. Wait for WPF to merge each pair. 4. Execute Rowhammer on the fused pages for finding flips.

Note that we control the order of these pages in physical memory based on the hash of their contents as discussed in §1.2.2, necessary for performing double-sided Rowhammer. Once we find some exploitable bit flips, we trigger copy-on-write to release all the pages back to the system. In this stage, we write security-sensitive data on a large number of pages, such that every page is duplicated extacly once. After another fusion pass, we complete the attack by triggering Rowhammer again to corrupt the security-sensitive data, very much like the original Flip Feng Shui.

Figure 1.3 shows a near-perfect physical memory reuse between the two page fusion passes with WPF. Note the physical memory pages at the end of available memory to the guest (2 GB or 524,288 physical pages). This shows that we can perform Flip Feng Shui even when

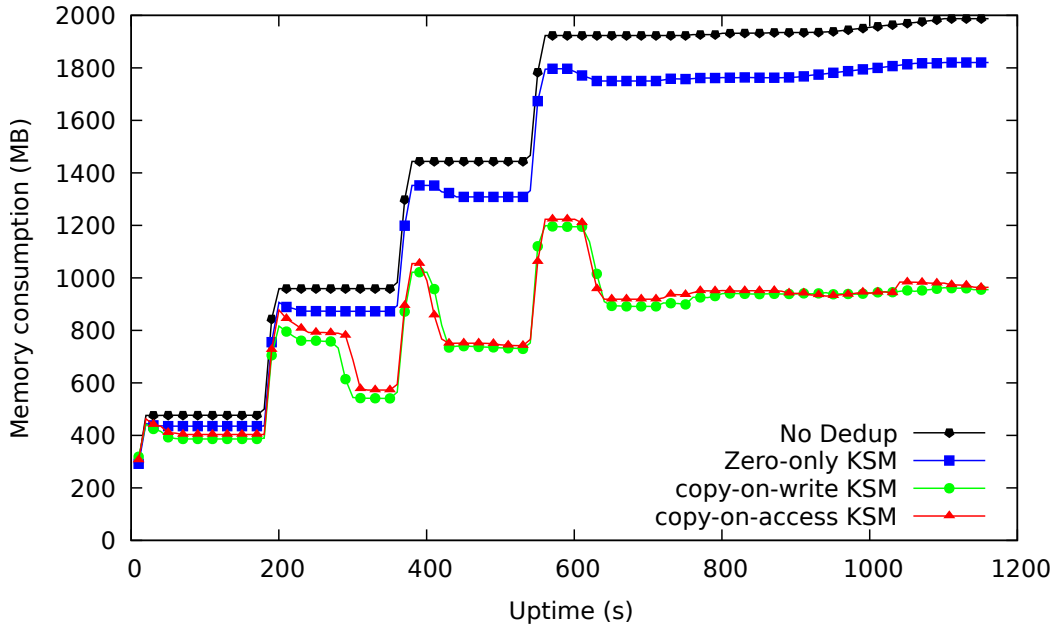


FIGURE 1.4: Comparing the effect of copy-on-access with copy-on-write on fusion rates.

new pages are allocated to back merged pages. One potential source of unreliability is the fact that the order in which processes are selected for page fusion is not known to the attacker as described earlier in §1.2.2. Further, the content hash of the target pages influence the order of physical memory allocation. We resolve both problems by allocating a large number of pair-wise duplicated targets (similar to [26]) to minimize the impact of other processes in the system.

1.5.3 Summary

Table 1.1 summarizes the attacks that we described with the underlying issues that permit information leakage or physical memory massaging. Our new merge-based information disclosure attacks show that just reading from fused pages is enough to leak information, obviating even the need to rely on copy-on-write events. Further, it is possible to observe changes in the virtual memory translation to detect page fusion indirectly without accessing the target page at all. Finally, our reuse-based attack shows that it is possible to perform Flip Feng Shui even when new allocations are used to back fused pages.

We implemented all these attacks on page fusion to verify their practicality. We further performed a deeper analysis of the attack surface of page fusion involving other side channels (e.g., the TLB and the DRAM row buffer) which we omit due to space limitation. Our conclusion is that the design principles that we describe next protect against all these attacks.

1.6 Design Principles

To stop all the potential attacks on page fusion, including the ones we described in §1.4 and §1.5, we follow two main design principles. The first principle stops information disclosure and the second principle stops Flip Feng Shui.

1.6.1 Stopping Information Disclosure

To protect against information disclosure attacks, we should stop the ability of an attacker to detect whether memory pages are fused via either merge or unmerge events. We can achieve this by ensuring that every page that an attacker tests always behaves the same, whether it is merged or not. We call this principle Same Behavior (**SB**) and discuss how we can enforce it.

Merge-based attacks It should be clear to the reader by now that sharing memory between mutually distrusting parties is not safe even if shared pages are write-protected. Hence, to preserve security, no memory pages should be shared between distrusting parties. However, not sharing memory pages across security boundaries means that we need to disable page fusion. To resolve this dichotomy, we propose sharing pages with the same contents that are not accessed by either party. We do this by removing all access permissions to pages that are shared. Any access to a shared page will cause a trap and lead to an explicit copy of the page contents to a new page in a copy-on-access fashion. We call this mechanism share xor fetch or, in short, **S⊕F**. **S⊕F** ensures sharing *only* pages that are not accessed or prefetched (countering implicit accesses via the `prefetch` x86 instruction [78]). This design converges to two sets of pages: pages in the working set that are not fused and idle pages that are candidates for fusion.

S⊕F potentially reduces fusion rates as pages which are (continuously) read or executed can no longer be shared. To investigate this, we modify KSM to unmerge on any page fault and compare it with the baseline which only unmerges on writes. Figure 1.4 shows the result of the experiment when starting four VMs with five minutes between launch times, each running an Apache server. After the page fusion process stabilizes, we can see that copy-on-access only marginally reduces fusion rates (1%). The reason for this is the fact that most of the fusion benefits come from idle pages in the system (e.g., the page cache) as we will later show in our evaluation. We also experimented with merging only zero pages to mitigate information disclosure as proposed before [26]. Compared with when page fusion is off, zero-pages account for only 16% of all duplicate pages, as shown in Figure 1.4. This is insufficient in cloud settings and further justifies our efforts in securing full (vs. zero-page only) page fusion. We also note that zero-page fusion is not by itself secure against Flip Feng Shui. We will further experiment with the effects of **S⊕F** on fusion rates and performance later in our evaluation.

Take-away: **S⊕F** prevents attacks that disclose information based on detecting a merge event because while the attacker can check whether a page is a candidate for fusion, she cannot infer whether it is actually fused. We now discuss how we prevent information disclosure through unmerged-based attacks.

Unmerge-based attacks While **S⊕F** protects page fusion against merge-based attacks, attackers can still use the difference in the behavior of merged and unmerged pages to detect unmerge events by, for example, measuring if accessing a page generates a lengthy page fault. To resolve this problem, all memory pages should behave the same, whether they are merged or not—enforcing **SB**.

A simple way to enforce this is by a mechanism which we call Fake Merging (**FM**). **FM** ensures that pages that are not merged behave the same as pages that are. For this purpose, fake merging removes all access permissions and performs copy-on-access for non-shared pages as well. Again, **FM** sounds prohibitively expensive, but we will show that in realistic settings the performance penalty is negligible. The main reason is that page fusion systems take a long time to scan memory in order not to interfere with the main computation and, as a result, the performance penalty is amortized over long periods of time. Furthermore, we can apply working set estimation to reduce the number of additional page faults caused by

FM without compromising security. Our working set-based optimization, discussed in §1.7.2, exploits the intuition that **SB** naturally converges to performing fusion on cold (and highly fusable) pages in the system.

Take-away: **FM** prevents the popular copy-on-write side channel since all first accesses to attacker-controlled pages result in copy-on-access until the next fusion pass.

1.6.2 Flip Feng Shui attacks

To defend against Flip Feng Shui-like attacks, we need to eliminate the attacker's capability to control how the page fusion system chooses the physical page that backs the merged pages. We enforce this by making sure that the page fusion system does not use physical memory in a predictable manner. We achieve this by properly randomizing page allocations on merge events. We call this principle Randomized Allocation (**RA**).

To protect against merge-based attacks [143], we need to always allocate a new page on a merge event. Unfortunately, as we showed in our new Flip Feng Shui attack against Windows, simply allocating a new page to back a shared page is not enough to prevent an attacker's ability to massage the physical memory. The reason is that efficient physical memory allocators often promote predictable reuse to reduce overhead. Hence, **RA** should be enforced either by randomizing the system-wide page allocator or at the page fusion system itself. Since randomizing the system-wide page allocator has non-trivial performance and usability implications due to the inherent fragmentation, we opt for enforcing **RA** at the page fusion system directly.

Take-away: **RA** prevents Flip Feng Shui-like attacks since attackers cannot predict the physical page that may back a fused page.

1.6.3 Discussion

Enforcing **SB** has interesting implications for page fusion. Pages in the working set will no longer fuse with other (idle) pages. This design basically partitions the candidate memory into two sets: the working set which is not considered for page fusion and the idle set which may or may not get merged. It becomes impossible for the attacker to tell whether pages in the idle set are merged since we remove all access/fetch permissions to these pages. Without access/fetch permissions, these pages cannot be stored on shared resources in the system such as the LLC or the DRAM row buffer, mitigating the new attacks we discussed in §1.5.1.

Enforcing **RA** will stop Flip Feng Shui and its new reuse-based variant which we discussed in §1.5.2) by randomizing allocations that back fused pages. In Table 1.1, we also detail which principle stops which classes of attack.

1.7 Implementation

We now discuss how VUsion adheres to **SB** and **RA**. We implemented VUsion on top of the Linux kernel version 4.10.0-rc6 and reused most of KSM's original implementation and kernel tracing functionality. Our patch changes only 846 lines of code in the Linux kernel. This suggests that the proposed modifications are realistic and that VUsion contributes a practical page fusion system for production use. We assume 4 KB pages in this section and expand our implementation to huge pages in the next section.

1.7.1 Enforcing the Design Principles

Share XOR Fetch To implement **S⊕F**, we need to intercept all accesses to a given page. Modifying the present bit in the PTEs to trigger a page fault on access is one possibility, but it

requires intrusive changes to the Linux kernel as this bit is used for tracking memory pages in many places in Linux. Instead, we opted to use the reserved bits. According to the Intel and AMD manuals [87, 2], if the reserved bits are set, the processor generates a page fault on access regardless of the permission bits in the corresponding PTE. In the page fault handler, we check these bits and perform copy-on-access if they are set. To prevent `prefetch`-based attacks, in turn, we also set the *Caching Disabled* bit in the PTEs.

Fake Merging To enforce **SB**, we also set the reserved bits in PTEs of non-shared pages and perform copy-on-access similarly to pages that are shared. We ensure that both shared and non-shared pages follow the same code paths to avoid opening other timing channels that an attacker could use to detect merge events. More specifically, we make three different design decisions compared to KSM.

(i) KSM uses the unstable tree to avoid write-protecting the pages that are being considered for fusion. This opens up a side channel since pages that become merged are immediately write-protected compared to pages that remain in the unstable tree. Fortunately, to enforce **SB**, we need to remove access permission from *any* page that is being considered for fusion. Hence, VUsion simply does not require an unstable tree and is hence protected from this side channel. (ii) Without additional care, pages that are merged take shorter during copy-on-access than pages that are fake merged. This is due to the fact that pages which are fake merged can be freed in the page fault handler, as references to the page drop to zero. This entails an expensive interaction with the buddy allocator. To counter this, we perform deferred free by queueing these pages and freeing them in the background. Note that deferred frees are already common practice in the kernel with the advent of RCU lock. The real merge also queues a dummy request to ensure the execution of the same instructions for both merge and fake merge. (iii) An advanced attacker could perform a page coloring attack on the page fault handler during copy-on-access to infer the color of the source page. If this is done across multiple scans, the attacker can infer a merge with high probability if the color of the source page does not change. We hence select new (random) physical pages to back pages that are merged or faked merged during each scan. Thanks to our working set-based optimization detailed below, these extra costs are only incurred for cold pages with little impact on performance.

Randomized Allocation We reserve 128 MB of physical memory in a cache to add 15 bits of entropy to physical memory allocations performed by VUsion during both merging and unmerging. Under these conditions, an attacker seeking to abuse page fusion in a Flip Feng Shui attack can only mount a probabilistic (and thus unreliable) attack, where a specific vulnerable template is controllably reused by the allocator with a probability of only 2^{-15}, providing much more entropy than the fairly predictable standard page allocator [159].

1.7.2 Working Set Estimation

A naive version of VUsion would assume that all the pages in the system are idle in each fusion round and the pages that are actually in the working set will trigger a page fault regardless of whether they were merged. This implementation, while secure, results in a considerable number of page faults when removing access to pages in the working set.

To address this problem, VUsion estimates the working set of registered memory and does not consider it for fusion. Doing so allows it to significantly reduce the number of page faults, improving performance. To this end, we use the idle page tracking facility in the Linux kernel [85]. During a scan, we check if the page has not been accessed for a period that can be controlled in VUsion. If that is the case, then VUsion considers the page for fusion.

Note that working set estimation does not reveal any information to the attacker except that the page is being considered for fusion (but may or may not be actually fused) since it is

idle. In other words, any side channel attack can only leak already known information: which of the attacker's own pages are in the working set.

1.8 Transparent Huge Pages

Huge pages are becoming increasingly important for modern systems as the working sets of applications increase in size. With many workloads, virtualization also greatly benefits from huge pages, owing to the higher cost of a TLB miss in the guest VM [109, 21, 73, 162, 104]. `khugepaged` is a Linux kernel daemon that runs in the background and transparently collapses consecutive physical pages into huge pages (THPs). Conversely, KSM breaks up THPs again whenever there is a sharing opportunity in them. Unfortunately, as discussed in §1.5, this opens up new side channels to detect merge events. Here, we discuss a secure implementation of THPs for our design, making VUsion deployable in practice. Our design for THPs follows that of Ingens [104] while addressing security issues such as translation attacks.

1.8.1 Handling Idle and Active Pages

Since the difference between huge and normal pages can be used to detect a merge event, we have to ensure that pages that are being considered for fusion are either all huge or normal to enforce **SB**. Since VUsion considers only idle pages for fusion, the size of the pages does not affect performance. Hence, we should opt for maximizing fusion rate. Since even a single byte difference makes it impossible to merge pages, sharing opportunities will be greater for normal pages. As a result, every time we consider a THP for fusion, we first break it up into normal pages. As mentioned earlier, the only information this provides to the attacker is that this THP is idle and a candidate for fusion. Since all pages considered for fusion are now small, we stop the attacks based on the difference in translation of huge and normal pages.

The pages in the working set are not candidates for fusion, but they are important for performance. It is possible that a huge page becomes partially idle and partially active. This creates a performance versus capacity trade-off. On x86-64, for example, there are 512 pages in a huge page. At any point in time, any number K of these pages could be active. If we consider the huge page active if $K \geq n$ (at least n active 4 KB pages), then $n = 1$ provides the best possible performance (conserving huge pages) while bigger values of n will provide more fusion opportunity, increasing available capacity. VUsion provides support for both high performance (a la Ingens [104]) or maximum fusion rate (a la KSM) while preserving security. Recent work shows how one can optimize n dynamically depending on the workload [79].

1.8.2 Securing khugepaged

As mentioned, we need to collapse normal pages that become active into huge pages to improve performance. Fortunately the background `khugepaged` thread performs this for free. However, we must be careful to prevent it from collapsing (fake) merged pages back into huge pages. Otherwise, an attacker can perform the translation attack using pages that are next to the target page.

To this end, we again use the idle page tracking mechanism available in the Linux kernel. If one (or more) of 512 pages that can potentially form a huge page is active, we will (fake) unmerge the other 511 pages if any of them is (fake) merged. As part of transforming 512 contiguous virtual pages into a huge virtual page, khugepaged will copy the contents of these pages into 512 contiguous physical pages. This is safe because all of these pages are first (fake) unmerged. This way, `khugepaged` preserves the **SB** semantics by securely collapsing pages in the working set into huge pages.

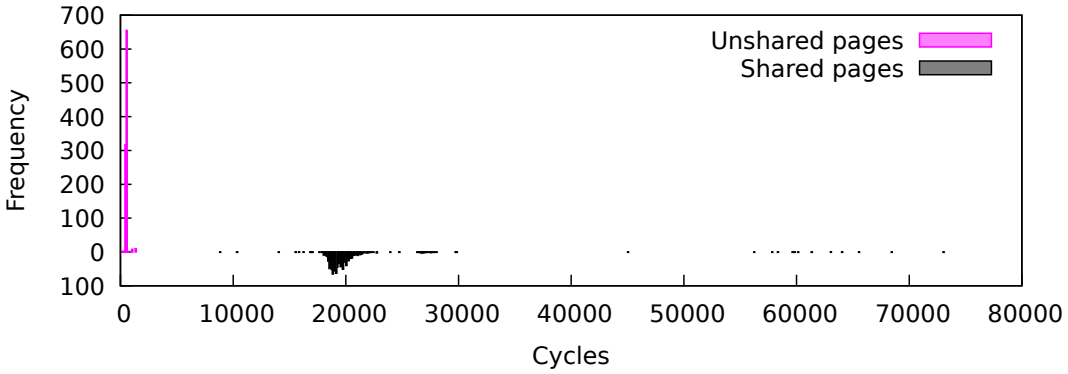


FIGURE 1.5: Freq. dist. of timing 1,000 writes in KSM.

To summarize, VUsion enforces **SB** on huge pages by breaking them before (fake) merging them, and khugepaged only collapses pages if their surrounding pages are first (fake) unmerged before forming a huge page with them.

1.9 Evaluation

We evaluate three aspects of VUsion compared to original Linux/KSM: 1. Does VUsion stop all the attacks reviewed in §1.4 and §1.5? 2. How does the performance compare? 3. How do the fusion rates compare?

Benchmarks We run synthetic microbenchmarks to evaluate the security of VUsion. To evaluate performance and memory saving, we use various benchmarks that stress different parts of the system. To fully stress the memory subsystem, we use `Stream` [151], a synthetic memory bandwith benchmark. We further use SPEC CPU2006 and PARSEC as general-purpose memory-intensive benchmarks, Postmark as a file system benchmark, and Apache (httpd), memcached and Redis as server benchmarks. Unless otherwise specified, the benchmarks run with four VMs configured with a virtual core and 2 GB of RAM. One of the VMs runs the benchmark while others provide load for page fusion. We also test a diverse set of VMs to observe the effects on fusion rates.

Testbed We use a 4-core Intel Xeon E3-1240 v5 processor running at 3.5 GHz with 24 GB of DDR4 memory as our evaluation testbed. We further experimented with a dual processor Xeon E5-2650 v2 system with 32 GB of memory and obtained similar results which we do not include for brevity. we configure both VUsion and KSM with KSM default values (i.e., T = 20 ms and N = 100 pages). We run server benchmarks using a client machine over a 1 Gbps network. We configure the VMs' virtual NICs using virtio with vhost enhancements for high-performance I/O.

1.9.1 Security

We show that VUsion enforces **SB** and **RA** by timing reads and writes to pages that are shared or unshared, and checking whether the page allocations are truly random.

Enforcing SB Figure 1.5 shows the frequency distribution of timing 1,000 writes in KSM after a fusion pass. The two distinct peaks for shared and unshared pages show the copy-on-write side channel present in KSM. Figure 1.6 shows the results of 1,000 reads in VUsion. In

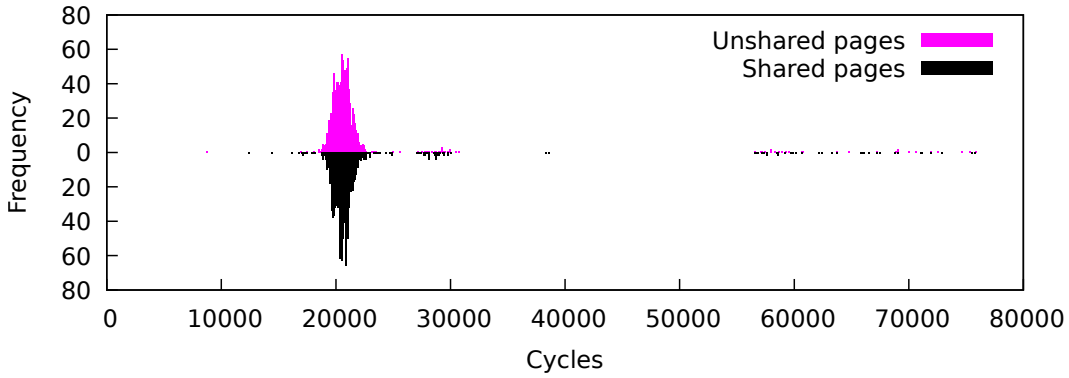


FIGURE 1.6: Freq. dist. of timing 1,000 reads in VUsion.

	copy MB/s	scale MB/s	add MB/s	triad MB/s
No Dedup	11109	10690	12463	12342
KSM	11035	10644	12431	12291
VUsion	11019	10695	12423	12280
VUsion THP	11022	10646	12441	12271

TABLE 1.2: Performance of the Stream benchmark.

the case of VUsion, there is no visible difference between shared and unshared pages since both cases trigger copy-on-access events. The results for writes are similar to reads. To gain more confidence, we perform a Kolmogorov-Smirnov test to see whether the timing events for merged and unmerged pages follow the same distribution in VUsion. The calculated p-value is high (0.36) which means that we do not reject the hypothesis that these events are from the same distribution.

These tests show that VUsion conforms to the **SB** principle that we described in §1.6. We note that our analysis can guarantee correctness, but not the complete absence of other (arbitrary) side channels. While recent work shows promising results for formal constant-time verification [7], doing so is difficult in the context of the Linux kernel but an interesting direction for future work. Further, our experiments with the `prefetch` instruction confirm that setting the "Caching Disabled" bit in the PTEs of (fake) merged pages stops the recently reported side channel [78] where pages can be *prefetched* into the cache without access permissions.

Enforcing RA We record the offsets of pages chosen for merge and fake merge in VUsion when executing two VMs. We then perform a Kolmogorov-Smirnov goodness of fit test against the uniform distribution. The calculated p-value is high (0.44) which means that the test does not reject our hypothesis that physical page allocations in VUsion follow the uniform distribution.

1.9.2 Performance

To gain a complete understanding of possible performance issues when enforcing **S⊕F**, we first quantify the nature of common merge events. Table 1.3 shows the type of merged pages in one of the four VMs. Interestingly, most possibilities for fusion come from idle pages in the system (i.e., buddy) allocator and the page cache. To experiment with the former, we evaluate VUsion with the `Stream` microbenchmark as well as the memory-intensive SPEC and PARSEC benchmarks. To experiment with the latter, we evaluate VUsion with the file system-intensive Postmark benchmark. We also complement our analysis with a server

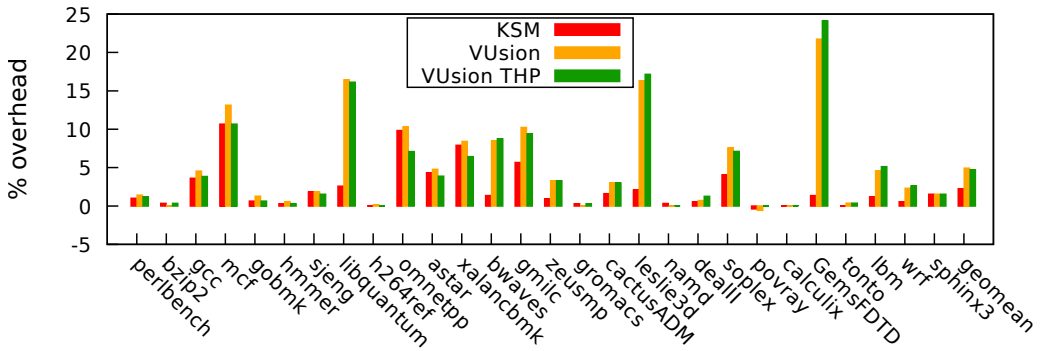


FIGURE 1.7: Performance overhead on SPEC CPU2006.

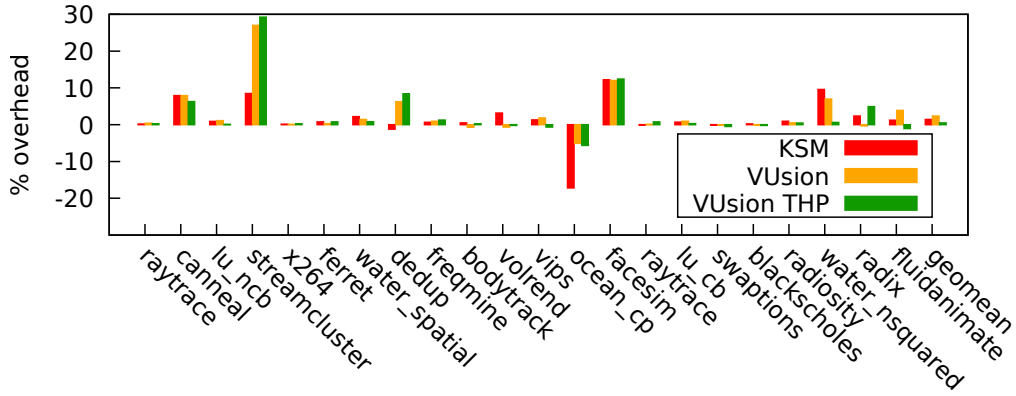


FIGURE 1.8: Performance overhead on PARSEC.

application benchmark, important to evaluate the impact on common cloud workloads. Our results are as follows.

Stream Table 1.2 shows the available memory bandwidth when running Stream in a VM with/without KSM and in VUsion with/without THP enhancements. In all cases, the additional overhead introduced by KSM or VUsion is below 1%. This is due to the fact that the default scanning rate is slow (5000 pages/second) and the few additional page faults only marginally affect the memory bandwidth.

SPEC CPU2006 Figure 1.7 shows the performance overhead of KSM and VUsion with-/without THP enhancements compared to when page fusion is turned off over the general-purpose SPEC CPU2006 benchmark suite. Considering the geometric mean, KSM adds 2.2% overhead to the baseline. VUsion adds another 2.7% (overall 4.9%) and enabling huge pages adds 2.4% (overall 4.6%). Most of the benchmarks are insensitive to the additional page faults caused by enforcing **S**$\oplus$**F**. Similar to `Stream`, the additional page faults are bounded by the number of pages that become active over a page fusion period (i.e., a few hundred seconds). Hence, we conclude that VUsion enforces **SB** and **RA** with minimal performance overhead in a general-purpose workload.

PARSEC To study the behavior of the system under concurrent workloads, we experiment with PARSEC. We increase the number of virtual cores to four to increase possible parallelism

	page cache (%)	buddy (%)	kernel (%)	rest (%)
KSM	51.8	38.4	6.9	2.9
VUsion	51.2	38.6	6.6	3.6
VUsion THP	50.4	32.8	6.3	10.5

TABLE 1.3: Contribution of different page types to page fusion.

	mean (tx/s)	min (tx/s)	max (tx/s)
No dedup	3237.3	3191	3289
KSM	3221.7	3215	3232
VUsion	3178.7	3154	3191
VUsion THP	3246.3	3222	3285

TABLE 1.4: Performance of the Postmark benchmark.

in our test VM. Figure 1.8 shows the outcome. `fmm` and `barnes` require more than 8 GB of RAM to execute and the `netapps` category hangs in our unmodified setting and hence we excluded. KSM adds 1.7% of performance overhead on top of the baseline. Consideraig the geometric mean, VUsion slightly degrades KSM performance by 0.5% (overall 2.2%) while VUsion's THP enhancements improve KSM's performance by 1.4% (overall 0.8%). These results further prove that VUsion introduces low overhead and can even improve KSM's performance.

Postmark Table 1.4 shows the number of transactions per second in Postmark, a benchmark emulating a mailserver that heavily interacts with the file system. KSM degrades performance by 1.5% while VUsion degrades performance by 2.9%. VUsion with THP enhancements slightly improves the performance relative to KSM (0.2% improvement over baseline). These results suggest that VUsion can secure page fusion without performance penalty in workloads that benefit from it the most.

Apache We use Apache 2.4.18 with the default prefork module and wrk [168] to generate load on the server at remote CPU saturation using 20 concurrent connections and 10 threads for a duration of 500 seconds. Table 1.5 shows the throughput and various latency percentiles that we achieve under different configurations. In the case of throughput, for Apache, KSM incurs 20.0% of overhead on the baseline while VUsion adds a marginal 0.4% overhead. THP enhancements in VUsion improve the performance relative to KSM by 12.7%. Latency follows a similar trend: VUsion provides similar performance to KSM while the THP enhancements in VUsion improve the latency relative to KSM.

Figure 1.9 shows the number of huge pages during runtime of the Apache benchmark. As expected, the number of huge pages is higher in VUsion with THP compared to KSM. More importantly, these huge pages are part of the working set, improving performance as reported in Table 1.5. During the runtime of the benchmark, the VM allocates more memory with demand paging. Initially the allocations are backed by huge pages, but VUsion without THP enhancements breaks them down when considering them for fusion.

Key-value stores To experiment with server applications that have a large memory footprint, we experiment with Redis (version 3.0.6) and Memcached (version 1.4.25), two popular key-value stores. We use memtier_benchmark [121] with the default configuration for generating load: using 4 threads, 50 clients, a set/get ratio of 1:10, and 32-byte objects from a 10 million key space.

Table 1.6 shows the throughput of Redis and Memcached. Redis follows a similar trend as Apache discussed earlier: KSM and VUsion provide similar throughput and the THP enhancements in VUsion improve overall performance. The throughput of Memcached with

	kreq/s (rel.)	lat. 75%	lat. 90%	lat. 99%
No Dedup	22.03 (100%)	1.34 ms	1.95 ms	4.49 ms
KSM	18.42 (83.6%)	1.59 ms	2.34 ms	5.87 ms
VUsion	18.28 (82.3%)	1.64 ms	2.47 ms	6.51 ms
VUsion THP	21.18 (96.1%)	1.37 ms	2.03 ms	5.55 ms

TABLE 1.5: Performance of the Apache server.

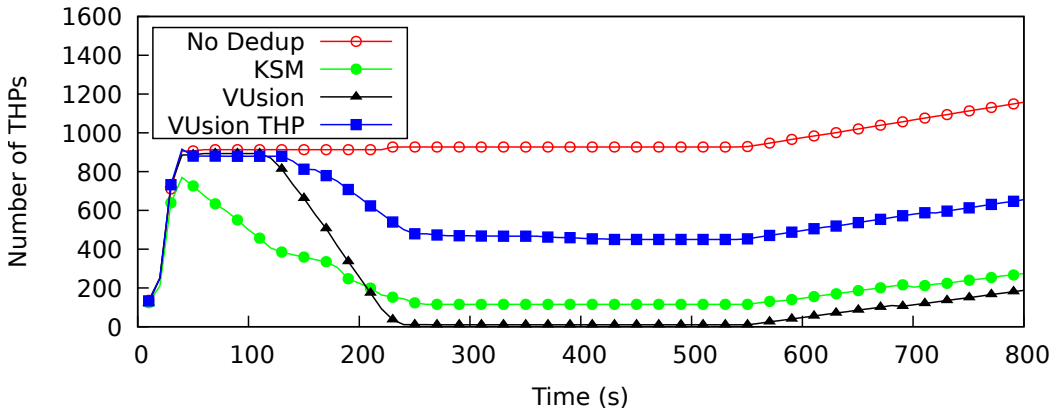


FIGURE 1.9: Conserving THPs with the Apache benchmark.

VUsion is more severely impacted (5.3% worse than KSM), but the THP enhancements bring the throughput of VUsion close to KSM. Table 1.7 shows the latency of GET and SET requests in both key-value stores. We observe similar trends again: the latency with VUsion is marginally impacted compared with KSM and the THP enhancements in VUsion improve the results, even at the tail but with the exception of SET requests (4.5% overhead compared to KSM).

1.9.3 Fusion Rates

To study fusion rates with VUsion, we showcase three scenarios. The first scenario shows fusion rates of idle VMs and how quickly VUsion fuses memory compared to KSM. The second scenario shows the scalability of VUsion, namely when starting many different VMs of different types. The third scenario shows the memory consumption during our Apache server benchmark.

Idle VMs Figure 1.10 shows the total memory consumption of four VMs started 5 minutes after each other under different systems. This shows that, in an idle setting (expected on many cloud hosts), VUsion's fusions rates converges to that of KSM despite the conservative **S⊕F** policy. VUsion, however, takes longer to merge pages when compared to KSM. As we discussed in §1.7.1, KSM merges pages as soon as it finds a match in its stable tree. VUsion, however, waits one scan round before (fake) merging to enforce **SB**. Further, waiting one round allows VUsion to reduce page faults by focusing on pages that are not in the working set.

Diverse VMs To understand the effects of VUsion on fusion rate in a more realistic setting, we experiment with 44 VM images from our DAS4 cloud deployment registered by various teams. These images include various Linux distributions and software stacks. We start 16 VMs at the same time using randomly selected VM images and report the consumed memory over time in Figure 1.11. The results are similar to our synthetic test; VUsion achieves a

	Redis (kreq/s)	Memcached (kreq/s)
No dedup	175.30 (100%)	167.5 (100%)
KSM	155.66 (88.8%)	163.97 (97.9%)
VUsion	155.09 (88.4%)	155.11 (92.6%)
VUsion THP	163.8 (93.4%)	163.87 (97.8%)

TABLE 1.6: Throughput of Redis and Memcached.

	Redis SET (ms)			Memcached SET (ms)		
Percentile	90.0	99.0	99.9	90.0	99.0	99.9
No Dedup	1.6	2.4	4.9	1.7	2.5	3.5
KSM	1.7	2.8	6.7	1.8	3.2	6.3
VUsion	1.8	3.0	7.3	2.0	3.6	6.3
VUsion THP	1.6	2.8	7.0	1.8	2.9	4.7

	Redis GET (ms)			Memcached GET (ms)		
Percentile	90.0	99.0	99.9	90.0	99.0	99.9
No Dedup	1.6	2.4	5.0	1.7	2.5	3.5
KSM	1.7	2.8	7.7	1.8	3.2	6.2
VUsion	1.8	3.0	7.0	2.0	3.6	6.2
VUsion THP	1.6	2.7	6.7	1.8	2.9	4.7

TABLE 1.7: Latency of Redis and Memcached.

similar fusion rate compared to KSM. VUsion with THP enhancements conserves huge pages that are in the working set while reducing fusion rate by 61%. These results show that VUsion provides its users with security with a trade-off between page fusion and performance.

Apache Figure 1.12 shows the memory consumption during the Apache benchmark which we reported on earlier. We start four VMs together and start the benchmark on one of them after 360 seconds. Again VUsion achieves a similar fusion rate compared to KSM while similarly degrading fusion rate when conserving huge pages to improve performance. We also notice memory consumption increasing during the benchmark period for all cases. This is due to Apache's self-balancing strategy, which gradually expands the number of worker processes to serve many parallel requests to improve its throughput.

1.10 Related Work

1.10.1 Attacks

Page fusion has been previously used in various attacks: as a prelude to a `FLUSH+RELOAD` attack [172], as a side channel to fingerprint software or build covert channels [153, 135, 88, 170], and as a way to brute-force ASLR [18, 26] or passwords [26]. Finally, the Flip Feng Shui attack [143] uses page fusion for physical memory massaging to compromise cryptographic keys of a victim VM.

In all cases, the traditional mitigation is to disable page fusion, wasting memory. An alternative is to disable active page fusion and only fuse swapped pages within a compressed in-memory cache. This is the approach taken by the current *Windows Memory Combining* implementation (previously the name of the active page fusion system on Windows, now disabled). This design, however, misses substantial fusion opportunities compared to active page fusion. In contrast, VUsion preserves page fusion benefits, mitigates all existing attacks, and even protects against a number of new attack vectors that we presented in this chapter.

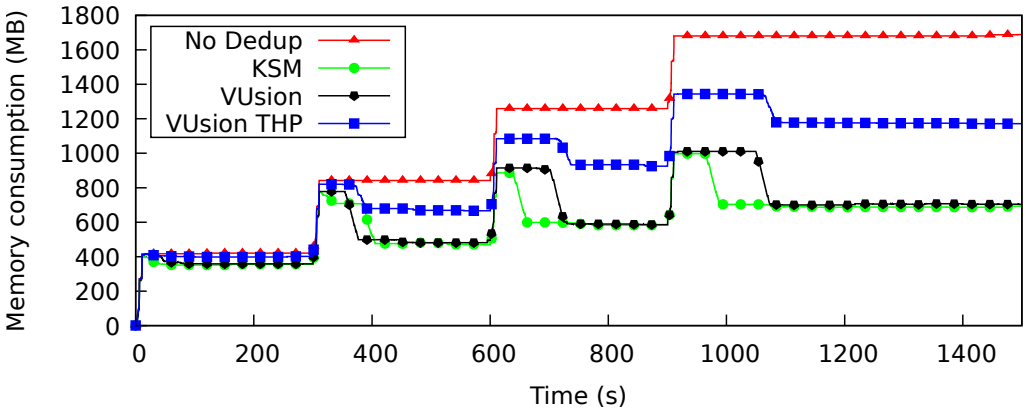


FIGURE 1.10: Memory consumption of idle VMs.

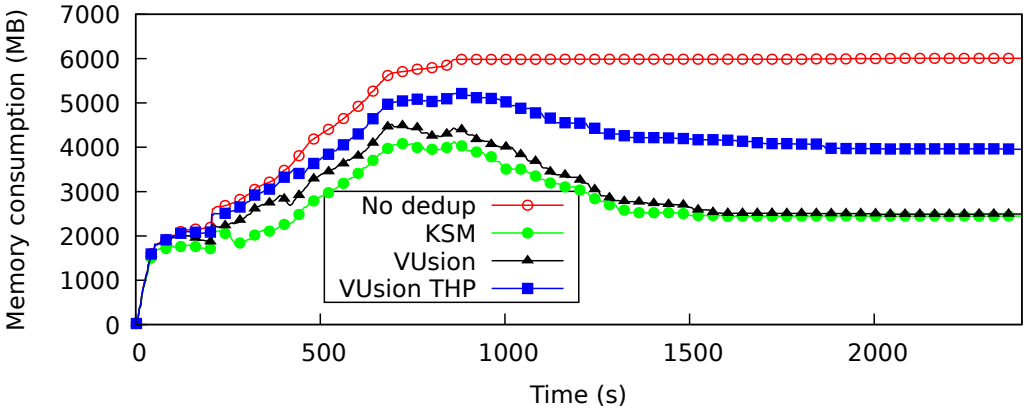


FIGURE 1.11: Memory consumption of different VMs.

1.10.2 Defenses

The only existing defense against information leakage via page fusion is HexPADS [137]. HexPADS is an anomaly detection system that uses performance counters to detect suspicious behavior. Given the anomaly detection nature of HexPADS, it is prone to false positives and false negatives, providing attackers with the opportunity to tune their attacks and easily bypass HexPADS. Furthermore, HexPADS does not protect against physical memory massaging. In comparison, VUsion secures page fusion by design, improving performance as a by-product, and does not have any of the aforementioned weaknesses.

The **S⊕F** design principle relies on the copy-on-access technique, which has also been previously used in different applications such as post-copy live migration [83] and defending against cache attacks [174]. In contrast, VUsion uses copy-on-access as a building block for securing page fusion and combines it with protection against `prefetch`-based and other attacks.

Timing attacks against page fusion and side-channel attacks in general can be (partially) mitigated by reducing the timer accuracy. At the software level, major browsers such as Chrome, Firefox, and Microsoft Edge have reduced the accuracy of their timers to prevent side-channel attacks from JavaScript. Kohlbrenner and Shacham [100] propose introducing noise in the timer and the event loop of JavaScript to hinder timing measurement of system events. At the hardware level, TimeWarp [118] reduces the fidelity of timers and performance counters to make it difficult for attackers to distinguish between different microarchitectural events. Unfortunately, degrading the timer has performance implications and as shown by

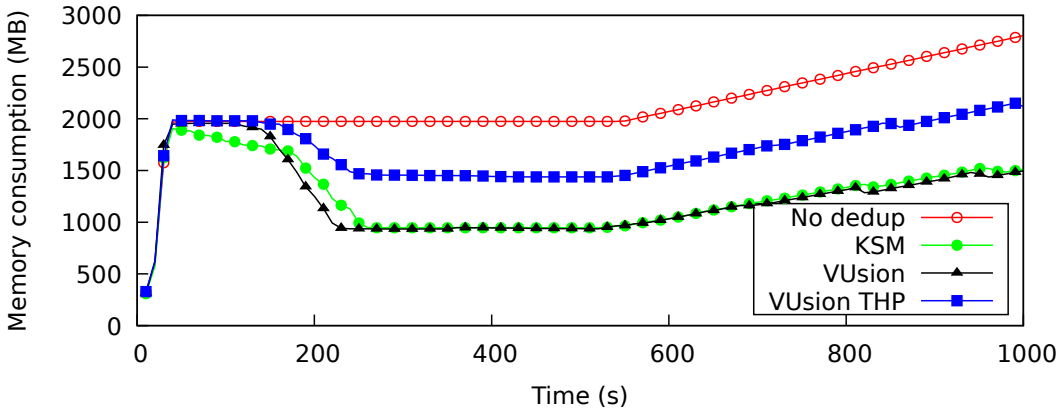


FIGURE 1.12: Memory consumption of the Apache benchmark.

Gras et al. [74] can easily be bypassed.

1.11 Conclusion

Page fusion reduces memory pressure in modern cloud and consumer platforms, but existing approaches have been plagued by security weaknesses that lead to information disclosure and control over physical memory. This chapter shows that these weaknesses are not fundamental and proposes a secure page fusion system with marginal degradation of fusion rates. The new design stops known and new attacks against page fusion, while also addressing inefficiencies. Our solution provides evidence that it is possible to support both secure and efficient page fusion in real-world settings.

Chapter 2

ZebRAM

The Rowhammer vulnerability common to many modern DRAM chips allows attackers to trigger bit flips in a row of memory cells by accessing the adjacent rows at high frequencies. As a result, they are able to corrupt sensitive data structures (such as page tables, cryptographic keys, object pointers, or even instructions in a program), and circumvent all existing defenses.

This chapter introduces ZebRAM, a novel and comprehensive software-level protection against Rowhammer. ZebRAM isolates every DRAM row that contains data with guard rows that absorb any Rowhammer-induced bit flips; the only known method to protect against all forms of Rowhammer. Rather than leaving guard rows unused, ZebRAM improves performance by using the guard rows as efficient, integrity-checked and optionally compressed swap space. ZebRAM requires no hardware modifications and builds on virtualization extensions in commodity processors to transparently control data placement in DRAM. Our evaluation shows that ZebRAM provides strong security guarantees while utilizing all available memory.

2.1 Introduction

The Rowhammer vulnerability, a defect in DRAM chips that allows attackers to flip bits in memory at locations to which they should not have access, has evolved from a mere curiosity to a serious and very practical attack vector for compromising PCs [26], VMs in clouds [143, 171], and mobile devices [67, 159]. Rowhammer allows attackers to flip bits in DRAM rows simply by repeatedly reading neighboring rows in rapid succession. Existing software-based defenses have proven ineffective against advanced Rowhammer attacks [16, 28], while hardware defenses are impractical to deploy in the billions of devices already in operation [107]. This chapter introduces ZebRAM, a comprehensive software-based defense preventing all Rowhammer attacks by isolating every data row in memory with guard rows that absorb any bit flips that may occur.

Practical Rowhammer attacks Rowhammer attacks can target a variety of data structures, from page table entries [148, 159, 160, 171] to cryptographic keys [143], and from object pointers [26, 67, 155] to opcodes [77]. These target data structures may reside in the kernel [148, 159], other virtual machines [143], the same process address space [26, 67], and even on remote systems [155]. The attacks may originate in native code [148], JavaScript [26, 76], or from co-processors such as GPUs [67] and even DMA devices [155]. The objective of the attacker may be to escalate privileges [26, 159], weaken cryptographic keys [143], compromise remote systems [155], or simply lock down the processor in a denial-of-service attack [90].

Today's defenses are ineffective Existing hardware-based Rowhammer defenses fall into three categories: refresh rate boosting, target row refresh, and error correcting codes. Increasing the refresh rate of DRAM [96] makes it harder for attackers to leak sufficient charge from a row before the refresh occurs, but cannot prevent Rowhammer completely without unacceptable performance loss and power consumption increase. The target row refresh (TRR) defense, proposed in the LPDDR4 standard, uses hardware counters to monitor DRAM row accesses and refreshes specific DRAM rows suspected to be Rowhammer victims. However, TRR is not widely deployed; it is optional even in DDR4 [92]. Moreover, researchers still regularly observe bit flips in memory that is equipped with TRR [147]. As for error correcting codes (ECC), the first Rowhammer publication already argued that even ECC-protected DRAM is susceptible to Rowhammer attacks that flip multiple bits per memory word [96]. While this is complicating attacks, they do not stop fully stop them as shown by the recent ECCploit attack [40]. Furthermore, ECC memory is unavailable on most consumer devices.

Software defenses do not suffer from the same deployment issues as hardware defenses. These solutions can be categorized into primitive weakening, detection, and isolation.

Primitive weakening makes some of the steps in Rowhammer attacks more difficult, for instance by making it harder to obtain physically contiguous uncached memory [148], or to create the cache eviction sets required to access DRAM in case the memory *is* cached. Research has already shown that these solutions do not fundamentally prevent Rowhammer [67].

Rowhammer detection uses heuristics to detect suspected attacks and refresh victim rows before they succumb to bit flips. For instance, ANVIL uses hardware performance counters to identify likely Rowhammer attacks [16]. Unfortunately, hardware performance counters are not available on all CPUs, and some Rowhammer attacks may not trigger unusual cache behavior or may originate from unmonitored devices [67].

A final, and potentially very powerful defense against Rowhammer is to *isolate* the memory of different security domains in memory with unused *guard rows* that absorb bit flips. For instance, CATT places a guard row between kernel and user memory to prevent Rowhammer attacks against the kernel from user space [28]. Unfortunately, CATT does not prevent Rowhammer attacks between user processes, let alone attacks *within* a process that aim to subvert cryptographic keys [143]. Moreover, the lines between security domains are often blurry, even in seemingly clear-cut cases such as the kernel and user-space, where the shared page cache provides ample opportunity to flip bits in sensitive memory areas and launch devastating attacks [77].

ZebRAM: isolate everything from everything Given the difficulty of correctly delineating security domains, the only *guaranteed* approach to prevent all forms of Rowhammer is to isolate *all* data rows with guard rows that absorb bit flips, rendering them harmless. The guard rows, however, break compatibility: buddy allocation schemes (and certain devices) require physically-contiguous memory regions. Furthermore, the drawback of this approach is obvious—sacrificing 50% of memory to guard rows is extremely costly. This chapter introduces ZebRAM, a novel, comprehensive and compatible software protection against Rowhammer attacks that isolates everything from everything else *without* sacrificing memory consumed by guard rows. To preserve compatibility, ZebRAM remaps physical memory using existing CPU virtualization extensions. To utilize guard rows, ZebRAM implements an efficient, integrity-checked and optionally compressed swap space in memory.

As we show in Section 2.7, ZebRAM incurs an overhead of 5% on the SPEC CPU 2006 benchmarks. While ZebRAM remains expensive in the memory-intensive `redis` instance, our evaluation shows that ZebRAM's in-memory swap space significantly improves performance compared to our basic solution that leaves the guard rows unused, in some cases eliminating over half of the observed performance degradation. In practice, the recent Meltdown/Spectre vulnerabilities show that for a sufficiently serious threat, even expensive

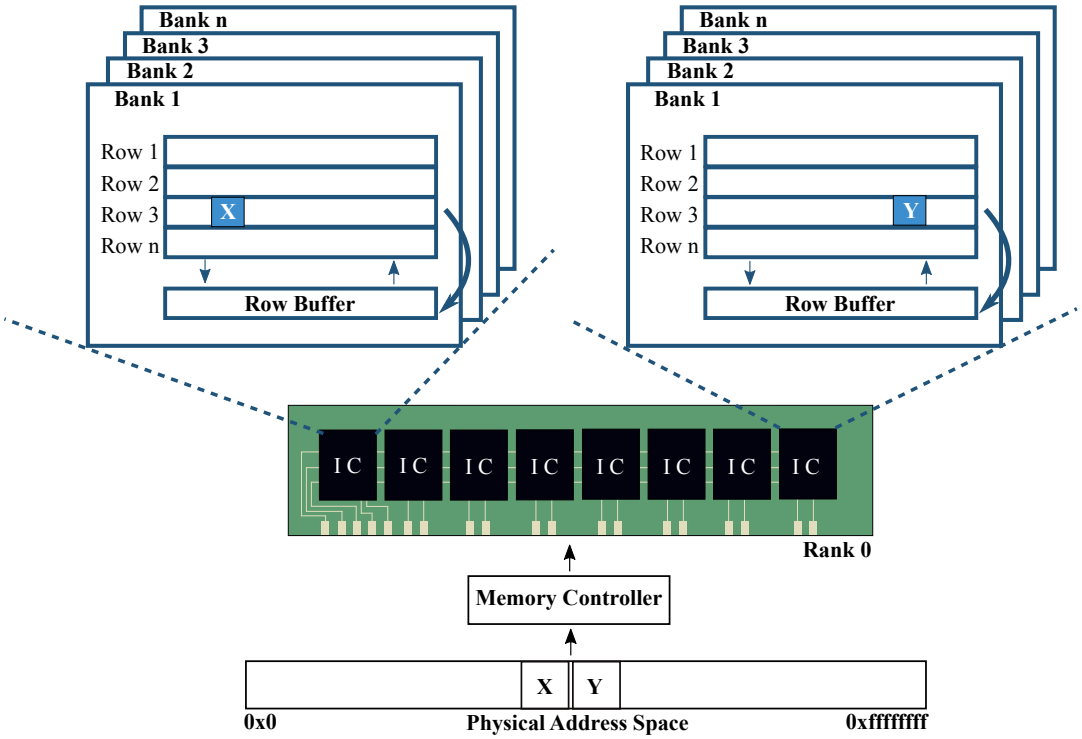


FIGURE 2.1: DRAM organization and example mapping of two consecutive addresses.

fixes are accepted [126]. First and foremost, however, this work investigates an extreme point in the design space of Rowhammer defenses: the first complete protection against all forms of Rowhammer, without sacrificing memory, at a cost that is a function of the workload.

Contributions Our contributions are the followings:

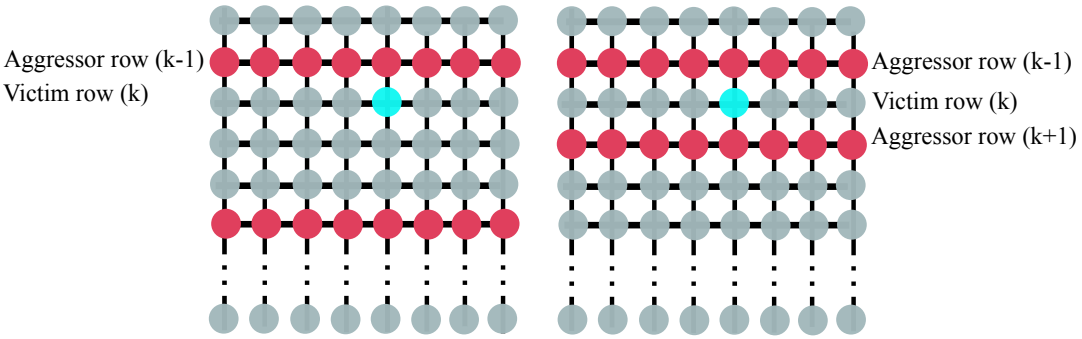
- We describe ZebRAM, the first comprehensive software protection against all forms of Rowhammer.
- We introduce a novel technique to utilize guard rows as fast, memory-based swap space, significantly improving performance compared to solutions that leave guard rows unused.
- We implement ZebRAM and show that it achieves both practical performance and effective security in a variety of benchmark suites and workloads.
- ZebRAM is open source to support future work.

2.2 Background

This section discusses background on DRAM organization, the Rowhammer bug, and existing defenses.

2.2.1 DRAM Organization

We now discuss how DRAM chips are organized internally, which is important knowledge for launching an effective Rowhammer attack. Figure 2.1 illustrates the DRAM organization.



(a) Single-sided Rowhamamer attack (b) Double-sided Rowhammer attack

FIGURE 2.2: Flipping a bit in a neighboring DRAM row through single-sided (a) and double-sided (b) Rowhammer attacks.

The most basic unit of DRAM storage is a *cell* that can hold a single bit of information. Each DRAM cell consists of two components: a capacitor and a transistor. The capacitor stores a bit by retaining electrical charge. Because this charge leaks away over time, the memory controller periodically (typically every 64 ms) reads each cell and rewrites it, restoring the charge on the capacitor. This process is known as *refreshing*.

DRAM cells are grouped into *rows* that are typically 1024 cells (or *columns*) wide. Memory accesses happen at row granularity. When a row is accessed, the contents of that row are put in a special buffer, called the *row buffer*, and the row is said to be *activated*. After the access, the activated row is written back (i.e., recharged) with the contents of the row buffer.

Multiple rows are stacked together to form *banks*, with multiple banks on a DRAM *integrated circuit (IC)* and a separate row buffer per bank. In turn, DRAM ICs are grouped into *ranks*. DRAM ICs are accessed in parallel; for example, in a DIMM that has eight ICs of 8 bits wide each, all eight ICs are accessed in parallel to form a 64 bit *memory word*.

To address a memory word within a DRAM rank, the system memory controller uses three addresses for the bank, row and column, respectively. Note that the mapping between a physical memory address and the corresponding rank-index, bank-index and row-index on the hardware module is nonlinear. Consequently, two consecutive physical memory addresses can be mapped to memory cells that are located on different ranks, banks, or rows (see Figure 2.1). As explained next, knowledge of the address mapping is vital to effective Rowhammer.

2.2.2 The Rowhammer Bug

As DRAM chips become denser, the capacitor charge reduces, allowing for increased DRAM capacity and lower energy consumption. Unfortunately, this increases the possibility of memory errors owing to the smaller difference in charge between a “0” bit and a “1” bit.

Research shows that it is possible to force memory errors in DDR3 memory by activating a row many times in quick succession, causing capacitors in neighboring *victim* rows to leak their charge before the memory controller has a chance to refresh them [96]. This rapid activation of memory rows to flip bits in neighboring rows is known as the *Rowhammer attack*. Subsequent research has shown that bit flips induced by Rowhammer are highly reproducible and can be exploited in a multitude of ways, including privilege escalation attacks and attacks against co-hosted VMs in cloud environments [26, 76, 140, 143, 148, 159, 171].

The original Rowhammer attack [148] is now known as *single-sided* Rowhammer. As Figure 2.2 shows, it uses many rapid-fire memory accesses in one *aggressor* row $k-1$ to induce bit flips in a neighboring victim row k. A newer variant called *double-sided* Rowhammer hammers rows $k-1$ and $k+1$ on both sides of the victim row k, increasing the likelihood of a bit flip (see Figure 2.2). Recent research shows that bit flips can also be induced

by hammering only one memory address [77] (*one-location* hammering). Regardless of the type of hammering, Rowhammer can only induce bit flips on directly neighboring DRAM rows.

In contrast to single-sided Rowhammer, the double-sided variant requires knowledge of the mapping of virtual and physical addresses to memory rows. Since DRAM manufacturers do not publish this information, this necessitates reverse engineering the DRAM organization.

2.2.3 Rowhammer Defenses

Research has produced both hardware- and software-based Rowhammer defenses.

The original hardware defense proposed by Kim et al. [96] doubles the refresh rate. Unfortunately, this has been proven insufficient to defend against Rowhammer [16]. Other hardware defenses include error-correcting DRAM chips (ECC memory), which can detect and correct a 1-bit error per ECC word (64-bit data). Unfortunately, ECC memory cannot correct multi-bit errors [4, 107] and is not readily available in consumer hardware. The new LPDDR4 standard [93] specifies two features which together defend against Rowhammer: *Target Row Refresh (TRR)* enables the memory controller to refresh rows adjacent to a certain row, and *Maximum Activation Count (MAC)* specifies a maximum row activation count before adjacent rows are refreshed. Despite these defenses, Gruss et al. [147] still report bit flips in TRR memory.

ANVIL [16], a software defense, uses Intel's performance monitoring unit (PMU) to detect physical addresses that cause many cache misses indicative of Rowhammer.[1] It then recharges suspected victim rows by accessing them. Unfortunately, the PMU does not accurately capture memory accesses through DMA, and not all CPUs feature PMUs. Moreover, the current implementation of ANVIL does not accurately take into account DRAM address mapping and has been reported to be ineffective because of it [154].

Another software-based defense, B-CATT [28], implements a bootloader extension to blacklist all the locations vulnerable to Rowhammer, thus wasting the memory. However, Gruss et al. [77] show that this approach is not practical as it may blacklist over 95% of memory locations; similar results were reported by Tatar et al. [154] showing DIMMs with 99+% vulnerable memory locations. In addition, in our experiments, we have observed different bit flip patterns over time for the same module, making B-CATT incomplete.

Yet another software-based defense called CATT [28] proposes an alternative memory allocator for the Linux kernel that isolates user and kernel space in physical memory, thus ensuring that user-space attackers cannot flip bits in kernel memory. However, CATT does not defend against attacks between user-space processes, and previous work [77] shows that CATT can be bypassed by flipping bits in the code of the `sudo` program.

2.3 Threat Model

The Rowhammer attacks found in prior research aim for privilege escalation [26, 140, 143, 148, 159, 171, 76], compromising co-hosted virtual machines [143, 171] or even attacks over the network [155]. Our approach, ZebRAM, addresses all these attacks through its principle of isolating memory rows from each other. Our prototype implementation of ZebRAM focuses only on virtual machines, stopping all of the aforementioned attacks launched from or at a victim virtual machine, assuming the hypervisor is trusted. We discuss possible alternative implementations (e.g., native) in Section 2.9.2.

[1] Rowhammer attacks repeatedly clear hammered rows from the CPU cache to ensure that they hammer DRAM memory, not the cache.

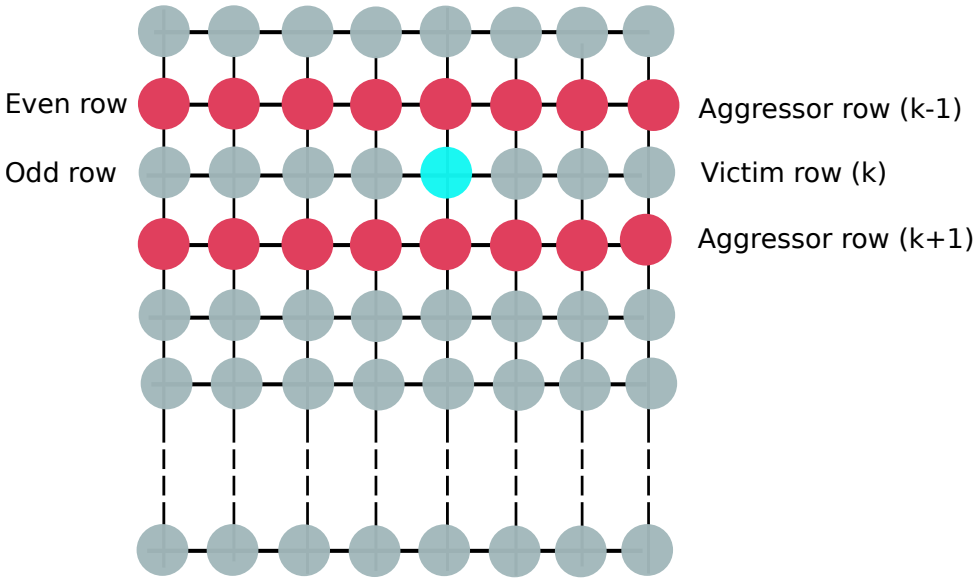


FIGURE 2.3: Hammering even-numbered rows can only induce bit flips in odd-numbered rows and vice versa.

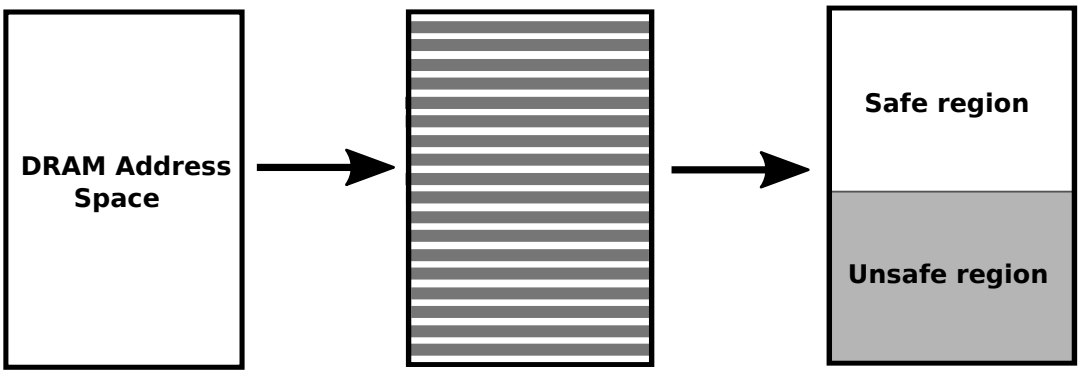


FIGURE 2.4: Splitting the memory into safe and unsafe regions using even and odd rows in a zebra pattern.

2.4 Design

To build a comprehensive solution against Rowhammer attacks, we should consider Rowhammer's fault model: bit flips only happen in adjacent rows when a target row is hammered as shown in Figure 2.3. Given that any row can potentially be hammered by an attacker, all rows in the system can be abused. To protect against Rowhammer in software, we can follow two approaches: we either need to protect the entire memory against Rowhammer or we need to limit the rows that the attacker can access. Protecting the entire memory is not secure even in hardware [107, 159] and software attempts have so far been shown to be insecure [77]. Instead, we aim to design a system where an attacker can only hammer a subset of rows directly.

Basic ZebRAM In order to make sure that Rowhammer bit flips cannot target any data, we should enforce the invariant that all *adjacent rows are unused.* This can be done by making sure that either all odd or all even rows are unused by the system. Assuming odd rows are unused, all even rows will create a *safe region* in memory; it is not possible for an attacker to flip bits in this safe regions simply because all the odd rows are inaccessible to the attacker. The attacker can, however, flip bits in the odd rows by hammering the even rows in the safe

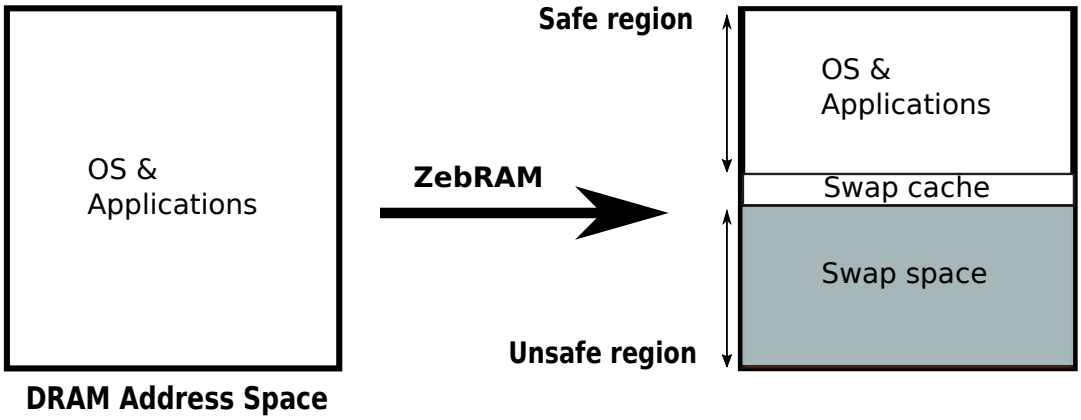


FIGURE 2.5: ZebRAM logically divides system memory into a safe region for normal use, a swap space made from the unsafe region, and a swap cache to protect the safe region from accesses made to the unsafe region.

region. Hence, we call the odd rows the *unsafe region* in memory. Given that the unsafe region is unused, the attacker cannot flip bits in the data used by the system. This simple design with its zebra pattern shown in Figure 2.4 already stops all Rowhammer attacks. It however has an obvious downside: it wastes half of the memory that makes up the unsafe region. We address this problem later when we explain our complete ZebRAM design.

A more subtle downside in this design is incompatibility with the Buddy page allocation scheme used in commodity operating systems such as Linux. Buddy allocation requires contiguous regions of physical memory in order to operate efficiently and forcing the system not to use odd rows does not satisfy this requirement. Ideally, our design should utilize the unsafe region while providing (the illusion of) a contiguous physical address space for efficient buddy allocation as shown on the right side of Figure 2.4. To address this downside, our design should provide a translation mechanism that creates a linear physical address space out of the safe region.

ZebRAM If we can find a way to securely use the unsafe region, then we can gain back the memory wasted in the basic ZebRAM design. We need to enforce two invariants if we want to make use of the unsafe region for storing data. First, we need to make sure that we properly handle potential bit flips in the unsafe region. Second, we need to ensure that accessing the unsafe region does not trigger bit flips in the safe region. Our proposed design, ZebRAM, shown in Figure 2.5 satisfies all these requirements. To handle bit flips in the unsafe region, ZebRAM performs software integrity checks and error correction whenever data in the unsafe region is accessed. To protect the safe region from accesses to the unsafe region, ZebRAM uses a cache in front of the unsafe region. This cache is allocated from the safe region and ZebRAM is free to choose its size and replacement policy in a way that protects the safe region. Finally, to provide backward-compatibility with memory management in commodity systems, ZebRAM can employ translation mechanisms provided by hardware (e.g., virtualization extensions in commodity processors) to translate even rows into a contiguous physical address space for the guest.

To maintain good performance, ZebRAM ensures that accesses to the safe region proceed without interposition. As mentioned earlier, this can potentially cause bit flips in the unsafe region. Hence, all accesses to the unsafe region should be interposed for bit flip detection and correction. To this end, ZebRAM exposes the unsafe region as a swap device to the protected operating system. With this design, ZebRAM reuses existing page replacement policies of the operating system to decide which memory pages should be evicted to the swap (i.e., unsafe

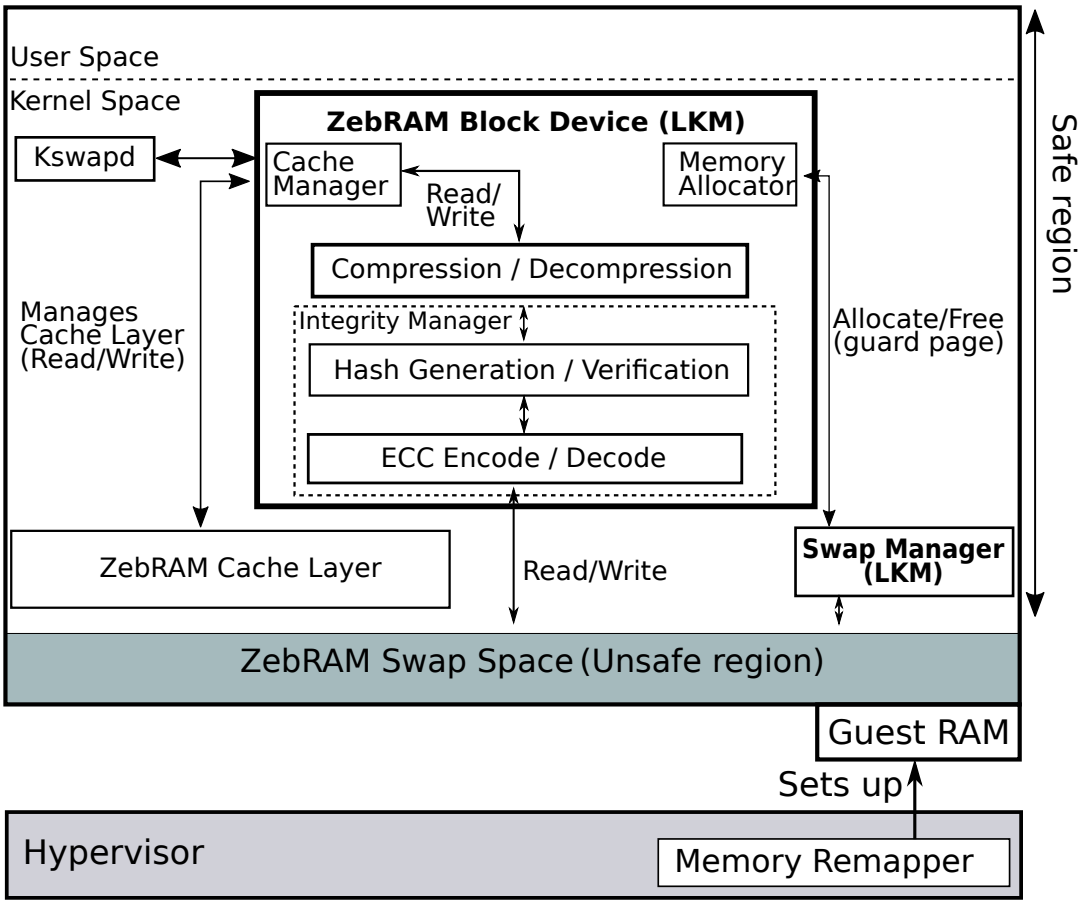


FIGURE 2.6: ZebRAM Components.

region). Given that most operating systems use some form of Least Recently Used (LRU), the working set of the system remains in the safe region, preserving performance. Once the system needs to access a page from the unsafe region, the operating system selects a page from the safe region (e.g., based on LRU) and creates necessary meta data for bit flip detection (and/or correction) using the contents of the page and writes it to the unsafe region. At this point, the system can bring the page to the safe region from the unsafe region. But before that, it uses the previously calculated meta data to perform bit flip detection and correction. Note that the swap cache (for protecting the safe region) is essentially part of the safe region and is treated as such by ZebRAM.

Next, we discuss our implementation of ZebRAM's design before analyzing its security guarantees and evaluating its performance.

2.5 Implementation

In this section, we describe a prototype implementation of ZebRAM on top of the Linux kernel. Our prototype protects virtual machines against Rowhammer attacks and consists of the following four components: the *Memory Remapper*, the *Integrity Manager*, the *Swap Manager*, and the *Cache Manager*, as shown in Figure 2.6. Our prototype implements Memory Remapper in the hypervisor and the other three components in the guest OS. It is possible to implement all the components in the host to make ZebRAM guest-transparent. We discuss alternative implementations and their associated trade-offs in Section 2.9.2. We now discuss these components as implemented in our prototype.

2.5.1 ZebRAM Prototype Components

Memory Remapper implements the split of physical memory into a safe and unsafe region. One region contains all the even-numbered rows, while the other contains all the odd-numbered rows. Note that because hardware vendors do not publish the mapping of physical addresses to DRAM addresses, we need to reverse engineer this mapping following the methodology established in prior work [138, 171, 154].

Because Rowhammer attacks only affect directly neighboring rows, a Rowhammer attack in one region can only incur bit flips in the other region, as shown in Figure 2.3. In addition, ZebRAM supports the conservative option of increasing the number of guard rows to defend against Rowhammer attacks that target a victim row not directly adjacent to the aggressor row. However, our experience with a large number of vulnerable DRAM modules shows that with the correct translation of memory pages to DRAM locations, bit flips trigger exclusively in rows adjacent to a row that is hammered.

Integrity Manager protects the integrity of the unsafe region. Our software design allows for a flexible choice for error detection and correction. For error correction, we use a commonly-used Single-Error Correction and Double-Error Detection (SECDED) code. As shown in recent work [40], SECDED and other similar BCH codes can still be exploited on DIMMs with large number of bit flips. Our database of Rowhammer bit flips from 14 vulnerable DIMMs [154] shows that only 0.00015% of all memory words with bit flips can bypass our SECDED code (found in 2 of the 14 vulnerable DIMMs) and 0.13% of them can cause a detectable corruption (found in 7 of the 14 vulnerable DIMMs). To provide strong detection guarantees, while providing correction possibilities, ZebRAM provides the possibility to mix SECDED with collision resistant hash functions such as SHA-256 at the cost of extra performance overhead.

Swap Manager uses the unsafe region to implement an efficient swap disk in memory, protected by the Integrity Manager and accessible only by the OS. Using the unsafe region as a swap space has the advantage that the OS will only access the slow, integrity-checked unsafe region when it runs out of fast safe memory. As with any swap disk, the OS uses efficient page replacement techniques to minimize access to it. To maximize utilization of the available memory, the Swap Manager also implements a compression engine that optionally compresses pages stored in the swap space.

Note that ZebRAM also supports configurations with a dedicated swap disk (such as a hard disk or SSD) in addition to the memory-based swap space. In this case, ZebRAM swap is prioritized above any other swap disks to maximize efficiency.

Cache Manager implements a fully associative cache that speeds up access to the swap space while simultaneously preventing Rowhammer attacks against safe rows by reducing the access frequency on memory rows in the unsafe region. The swap cache is faster than the swap disk because it is located in the safe region and does not require integrity checks or compression. Because attackers must clear the swap cache to be able to directly access rows in the unsafe region, the cache prevents attackers from efficiently hammering guard rows to induce bit flips in safe rows.

Because the cache layer sits in front of the swap space, pages swapped out by the OS are first stored in the cache, in uncompressed format. Only if the cache is full does the Cache Manager flush the *least-recently-added (LRA)* entry to the swap disk. The LRA strategy is important, because it ensures that attackers must clear the *entire* cache after every row access in the unsafe region.

2.5.2 Implementation Details

We implemented ZebRAM in C on an Intel Haswell machine running Ubuntu 16.04 with kernel v4.4 on top a Qemu-KVM v2.11 hypervisor. Next we provide further details on the implementation various components in the ZebRAM prototype.

Memory Remapper To efficiently partition memory into guard rows and safe rows, we use *Second Level Address Translation (SLAT)*, a hardware virtualization extension commonly available in commodity processors. To implement the Memory Remapper component, we patched Qemu-KVM's `mmap` function to expose the unsafe memory rows to the guest machine as a contiguous memory block starting at physical address `0x3ffe0000`. We use a translation library similar to that of Throwhammer [155] for assigning memory pages to odd and even rows in the Memory Remapper component.

Integrity Manager The Integrity Manager and Cache Manager are implemented as part of the ZebRAM block device, and comprise 369 and 192 LoC, respectively. The Integrity Manager uses `SHA-256` algorithm for error detection, implemented in mainline Linux, to hash swap pages, and keeps the hashes in a linear array stored in safe memory. Additionally, the Integrity Manager by default uses an ECC derived from the extended Hamming(63,57) code [80], expurgated to have a message size an integer multiple of bytes. The obtained ECC is a $[64, 56, 4]_2$ block code, providing single error correction and double error detection (SECDED) for each individual (64-bit) memory word—functionally on par with hardware SEC-DED implementations.

Swap Manager The Swap Manager is implemented as a Loadable Kernel Module (LKM) for the guest OS that maintains a stack containing the Page Frame Numbers (PFNs) of free pages in the swap space. It exposes the RAM-based swap disk as a readable and writable block device that we implemented by extending the `zram` compressed RAM block device commonly available in Linux distributions. We changed `zram`'s `zsmalloc` slab memory allocator to only use pages from the Swap Manager's stack of unsafe memory pages. To compress swap pages, we use the LZO algorithm also used by `zram` [114]. The Swap Manager LKM contains 456 LoC while our modifications to `zram` and `zsmalloc` comprise 437 LoC.

Cache Manager The Cache Manager implements the swap cache using a linear array to store cache entries and a radix tree that maps ZebRAM block device page indices to cache entries. By default, ZebRAM uses 2% of the safe region for the swap cache.

Guest Modifications The guest OS is unchanged except for a minor modification that uses Linux's boot memory allocator API (`alloc_bootmem_low_pages`) to reserve the unsafe memory block as swap space at boot time. Our changes to Qemu-KVM comprise 2.6K lines of code (LoC), while the changes to the guest OS comprise only 4 LoC. Furthermore, the Linux kernel may eagerly write dirty pages into the swap device based on its `swappiness` tunable. In ZebRAM, we use a swappiness of 10 instead of the default value of 60 to reduce the number of unnecessary writes to the unsafe region.

2.6 Security Evaluation

This section evaluates ZebRAM's effectiveness in defending against traditional Rowhammer exploits. Additionally, we show that ZebRAM successfully defends even against more

Run no.	1 bit flip in 64 bits	2 bit flips in 64 bits	Total bit flips	ZebRAM detection performance Detected bit flips	Corrected bit flips
1	4,698	2	4,702	4,702	4,698
2	5,132	0	5,132	5,132	5,132
3	2,790	0	2,790	2,790	2,790
4	4,216	1	4,218	4,218	4,216
5	3,554	0	3,554	3,554	3,554

TABLE 2.1: ZebRAM's effectiveness defending against a ZebRAM-aware Rowhammer exploit.

advanced ZebRAM-aware Rowhammer exploits. We evaluated all attacks on a Haswell i7-4790 host machine with 16GB RAM running our ZebRAM-based Qemu-KVM hypervisor on Ubuntu 16.04 64-bit. The hypervisor runs a guest machine with 4GB RAM and Ubuntu 16.04 64-bit with kernel v4.4, containing all necessary ZebRAM patches and LKMs.

2.6.1 Traditional Rowhammer Exploits

Under ZebRAM's memory model, traditional Rowhammer exploits on system memory only hammer the safe region, and can therefore trigger bit flips only in the integrity-checked unsafe region by construction. We tested the most popular real-world Rowhammer exploit variants to confirm that ZebRAM correctly detects these integrity violations.

In particular, we ran the single-sided Rowhammer exploit published by Google's Project Zero,[2] as well as the one-location[3] and double-sided[4] exploits published by Gruss et al. on our testbed for a period of 24 hours. During this period the single-sided Rowhammer exploit induced two bit flips in the unsafe region, while the one-location and double-sided exploits failed to produce any bit flips. ZebRAM successfully detected and corrected all of the induced bit flips.

The double-sided Rowhammer exploit failed due to ZebRAM's changes in the DRAM geometry, alternating safe rows with unsafe rows. Conventional double-sided exploits attempt to exploit a victim row k by hammering the rows $k-1$ and $k+1$ below and above it, respectively. Under ZebRAM, this fails because the hammered rows are not really adjacent to the victim row, but remapped to be separated from it by unsafe rows. Unaware of ZebRAM, the exploit thinks otherwise based on the information gathered from the Linux' `pagemap`—due to the virtualization-based remapping layer—and essentially behaves like an unoptimized single-sided exploit. Fixing this requires a ZebRAM-aware exploit that hammers two consecutive rows in the safe region to induce a bit flip in the unsafe region. As described next, we developed such an exploit and tested ZebRAM's ability to thwart it.

2.6.2 ZebRAM-aware Exploits

To further demonstrate the effectiveness of ZebRAM, we developed a ZebRAM-aware double-sided Rowhammer exploit. This section explains how the exploit attempts to hammer both the safe and unsafe regions, showing that ZebRAM detects and corrects all the induced bit flips.

Attacking the Unsafe Region

To induce bit flips in the unsafe region (where the swap space is kept), we modified the double-sided Rowhammer exploit published by Gruss et al. [76] to hammer every pair of two consecutive DRAM rows in the safe region (assuming the attacker is armed with an ideal

[2] https://github.com/google/rowhammer-test
[3] https://github.com/IAIK/flipfloyd
[4] https://github.com/IAIK/rowhammerjs/tree/master/native

ZebRAM-aware memory layout oracle) and ran the exploit five times, each time for 6 hours. As Table 2.1 shows, the first exploit run induced a total of 4,702 bit flips in the swap space, with 4,698 occurrences of a single bit flip in a 64-bit data word and 2 occurrences of a double bit flip in a 64-bit word. ZebRAM successfully corrected all 4,698 single bit flips and detected the double bit flips. As shown in Table 2.1, the other exploit runs produced similar results, with no bit flips going undetected. Note that ZebRAM can also detect more than two errors per 64-bit word due to its combined use of ECC and hashing, although our experiments produced no such cases.

Attacking the Safe Region

In addition to hammering safe rows, attackers may also attempt to hammer unsafe rows to induce bit flips in the safe region. To achieve this, an attacker must trigger rapid writes or reads of pages in the swap space. We modified the double-sided Rowhammer exploit to attempt this by opening the swap space with the *open* system call with the *O_DIRECT* flag, followed by repeated *preadv* system calls to directly read from the ZebRAM swap disk (bypassing the Linux page cache).

Because the swap disk only supports page-granular reads, the exploit must read an entire page on each access. Reading a ZebRAM swap page results in at least two memory copies; first to the kernel block I/O buffer, and next to user space. The exploit evicts the ZebRAM swap cache before each swap disk read to ensure that it accesses rows in the swap disk rather than in the cache (which is in the safe region). After each page read, we use a `clflush` instruction to evict the cacheline we use for hammering purposes. Note that this makes the exploit's job easier than it would be in a real attack scenario, where the exploit cannot use `clflush` because the attacker does not own the swap memory. A real attack would require walking an entire cache eviction buffer after each read from the swap disk.

We ran the exploit for 24 hours, during which time the exploit failed to trigger any bit flips. This demonstrates that the slow access frequency of the swap space—due to its page granularity, integrity checking, and the swap cache layer—successfully prevents Rowhammer attacks against the safe region.

To further verify the reliability of our approach, we re-tested our exploit with the swap disk's cache layer, compression engine, and integrity checking modules disabled, thus providing overly optimistic access speeds (and security guarantees) to the swap space for the Rowhammer exploit. Even in this scenario, the page-granular read enforcement of the swap device alone proved sufficient to prevent any bit flips. Our time measurements using `rdtsc` show that even in this optimistic scenario, memory dereferences in the swap space take 2,435 CPU cycles, as opposed to 200 CPU cycles in the safe region, removing any possibility of a successful Rowhammer attack against the safe region.

2.7 Performance Evaluation

This section measures ZebRAM's performance in different configurations compared to an unprotected system under varying workloads. We test several different kinds of applications, commonly considered for evaluation by existing systems security defenses. First, we test ZebRAM on the SPEC CPU2006 benchmark suite [81] to measure its performance for CPU-intensive applications. We also benchmark ZebRAM the popular `nginx` and Apache web servers, as well as the `redis` in-memory key-value store. Additionally, we present microbenchmark results to better understand the contributing factors to ZebRAM's overhead.

Testbed Similar to our security evaluation, we conduct our performance evaluation on a Haswell i7-4790 machine with 16GB RAM running Ubuntu 16.04 64-bit with our modified

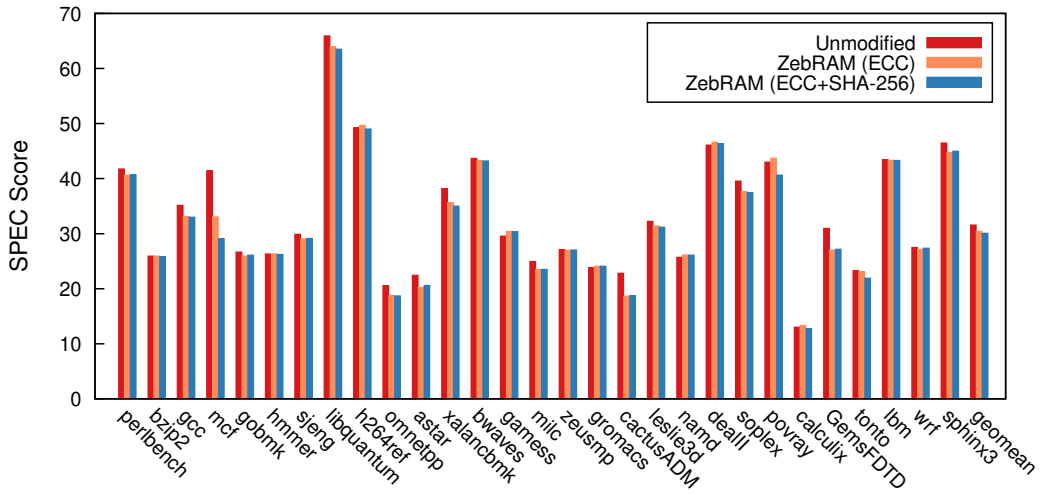


FIGURE 2.7: SPEC CPU 2006 performance results.

Qemu-KVM hypervisor. We run the ZebRAM modules and the benchmarked applications in an Ubuntu 16.04 guest VM with kernel v4.4 and 4GB of memory using a split of 2GB for the safe region and 2GB for the unsafe region. To establish a baseline, we use the same guest VM with an unmodified kernel and 4GB of memory. In the baseline measurements, the guest VM has direct access to all its memory, while in the ZebRAM performance measurements half of the memory is dedicated to the ZebRAM swap space. In all reported memory usage figures we include memory used by the Integrity Manager and Cache Manager components of ZebRAM. For our tests of server applications, we use a separate Skylake i7-6700K machine as the client. This machine has 16GB RAM and is linked to the ZebRAM machine via a 100Gbit/s link. We repeat all our experiments multiple times and observe marginal deviations across runs.

SPEC 2006 We compare performance scores of the SPEC 2006 benchmark suite in three different setups: (i) unmodified, (ii) ZebRAM configured to use only ECC, and (iii) ZebRAM configured to use ECC and SHA-256. The ZebRAM (ECC) and ZebRAM (ECC and SHA-256) show a performance overhead over the unmodified baseline of 4% and 5%, respectively (see Figure 2.7). The reason behind this performance overhead is that as the ZebRAM splits the memory in a zebra pattern, the OS can no longer benefit from huge pages. Also, note that certain benchmarks, such as `mcf`, exhibits more than 5% overhead because they use ZebRAM's swap memory as their working set do not fit in the safe region.

Web servers We evaluate two popular web servers: nginx (1.10.3) and Apache (2.4.18). We configure the virtual machine to use 4 VCPUs. To generate load to the web servers we use the wrk2 [169] benchmarking tool, retrieving a default static HTML page of 240 characters. We set up nginx to use 4 workers, while we set up Apache with the prefork module, spawning a new worker process for every new connection. We also increase the maximum number of clients allowed by Apache from 150 to 500. We configured the wrk2 tool to use 32 parallel keep-alive connections across 8 threads. When measuring the throughput we check that CPU is saturated in the server VM. We discard the results of 3 warmup rounds, repeat a one-minute run 11 times, and report the median across runs. Figure 2.8 shows the throughput of ZebRAM under two different configurations: (i) ZebRAM configured to use only ECC, and

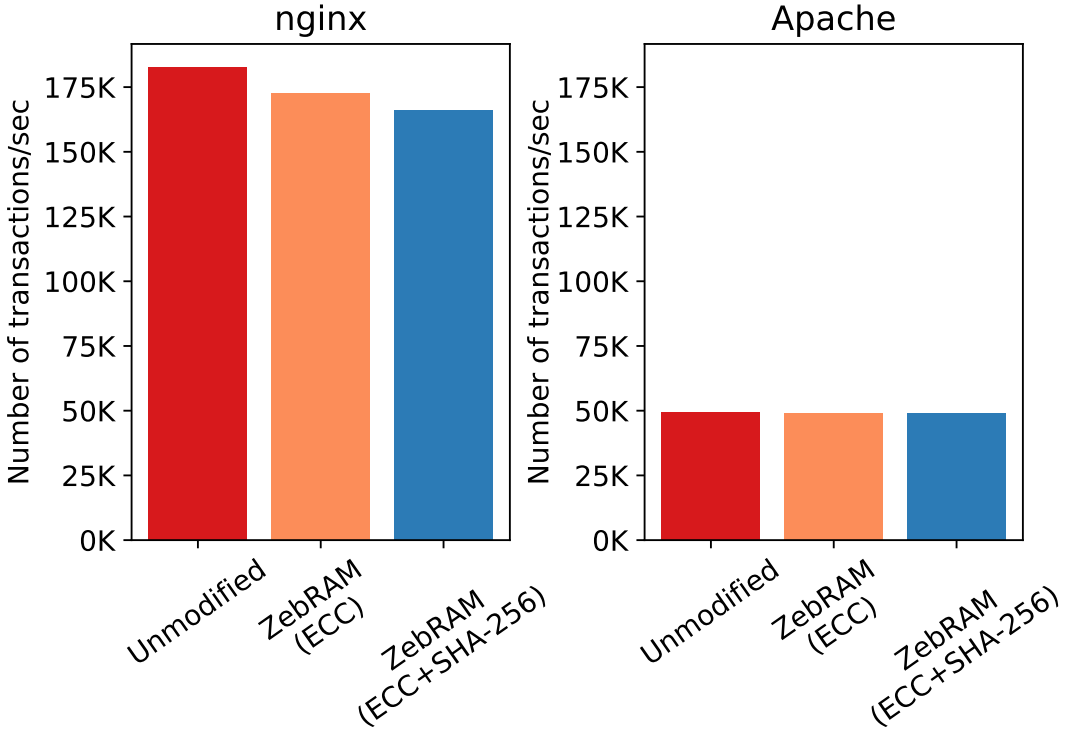


FIGURE 2.8: Nginx and Apache throughput at saturation.

(ii) ZebRAM configured to use ECC and SHA-256. Besides throughput, we want to measure ZebRAM's latency impact. We use wrk2 to throttle the load on the server (using the rate parameter) and report the 99th percentile latency as a function of the client request rate in Figure 2.9.

The baseline achieves 182k and 50k requests per second on Nginx and Apache respectively. The ZebRAM's first configuration (only ECC) reaches 172k and 49k while the second configuration reaches 166k and 49k.

Before saturation, the results show that ZebRAM imposes no overhead on the 99th percentile latency. After then, both configurations of ZebRAM show a similar trend with linearly higher 99th percentile response time.

Overall, ZebRAM's performance impact on both web servers and SPEC benchmarks is low and mostly due to the inability to efficiently use Linux' THP support. This is expected, since as long as the working set can comfortably fit in the safe region (e.g., around 400MB for our web server experiments) the unsafe memory management overhead is completely masked. We isolate and study such overhead in more detail in the following.

Microbenchmarks To drill down the overhead of each single feature of ZebRAM, we measure the latency of swapping in a page from the ZebRAM device under different configurations. To measure the latency, we use a small binary that sequentially writes on every page of a large eviction buffer in a loop. This ensures that, between two accesses to the same page, we touch the entire buffer, evicting that page from memory. To be sure that Linux swaps in just one page for every access, we set the page-cluster configuration parameter to 0. In this experiment, two components interact with ZebRAM: our binary triggers swap-in events from the ZebRAM device while the `kswapd` kernel thread swaps pages to the ZebRAM device to free memory. The interaction between them is completely different if the binary uses exclusively loads to

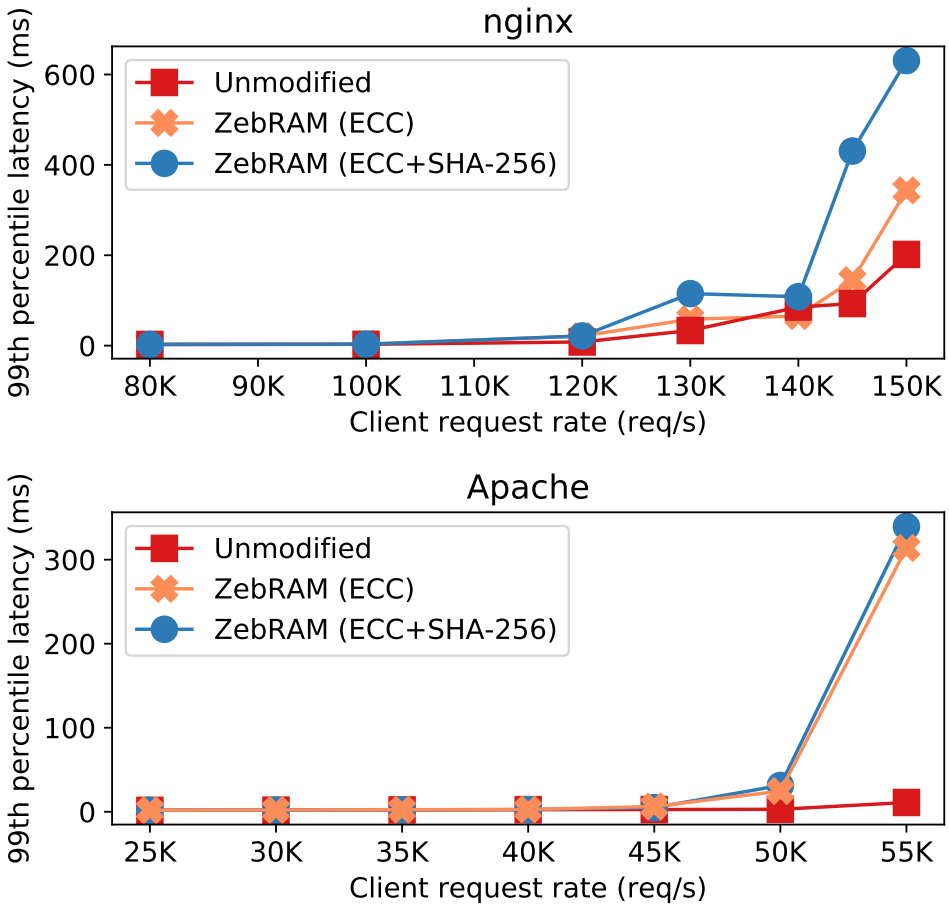


FIGURE 2.9: Nginx and Apache latency (99th percentile).

stress the memory. This is because the kernel would optimize out unnecessary flushes to swap and batch together TLB invalidations. Hence, we choose to focus on stores to study the performance in the worst-case scenario and because read-only workloads are less common than mixed workloads.

We reserve a core exclusively for the binary so that `kswapd` does not (directly) steal CPU cycles from it. We measure 1,000,000 accesses for each different configuration. Table 2.2 presents our results.We also run the binary in a loop and profile its execution with the `perf` Linux tool to measure the time spent in different functions. Due to function inlining, it is not always trivial to map a symbol to a particular feature. Nevertheless, `perf` can provide insights into the overhead at a fine granularity. In the first configuration, we disable the all features of ZebRAM and perform only memory copies into the ZebRAM device. As the copy operation is fast, the `perf` tool reports that just 4% percent of CPU cycles are spent copying. Interestingly, 47% of CPU cycles are spent serving Inter Process Interrupts from other cores. This is because, while we are swapping in, `kswapd` on another core is busy freeing memory. For this purpose, `kswapd` needs to unmap pages that are on their way to be swapped out from the process's page tables. This introduces TLB shootdowns (and IPIs) to invalidate other cores' TLB stale entries. It is important to notice that the faster we swap in pages, the faster `kswapd` needs to free memory. This unfortunately results in a negative feedback loop that represent one of the major sources of overhead when the large number of swap-in events continuously force `kswapd` to wake up.

Configuration	median (ns)	90th (ns)	99th (ns)
copy	2,362.0	4,107.0	8,167.0
SHA-256	13,552.0	14,209.0	17,092.0
cache + comp + SHA-256	8,633.0	13,191.0	18,678.0
cache + comp + SHA-256 + ECC	9,862.0	15,118.0	20,794.0

TABLE 2.2: Page swap-in latency from the ZebRAM device.

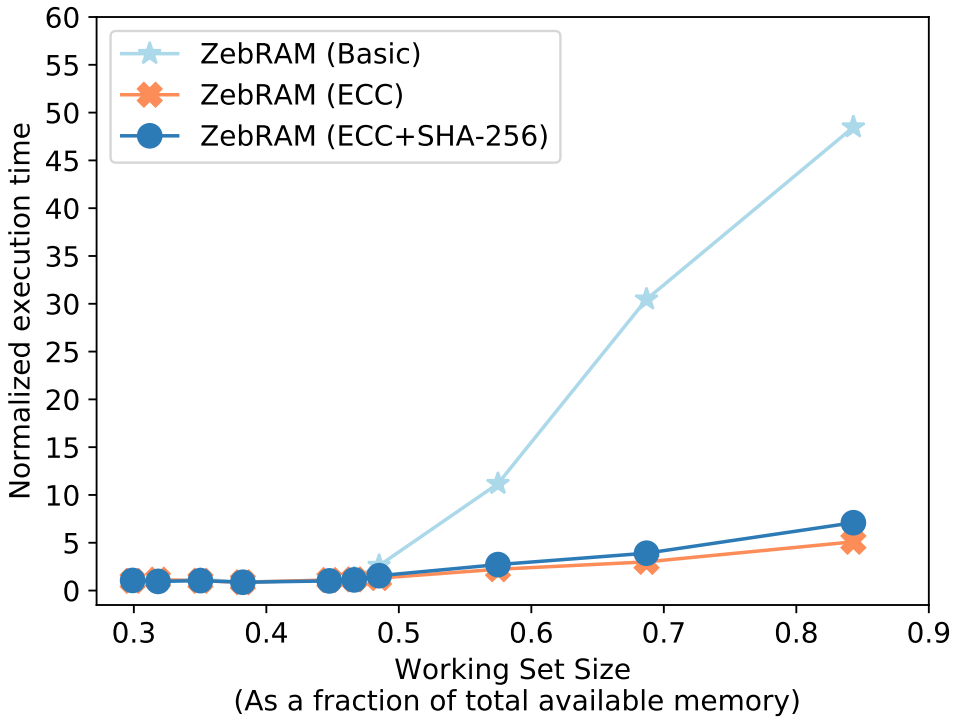


FIGURE 2.10: Redis throughput at saturation.

Adding hashing (SHA-256) on top of the previous configuration shows an increase in latency, which is also reflected in the CPU cycles breakdown. The `perf` tool reports that 55% of CPU cycles are spent swapping in pages (copy + hashing), while serving IPIs accounts for 29%. Adding cache and compression on top of SHA-256 decreases the latency median and increases the 99th percentile. This is because, on a cache hit, the ZebRAM only needs to copy the page to userspace; however, on a cache miss, it has to verify the hash of the page and decompress the page too. The `perf` tool reports 42% of CPU cycles are spent in the decompression routine and 26% in serving IPI requests for other cores and less than 5% in hashing and copying. This confirms the presence of the swap-in/swap-out feedback loop under high memory pressure. Adding ECC marginally increases the latency, the `perf` tool reports similar CPU usage breakdown for the version without ECC.

Larger working sets As expected, ZebRAM's overheads are mostly associated to swap-in/swap-out operations, which are masked when the working set can fit in the safe region but naturally become more prominent as we grow the working set. In this section, we want to evaluate the impact of supporting increasingly larger working sets compared to a more traditional swap implementation. For this purpose, we evaluate the performance of a key-value store in four different setups: (i) unmodifed system, (ii) the basic version of ZebRAM (iii)

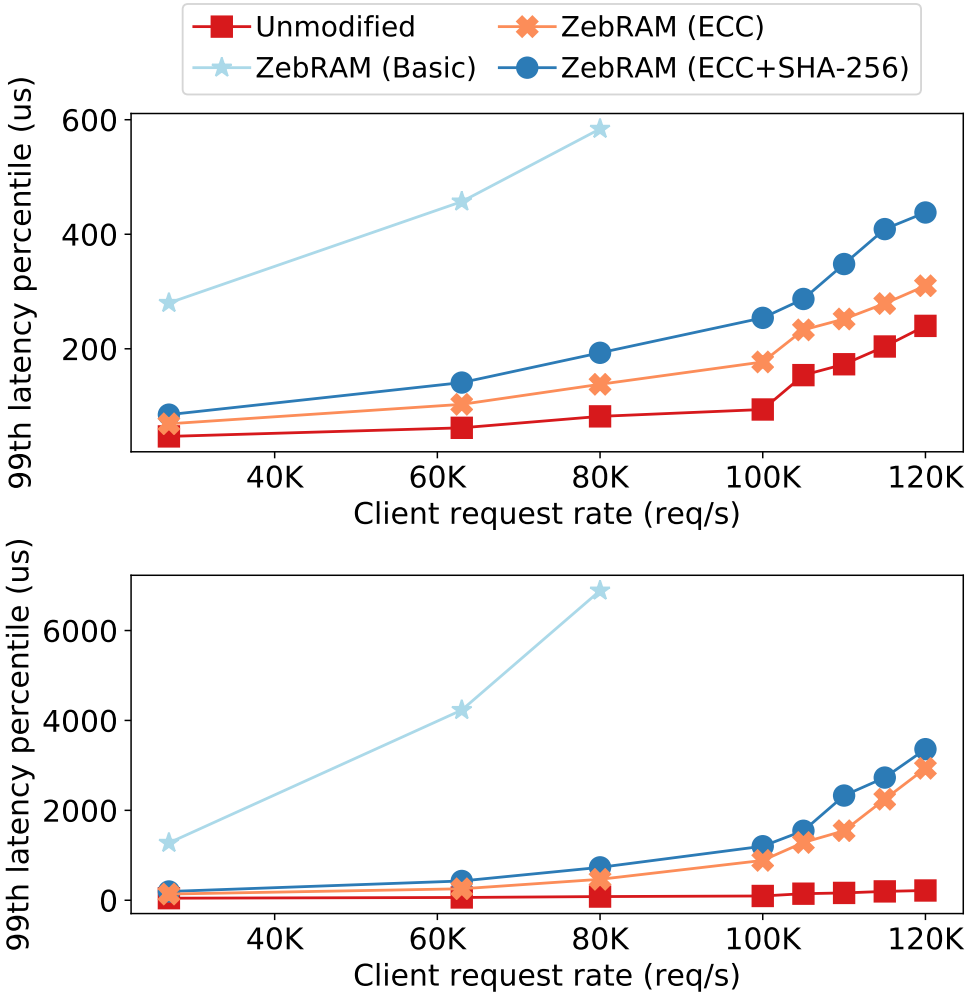


FIGURE 2.11: Redis latency (99th percentile). The working set size is 50% of RAM (top) and 70% of RAM (bottom).

ZebRAM configured with ECC, and (iv) ZebRAM configured with ECC and SHA-256. The basic version of ZebRAM uses just one of the two domains in which ZebRAM splits the RAM and swaps to a fast SSD disk when the memory used by the OS does not fit into it. We use YCSB[42] to generate load and induce a target working set size against a redis (4.0.8) key-value store. We setup YCSB to use 1KB objects and perform a 90/10 read/write operations ratio. Each test runs for 20 seconds and, for each configuration, we discard the results of 3 warmup rounds and report the median across 11 runs. We configure YCSB to access the dataset key space uniformly and we measure the throughput at saturation for different data set sizes.

Figure 2.10 depicts the reported normalized execution time as a function of the working set size (in percentage compared to the total RAM size). As shown in the figure, when the working set size is small enough (e.g., 44%) the OS hardly reclaims any memory, hence the unsafe region remains unutilized and the normalized execution time is only 1.08x for the basic version of ZebRAM while the normalized execution time is between 1.04x and 1.10x for all other configurations of ZebRAM. As we increase the working set size, the OS starts reclaiming pages and the normalized execution time increases accordingly. However, the increase is much more gentle for ZebRAM compared to the basic version of ZebRAM

and the gap becomes more significant for larger working set sizes. For instance, for a fairly large working set size (e.g., 70%), ZebRAM (ECC) has 3.00x normalized execution time, and ZebRAM (ECC and SHA-256) has 3.90x, compared to the basic version of ZebRAM at 30.47x.

To study the impact of ZebRAM on latency, we fix the working set size to 50% and 70% of the total RAM and repeat the same experiment while varying the load on the server. Figure 2.11 presents our results for the 99th latency percentile. At 50%, results of (i) the ZebRAM configured with ECC, (ii) the ZebRAM configured with ECC and SHA-256, and (iii) baseline (unmodified) follow the same trend. The ZebRAM's first configuration (only ECC) reports a 99th latency percentile of 138us for client request rates below 80,000, compared to 584us for ZebRAM (basic). At 70%, the gap is again more prominent, with ZebRAM reporting a 99th latency percentile of 466us and ZebRAM (basic) reporting 6,887us.

Overall, ZebRAM can more gracefully reduce performance for larger working sets compared to a traditional (basic ZebRAM) swap implementation, thanks to its ability to use an in-memory cache and despite the integrity checks required to mitigate Rowhammer. As our experiments demonstrate, given a target performance budget, ZebRAM can support much larger working sets compared to the ZebRAM's basic implementation, while providing a strong defense against arbitrary Rowhammer attacks. This is unlike the basic ZebRAM implementation, which optimistically provides no protection against similar bit flip-based attacks. Unfortunately, such attacks, which have been long-known for DRAM [96], have recently started to target flash memory as well [31, 103].

2.8 Related work

This section summarizes related work on Rowhammer attacks and defenses.

Attacks In 2014, Kim et al. [96] were the first to show that it is possible to flip bits in DDR3 memory on x86 CPUs simply by accessing other parts of memory. Since then, many studies have demonstrated the effectiveness of Rowhammer as a real-world exploit in many systems.

The first practical Rowhammer-based privilege escalation attack, by Seaborn and Dullien [148], targeted the x86 architecture and DDR3 memory, hammering the memory rows by means of the native x86 `clflush` instruction that would flush the cache and allow high-frequency access to DRAM. By flipping bits in page table entries, the attack obtained access to privileged pages.

Not long after these earliest attacks, researchers greatly increased the threat of Rowhammer attacks by showing that is possible to launch them from JavaScript also, allowing attackers to gain arbitrary read/write access to the browser address space from a malicious web page [26, 76].

Moreover, newer attacks started flipping bits in memory areas other than page table entries, such as object pointers (to craft counterfeit objects [26]), opcodes [77], and even application-level sensitive data [143].

For instance, Flip Feng Shui demonstrated a new attack on VMs in cloud environments that flipped bits in RSA keys in victim VMs to make them easy to factorize, by massaging the physical memory of the co-located VMs to land the keys on a page that was hammerable by the attacker. Around the same time, other researchers independently also targeted RSA keys with Rowhammer but now for fault analysis [23]. Concurrently, also, Xiao et al. [171] presented another cross-VM attack that manipulates page table entries in Xen.

Where the attacks initially focused on PCs with DDR3 configurations, later research showed that ARM processors and DDR4 memory chips are also vulnerable [159]. While this opened the way for Rowhammer attacks on smartphones, the threat was narrower than on

PCs, as the authors were not yet able to launch such attacks from JavaScript. This changed recently, when research described a new way to launch Rowhammer attacks from JavaScript on mobile phones and PC, by making use of the GPU. Hammering directly from the GPU by way of WebGL, the authors managed to compromise a modern smart phone browser in under two minutes. Moreover, this time the targeted data structures are doubles and pointers: by flipping a bit in the most significant bytes, the attack can turn pointers into doubles (making them readable) and doubles into pointers (yielding arbitrary read/write access).

All Rowhammer attacks until that point required local code execution. Recently, however, researchers demonstrated that even remote attacks on servers are possible [155], by sending network traffic over high-speed network to a victim process, using RDMA NICs. As the server that is receiving the network packets is using DMA to write to its memory, the remote attacker is able to flip bits in the server. By carefully manipulating the data in a key-value store, they show that it is possible to completely compromise the server process.

It should be clear that Rowhammer exploits have spread from a narrow and arcane threat to target two of the most popular architectures, in all common computing environments, different types of memory (and arguably flash [31]), while covering most common threat models (local privilege escalation, hosted JavaScript, and even remote attacks). ZebRAM protects against all of the above attacks.

Defenses Kim et al. [96] propose hardware changes to mitigate Rowhammer by increasing row refresh rates or using ECC. These defenses have proven insufficient [16] and infeasible to deploy on the required massive scale. The new LPDDR4 standard [93] specifies two features which together defend against Rowhammer: TRR and MAC. Despite these defenses, van der Veen et al. still report bit flips on a Google pixel phone with LPDDR4 memory [158] and Gruss et al. [147] report bit flips in TRR memory. While nobody has demonstrated Rowhammer attacks against ECC memory yet, the real problem with such hardware solutions is that most systems in use today do not have ECC, and replacing all DRAM in current devices is simply infeasible.

In order to protect from Rowhammer attacks, many vendors simply disabled features in their products to make life harder for attackers. For instance, Linux disabled unprivileged access to the *pagemap* [148], Microsoft disabled memory deduplication [48] to defend from the Dedup Est Machina attack [26], and Google disabled [156] the ION contiguous heap in response to the Drammer attack [159] on mobile ARM devices. Worryingly, not a single defence is currently deployed to protect from the recent GPU-based Rowhammer attack on mobile ARM devices (and PCs), even though it offers attackers a huge number of vulnerable devices.

Finally, researchers have proposed targeted software-based solutions against Rowhammer. ANVIL [16] relies on Intel's performance monitoring unit (PMU) to detect and refresh likely Rowhammer victim rows. An improved version of ANVIL requires specialized Intel PMUs with a fine-grained physical to DRAM address translation. Unfortunately, Intel's (and AMD's) PMUs do not capture precise address information when memory accesses bypass the CPU cache through DMA. Hence, this version of ANVIL is vulnerable to off-CPU Rowhammer attacks. Unlike ANVIL, ZebRAM is secure against off-CPU attacks, since device drivers transparently allocate memory from the safe region.

CATT [28] isolates (only) user and kernel space in physical memory so that user-space attackers cannot trigger bit flips in kernel memory. However, research [77] shows CATT to be bypassable by flipping opcode bits in the `sudo` program code. Moreover, CATT does not defend against attacks that target co-hosted VMs at all [28]. In contrast, ZebRAM protects against co-hosted VM attacks, attacks against the kernel, attacks between (and even within) user-space processes and attacks from co-processors such as GPUs.

Other recent software-based solutions have targeted specific Rowhammer attack variants. GuardION isolates DMA buffers to protect mobile devices against DMA-based Rowhammer attacks [160]. ALIS isolates RDMA buffers to protect RDMA-enabled systems against Throwhammer [155]. Finally, VUSion, described in Chapter 1, randomizes page frame allocation to protect memory deduplication-enabled systems against Flip Feng Shui.

2.9 Discussion

This section discusses feature and performance tradeoffs between our ZebRAM prototype and alternative ZebRAM implementations.

2.9.1 Prototype

Because the ZebRAM prototype relies on the hypervisor to implement safe/unsafe memory separation, and on a cooperating guest kernel for swap management, both host and guest need modifications. In addition, the guest physical address space will map highly non-contiguously to the host address space, preventing the use of huge pages. The guest modifications, however, are small and self-contained, do not touch the core memory management implementation and are therefore highly compatible with mainline and third party LKMs.

2.9.2 Alternative Implementations

In addition to our implementation presented in Section 2.5, several alternative ZebRAM implementations are possible. Here, we compare our ZebRAM implementation to alternative hardware-based, OS-based, and guest-transparent virtualization-based implementations.

Hardware-based Implementing ZebRAM at the hardware level would require a physical-to-DRAM address mapping where sets of odd and even rows are mapped to convenient physical address ranges, for instance an even lower-half and an odd upper-half. This can be achieved with by a fully programmable memory controller, or implemented as a configurable feature in existing designs. With such a mapping in place, the OS can trivially separate memory into safe and unsafe regions. In this model, the Swap Manager, Cache Manager and Integrity Manager are implemented as LKMs just as in the implementation from Section 2.5. In contrast to other implementations, a hardware implementation requires no hypervisor, allows the OS to make use of (transparent) huge pages and requires minimal modifications to the memory management subsystem. While a hardware-supported ZebRAM implementation has obvious performance benefits, it is currently infeasible to implement because memory controllers lack the required features.

OS-based Our current ZebRAM prototype implements the Memory Remapper as part of a hypervisor. Alternatively, the Memory Remapper can be implemented as part of the bootloader, using Linux' boot memory allocator to reserve the unsafe region for use as swap space. While this solution obviates the use of a hypervisor, it also results in a non-contiguous physical address space that precludes the use of huge pages and breaks DMA in older devices. In addition, it is likely that this approach requires invasive changes to the memory management subsystem due to the very fragmented physical address space.

Transparent Virtualization-based While our current ZebRAM implementation requires minor changes to the guest OS, it is also possible to implement a virtualization-based variant of ZebRAM that is completely transparent to the guest. This entails implementing the ZebRAM

swap disk device in the host and then exposing the disk to the guest OS as a normal block device to which it can swap out. The drawback of this approach is that it degrades performance by having the hypervisor interposed between the guest OS and unsafe memory for each access to the swap device, a problem which does not occur in our current implementation. The clear advantage to this approach is that it is completely guest-agnostic: guest kernels other than Linux, including legacy and proprietary ones are equally well protected, enabling existing VM deployments to be near-seamlessly transitioned over to a Rowhammer-safe environment.

2.10 Conclusion

We have introduced ZebRAM, the first comprehensive software defense against all forms of Rowhammer. ZebRAM uses guard rows to isolate all memory rows containing user or kernel data, protecting these from Rowhammer-induced bit flips. Moreover, ZebRAM implements an efficient integrity-checked memory-based swap disk to utilize the memory sacrificed to the guard rows. Our evaluation shows ZebRAM to be a strong defense able to use all available memory at a cost that is a function of the workload. To aid future work, we release ZebRAM as open source.

Chapter 3

OpenCAL

3.1 Introduction

Scientific Computing [71] is a broad and constantly growing multidisciplinary research field that uses formal paradigms to study complex problems and solve them through simulation by using advanced computing capabilities.

Different formal paradigms have been proposed to provide the abstraction context in which problems are formalized. Partial Differential Equations (PDEs) were probably the first to be largely employed for describing a wide variety of phenomena. Unfortunately, PDEs can be analytically solved only for a small set of simplified problems [119] and Numerical Methods have to be employed to obtain approximate solutions for real situations. Among them, the Finite Differences Method (FDM) was one of the first considered, still currently employed, to address a wide variety of phenomena such as acoustics [34, 27], heat [141, 45], computational fluid dynamics (CFD) [36, 60], and quantum mechanics [84, 65].

Besides other solutions proposed for numerically approximating PDEs like, for instance, Finite Elements and Finite Volume Methods, further formal paradigms were more recently proposed for modeling complex systems as result of studies in Computer Science. Among them, Cellular Automata (CA) [125] are Turing-equivalent [39, 41] parallel computational models. CA are widely studied from a theoretical point of view [166, 105, 165, 127], and their application domains vary from Artificial Life [106, 20] to Computational Fluid Dynamics [68, 120, 82, 5], besides many others. In the 80s, an extension of the original CA formalism was proposed to better model and simulate a specific set of complex phenomena [75]. Such an extension is known as Complex or Multi-Component Cellular Automata and was applied to the simulation of debris flows [49, 14], lava flows [54, 55, 129, 53], pyroclastic flows [13, 44], forest fires spreading [10, 12], hydrologic and eco-hydrologic modeling [122, 142, 123, 33], soil erosion [50], crowd dynamics [112, 164, 163], urban dynamics [25], besides others. In this chapter we will refer such an extension of the original CA paradigm as Extended Cellular Automata (XCA).

Independently from the adopted formal paradigm, the simulation of complex systems often requires Parallel Computing. OpenMP is the most widely adopted solution for parallel programming on shared memory computers [37]. It fully supports parallel execution on multi-core CPUs and, starting from the 4.0 specification, also includes support for accelerators like graphic processing units (GPUs) or Xeon Phi co-processors. Unfortunately, compilers like gcc currently do not fully support the OpenMP most recent specifications and, in practice, OpenMP-based applications still mainly run on CPUs [129, 8, 139]. However, in recent years, general purpose computing on graphic processing units (GPGPU), which exploits GPUs and many-core co-processors for general purpose computation, has gained wide acceptance as an alternative solution for high-performance computing, resulting in a rapid spread of applications in many scientific and engineering fields [134]. Most implementations are currently based on Nvidia CUDA (see e.g., [24, 52, 61, 51]), one of the first platforms proposed to exploit GPUs computational power on Nvidia hardware. An open alternative to CUDA is OpenCL

[150], an Application Program Interface (API) originally proposed by Apple and currently managed by Khronos Group for parallel programming on heterogeneous devices like CPUs, GPUs, Digital Signal Processors (DSPs), and Field-Programmable Gate Arrays (FPGAs). Interest in OpenCL is continuously growing and many applications can already be found in literature [115, 19, 62, 30]. However, an OpenCL parallelization of a scientific application is often a non-trivial task and, in many cases, requires a thorough re-factorization of the source code. For this reason, many computational layers were proposed, which make many-core co-processors computational power easier to be exploited. For instance, ArrayFire [117] is a mathematical library for matrix-based computation such as linear algebra, reductions, and Fast Fourier transform; clSpMV [152] is a sparse matrix vector multiplication library; clBlas [38] is an OpenCL parallelization of the Blas linear algebra library. Examples of higher level computational layers, which provide the abstraction of formal computational paradigms, are: OPS [145, 89] and OP2 [70, 144], which are open-source frameworks for the execution of structured and unstructured grid applications, respectively, on clusters of GPUs or multi-core CPUs; AQUAgpusph [32], which is a smoothed-particle hydrodynamics solver; ASL [3], an accelerated multi-physics simulation software based, among others, on the Lattice Boltzmann Method; CAMELot [59, 56] and libAuToti [149], which are a proprietary simulation environment and an efficient parallel library for XCA model development, respectively.

In this article we introduce OpenCAL (Open Computing Abstraction Layer), a new open source parallel computing abstraction layer for scientific computing. It provides the Extended Cellular Automata general formalism as a Domain Specific Language, allowing for the straightforward parallel implementation of a wide range of complex systems. Cellular Automata, Finite Differences and, in general, other structured grid-based methods are therefore supported. Different versions of the library allow to exploit both multi- and many-core shared memory devices, as well as distributed memory systems. Specifically, OpenMP- and OpenCL-based implementations have been developed, both of them providing optimized data structures and algorithms to speed-up the execution and allowing for a transparent parallelism to the user. A MPI-based implementation is also currently under development and allows to exploit many-core accelerators on interconnected systems.

Among the above cited software, OPS, ASL and CAMELot probably are the most similar to OpenCAL in terms of modeling and development approach, and could be considered as possible alternatives to the library proposed in this chapter. In particular, OPS provides a straightforward Domain Specific Language for structured grid-based modeling, even if it does not refer to any specific abstract computational formalism. Its main characteristic consists in allowing to obtain different parallel versions of a computational model starting from its serial implementation, thanks to a seamless code-generator approach. Both MPI-based distributed memory and CUDA/OpenCL many-core versions can be obtained in this way, with a minimal effort by the developer. Conversely, ASL provides different higher level modeling abstractions among which the Lattice Boltzmann Method, that is eventually a Cellular Automata-based paradigm. Nevertheless, it currently does not allow for parallel execution on distributed memory systems, which can be a great limitation in some cases. Eventually, CAMELot offers an integrated simulation environment for XCA development and allows for parallel execution on both shared and distributed memory systems thanks to the message passing paradigm, not permitting however the exploitation of modern many-core devices. With respect to the above cited software, OpenCAL provides both the higher CAMELot modeling approach and, similarly to OP2, allows for the execution on a wide range of shared and distributed parallel platforms (even if by adopting a classic library approach). In addition, OpenCAL provides different embedded strategies and optimization algorithms which allow to progressively improve the computational performance of different kinds of models and simulations.

In the following, the OpenCAL architecture is presented and the OpenMP- and OpenCL-based parallel implementations described. We also present and discuss the implementation of

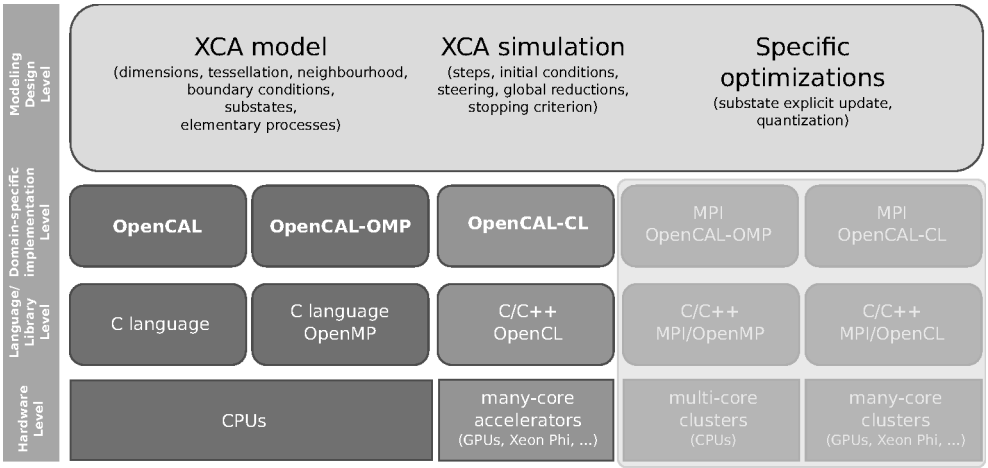


FIGURE 3.1: OpenCAL architecture. At the higher level of abstraction, the model, together with the simulation process and possible optimizations, is designed. The OpenCAL libraries can be found at the implementation abstraction layer, allowing for a straightforward implementation of the designed computational model. OpenCAL-based applications can be therefore executed at the hardware level on both multi-core CPUs and many-core devices. The execution on distributed memory systems is currently under development.

a first simple example of application for multi- and many-core devices to show how straightforward the OpenCAL-based model development is. We therefore consider the *SciddicaT* XCA landslide simulation model [15] as a more complex reference example for correctness and computational performance evaluation on multi-core CPUs, many-core GPUs, and also on a test multi-node GPU-based system. Specifically, we refer to three different versions of *SciddicaT*, which progressively exploit OpenCAL built-in features and, for each of them, we propose different implementations based on the serial and parallel versions of the library. Eventually, results of a further study performed to devise the best platform for execution, depending on the model's computational intensity and the domain extent, is presented. A general discussion concerning OpenCAL and future outcomes concludes the chapter.

3.2 An OpenCAL Overview: Software Architecture, Main Specifications and a First Example of Application

In this section we describe the software architecture, main structures and underlying algorithms of the OpenCAL library, besides a first example of application to highlight how easy model development is. The serial version of the library will be simply referred as OpenCAL in the following, while OpenCAL-OMP and OpenCAL-CL will refer to the OpenMP- and OpenCL-based parallelizations, respectively. Eventually, the preliminary distributed memory version of OpenCAL will be referred as OpenCAL-MPI. The main API data types and functions specifications are here presented. A full API description can be found in the OpenCAL user guide on GitHub.

3.2.1 Software Architecture

The OpenCAL architecture is depicted in Figure 3.1. At the higher level of abstraction, the Scientist conceptually designs the computational model, by referring to the Extended Cellular Automata general formalism. Structured grid-based models whose evolution is determined by

local rules, as well as by global laws or even by a combination of local and global operations, are therefore fully supported. At this level, domain topology and extent, boundary conditions, substates (each of them representing the set of admissible values of a given characteristic assumed to be relevant for the modeled system and its evolution), neighborhood (defining the pattern over which local rules are applied) and elementary processes (defining the local rules of evolution) are formalized. The simulation process is also designed at this level, by specifying the initial conditions of the system, optional global operations (e.g., steering or global reductions), and a termination criterion to stop the system evolution. Note that, being supported by OpenCAL, at this stage some specific optimizations can be applied. Specifically, the explicit updating feature allows to both redefine the elementary processes application order and to selectively update substates after the application of each elementary process, while the *active cells optimization*, also known as *quantization*, allows to restrict the computation to a subset of the whole computational domain, by excluding stationary cells.

The different versions of OpenCAL can be found in the implementation level. Since they provide high-level data structures and algorithms that match the higher abstraction level components, all of them allow for a straightforward implementation of the previously designed computational model, by also allowing to ignore low-level issues like memory management and I/O operations. All OpenCAL versions are written in C for the maximum efficiency and, as pointed out by the language/library level, the OpenMP and OpenCL APIs were considered for implementing the corresponding parallel versions of OpenCAL. Finally, at the hardware level, depending on the adopted version of the library, execution can be performed on single- and multi-core CPUs, as well as on many-core accelerators like GPUs, transparently to the user. Figure 3.1 also shows hybrid MPI/OpenMP and MPI/OpenCL parallel implementations of OpenCAL. The latter, a preliminary implementation of which is in an advanced development state, is that we will refer as OpenCAL-MPI.

3.2.2 OpenCAL Domain Specific API Abstractions

The OpenCAL API was designed to be clear and easy to use. For this purpose, it follows some naming conventions, the most important of which are listed below:

- `CALbyte`, `CALint`, and `CALreal` redefine the `char`, `int` and `double` C native scalar types, respectively;
- Derived data types start with the `CAL` prefix (or `CALCL` for some specific OpenCAL-CL data types), followed by a type identifier formed by one or more capitalized keywords, an optional suffix identifying the model dimension (e.g., `2D` or `3D`), and an eventual optional suffix specifying the basic scalar type, which can be `b`, `i`, or `r`, for `CALbyte`, `CALint` and `CALreal` derived types, respectively (e.g., `CALSubstate3Dr` represents an example of three-dimensional double precision-based data type - cf. below);
- Constants and enumerals start with the `CAL_` prefix, followed by one or more uppercase keywords separated by the _ character (e.g., the `CAL_TRUE` and `CAL_FALSE` Boolean enumerals);
- Functions are characterized by the `cal` prefix (or `calcl` for some specific OpenCAL-CL functions), followed by at least one capitalized keyword, and end with a suffix specifying the model dimension and the basic datatype (e.g., `calSet2Di` represents an example of an API function acting on a bi-dimensional integer based data type).

In the following, the `{arg1|arg2|...|argn}` and `[arg1|arg2|...|argn]` conventions will be adopted: the first one identifies a list of n mutually exclusive arguments, where one of the arguments is needed; the second is used to identify a set of n non-mutually

exclusive optional arguments. As an example, `calGet[X]{2D|3D}{b|i|r}()` function actually identifies a set of API functions with one optional and two mandatory suffixes: the first one, if present, indicates that the function is able to access neighborhood data (`X` is the symbol commonly used in the XCA formalism to refer to the neighborhood), while the other two indicate the domain dimension and the basic type of the data to be accessed, respectively.

One of the most important API objects is the *model*, which is the implementation level object corresponding to the XCA model formalized at the design level. It is simply declared as a pointer to the `CALModel{2D|3D}` built-in data type, and can straightforwardly be defined by means of the `calCADef{2D|3D}()` function. The model object essentially allows to define the domain dimensions (2D and 3D models are natively supported, even if 1D models can be defined as degenerate case of the 2D one), the size of each of them, the cell geometry (square, rectangular and hexagonal cells are supported), the domain topology (e.g., if a 2D domain has to be considered as bounded or as a torus) and the neighborhood pattern, besides embedding a built-in data structure needed by the quantization optimization algorithm (cf. below in this Section). As regards neighborhoods, a set of predefined patterns is provided (e.g., the `CAL_MOORE_NEIGHBORHOOD_{2D|3D}` enumeral refers to the Moore pattern), even if generic neighborhoods can be explicitly defined for maximum flexibility (by using the `CAL_CUSTOM_NEIGHBORHOOD_{2D|3D}` enumeral at definition time and then the `calAddNeighbor{2D|3D}()` function to add neighbors to the initially empty set). Besides the predefined Moore neighborhood, the von Neumann one and 2D-specific hexagonal neighborhoods are also provided by the API. The model object seamlessly manages both the data, mainly represented by *substates* (which are declared as `CALSubstate{2D|3D}{b|i|r}` objects), and the local rules of evolution for the system (i.e., the automaton *transition function*), expressed in terms of *elementary processes* (that are defined as callback functions or OpenCL kernels, depending on the specific OpenCAL implementation). For this purpose, both substates and elementary processes must be registered to the model object (by means of the `calAddSubstate{2D|3D}{b|i|r}()` and `calAddElementaryProcess{2D|3D}()` functions, respectively), which in this way can store pointers to each of them for subsequent seamless indirect access. Note that, a further device-side model object (of type `CALCLModel{2D|3D}`) is provided by the OpenCAL-CL API, which makes data transfer and parallel execution on OpenCL compliant devices transparent to the user. This latter object is declared as a pointer to `CALCLModel{2D|3D}`, and defined by means of the `calclCADef{2D|3D}()` function. Specifically, data transfer from the host to the device global memory is performed at definition time, while data is seamlessly copied back to the host at the end of the simulation process, by minimizing in this way time consuming host to/from device data movements during the computation. To further speed-up the device-side execution, the library also provides the `calclGlobaltoLocal[X]()` API function that can be used within the kernels to transfer data (i.e., central cell and neighborhoods states) from the global to the faster local memory.

According to the XCA computational paradigm, substates define specific characteristics considered to be relevant for the system initial state definition and its evolution. For instance, in a fluid-dynamic computational model, different substates can be used for modeling mass, viscosity and velocity field components. Each substate object has the same extent of the whole computational domain, so that each cell is characterized by specific substates values. Implicitly, this leads to a SoA (Structure of Arrays) approach, which proved to be the most effective in the case of parallel programming on GPUs (see e.g., [46]). For efficient access due to memory coalescence issues, substates objects are implemented by means of linearized arrays. Nevertheless, internal format is transparent to the user, which can access data by means of multidimensional indices and neighborhoods identifiers (e.g., the `calGet[X]{2D|3D}()` function allows to retrieve the current state of the central cell and - if the `X` optional suffix

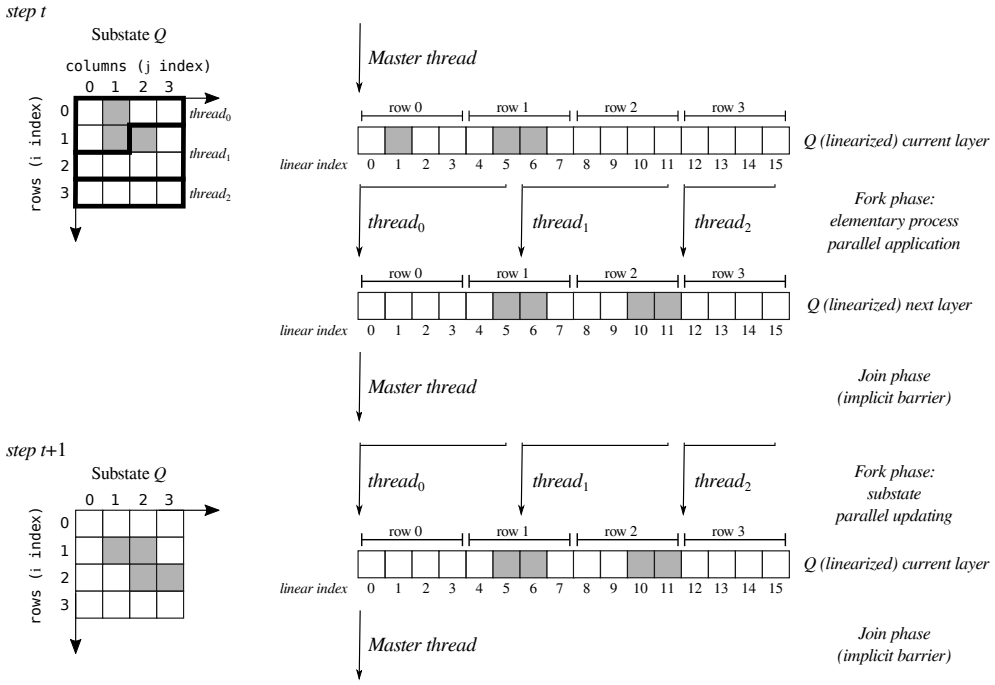


FIGURE 3.2: An example of OpenCAL-OMP parallel application of an elementary process to a substate Q and its subsequent parallel updating. The computational domain is initially partitioned by means of a pool of three threads (fork phase). These latter concurrently apply the elementary process by reading state values from the current layer and by updating new values to the next one. At the end of the elementary process application, threads implicitly synchronize by joining into the master one (join phase), and the parallel update phase starts. As before, a pool of threads concurrently copies the next layer into the current one and the new configuration of Q is obtained. A join phase eventually occurs, which ensures data consistency before the application of another elementary process.

is present - its neighbors, while the `calSet{2D|3D}()` one permits to update the central cell's state). Behind the scene, substates are defined by means of two *computational layers*: the *current* layer represents a read-only memory and is used for retrieving central and neighboring cells current states, while the *next* one is used only for updating the new value of the central cell. Once all new states have been written to the next layer, the substate is seamlessly updated (even if the update phase can be made explicit by means of the `calUpdateSubstate{2D|3D}{b|i|r}` function - cf. below in this Section), i.e. the next layer is copied into the current one, and the substate object is ready for further processing. Note that, as already stated, in the case of OpenCAL-CL, substates are updated device-side, by allowing to perform the whole computation process on the device. Eventually, each OpenCAL implementation also provides *single layer substates*, which only consist of the current computational layer. They are declared as standard double-layered objects, even if the *next* layer is lost at registration time, where the `calAddSingleLayerSubstate{2D|3D}{b|i|r}()` function must be used (instead of the `calAddSubstate{2D|3D}{b|i|r}()` one). Single layer substates can be considered for internal transformations processing, i.e. for those modeling specific rules which determine the substate change within the cell as a function of the central cell state only. Not needing to be updated, they represent a lighter memory and more efficient alternative to double-layer substates.

The system evolution is obtained by applying the elementary processes composing the

transition function in the same order in which they have been registered to the model object (even if the predefined order can be overridden - cf. below in this Section) and, after the application of each of them, by updating the involved double-layer substates. As anticipated, elementary processes are implemented by means of callback functions or OpenCL kernels and their execution is transparently performed by the library (even if, in case of host-side execution, elementary processes application, and also substates updating, can be made explicit - cf. below in this Section). According to the XCA paradigm, each cell must appear to be updated simultaneously to each other (*implicit parallelism*). For this purpose, a pool of concurrent threads/work-items should apply the elementary process simultaneously to each cell of the computational domain. However, depending on the domain dimensions, this is not always possible, even in the case of parallel execution on many-core devices. Nevertheless, implicit parallelism is guaranteed in each OpenCAL implementation thanks to double-layer substates. In fact, by using the current layer as read only memory and the next one for updating purposes only, cells appear to be simultaneously updated with respect to each other, even in the case of serial computation. In this respect, an elementary process is equivalent to a `parallel for` loop, which transparently applies its local rule of evolution simultaneously to each cell of the computational domain. In the case of parallel execution, a data parallel approach is adopted. In particular, the domain is decomposed in uniform chunks in the case of OpenCAL-OMP, while a one cell/one work-item decomposition is adopted in OpenCAL-CL. OpenCAL-MPI currently adopts a classical chunk-based domain decomposition with seamless halos exchange, and also a one cell/one work-item model at GPU level. Note that double layer substates allow for a lock-free parallelization in all cases. In fact, no race conditions can occur since, in particular, the update phase is limited by definition of the XCA computational paradigm to the memory location associated with the central cell. An example is shown in Figure 3.2 for the case of OpenCAL-OMP, where a pool of three threads concurrently process an uniformly partitioned domain for both elementary processes application and substates updating. In this case, the third thread is completely wasted, since it only processes a subset of stationary cells, and therefore a load unbalance occurs. In such a case, a dynamic scheduling is seamlessly adopted in OpenCAL-OMP to mitigate the unbalance among chunks. Regarding the OpenCAL-CL specific case, a grid of OpenCL work-items is adopted for a SIMD-based parallelization. Depending on the dimension of the computational model, two- or three-dimensional OpenCL index spaces (i.e. OpenCL NDRanges) are transparently considered, while a one-dimensional one is adopted in the case the quantization optimization is exploited (cf. below in this Section). The number of work-items to be adopted is evaluated for each model dimension by preliminary querying OpenCL for the (device-dependent) preferred work-group size multiple w_s (i.e. the warp/wavefront size in NVIDIA/AMD GPUs), and therefore by considering the smallest multiple of w_s which is greater than or equal to the model dimension. For instance, if $w_s = 32$ and the first dimension of the domain is 2000, the number of work-items in that dimension will be 2016, i.e. the first multiple of 32 which is greater than or equal to 2000, thus resulting in 16 redundant work-items. However, since redundant work-items do not map any cell of the computational domain, they immediately terminate their execution. Moreover, according to OpenCL, work-items are grouped in work-groups. The choice of the number of work-groups to be considered (and therefore the work-group size) depends on the device architecture and can be both transparently determined (default setting), or explicitly set for finer tuning.

In case of host-side execution, i.e. when OpenCAL and OpenCAL-OMP are considered, simulation execution is managed by a specific *simulation* object, that must be declared as a pointer to the `CALRun{2D|3D}` data type and then defined by means of the `calRunDef{2D|3D}()` function. Nevertheless, in the case of device-side execution, i.e. when OpenCAL-CL is considered, the role of the simulation object is played by the device-side model. Among others, the simulation object defines the substates updating policy: in case of implicit scheme, the built-in transition function is applied (i.e. elementary processes are

applied in the same order in which they have been registered to the model and all registered substates are updated after the application of each of them); when the explicit policy is adopted, the transition function must be overridden and elementary processes explicitly applied, as well as substates explicitly updated. Allowing to avoid the update of unmodified substates (therefore unneeded memory copy operations) the OpenCAL implicit naive approach, which is provided as first instance to allow the developer to completely ignore underlying data structures issues, can be overcome. For this and other purposes, the simulation object can optionally register one or more global callback functions, listed below:

- `init()`: It is executed once before the simulation loop and can be used used to set the initial conditions of the system.
- `globalTransition()`: It overrides the built-in transition function and can be used to redefine the execution order of the registered elementary processes and to perform selective substates updating. The function also allows to perform global operations over the computational domain, e.g., reductions. Built-in reductions allow to compute global minimum, maximum, sum, product, as well as logical and bit-wise AND, OR and NOT operations on the registered substates.
- `steering()`: It is executed at the end of each computational step and can be used to perform generic global operations, as well as global reductions.
- `stopCondition()`: It is checked after the steering function (if defined) at the end of each computational step and can be used to define a stopping criterion for the simulation. Differently to the other callbacks, which do not return any value, the function returns a Boolean value: *true* if the termination criterion is satisfied, *false* in the other case.

Algorithm 1 outlines the OpenCAL implicit simulation process, that applies the default model transition function if not differently specified. In the other case, the `globalTransition()` function is applied. The `init()` function, if defined, is called first and subsequently active cells (if quantization is enabled - cf. below in this Section) and substates are updated. Moreover, the `step` counter and the `halt` variable, that is used to check the simulation termination condition, are set to the initial step and to *false*, respectively. The main simulation loop follows, which is triggered by the `calRun{2D|3D}()` or `calclRun{2D|3D}()` function call, depending on the adopted OpenCAL implementation. At each step, after the application of each elementary process to each cell of the computational domain, active cells (if the quantization optimization is used) and substates are updated. If defined, the `steering()` global function is therefore called and active cells and substates again updated. The `stopCondition()` function is eventually called and the step counter increased. The simulation loop continues while the `halt` variable, whose value is set by the `stopCondition()` function, is *false* or the final step of computation is met.

3.2.3 The quantization optimization

In many grid-based simulations, system's dynamics only affects a small region of the whole computational domain. For instance, this is the case of topologically connected phenomena, like debris or lava flows. In these cases, a naive approach where the overall domain is processed can lead to a considerable waste of computational resources, even in the case stationary cells (i.e. those cells that do not change their state in the next computational step) are only checked and the application of the evolution rules skipped.

Different approaches have been proposed to improve the efficiency of the naive approach. Among them, the hyper-rectangular bounding box (HRBB) optimization, consisting in surrounding the simulated phenomenon by means of a fitting rectangle (or a parallelepiped, in the

Algorithm 1: OpenCAL main implicit simulation process.

```
init()                                          // Call the init() global function
if quantization then
    update(A)                                   // Update the array of active cells
forall q ∈ Q do
    update(q)                                   // Update the substate q
step ← initial_step
halt ← false
while ¬halt ∧ (step ≤ final_step ∨ final_step = CAL_RUN_LOOP) do
    forall e of σ do
        forall (A ≠ ∅ ∧ i ∈ A) ∨ i ∈ R do
            e(i)                                // Apply the elementary process e to the cell i
        if quantization then
            update(A)                           // Update the array of active cells
        forall q ∈ Q do
            update(q)                           // Update the substate q
    steering()                                  // Call the steering() global function
    if quantization then
        update(A)                               // Update the array of active cells
    forall q ∈ Q do
        update(q)                               // Update the substate q
    halt ← stopCondition()                      // Check the stop condition
    step ← step + 1
return
```

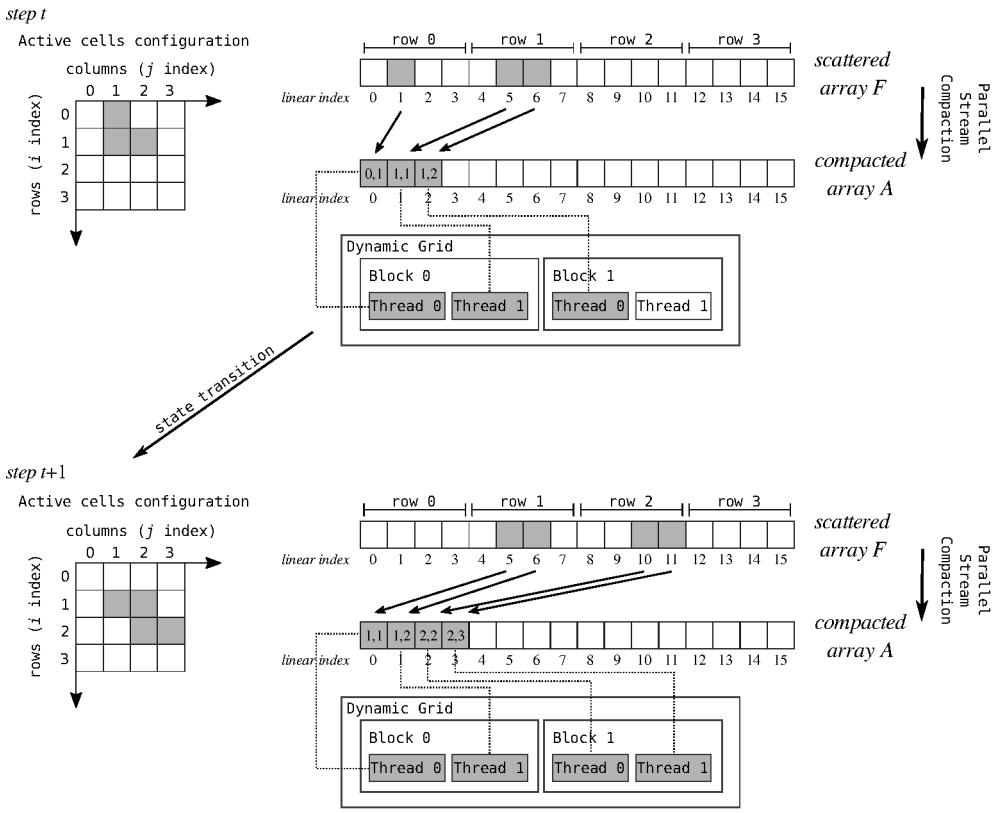


FIGURE 3.3: An example of application of the OpenCAL-CL parallel stream compaction algorithm. Active cells are represented in gray within a two-dimensional 4x4 matrix of flags, implemented as a linearized array, F. The parallel stream compaction algorithm processes F and produces the compacted array A as output, containing the coordinates of the active cells in its first part. A grid of work-items therefore processes data by adopting the one thread/one active cell policy. The process is therefore repeated at the next computational step.

case of a 3D model), by contextually restricting the computation to this specific sub-region, proved to be a simple but effective approach in different cases (see e.g., [52]). However, HRBB demonstrated its limit in the simulation of scattered phenomena, where the hyper-rectangle can easily grow up and include the whole domain, embedding a considerable number of inactive cells.

A more effective approach, which is also able to optimally distribute the computational load in case of parallel execution, consists in maintaining a dynamic set of coordinates only of the active cells during the simulation, by restricting the computation only to this set (see e.g., [61]). The activation state for a cell generally depends on the specific system to be simulated. In many cases, for instance in computational fluid-dynamics, a threshold-based criterion can be adopted. For this reason, this latter approach is commonly known as quantization. Even if more complex to be implemented, in many cases it outperforms the HRBB approach and, for this reason, was considered in OpenCAL.

Its implementation is based on a compacted array, A, containing the computational domain's active cells coordinates. A, which is initially empty, is generally defined at the system initialization stage. For this purpose, and also to maintain A updated, the activation value must be explicitly set (to *true*, which means that the cell is active, or to *false*, on the other hand) only for the cells that change state during the current computational step (a Boolean

working array F is updated in this preliminary step). An efficient stream compaction algorithm (as implemented in [66]) is therefore transparently applied at the end of each computational step to update the set A, based on the activation states stored in F. For illustrative purposes, an example of application of the OpenCAL-CL stream compaction algorithm is described in Figure 3.3. Note that, in this way, domain-sized data is processed only once during the stream compaction phase, while only the subset of cells belonging to A is involved in the remaining actual computation. The quantization optimization clearly introduces an overhead. However, depending on the domain dimension and the affected area (or volume), as well as on the computational intensity of the model, it can produce a considerably speed-up of the overall computational process.

Using the quantization optimization is quite straightforward. Firstly, it must be enabled at model object definition time by means of the `calCADef{2D|3D}()` function. Subsequently, the `calAddActiveCell[X]{2D|3D}()` function can be used to mark the central cell and its neighbors (if the `X` version of the function is considered) to be added to A, while the `calRemoveActiveCell{2D|3D}()` to mark the central cell to be removed. All these functions essentially write a 8-bit long Boolean value to F and, for this reason, there is no risk to obtain a corrupted value, even in the case of parallel execution (i.e. in the case two threads/work-items attempt to store their own value to the same memory word at the same time). Even in the case of OpenCAL-CL, if the same instruction is executed by more than one work-item (even belonging to different work-groups) to the same location in global memory (where F is stored), the access is serialized and at least one access is guaranteed (even if the actual thread performing the operation is undefined - cf. e.g., [47]). Eventually, in case of explicit update scheme, the `calUpdateActiveCells{2D|3D}` function must be explicitly invoked to update A after each add/remove phase is complete.

Note that, since the API allows to modify the neighboring cells activation state, the quantization optimization can lead to race conditions. Nevertheless, to avoid them it is sufficient to keep the add and remove phases disjoint, i.e. performed by different elementary processes. In fact, if the same elementary process could both add and remove cells to/from A, two different (central) cells could update the same (neighboring) cell to different activation states, and the resulting value in F before the stream compaction execution would depend on the application order of the elementary process to the cells.

3.2.4 Conway's Game of Life

As a first illustrative example of the library, we here present the OpenCAL implementation of the Turing complete Conway's Game of Life (simply Life in the following), one of the most simple, yet powerful example of CA [69]. It can be thought as an infinite two-dimensional grid of square cells, each of them being in one of two possible states, *dead* or *alive*. Every cell interacts with the eight adjacent neighbors belonging to the Moore neighborhood. At each time step, one of the following transitions occur: 1. Any alive cell with fewer than two alive neighbors dies, as if by loneliness; 2. Any alive cell with more than three alive neighbors dies, as if by overcrowding; 3. Any alive cell with two or three alive neighbors lives, unchanged, to the next generation; 4. Any dead cell with exactly three alive neighbors comes to life. Formally, Life can be defined as:

$$Life =< R, X, Q, \sigma >$$

where

1. R is the set of points, with integer coordinates, which defines a two-dimensional toroidal cellular space;

2. $X = \{(0,0), (-1,0), (0,-1), (0,1), (1,0), (-1,-1), (1,-1), (1,1), (-1,1)\}$ is the Moore neighborhood, i.e. the set of relative coordinates that, when added to the coordinate vector of the central cell, give the absolute coordinates of the neighboring cells;

3. $Q = \{0,1\}$ is the set of cell states, 0 representing the dead state, 1 the alive;

4. $\sigma : Q^9 \rightarrow Q$ is the deterministic cell transition function. It is composed by one elementary process, which implements the aforementioned transition rules.

```
1  #include <OpenCAL/cal2D.h>     //#include <OpenCAL-OMP/cal2D.h>
2  #include <OpenCAL/cal2DIO.h>   //#include <OpenCAL-OMP/cal2D.h>
3  #include <OpenCAL/cal2DRun.h>  //#include <OpenCAL-OMP/cal2D.h>
4  #include <stdlib.h>
5
6  struct CALModel2D* life;
7  struct CALSubstate2Di* Q;
8  struct CALRun2D* life_simulation;
9
10 void lifeTransitionFunction(struct CALModel2D* life, int i, int j);
11
12 int main() {
13   life = calCADef2D(8, 16, CAL_MOORE_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL, CAL_NO_OPT
         );
14   life_simulation = calRunDef2D(life, 1, 1, CAL_UPDATE_IMPLICIT);
15
16   Q = calAddSubstate2Di(life);
17   calAddElementaryProcess2D(life, lifeTransitionFunction);
18
19   calInitSubstate2Di(life, Q, 0);
20   calInit2Di(life, Q, 0, 2, 1);
21   calInit2Di(life, Q, 1, 0, 1);
22   calInit2Di(life, Q, 1, 2, 1);
23   calInit2Di(life, Q, 2, 1, 1);
24   calInit2Di(life, Q, 2, 2, 1);
25
26   calSaveSubstate2Di(life, Q, "./life_0000.txt");
27   calRun2D(life_simulation);
28   calSaveSubstate2Di(life, Q, "./life_LAST.txt");
29
30   calRunFinalize2D(life_simulation);
31   calFinalize2D(life);
32   return 0;
33 }
```

LISTING 3.1: An OpenCAL implementation of the Conway's Game of Life. To obtain the equivalent OpenCAL-OMP implementation it is sufficient to consider the (currently commented) OpenCAL-OMP header files instead of the OpenCAL ones (cf. lines 1-3).

In the following, two OpenCAL/OpenCAL-OMP and OpenCAL-CL implementations of Life are presented and commented. The program in Listings 3.1, containing the main application, and 3.2, containing the transition function, shows a possible OpenCAL/OpenCAL-OMP implementation of Life. Concerning the main application, header files are included at lines 1-3 that allow to define the required 2D model and substate, besides providing some basic I/O facilities. The model object, `life`, is declared at line 6, while lines 7 and 8 declare the required substate, `Q`, and simulation object, `life_simulation`, respectively. These objects are defined later in the `main` function at lines 13-14. In particular, the model definition function, `calCADef2D()`, takes the domain dimensions (an 8 rows $\times$ 16 columns domain is considered here), the neighborhood pattern (Moore in this case), the boundary topology (a toroidal domain is considered in the example to account for an unlimited domain) and the optimization to be used (the quantization optimization is not adopted in the example). Furthermore, the simulation object definition function, `calRunDef2D()`, requires the address of a model object to be evolved (which is `life` in this example), the initial and final simulation steps (set both to

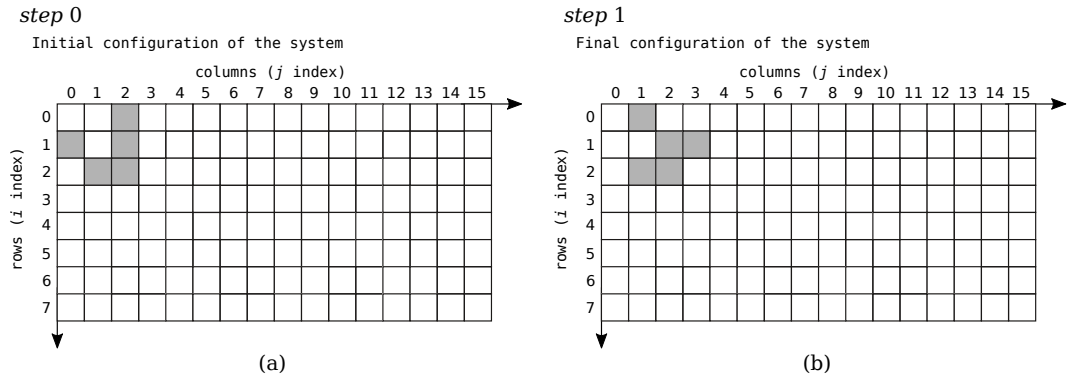


FIGURE 3.4: Graphical representation of one computational step of the Game of Life, showing the (a) initial and (b) final configurations of the system. Alive cells are represented in gray, dead cells in white.

one to perform a single computational step), and eventually the substates update policy (here set to implicit) as parameters. Line 16 allocates memory and registers the integer-based Q substate to the model object by means of the `calAddSubstate2Di()` function, while line 17 registers an elementary process to `life` by means of the `calAddElementaryProcess2D()` function. Here, `lifeTransitionFunction` is a developer-defined callback function implementing the model local rules. At this aim, the `calGet[X]2Di()` and `calSet2Di()` API functions are used for reading and updating purposes at cell level (cf. Listing 3.2).

```
1  void lifeTransitionFunction(struct CALModel2D* life, int i, int j) {
2    int sum = 0, n;
3    for (n=1; n<life->sizeof_X; n++)
4      sum += calGetX2Di(life, Q, i, j, n);
5
6    if ((sum == 3) || (sum == 2 && calGet2Di(life, Q, i, j) == 1))
7      calSet2Di(life, Q, i, j, 1);
8    else
9      calSet2Di(life, Q, i, j, 0);
10 }
```

LISTING 3.2: The OpenCAL/OpenCAL-OMP callback function implementing the elementary process of Game of Life application shown in Listing 3.1.

The `calInitSubstate2Di()` function at line 19 initializes the Q substate to 0 (for both the current and next layers), while lines 20-24 define a so called *glider* pattern (cf. Figure 3.4a) by means of the `calInit2Di()` function. The `calSaveSubstate2Di()` function at line 26 saves the Q substate to file, while the subsequent call to `calRun2D()` enters the simulation process (actually, only one computational step in this example), and returns to the `main` function when the simulation is terminated. The `calSaveSubstate2Di()` is called again at line 28 to save the new (last) system configuration, while the last two API calls release memory previously allocated by OpenCAL. The `return` statement at line 32 ends the program.

Figure 3.4 shows a graphical representation of the initial and final configurations of Life. As expected, the initially defined glider (Figure 3.4a) has evolved into the new correct configuration (Figure 3.4b).

```
1  #include <OpenCAL-CL/calcl2D.h>
2  #include <OpenCAL/cal2DIO.h>
3
4  #define KERNEL_SRC "./kernel"
5  #define KERNEL_LIFE_TRANSITION_FUNCTION "lifeTransitionFunction"
6  #define PLATFORM_NUM 0
7  #define DEVICE_NUM 0
8  #define DEVICE_Q 0
9  struct CALModel2D* life;
10 struct CALSubstate2Di* Q;
11
12 int main() {
13   struct CALCLDeviceManager* calcl_device_manager = calclCreateManager();
14   calclPrintPlatformsAndDevices(calcl_device_manager);
15   CALCLdevice device = calclGetDevice(calcl_device_manager, PLATFORM_NUM, DEVICE_NUM
         );
16   CALCLcontext context = calclCreateContext(&device);
17   CALCLprogram program = calclLoadProgram2D(context, device, KERNEL_SRC, NULL);
18   // <Missing>: Here source code as in lines 12-24 from Listing 1
19
20   struct CALCLModel2D* life_device = calclCADef2D(life, context, program, device);
21
22   CALCLkernel life_transition_function = calclGetKernelFromProgram(&program,
         LIFE_TRANSITION_FUNCTION);
23   calclAddElementaryProcess2D(life_device, &life_transition_function);
24
25   calSaveSubstate2Di(life, Q, "./life_0000.txt");
26   calclRun2D(life_device, 1, 1);
27   calSaveSubstate2Di(life, Q, "./life_LAST.txt");
28
29   calclFinalizeManager(calcl_device_manager);
30   calclFinalize2D(life_device);
31   calFinalize2D(life);
32   return 0;
33 }
```

LISTING 3.3: An OpenCAL-CL implementation of Conway's Game of Life.

According to OpenCL, a possible OpenCAL-CL implementation of Life is subdivided in two different parts: a device- and a host-side application. The host-side application, running on the CPU and controlling the computation on the compliant device (e.g., a GPU), is shown in Listing 3.3. The `calcl2D.h` header file is included at lines 1-2, together with the OpenCAL `cal2DIO.h` header for I/O purposes. The path of the directory containing the transition function elementary processes (implemented as OpenCL kernels) is defined at line 4, while the name of the only kernel required at line 5. Lines 6-7 define the OpenCL identifiers for the platform and device to be used. Note that OpenCAL-CL can query the system for platforms and compliant devices, by allowing the user to select them at run time. However, for the sake of simplicity, in this example the first device belonging to the first platform is set. The substate numerical handle `Q` is also defined at line 8, as it is required to refer to the object from both the host and device application. Lines 13-16 are needed to select the compliant device and to create an OpenCL context. These statements widely simplify the device management and can be considered as a kind of template to be used in each OpenCAL-CL application. Line 17 reads kernels (just one in this example) from file (contained in the directory specified at line 4), compile and groups them into an OpenCL program, to be used later to extract kernels for execution. The host-side object definition follows, together with the substate and its initialization (cf. line 18). Line 20 defines the `life_device` device-side object by means of the `calclCADef2D()` function, also by performing host to device data transfer transparently to the user. The elementary process (which actually is an OpenCL kernel) is therefore extracted from the previously compiled program by means of the `calclGetKernelFromProgram()` function at line 22. It returns an OpenCL kernel, which is subsequently registered to the device-side model by means of the `calclAddElementaryProcess2D()` function at line 23. Lines 25 and

27 are used to save the CA state before and after simulation execution, respectively. The CA simulation is executed by means of the `calclRun2D()` function at line 26. In this example, the only elementary process defined is executed in parallel on the compliant device in a transparently way to the user. Eventually, lines 29-31 perform memory deallocation for the previously defined objects. The return statement at line 32 terminates the program.

```
1  #include <OpenCAL-CL/calcl2D.h>
2  #define DEVICE_Q 0
3
4  __kernel void lifeTransitionFunction(__CALCL_MODEL_2D) {
5    calclThreadCheck2D();
6    int i = calclGlobalRow();
7    int j = calclGlobalColumn();
8    CALint sizeof_X = calclGetNeighborhoodSize();
9
10   int sum = 0, n;
11   for (n=1; n<sizeof_X; n++)
12     sum += calclGetX2Di(MODEL_2D, DEVICE_Q, i, j, n);
13
14   if ((sum==3) || (sum==2 && calclGet2Di(MODEL_2D, DEVICE_Q, i, j)==1))
15     calclSet2Di(MODEL_2D, DEVICE_Q, i, j, 1);
16   else
17     calclSet2Di(MODEL_2D, DEVICE_Q, i, j, 0);
18 }
```

LISTING 3.4: The OpenCAL-CL kernel implementing the elementary process of the Game of Life application shown in Listing 3.3.

The device-side kernel implementing the Life transition function is shown in Listing 3.4. The `calcl2D.h` header is included at line 1, and a numeric handle defined at line 2 to refer the Q substate device-side (this is needed to access the correct buffer in the device global memory - cf. The OpenCAL User Guide on GitHub). The transition rules are implemented as an OpenCL kernel at lines 4-18. In particular, line 5 checks for redundant work-items, while lines 6-7 get the indices corresponding to the integer coordinates of the cell that the kernel is going to process. Similarly, line 8 retrieves the neighborhood size by means of the `calclGetNeighborhoodSize()` function. Eventually, lines 10-17 implement the transition rules by using the `calclGet[X]2Di()` and `calclSet2Di()` functions for reading and updating purposes, respectively.

3.3 The *SciddicaT* XCA Example of Application

The *SciddicaT* computational fluid dynamic XCA model was selected as reference application to test OpenCAL in terms of numerical correctness and efficiency. Despite its simplicity, *SciddicaT* is able to simulate the dynamics of real non-inertial fluid-flows on complex topographic surfaces [15]. Specifically, three different versions of *SciddicaT* were considered, namely $SciddicaT_{naive}$, $SciddicaT_{ac}$, and $SciddicaT_{ac+esl}$, whose formal definitions allow for the adoption of progressively more efficient OpenCAL features. For each of them, OpenCAL-, OpenCAL-OMP- and OpenCAL-CL-based double precision implementations have been developed. OpenCAL-MPI implementations of the $SciddicaT_{naive}$ and $SciddicaT_{ac}$ models have also been considered for preliminary testings. In the following, the different versions of *SciddicaT* are formally defined. However, source code details are omitted, since they would burden the discussion. Please refer to the OpenCAL user guide on GitHub for a full description of each of them.

3.3.1 The $SciddicaT_{naive}$ Example of Application

$SciddicaT_{naive}$ is the first, naive, version of the *SciddicaT* fluid-flow model considered in this work. It is formally defined as:

$$SciddicaT_{naive} =< R, X, Q, P, \sigma >$$

where R is the two-dimensional computational domain, subdivided in square cells of uniform size, while X is the von Neumann neighborhood (a geometrical pattern identifying the set of four cells located to the north, east, west and south directions, adjacent to the central one). Q is the set of cell states. It is subdivided in the following substates:

- Q_z is the set of values representing the topographic altitude (i.e. elevation a.s.l.);
- Q_h is the set of values representing the fluid thickness;
- Q_o^4 are the sets of values representing the outflows from the central cell to the four neighbors.

$P = \{p_\epsilon, p_r\}$ is the set of parameters ruling the model dynamics. In particular, p_ϵ specifies the minimum thickness below which the fluid cannot outflow the cell due to the effect of adherence, while p_r is the relaxation rate parameter, which essentially is an outflow damping factor.
$\sigma : Q^5 \to Q$ is the deterministic cell transition function. It is composed by two elementary processes, listed below in the same order they are applied:

- $\sigma_1 : (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \to Q_o^4$ computes outflows from the central cell to the four neighboring ones by applying the *minimization algorithm of the differences* [75]. As a simplification of the adherence effect, a preliminary control avoids outflows computation where the fluid thickness is smaller than or equal to p_ϵ. If this is not the case, the resulting outflows are given by $q_o(0,m) = f(0,m) \cdot p_r \ (m = 0, \ldots, 3)$, being $f(0,m)$ the outgoing flows towards the 4 adjacent cells, as computed by the minimization algorithm, and $p_r \in]0,1]$ a relaxation factor considered to damp outflows in order to obtain a smoother convergence to the global equilibrium of the system. The Q_o^4 substates are updated accordingly with the values of the computed outflows.
- $\sigma_2 : Q_h \times (Q_o^4)^5 \to Q_h$ determines the value of debris thickness inside the cell by considering mass exchange in the cell neighborhood: $h^{t+1}(0) = h^t(0) + \sum_{m=0}^{3}(q_o(0,m) - q_o(m,0))$. Here, $h^t(0)$ and $h^{t+1}(0)$ are the mass thickness inside the cell at the t and $t+1$ computational steps, respectively, while $q_o(m,0)$ represents the inflow from the $n = (m+1)^{th}$ neighboring cell. The Q_h substate is updated accordingly to account for the mass balance within the cell.

The initial condition of the system is defined by the Digital Elevation Model (DEM) of the surface over which the mass will flow down and by a map of the mass thickness in each cell of the landslide source, also provided as a raster map. This information is used to set up the Q_z and Q_h substates, while the Q_o^4 substates are set to zero everywhere. The evolution of the system is therefore obtained by applying the elementary processes in the order in which they are defined to each cell of the cellular space, and by performing substates updating after the application of each elementary process .

The $SciddicaT_{naive}$ Simulation of the Tessina Landslide

$SciddicaT_{naive}$ was applied to the simulation of the Tessina landslide [15], occurred in Northern Italy in 1992. The event happened in the Tessina valley between altitudes of 1220 m and 625 m a.s.l., with a total longitudinal extension of nearly 3 km and a maximum width of about 500 m.

The topographic surface over which the landslide developed was discretized as a DEM of 410 rows per 294 columns, with square cells of 10 m side, for a total of 102,540 cells.

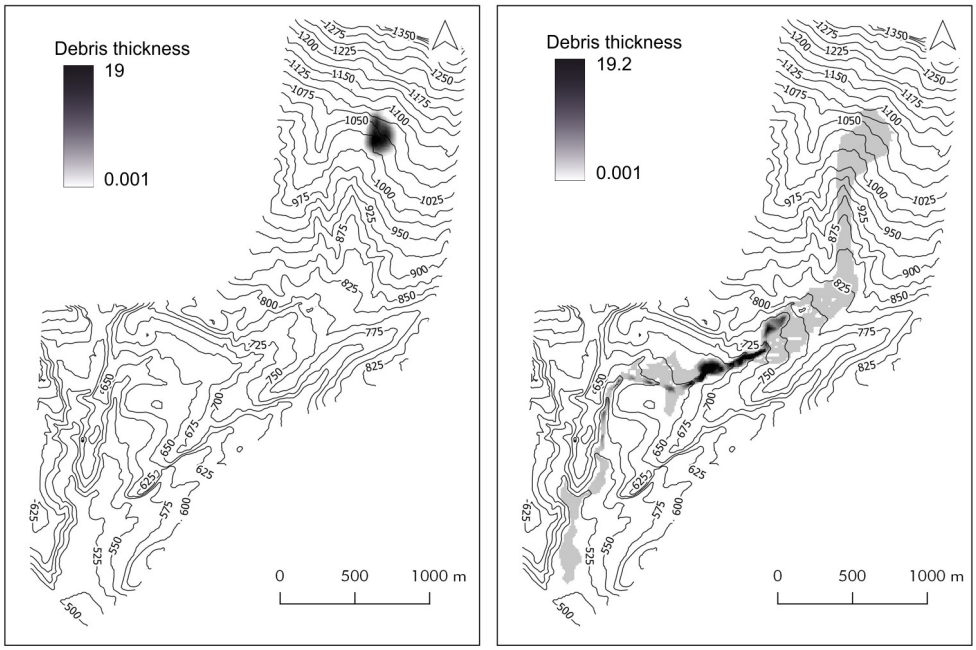


FIGURE 3.5: *SciddicaT* simulation of the 1992 Tessina (Italy) landslide: landslide source on the left; final landslide path on the right.

The landslide source, specifying the location and thickness of the detachment area, was also described by means of a raster map of the same dimensions.

According to [15], the model parameters p_ϵ and p_r were set to the values 0.001 and 0.5, respectively, and 4000 simulation steps were considered. Outcomes (cf. Figure 3.5) achieved by considering the serial and the different parallel implementations of $SciddicaT_{naive}$ perfectly matched, by confirming the numerical correctness of the library. Simulation outcome also matched that obtained by Avolio et al. [15].

3.3.2 The $SciddicaT_{ac}$ Example of Application

The $SciddicaT_{ac}$ computationally improved version of $SciddicaT$ exploits the OpenCAL active cells quantization optimization. This is possible since, in the case of a fluid-flow model - like the one here considered - only cells involved in mass variation are interested in a state change to the next computational step. As a consequence, it is possible to initialize the set of active cells to those cells containing mass. Moreover, if during the computation an outflow is computed from an active cell towards a non-active neighbor, the latter can be added to the set of active cells and then considered for subsequent state change. Similarly, if the mass of an active cell drops below a given threshold, the cell can be eliminated. In the case of $SciddicaT_{ac}$, this happens when debris thickness becomes lower than or equal to p_ϵ. In order to account for these processes, the formal definition of the XCA fluid-flow model is modified, by adding the set of active cells, A. The optimized $SciddicaT_{ac}$ model is now defined as:

$$SciddicaT_{ac} =< R, A, X, Q, P, \sigma >$$

where $A \subseteq R$ is the set of active cells, while the other components are defined as in the formal definition of $SciddicaT_{naive}$. The transition function is now defined as $\sigma : A \times Q^5 \rightarrow Q \times A$, denoting that it is applied only to the cells in A and that it can add/remove active cells. More in detail, the σ_1 elementary process has to be modified, as it can activate new cells. Moreover, a new elementary process, σ_3, has to be added in order to remove cells that cannot produce

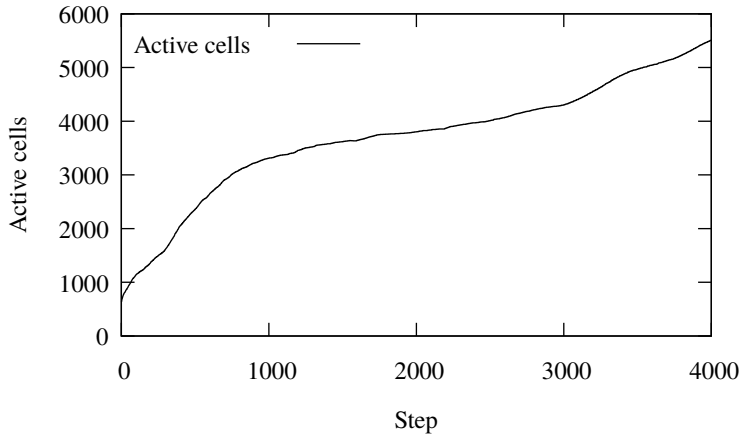


FIGURE 3.6: Number of active cells during the $SciddicaT_{ac}$ simulation of the Tessina landslide shown in Figure 3.5.

outflows during the next computational step due to the fact that their debris thickness is negligible. The new sequence of elementary processes is listed below, in the same order in which they are applied.

- $\sigma_1 : A \times (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \rightarrow Q_o^4 \times A$ determines the outflows from the central cell to the neighbors. Each time an outflow is computed, the neighbor receiving the flow is added to the set of active cells.
- $\sigma_2 : A \times Q_h \times (Q_o^4)^5 \rightarrow Q_h$ determines the value of debris thickness inside the cell by considering mass exchange in the cell neighborhood. This elementary process does not differ with respect to that of the $SciddicaT_{naive}$ model.
- $\sigma_3 : A \times Q_h \times p_\epsilon \rightarrow A$ removes a cell from A if its debris thickness is lower than or equal to the p_ϵ threshold.

The $SciddicaT_{ac}$ Simulation of the Tessina Landslide

The simulation of the Tessina landslide performed by the different implementations of $SciddicaT_{ac}$ resulted numerically correct, perfectly matching those of $SciddicaT_{naive}$ (cf. Figure 3.5). Nevertheless, it is worth to note that the number of (active) cells involved in the computation during the simulation vary between 637, corresponding to the number of cells defining the landslide source, and 5,509. The resulting mean value of cells processed per step is 3,277, corresponding to about the 3.2% of the whole computational domain. Figure 3.6 shows how the number of active cells varies when the $SciddicaT_{ac}$ computational step is increased.

3.3.3 The $SciddicaT_{ac+esl}$ Example of Application

The further computationally improved $SciddicaT_{ac+esl}$ model exploits both the active cell optimization and the explicit simulation loop feature. The model formal definition does not differ from that of $SciddicaT_{ac}$, as well as the simulation outcome of the 1992 Tessina landslide, and therefore are omitted. The only difference with respect to $SciddicaT_{ac}$ consists in the fact that the transition function is overridden in $SciddicaT_{ac+esl}$, allowing to avoid unnecessary time consuming substate updating after the elementary processes execution.

Indeed, the σ_1 elementary process only processes the active cells structure and the outflows substates, so that both the Q_z and Q_h substates do not need to be updated. Moreover, since the σ_2 only changes the debris thickness by evaluating the incoming and outgoing flow mass balance, only the Q_h substate is updated. Eventually, only the active cells structure is updated after the application of the σ_3 elementary process, since this latter simply removes from A cells that have become inactive, by leaving all model substates unchanged.

3.4 Computational Results and Discussion

In order to evaluate OpenCAL from a computational point of view, the different versions of *SciddicaT* presented in the previous Section were considered and the Tessina landslide taken into account as simulation reference case study for a first set of tests (*standard tests*). In particular, a total of ten benchmark simulations were executed for each *SciddicaT* implemented versions, and the speed-up evaluated with respect to the serial implementation of $SciddicaT_{naive}$, by considering the minimum recorded execution times. Furthermore, in order to better assess the impact of local memory usage in OpenCAL-CL, a further implementation based on $SciddicaT_{naive}$ was considered, namely $SciddicaT_{local}$. In this version, a 8×8 work-group size was considered and data, i.e., the substates values of the cells belonging to the neighborhood, transparently transferred from the global to the fast local device memory by using the `calclGlobaltoLocal[X]()` API function (cf. Section 3.2.2). In addition, due to the low transition function computational intensity of *SciddicaT* (i.e., the model is more a memory-bound rather than compute-bound application) and the data-set dimension, which are not adequate to take significant advantage of the adopted GPUs, two additional stress tests were carried out: the transition functions were fictitiously made computationally heavier by reapplying them, at each step, for a total of 200 times (*transition function stress tests*), and the landslide source replicated for a total of 100 times over a wider computational domain by considering a DEM of 13,401,890 cells (*computational domain stress tests*). These latter tests were also considered to evaluate the preliminary OpenCAL-MPI versions of *SciddicaT*, both in terms of correctness and performance.

In all cases, OpenCAL and OpenCAL-OMP benchmarks were executed on a 8-core/16 threads Intel Xeon 2.0GHz E5-2650 CPU based workstation. One thread was considered for testing the different OpenCAL versions of *SciddicaT*, while 2, 4, 8 and 16 threads were employed for benchmark experiments concerning OpenCAL-OMP implementations. Moreover, two devices were adopted for testing the different versions of the OpenCAL-CL implementations of *SciddicaT*, namely a GTX 980 (Maxwell architecture) and a Tesla K40 (Kepler architecture) graphic processor. In particular, the former has 2048 CUDA cores, 4 GB global memory and 112 GB/s theoretical bandwidth communication for double precision data between CPU and GPU, while the latter device has 2880 cores, 12 GB global memory and 144 GB/s double precision high-bandwidth. Eventually, a Gigabit Ethernet interconnected dual-node test system with a GTX 980 GPU per node, which is the configuration used for development purposes, was considered for preliminary evaluation of the OpenCAL-MPI versions of *SciddicaT* .

3.4.1 Standard Tests

The speed-up and execution times of the Tessina landslide simulation related to the OpenCAL and OpenCAL-OMP different versions of *SciddicaT* are shown in Figure 3.7. Here, it is worth to note how the optimizations progressively introduced are effective and, as expected, execution times decrease steadily in all cases. In fact, even in the case of the serial OpenCAL-based implementations, the execution time decreases significantly from about 78 seconds, registered by $SciddicaT_{naive}$, to about 5 seconds, for the fully optimized $SciddicaT_{ac+esl}$

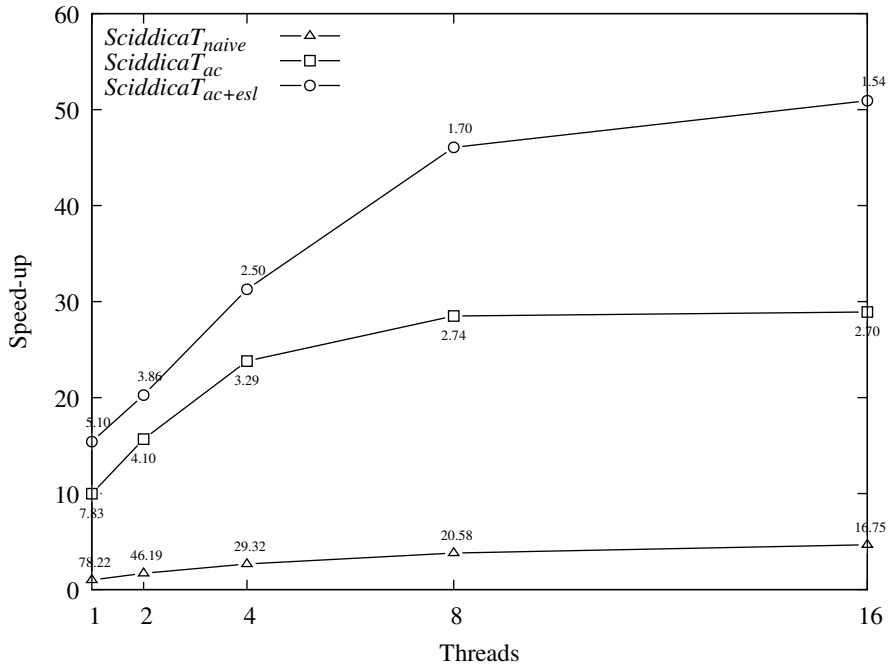


FIGURE 3.7: Speed-up achieved by the different OpenCAL-OMP versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown in correspondence of each speed-up vertex. The considered case study is the Tessina Landslide (cf. Figure 3.5). The adopted CPU was an Intel Xeon 2.0GHz E5-2650 CPU.

version. As expected, $SciddicaT_{ac+esl}$ is the version exhibiting the best performance, running about 51 times faster on 16 threads with respect to the reference simulation.

The benchmark results of the OpenCAL-CL versions of *SciddicaT* are shown in Figure 3.8. Here, as expected, $SciddicaT_{ac}$ resulted the more performing on both devices. Unexpectedly, however, all the experiments executed on the GTX 980 have outclassed the simulations performed on the Tesla K40, notwithstanding the first one being a gaming oriented GPU, while the latter is a HPC dedicated device. This result can be explained taking into account GPU hardware issues: for instance, the K40, though having more cores than the GTX 980, has a lower CUDA core clock-rate (745MHz vs 1126MHz) and lower memory clock-rate (6008 MHz vs 7012 MHz). Also cache issues could justify the results, since the K40 has less L1 and L2 level cache memories than the GTX 980, the latter benefiting from Nvidia's hardware improvements carried out in the more recent Maxwell architectures with respect to the Kepler ones. Moreover, independently from the adopted device, the *SciddicaT* version exploiting the GPU local memory did not resulted faster than the corresponding global memory version. This can be justified by the low transition function computational intensity, whereby work-items do not access data in local memory a sufficient number of times to result in better trade-off and thus better performances.

In addition, in the case of $SciddicaT_{ac}$, it is worth to note that the CPU performs better than the considered GPUs: 2.70 seconds on 16 threads, against 2.98 and 3.96 seconds on the GTX 980 and the Tesla K40, respectively. This can be explained by considering that the mesh generated by the quantization algorithm is too small to exploit the GPU latency thread hiding mechanism at best [97]. In fact, the mean number of cells processed per step is 3,277 (cf. Section 3.3.2), which is of the same order of magnitude of the number of cores of the adopted GPUs (cf. above in this Section). This also leads to a waste of bandwidth.

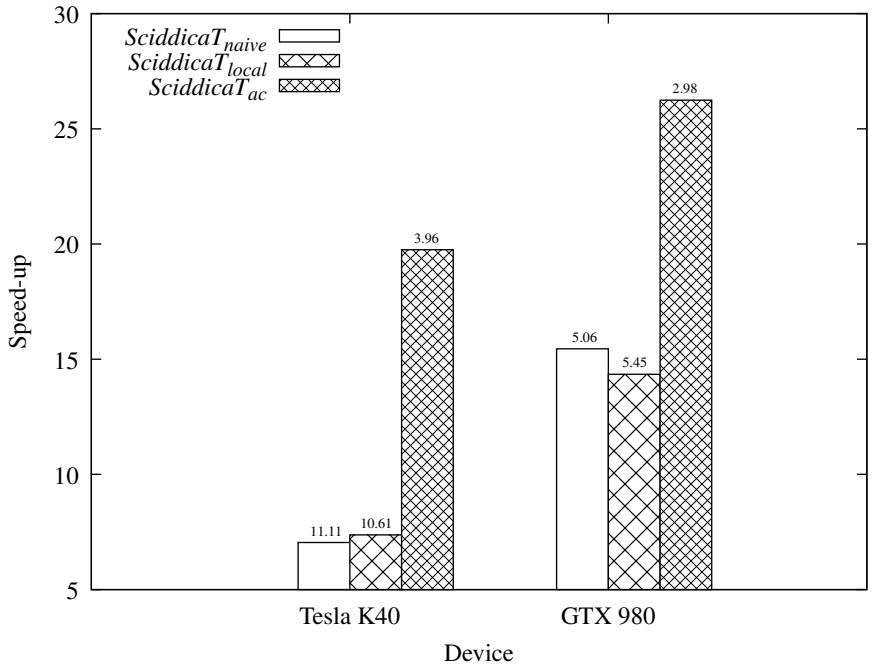


FIGURE 3.8: Speed-up achieved by the different OpenCAL-CL versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown on top of each speed-up bar. The considered case study is the Tessina Landslide (cf. Figure 3.5). The adopted OpenCL compliant devices were a Nvidia Tesla K40 and an Nvidia GTX 980.

In fact, while the other versions were able to adequately exploit the available bandwidth (e.g., the $SciddicaT_{naive}$ version reached about 88 GB/s on the Tesla K40 GPU), the one exploiting the quantization optimization was not able to take advantage of it (achieving 10 GB/s only). Eventually, a further study performed on the most time consuming kernels has shown that the achieved bandwidth is significantly higher for the CPU. Particularly indicative is the value measured for the more time consuming kernel, i.e. the one implementing the stream compaction algorithm. This latter, which takes alone about the 55% of the overall execution time on both the adopted CPU and GPUs versions, exploits the bandwidth the 35% better when the simulation is executed on the CPU, while the other kernels exploit better the bandwidth when the execution is performed on the GPUs, even though in this case they are all characterized by negligible percentage of the overall execution time (about the 3%). In other words, the standard test case here considered is simply too small to make decent use of the considered GPUs and, consequently it is not surprising that the CPU performs better in this specific case.

3.4.2 Transition Function Stress Tests

As anticipated, in order to evaluate performances when considering computationally intensive state transitions, further tests were carried out by fictitiously increasing the complexity of the *SciddicaT* transition function. This was done by reapplying the transition function σ for a total of 200 times during each simulation step, excluding data transfer (e.g., from global to local memory, in the case of $SciddicaT_{local}$).

Results of the benchmarks executed on the CPU are shown in Figure 3.9, both in terms of execution time and speed-up. A more pronounced timing decrease is here observed for each *SciddicaT* version as the number of threads is increased, with a maximum speed-up of about

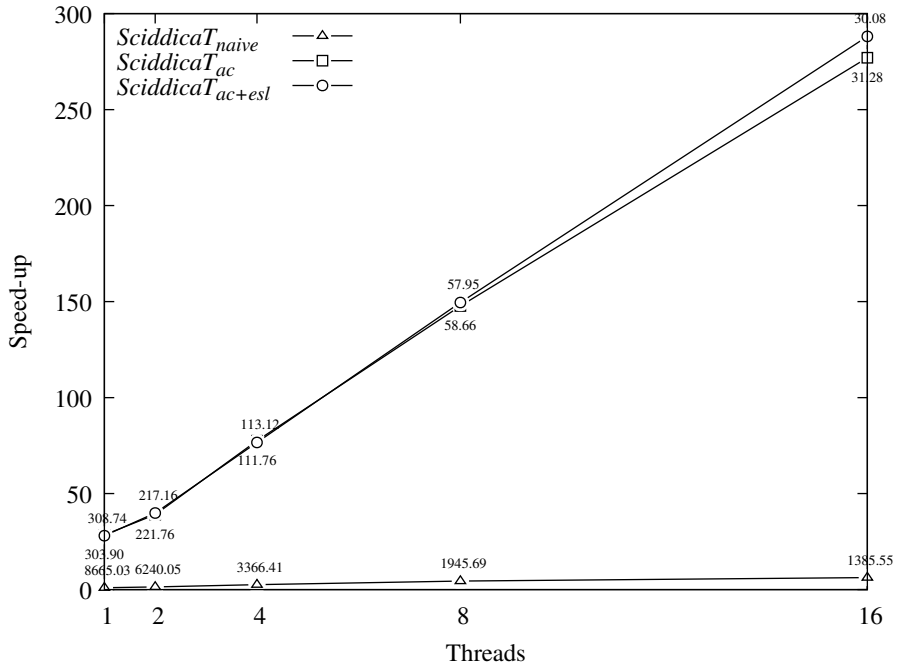


FIGURE 3.9: Speed-up achieved during the *transition function stress test* by the different OpenCAL-OMP versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown in correspondence of each speed-up vertex. The considered case study is the Tessina Landslide (cf. Figure 3.5). The adopted CPU was an Intel Xeon 2.0GHz E5-2650 CPU.

289 for the $SciddicaT_{ac+esl}$ execution on 16 threads. Here, the implementations exploiting the active cells optimization outperform the naive one of two orders of magnitude.

Figure 3.10 shows the benchmark results of the different OpenCAL-CL versions of *SciddicaT* on the considered graphic hardware for the transition function stress test. Here, conversely from the standard tests, the *SciddicaT* version exploiting the GPU local memory resulted significantly faster than the corresponding global memory version on both the considered devices. Specifically, the Tesla K40 reported the best result, highlighting a better local memory system (i.e., better trade-off between local memory access/transfer) with respect to the GTX 980 GPU. Nevertheless, the $SciddicaT_{ac}$ performances were always better than any CPU/GPU version (the GTX 980 performing better), demonstrating even in this case the validity of the active cells optimization. It is worth to note that this time the best GPU performance registered by the *SciddicaT* OpenCAL-CL versions significantly overcame the one registered on the CPU. Specifically, the $SciddicaT_{ac}$ ran about 441 times faster than the serial version of $SciddicaT_{naive}$, against the best 289 speed-up registered on the CPU, pointing out, as expected, the full suitability of GPGPU solutions in the case of sufficiently computationally intense simulation models.

3.4.3 Computational Domain Stress Tests

In order to evaluate performances when larger computational domains are taken into account, further tests were carried out by considering a DEM of 3,593 rows per 3,730 columns, with square cells of 10 m side. Moreover, the landslide source was uniformly replicated 100 times over the extended DEM and a simulation executed for each combination of *SciddicaT* versions and available devices. Figure 3.11 shows the simulation outcomes obtained by considering the larger DEM and the 100 landslide sources.

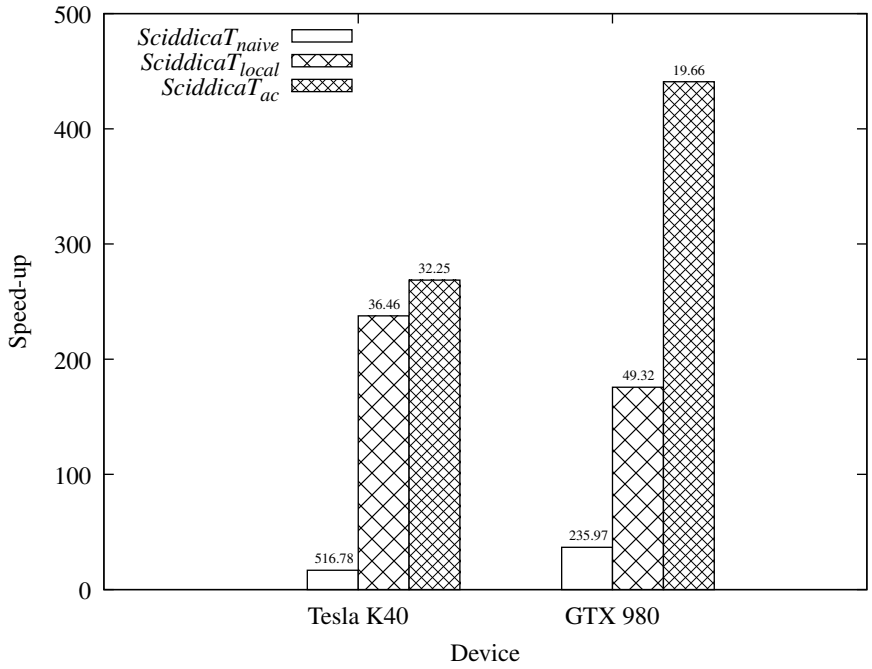


FIGURE 3.10: Speed-up achieved during the *transition function stress test* by the different OpenCAL-CL versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown on top of each speed-up bar. The considered case study is the Tessina Landslide (cf. Figure 3.5). The adopted OpenCL compliant devices were a Nvidia Tesla K40 and an Nvidia GTX 980.

Computational results of the OpenCAL-OMP versions of *SciddicaT* are shown in Figure 3.12. Similarly to the standard tests, a slight timing decrease is observed for all cases as the number of threads is increased. Speed-up values increase accordingly to the adopted optimizations, resulting $Sciddica_{ac+esl}$ the fastest version with a value of about 21.

Moreover, benchmark results of the OpenCAL-CL different versions of *SciddicaT* on the computational domain stress tests are shown in Figure 3.13. Here, as for the standard tests, the *SciddicaT* version exploiting the GPU local memory did not result significantly faster than the corresponding global memory version on both the considered devices, by confirming that local memory has to be accessed an elevated number of times to take an effective advantage compared to the global one (thus resulting more useful for higher computationally complex models). However, the $SciddicaT_{ac}$ versions performances resulted always better than all CPU/GPU versions (the GTX 980 performing better), demonstrating even in this case the validity of the active cells optimization. Even in this case, the best GPU performance was better than the one obtained for the CPU. In particular, the $SciddicaT_{ac}$ ran about 121 times faster on the GTX 980 than the serial version of $SciddicaT_{naive}$, against the best 21 absolute speed-up registered on the CPU pointing out, as expected, the usefulness of GPGPU solutions also in the case of extended computational domains. The result is justified by the same reasons highlighted for the standard test case. In particular, the higher dimension of the computational domain stress test mesh permits the GPUs to always perform better than the CPU in terms of achieved bandwidth for all the considered *SciddicaT* versions (the $SciddicaT_{ac}$ version included, which achieves about 77 GB/s of bandwidth on the Tesla K40, against the 10 GB/s achieved on the smaller mesh), consequently allowing to hide the thread latency in all cases, and thus to exploit better the GPU computational power. Eventually, it is worth to note that, as in the standard tests, the GTX 980 outperformed the Tesla K40, confirming that a

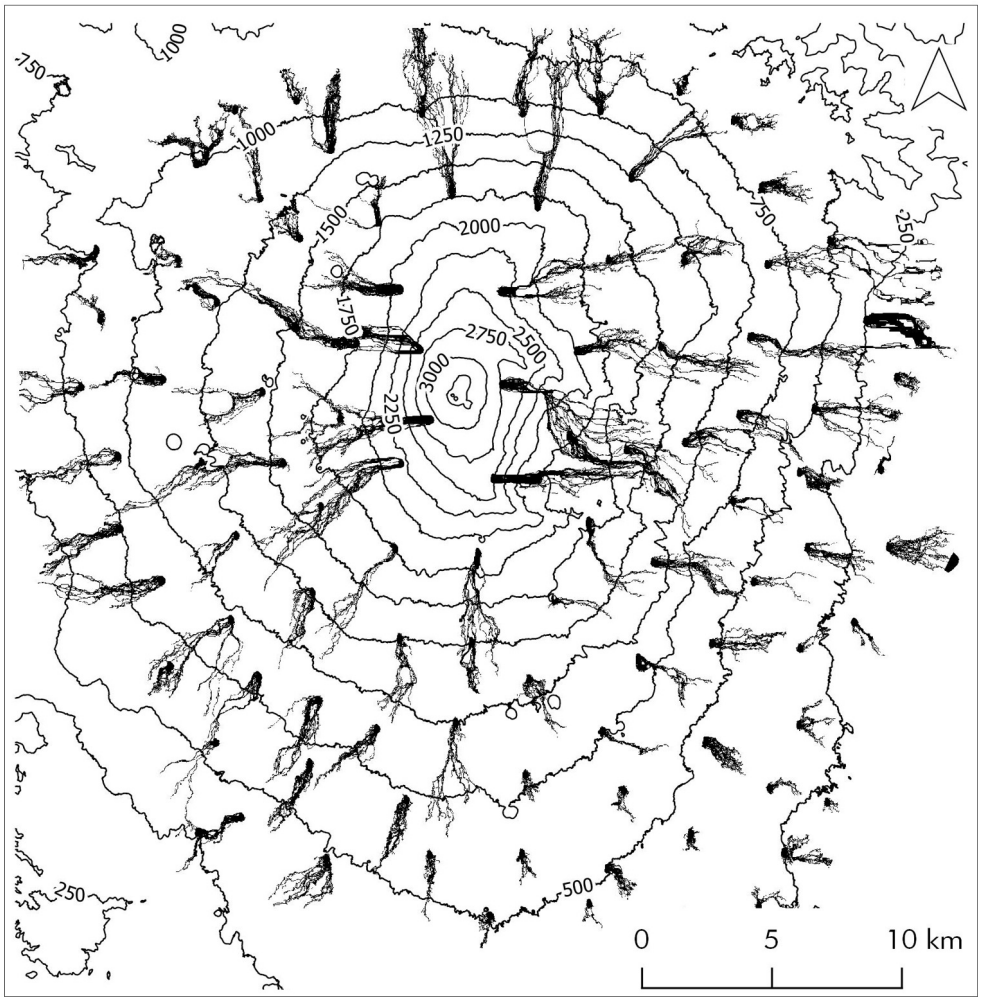


FIGURE 3.11: *SciddicaT* simulation stress test of 100 landslide sources distributed over a DEM of 3593 rows per 3730 columns, with square cells of 10 m side. Landslides paths are represented in black.

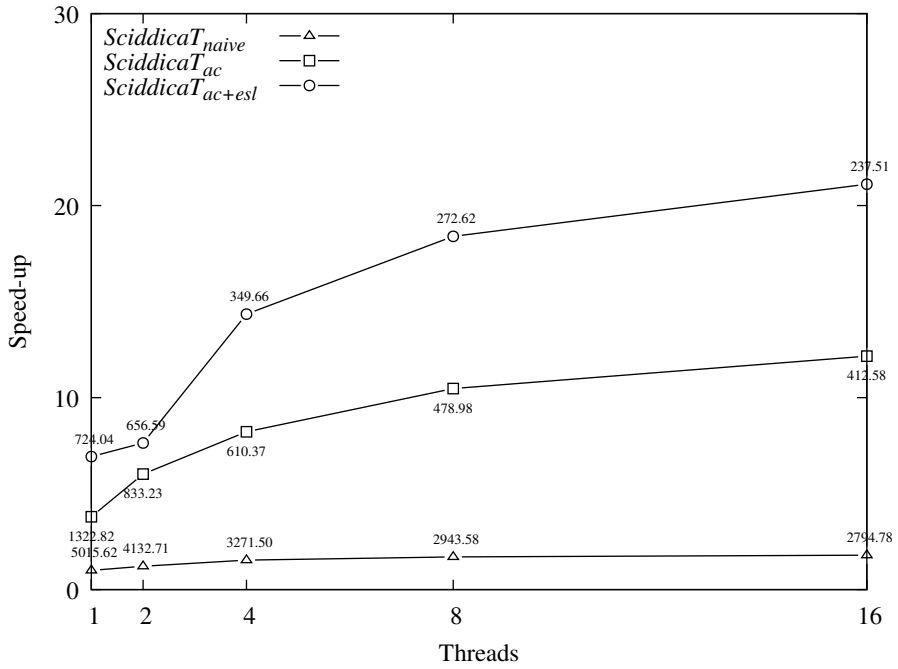


FIGURE 3.12: Speed-up achieved during the *computational domain stress test* by the different OpenCAL-OMP versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown in correspondence of each speed-up vertex. The considered case study is the simulation shown in Figure 3.11). The adopted CPU was an Intel Xeon 2.0GHz E5-2650 CPU.

gaming-oriented device is a preferable solution in case of models with low computational intensity.

Figure 3.13 also shows preliminary benchmark results on two dual-GPU computing systems. The first one consists of a single workstation equipped with two GTX 980 GPUs, while the second of a dual-node cluster with a single GTX 980 per node, interconnected through a Gigabit network. The computational domain was subdivided equally between the available GPUs (specifically, the first 1796 rows were assigned to the first GPU, the remaining 1797 to the second one), and two MPI processes ran for each performed test.

The performances of the dual-GPU workstation were more than satisfying. $SciddicaT_{naive}$ and $SciddicaT_{local}$ scaled super-linearly, being respectively 2.9 and 2.4 times faster than the single GPU execution. The reasons beneath this super-linearity are not currently clear and will be investigated in a future work. Even $SciddicaT_{ac}$ showed a good result, being about 1.53 times faster than the single GPU execution and the fastest in absolute terms.

In contrast, $SciddicaT_{naive}$ and $SciddicaT_{local}$ scaled sub-linearly on the dual-node system, being respectively 1.92 and 1.51 times faster than the single GPU corresponding executions, while $SciddicaT_{ac}$ showed a slowdown, with a speed-up of 0.31. In order to understand the reasons of this behavior, and also to assess the correctness of the OpenCAL-MPI implementation, we profiled the performances of the interconnection network by adapting a simple MPI ping-pong benchmark application in order to emulate the $SciddicaT_{ac}$ MPI communications. In particular, a total of 48,000 179,040 bytes-long buffers were sent, corresponding to twelve 179,040 bytes-long halos (top and bottom halos were sent for each elementary process/global function) sent for each of the 4000 $SciddicaT_{ac}$ simulation steps. The resulting communication time of 95,97 s (and mean latency of $3.9 \cdot 10^{-3}$ s per message) is in good agreement with the one measured during the $SciddicaT_{ac}$ simulation, which is 96,44 s. We can

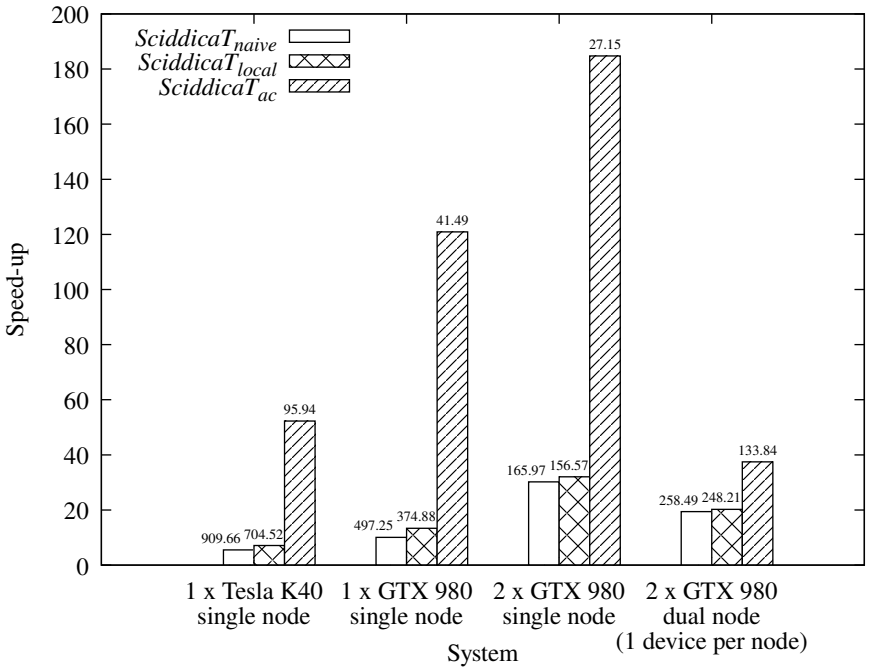


FIGURE 3.13: Speed-up achieved during the *computational domain stress test* by the different OpenCAL-CL versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown on top of each speed-up bar. The considered case study is the simulation shown in Figure 3.11. A single node workstation (alternatively equipped with a Nvidia Tesla K40, a Nvidia GTX 980, and two Nvidia GTX 980), and a dual-node cluster with a Gigabit interconnection network (equipped with an Nvidia GTX 980 per node) were adopted.

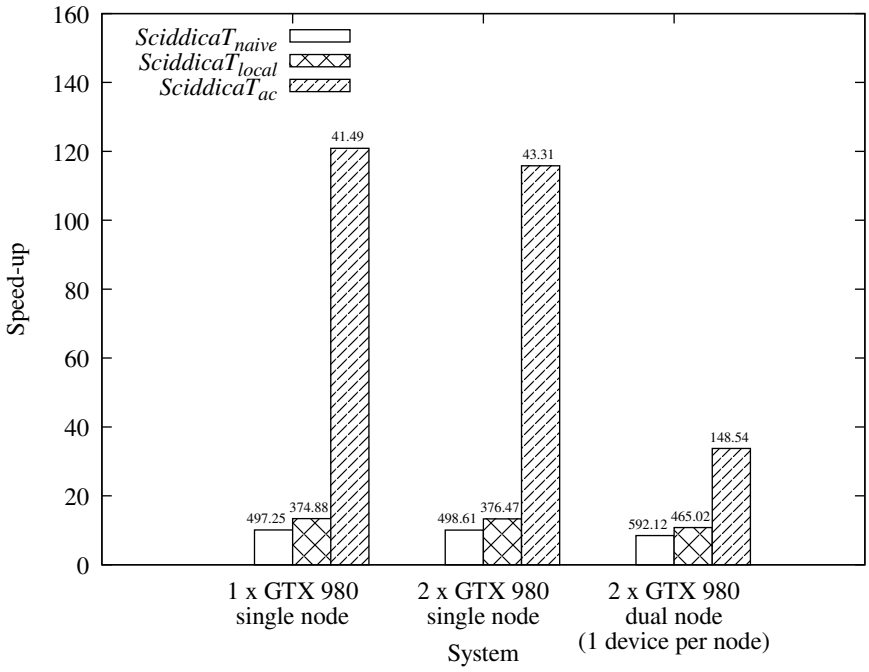


FIGURE 3.14: Speed-up achieved during the weak scaling analysis performed by doubling the data considered on the *computational domain stress test* (cf. Figure 3.11) by the different OpenCAL-MPI versions of the *SciddicaT* fluid-flow model. Single GPU/single domain results are also reported for comparison (cf. Figure 3.13). Elapsed times in seconds are shown on top of each speed-up bar. A single node workstation (alternatively equipped with one or two Nvidia GTX 980), and a dual-node cluster with a Gigabit interconnection network (equipped with an Nvidia GTX 980 per node) were adopted.

therefore assert that, even if preliminary, no communication issues are present in the current implementation of OpenCAL-MPI. In addition, by considering that 32.4 s and 2.55 s were spent in actual computation and in OpenCL read/write buffer enqueuing operations on the PCI Express bus, respectively, the MPI communication resulted to be the 72% of the $SciddicaT_{ac}$ total simulation time. The slowdown observed on the dual-node system is therefore explained by the poor performance of the interconnection network (that does not permit to reduce the communication time below a significant value), and by the high computational speed-up due to the active cells optimization (that considerably reduces the actual computing time), resulting in an high communication/computation time ratio. Network bandwidth was also profiled for completeness, resulting in about 34MB/s. This value does not saturate the Gigabit channel and therefore did not represent a bottleneck in the $SciddicaT_{ac}$ execution. Nevertheless, adopting an interconnection network specifically designed for high performance computing, as well as reducing the amount of data exchanged during the simulation, would result in better scalability.

Finally, we also performed a further weak-scaling analysis in which we doubled the computational domain represented in Figure 3.11 in order to evaluate the scalability on the dual-GPU systems. In particular, the domain was duplicated along the rows (resulting in 7,186 rows by 3,730 columns). This allowed to keep the amount of data exchanged over the network unchanged with respect to the one of the previous tests, since OpenCAL-MPI decomposes the domain along rows. The doubled domain was equally subdivided on the two GPUs in order to obtain an equal computational workload per device. The results of this analysis are shown in Figure 3.14, together with those of the single GPU/single domain simulations, which

are reported for comparison. The different versions of *SciddicaT* evidenced an almost ideal efficiency on the workstation equipped with two GTX 980 GPUs. Conversely, $SciddicaT_{naive}$ and $SciddicaT_{local}$ showed a slight slowdown when the dual-node/dual domain configuration was considered, with an efficiency of 0.84 and 0.8, respectively. As for the previous experiments on the same dual-node test cluster, $SciddicaT_{ac}$ evidenced a more marked slowdown, with efficiency of 0.28. Nevertheless, such results are in agreement with those of the previous strong-scaling analysis and are justified by the same considerations concerning the performances of the interconnection network and the computational efficiency of the different versions of *SciddicaT*.

3.5 Conclusions and Outlooks

In this article we presented the first release of OpenCAL, a new open source computing abstraction layer for Scientific Computing, representing a domain specific language for Extended Cellular Automata and, in general, for structured grid-based computational methods.

Besides the serial implementation, two different parallel versions were developed, namely OpenCAL-OMP and OpenCAL-CL, based on OpenMP and OpenCL, respectively. The first one allows to exploit multi-core CPUs on shared memory computers, while the second a wide range of heterogeneous devices like GPUs, FPGAs and other many-core co-processors. In general, an OpenMP-based parallelization is more straightforward than one based on OpenCL and, when compilers will fully support the 4.0/4.5 OpenMP specifications, it will be possible to execute OpenMP-based applications on both multi-core CPUs and many-core high-performance devices. On the other hand, an OpenCL-based parallelization allows to exploit a wide range of high-performance many-core devices straight away and, allowing for better tuning, permits in principle to exploit computational devices more efficiently. For these reasons, both the OpenMP and OpenCL parallelization of OpenCAL are currently maintained. In addition, preliminary results regarding OpenCAL-MPI, the currently under development MPI/OpenCL parallel version of OpenCAL, were presented.

Each version was designed to be as reliable and fast as possible and, for this purpose, the C language was adopted and optimized data structures and efficient algorithms were considered. Specifically, linearized arrays were adopted to represent both one-dimensional and higher order structures like substates and neighborhoods for exploiting memory coalescence and also to permit a more straightforward OpenCL parallelization. Moreover, the quantization optimization, which allows to restrict the application of the transition function only to non-stationary cells, was implemented in each version, while the explicitation of the global transition function, allowing for selective substates updating, was implemented only in the OpenCAL and OpenCAL-OMP versions (future work will regard the implementation also in OpenCAL-CL).

The *SciddicaT* XCA landslide simulation model was considered to assess numerical correctness and computational efficiency of each OpenCAL version. Note that, such kind of light-weight computational models have a great relevance in practice, for instance when massive simulation approaches are considered [113, 43] or in real-time applications [157, 128]. The OpenCAL and OpenCAL-OMP implementations of three different versions of *SciddicaT* were considered, from a naive one, $SciddicaT_{naive}$, to a version supporting the quantization optimization, $SciddicaT_{ac}$, up to a fully optimized version, $SciddicaT_{ac+esl}$, supporting both the quantization and the explicitation of the global transition function. The first two versions of *SciddicaT* were also implemented in OpenCAL-CL. In addition, a naive version of *SciddicaT* exploiting the local memory, namely $SciddicaT_{local}$, was implemented in OpenCAL-CL to evaluate the role of different GPU memory levels.

The Tessina landslide was considered to evaluate correctness and timings on a 16-thread Intel Xeon CPU based workstation and two Nvidia GPUs. Numerical correctness was confirmed by all the simulation outcomes, which perfectly matched the reference simulation obtained by adopting the OpenCAL implementation of $SciddicaT_{naive}$. Regarding computational performance, the different $SciddicaT$ versions demonstrated to be able to efficiently exploit the computational power of the heterogeneous devices considered in this work, by reducing the execution time of all the performed benchmarks accordingly to the progressively adopted optimizations. However, the best result obtained by $SciddicaT_{ac}$ using 16 threads on the CPU surprisingly was better than the best one obtained by $SciddicaT_{ac}$ on the GPU. This result is justified by the poor ability of the model to exploit both the GPU thread latency hiding mechanism and the bandwidth on the smaller mesh resulting by the application of the quantization algorithm. Nevertheless, subsequent stress tests performed by fictitiously complicating the transition function execution, and a further set of tests where the computational domain was considerably increased with respect to that originally considered, overturned the results and GPUs significantly resulted faster than the CPU, pointing out their usefulness in case of simulations of compute-bound models. GPU local memory provided an actual advantage only in the case of the first set of stress tests, pointing out that data must be accessed an adequate number of times to be effective. Here, in particular, the Tesla K40 resulted more efficient with respect to the GTX 980, even if based on the previous Nvidia hardware architecture, due to its more compute-bound devoted architecture and better management of the local memory. Eventually, the increased computational domain was also considered for a preliminary evaluation of the OpenCAL-MPI implementation of the library on two dual-GPU systems, the first consisting on a single workstation, the second on a dual-node cluster with a Gigabit interconnection network. In the first case, all the versions of $SciddicaT$ achieved very high scalability values, with even super-linear effects for $SciddicaT_{naive}$ and $SciddicaT_{local}$. Such super-linearity will be better investigated in a future work. As regards the dual-node system, $SciddicaT_{naive}$ and $SciddicaT_{local}$ showed a good speed-up, while $SciddicaT_{ac}$ was characterized by a slow-down. The reason of this behavior was investigated, resulting in a high communication/computation time ratio due to both the poor performances of the interconnection network and the high computational efficiency of $SciddicaT_{ac}$ at the single GPU level. A further weak-scaling analysis was also performed, which confirmed the same conclusions of the previous strong-scaling tests.

Though preliminary, results achieved confirm correctness and efficiency of the different OpenCAL versions, by highlighting their reliability for numerical model development of complex systems in the field of Scientific Computing and their execution on parallel heterogeneous devices. Moreover, since the implementations do not significantly differ from an OpenCAL version to another, it is possible to seamlessly obtain two different CPU/GPU parallel versions of the same model with a minimum effort and, therefore, to test them on the available hardware to select the best platform for execution. In fact, as shown for the case of $SciddicaT$, the best choice can deeply depend on both the computational complexity of the transition function and on the extent of the computational domain, and the best solution can not be determined *a priori*.

Nevertheless, a fine tuning of underlying data structures and algorithms will be performed in order to make OpenCAL still more performing and the MPI-based implementation, here only preliminary presented, will be completed to allow OpenCAL to exploit the computational power of distributed memory systems. As regards OpenCAL-CL, a multi-GPU support is currently under development to intelligently scale the overall system performances. Further tests on CPUs/GPUs heterogeneous systems will also be performed. Subsequent releases will also progressively support further computational paradigms, like the Lattice Boltzmann method, the Smoothed Particle Hydrodynamics (SPH), as well as other mesh-free numerical methods, with the aim to achieve a general software abstraction layer for computation.

OpenCAL is released under the Lesser GNU Public License (LGPL) version 3 and, together with a comprehensive installation and user manual accompanied by numerous examples, is currently freely available on GitHub at https://github.com/OpenCALTeam/opencal.

Chapter 4

Conclusion

This dissertation focused on modern operating system mitigations against sophisticated attacks such as side-channel and rowhammer.

In Chapter 1 we first analyzed the attack surface of page fusion, a feature used in modern operating systems to save memory. The analysis showed that this optimization, when developed in a relatively naive way, opens different side-channels that attackers can easily abuse. Moreover, when combined with an hardware glitch like rowhammer, this attack can lead to a complete compromise of a system. After this analysis we showed a principled approach, named VUsion, that mitigates these side-channels, and protects even from attackers that can use the rowhammer attack to flip arbitrary bits. While VUsion design looks bad performance-wise at a first glance, we showed this is not the case in real-life scenarios, and the prototype shows a performance overhead of less than 5% on the CPU SPEC benchmark suite.

In Chapter 2 we showed ZebRAM, a complete, software based and legacy compatible defense against rowhammer attacks. The main idea behind the defense is isolation. In ZebRAM, the operating system can directly access only half of the DRAM. Which DRAM is directly accessible is not arbitrary, but based on how DRAM-addresses are physically mapped in the DRAM chip and on how the rowhammer attack works. This is done in such a way that direct accesses to RAM can induce bitflips only in the not directly-accessible part, that "absorbs" the bitflips.

Wasting half of the RAM is, performance-wise, prohibitive. In ZebRAM we implemented an integrity checking system that allows the operating system to use the not-directly accessible part of the RAM as a compressed in-memory swap device. Another advantage of this approach is that we can re-utilize state of the art page reclaiming algorithms to maintain frequently accessed data in the directly accessible part of the RAM, reducing the performance toll. In Chapter 3 we proposed a general purpose cellular automata framework that can be adopted to develop several security critical applications. The library is released under an opensource license.

Overall, this work highlighted the danger of side-channel and rowhammer attacks. What makes this attacks really dangerous is that they do not exploit any programmer mistake but rather the leaking of information by other means or they maliciously induce hardware glitches. These new attacks pose completely new challenges if compared to typical threats as memory corruption exploitation. This work showed numerous examples of this kind of attacks and at the same time proposed mitigations to stop this threats in software, with negligible performance overhead in most scenarios. We showed a prototype implementation (on top of the Linux kernel) for both the designs proposed in Chapter 1 and Chapter 2. The source code of these prototypes is available online.

Bibliography

[1] Martín Abadi et al. “Control-flow Integrity”. In: CCS. 2005.

[2] Advanced Micro Device. *AMD64 Architecture Programmer’s Manual Volume 2: System Programming*. 2013.

[3] *Advanced Simulation Library*. `http://asl.org.il/`. Accessed: 2017-10-16.

[4] Barbara Aichinger. “DDR Memory Errors caused by Row Hammer”. In: HPEC. 2015.

[5] C.K. Aidun and J.R. Clausen. “Lattice-boltzmann method for complex flows”. In: *Annual Review of Fluid Mechanics* (2010).

[6] Periklis Akritidis et al. “Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors”. In: SEC. 2009.

[7] José Bacelar Almeida et al. “Verifying Constant-Time Implementations”. In: SEC. 2016.

[8] Amit Amritkar, Surya Deb, and Danesh Tafti. “Efficient parallel CFD-DEM simulations using OpenMP”. In: *Journal of Computational Physics* (2014).

[9] Starr Andersen and Vincent Abella. *Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies*. 2004.

[10] B. Arca, T. Ghisu, and G.A. Trunfio. “GPU-accelerated multi-objective optimization of fuel treatments for mitigating wildfire hazard”. In: *Journal of Computational Science* (2015).

[11] Andrea Arcangeli, Izik Eidus, and Chris Wright. “Increasing Memory Density by Using KSM”. In: OLS. 2009.

[12] M.V. Avolio, S. Di Gregorio, and G.A. Trunfio. “A randomized approach to improve the accuracy of wildfire simulations using cellular automata”. In: *Journal of Cellular Automata* (2014).

[13] M.V. Avolio et al. “Pyroclastic flows modelling using cellular automata”. In: *Computers and Geosciences* (2006).

[14] M.V Avolio et al. “SCIDDICA-SS3: a new version of cellular automata model for simulating fast moving landslides”. In: *The Journal of Supercomputing* (2013).

[15] MV Avolio et al. “Simulation of the 1992 Tessina landslide by a cellular automata model and future hazard scenarios”. In: *International Journal of Applied Earth Observation and Geoinformation* (2000).

[16] Zelalem Birhanu Aweke et al. “ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks”. In: ASPLOS. 2016.

[17] Sean Barker et al. “An Empirical Study of Memory Sharing in Virtual Machines”. In: USENIX ATC. 2012.

[18] Antonio Barresi et al. “CAIN: Silently Breaking ASLR in the Cloud”. In: WOOT. 2015.

[19] J. Bedorf, E. Gaburov, and S. Portegies Zwart. “A sparse octree gravitational N-body code that runs entirely on the GPU processor”. In: *Journal of Computational Physics* (2012).

[20] R.D. Beer. “Autopoiesis and cognition in the game of life”. In: *Artificial Life* (2004).

[21] Ravi Bhargava et al. “Accelerating Two-dimensional Page Walks for Virtualized Systems”. In: ASPLOS. 2008.

[22] Sandeep Bhatkar and Daniel C. DuVarney. “Efficient Techniques for Comprehensive Protection from Memory Error Exploits”. In: SEC. 2005.

[23] Sarani Bhattacharya and Debdeep Mukhopadhyay. “Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis”. In: CHESS. 2016.

[24] I. Blecic, A. Cecchini, and G.A. Trunfio. “Cellular automata simulation of urban dynamics through GPGPU”. In: *Journal of Supercomputing* (2013).

[25] I. Blecic, A. Cecchini, and G.A. Trunfio. “How much past to see the future: a computational study in calibrating urban cellular automata”. In: *International Journal of Geographical Information Science* (2015).

[26] Erik Bosman et al. “Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector”. In: S&P. 2016.

[27] A. Brański and E. Pręd ka. “Description of the room acoustic field with meshless methods”. In: *Proceedings - 7th Forum Acusticum 2014, Krakow, Poland*. 2014.

[28] Ferdinand Brasser et al. “CAn’t Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory”. In: SEC. 2017.

[29] Sergey Bratus et al. “Exploit Programming: From Buffer Overflows to “Weird Machines” and Theory of Computation”. In: ;login: 2011.

[30] W.M. Brown et al. “Implementing molecular dynamics on hybrid high performance computers - Short range forces”. In: *Computer Physics Communications* (2011).

[31] Yu Cai et al. “Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques”. In: *Proceedings of the Symposium on High-Performance Computer Architecture*. 2017.

[32] J.L. Cercos-Pita. “AQUAgpusph, a new free 3D SPH solver accelerated with OpenCL”. In: *Computer Physics Communications* (2015).

[33] G. Cervarolo, G. Mendicino, and A. Senatore. “A coupled ecohydrological-three-dimensional unsaturated flow model describing energy, H2O and CO2 fluxes”. In: *Ecohydrology* (2010).

[34] A. Chaigne and A. Askenfelt. “Numerical simulations of piano strings. I. A physical model for a struck string using finite difference methods”. In: *Journal of the Acoustical Society of America* (1994).

[35] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. “An Empirical Study on Memory Sharing of Virtual Machines for Server Consolidation”. In: ISPA. 2011.

[36] Keun-Shik Chang and Chang-Joon Song. “Interactive vortex shedding from a pair of circular cylinders in a transverse arrangement”. In: *International Journal for Numerical Methods in Fluids* (1990).

[37] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. Cambridge, MA, USA: The MIT Press, 2007.

[38] *clBlas*. https://github.com/clMathLibraries/clBLAS. Accessed: 2016-05-13.

[39] E. F. Codd. *Cellular Automata*. Orlando, FL, USA: Academic Press, Inc., 1968.

[40] Lucian Cojocar et al. "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks". In: S&P. 2019.

[41] Matthew Cook. "Universality in Elementary Cellular Automata". In: *Complex Systems* (2004).

[42] Brian F. Cooper et al. "Benchmarking cloud serving systems with YCSB." In: SoCC. 2010.

[43] Gino M Crisci et al. "Predicting the impact of lava flows at Mount Etna, Italy". In: *Journal of Geophysical Research: Solid Earth* (2010).

[44] G.M. Crisci et al. "PYR: A Cellular Automata model for pyroclastic flows and application to the 1991 Mt. Pinatubo eruption". In: *Future Generation Computer Systems* (2005).

[45] H.M. Şahin et al. "Determination of unidirectional heat transfer coefficient during unsteady-state solidification at metal casting-chill interface". In: *Energy Conversion and Management* (2006).

[46] *CUDA C Best Practices Guide*. NVIDIA Corporation, 2017.

[47] *CUDA C Programming Guide*. NVIDIA Corporation, 2017.

[48] *CVE-2016-3272*. Mar. 2016. URL: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3272.

[49] D. D'Ambrosio, S. Di Gregorio, and G. Iovine. "Simulating debris flows through a hexagonal cellular automata model: SCIDDICA S3-hex". In: *Natural Hazards and Earth System Science* (2003).

[50] D. D'Ambrosio et al. "A cellular automata model for soil erosion by water". In: *Physics and Chemistry of the Earth, Part B: Hydrology, Oceans and Atmosphere* (2001).

[51] D. D'Ambrosio et al. "Cellular automata and GPGPU: An application to lava flow modeling". In: *International Journal of Grid and High Performance Computing* (2012).

[52] D. D'Ambrosio et al. "Efficient application of GPGPU for lava flow hazard mapping". In: *Journal of Supercomputing* (2013).

[53] D. D'Ambrosio et al. "Lava invasion susceptibility hazard mapping through cellular automata". In: *Lecture Notes in Computer Science* (2006).

[54] D. D'Ambrosio et al. "Meta-model assisted evolutionary optimization of cellular automata: An application to the SCIARA model". In: *Lecture Notes in Computer Science* (2012).

[55] D. D'Ambrosio et al. "Optimizing cellular automata through a meta-model assisted memetic algorithm". In: *Lecture Notes in Computer Science* (2012).

[56] Donato D'Ambrosio and William Spataro. "Parallel evolutionary modelling of geological processes". In: *Parallel Computing* (2007).

[57] Donato D'Ambrosio et al. "The Open Computing Abstraction Layer for Parallel Complex Systems Modeling on Many-Core Systems". In: *J. Parallel Distrib. Comput.* (2018).

[58] Sourav Das and Dipanwita RoyChowdhury. “CAR30: A new scalable stream cipher with rule 30”. In: *Cryptography and Communications* (2013).

[59] Giuseppe Dattilo and Giandomenico Spezzano. “Simulation of a cellular landslide model with CAMELOT on high performance computers”. In: *Parallel Computing* (2003).

[60] X. Deng et al. “Further studies on Geometric Conservation Law and applications to high-order finite difference schemes with stationary grids”. In: *Journal of Computational Physics* (2013).

[61] S. Di Gregorio et al. “Accelerating wildfire susceptibility mapping through GPGPU”. In: *Journal of Parallel and Distributed Computing* (2013).

[62] P. Du et al. “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming”. In: *Parallel Computing* (2012).

[63] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. “Jump over ASLR: Attacking Branch Predictors to Bypass ASLR”. In: MICRO-49. 2016.

[64] Dmitry Evtyushkin et al. “BranchScope: A New Side-Channel Attack on Directional Branch Predictor”. In: ASPLOS. 2018.

[65] A. Farrokhabadi et al. “Theoretical modeling of the Casimir force-induced instability in freestanding nanowires with circular cross-section”. In: *Physica E: Low-Dimensional Systems and Nanostructures* (2014).

[66] G. Filippone et al. “CUDA dynamic active thread list strategy to accelerate debris flow simulations”. In: PDP. 2015.

[67] Pietro Frigo et al. “Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU”. In: S&P. 2018.

[68] U. Frish, B. Hasslacher, and Y. Pomeau. “Lattice gas automata for the Navier-Stokes Equation”. In: *Physical Review Letters* (1986).

[69] Martin Gardner. “Mathematical Games: The fantastic combinations of John Conway’s new solitaire game "life"”. In: *Scientific American* (1970).

[70] M.B. Giles et al. “Designing OP2 for GPU architectures”. In: *Journal of Parallel and Distributed Computing* (2013).

[71] Gene H Golub and James M Ortega. *Scientific computing: an introduction with parallel computing*. London, UK: Academic Press, 2014.

[72] Google. *Android Low RAM Configuration*. 2017. URL: https://goo.gl/Rz4B6I.

[73] Abel Gordon et al. “ELI: Bare-metal Performance for I/O Virtualization”. In: ASPLOS. 2012.

[74] Ben Gras et al. “ASLR on the Line: Practical Cache Attacks on the MMU”. In: NDSS. 2017.

[75] S. Di Gregorio and R. Serra. “An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata”. In: *Future Generation Computer Systems* (1999).

[76] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: DIMVA. 2016.

[77] Daniel Gruss et al. “Another Flip in the Wall of Rowhammer Defenses”. In: *arXiv preprint arXiv:1710.00551* (2017).

[78] Daniel Gruss et al. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: CCS. 2016.

[79] Fan Guo et al. “SmartMD: A High Performance Deduplication Engine with Mixed Pages”. In: ATC. 2017.

[80] Richard W Hamming. “Error detecting and error correcting codes”. In: *Bell Labs Technical Journal* (1950).

[81] John L. Henning. “SPEC CPU2006 memory footprint”. In: ACM SIGARCH Computer Architecture’07.

[82] F. Higuera and J. Jimenez. “Boltzmann approach to lattice gas simulations”. In: *Europhysics Letters* (1989).

[83] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. “Post-copy Live Migration of Virtual Machines”. In: OSR. 2009.

[84] J. Hu et al. “Electron behavior in hydrogen atom under electric fields”. In: ICPADM. 2015.

[85] *Idle Page Tracking*. 2015. URL: https://www.kernel.org/doc/Documentation/vm/idle_page_tracking.txt.

[86] Intel. *Intel Clear Containers: Building a Virtualization Continuum*. White paper. 2017.

[87] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. 2016.

[88] Gorka Irazoqui et al. “Know Thy Neighbor: Crypto Library Detection in Cloud”. In: PETS. 2015.

[89] S.P. Jammy et al. “Block-structured compressible Navier-Stokes solution using the OPS high-level abstraction”. In: *International Journal of Computational Fluid Dynamics* (2016).

[90] Yeongjin Jang et al. “SGX-Bomb: Locking Down the Processor via Rowhammer Attack”. In: SysTEX. 2017.

[91] JEDEC Solid State Technology Association. *DDR4 SDRAM Specification*. JESD79-4A. 2013.

[92] JEDEC Solid State Technology Association. “DDR4 SDRAM Specification”. In: *JESD79-4B* (2017).

[93] JEDEC Solid State Technology Association. “Low Power Double Data 4 (LPDDR4)”. In: *JESD209-4A* (2015).

[94] Samira Khan, Donghyuk Lee, and Onur Mutlu. “PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM”. In: DSN. 2016.

[95] D. Kim, P. J. Nair, and M. K. Qureshi. “Architectural Support for Mitigating Row Hammering in DRAM Memories”. In: *IEEE Computer Architecture Letters* (2015).

[96] Yoongu Kim et al. “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors”. In: ISCA. 2014.

[97] DB Kirk and WmW Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Burlington, Ma, USA: Morgan Kaufmann Publishers Inc, 2010.

[98] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: S&P. 2019.

[99] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *CRYPTO*. 1996.

[100] David Kohlbrenner and Hovav Shacham. “Trusted Browsers for Uncertain Times”. In: SEC. 2016.

[101] Radhesh Krishnan Konoth et al. “ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks”. In: OSDI. 2018.

[102] Taddeus Kroes et al. “Delta pointers: buffer overflow checks without the checks”. In: EuroSys. 2018.

[103] Anil Kurmus et al. “From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks”. In: WOOT. 2017.

[104] Youngjin Kwon et al. “Coordinated and Efficient Huge Page Management with Ingens”. In: OSDI. 2016.

[105] C.G. Langton. “Computation at the edge of caos: phase transition and emergent computation”. In: *Physica D* (1990).

[106] C.G. Langton. “Studying Artificial Life with Cellular Automata”. In: *Physica D* (1986).

[107] Mark Lanteigne. In: *How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware* (2016).

[108] Elias Levy. “Smashing The Stack For Fun And Profit”. In: *phrack* (1996).

[109] *Linux kernel: Transparent Hugepage Support*. https://www.kernel.org/doc/Documentation/vm/transhuge.txt Retrieved 27.11.207.

[110] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: S&P. 2018.

[111] Fangfei Liu et al. “Last-Level Cache Side-Channel Attacks are Practical”. In: S&P. 2015.

[112] Robert Lubaś, Jarosław Wąs, and Jakub Porzycki. “Cellular Automata as the basis of effective and realistic agent-based models of crowd behavior”. In: *The Journal of Supercomputing* (2016).

[113] F. Luca’ et al. “Integrating geomorphology, statistic and numerical simulations for landslide invasion hazard scenarios mapping: An example in the Sorrento Peninsula (Italy)”. In: *Computers and Geosciences* (2014).

[114] *LZO*. http://www.oberhumer.com/opensource/lzo/, Retrieved 09.09.2018.

[115] M. Macri et al. “Efficient Lava Flows Simulations with OpenCL: A Preliminary Application for Civil Defence Purposes”. In: 3PGCIC. 2015.

[116] Swapan Maiti, Shamit Ghosh, and Dipanwita Roy Chowdhury. “On the Security of Designing a Cellular Automata Based Stream Cipher”. In: *Information Security and Privacy*. Cham: Springer International Publishing.

[117] J. Malcolm et al. “ArrayFire: A GPU acceleration platform”. In: SPIE. 2012.

[118] Robert Martin, John Demme, and Simha Sethumadhavan. “TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks”. In: ISCA. 2012.

[119] Sandip Mazumder. “Chapter 1 - Introduction to Numerical Methods for Solving Differential Equations”. In: *Numerical Methods for Partial Differential Equations*. Cambridge, MA, USA: Academic Press, 2016.

[120] G.R. McNamara and G. Zanetti. “Use of the Boltzmann equation to simulate lattice-gas automata”. In: *Physical Review Letters* (1988).

[121] *memtier benchmark: A High-Throughput Benchmarking Tool for Redis and Memcached*. 2017. URL: https://github.com/RedisLabs/memtier_benchmark.

[122] G. Mendicino, J. Pedace, and A. Senatore. "Stability of an overland flow scheme in the framework of a fully coupled eco-hydrological model based on the Macroscopic Cellular Automata approach". In: *Communications in Nonlinear Science and Numerical Simulation* (2015).

[123] Giuseppe Mendicino et al. "Three-dimensional unsaturated flow modeling using cellular automata". In: *Water Resources Research* (2006).

[124] Ingo Molnar. *LKML: [announce] NX (No eXecute) support for x86 in Linux Kernel*. 2004. URL: https://lkml.org/lkml/2004/6/2/228.

[125] John von Neumann. *Theory of Self-Reproducing Automata*. Ed. by Arthur W. Burks. Champaign, IL, USA: University of Illinois Press, 1966.

[126] Lily Hay Newman. "The Hidden Toll of Fixing Meltdown and Spectre". In: *WIRED* (2018).

[127] S. Ninagawa. "Dynamics of universal computation and 1/f noise in elementary cellular automata". In: *Chaos, Solitons and Fractals* (2015).

[128] V.G. Ntinas et al. "Parallel Fuzzy Cellular Automata for Data-Driven Simulation of Wildfire Spreading". In: *Journal of Computational Science* (2016).

[129] M. Oliverio et al. "OpenMP parallelization of the SCIARA Cellular Automata lava flow model: Performance analysis on shared-memory computers". In: ICCS. 2011.

[130] Marco Oliverio et al. "Secure Page Fusion with VUsion". In: SOSP. 2017.

[131] *OpenBSD cvs archive: disabling SMT*. 2018. URL: https://www.mail-archive.com/source-changes@openbsd.org/msg99141.html.

[132] Yossef Oren et al. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications". In: CCS. 2015.

[133] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: CT-RSA. 2006.

[134] J.D. Owens et al. "A survey of general-purpose computation on graphics hardware". In: *Computer Graphics Forum* (2007).

[135] R. Owens and Weichao Wang. "Non-Interactive OS Fingerprinting Through Memory De-Duplication Technique in Virtual Machines". In: IPCCC. 2011.

[136] PaX Team. "Address Space Layout Randomization". In: Phrack. 2003.

[137] Mathias Payer. "HexPADS: A Platform to Detect "Stealth" Attacks". In: ESSoS. 2016.

[138] Peter Pessl et al. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: SEC. 2016.

[139] Antoniu Pop and Albert Cohen. "OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs". In: *ACM Transactions on Architecture and Code Optimization* (2013).

[140] Rui Qiao and Mark Seaborn. "A New Approach for Rowhammer Attacks". In: HOST. 2016.

[141] P. Rana and R. Bhargava. "Flow and heat transfer of a nanofluid over a nonlinearly stretching sheet: A numerical study". In: *Communications in Nonlinear Science and Numerical Simulation* (2012).

[142] G. Ravazzani, D. Rametta, and M. Mancini. “Macroscopic cellular automata for groundwater modelling: A first approach”. In: *Environmental Modelling and Software* (2011).

[143] Kaveh Razavi et al. “Flip Feng Shui: Hammering a Needle in the Software Stack”. In: SEC. 2016.

[144] I.Z. Reguly et al. “Acceleration of a Full-Scale Industrial CFD Application with OP2”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.5 (2016).

[145] I.Z. Reguly et al. “The OPS domain specific abstraction for multi-block structured grid computations”. In: WOLFHPC Held in Conjunction with SC. 2014.

[146] Charles Schmidt and Tom Darby. *The What, Why, and How of the 1988 Internet Worm*. 1988. URL: https://ethics.csc.ncsu.edu/abuse/wvt/worm/darby/worm.html.

[147] Michael Schwarz, Daniel Gruss, and Moritz Lipp. “Another Flip in the Row”. In: BHUS. https://gruss.cc/files/us-18-Gruss-Another-Flip-In-The-Row.pdf Retrieved 09.09.2018. 2018.

[148] Mark Seaborn and Thomas Dullien. “Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges”. In: BHUS. 2015.

[149] Giuseppe Spingola et al. “Modeling Complex Natural Phenomena with the libAuToti Cellular Automata Library: An example of Application to Lava Flows Simulation.” In: PDPTA. 2008.

[150] J.E. Stone, D. Gohara, and G. Shi. “OpenCL: A parallel programming standard for heterogeneous computing systems”. In: *Computing in Science and Engineering* (2010).

[151] *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. 2016. URL: https://www.cs.virginia.edu/stream/.

[152] B.-Y. Su and K. Keutzer. “clSpMV: A cross-platform OpenCL SpMV framework on GPUs”. In: ICS. 2012.

[153] Kuniyasu Suzaki et al. “Memory Deduplication As a Threat to the Guest OS”. In: EUROSEC. 2011.

[154] Andrei Tatar et al. “Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer”. In: RAID. 2018.

[155] Andrei Tatar et al. “Throwhammer: Rowhammer Attacks over the Network and Defenses”. In: ATC. 2018.

[156] Patrick Tjin. “android-7.1.0_r7 (Disable ION_HEAP_TYPE_SYSTEM _CONTIG)”. In: *https://android.googlesource.com/device/google/marlin-kernel/+/android-7.1.0_r7* (2016).

[157] A. Tsiftsis, I.G. Georgoudas, and G.Ch Sirakoulis. “Real data evaluation of a crowd supervising system for stadium evacuation and its hardware implementation”. In: *IEEE Systems Journal* (2016).

[158] V. van der Veen et al. *Drammer: Deterministic Rowhammer Attacks on Mobile Platforms*. http://vvdveen.com/publications/drammer.slides.pdf, Retrieved 09.09.2018.

[159] V. van der Veen et al. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: CCS. 2016.

[160] Victor van der Veen et al. “GuardION: Practical Mitigation of DMA-based Rowhammer Attacks on ARM”. In: DIMVA. 2018.

[161] VMWare. *Disallowing inter-Virtual Machine Transparent Page Sharing*. 2015. URL: https://goo.gl/uH0zNP.

[162] Mark Wagner. "KVM Performance Improvements and Optimizations". In: *KVM Forum* (2011).

[163] Jarosław Wąs and Robert Lubaś. "Towards realistic and effective Agent-based models of crowd dynamics". In: *Neurocomputing* (2014).

[164] Jarosław Wąs, Hubert Mróz, and Paweł Topa. "GPGPU computing for microscopic simulations of crowd dynamics". In: *Computing and Informatics* (2015).

[165] S. Wolfram. *A new kind of Science*. Champaign: Wolfram Media Inc., 2002.

[166] S. Wolfram. "Universality and complexity in cellular automata". In: *Physica D* (1984).

[167] Stephen Wolfram. "Cryptography with Cellular Automata". In: CRYPTO. 1985.

[168] *WRK - a HTTP Benchmarking Tool*. 2017. URL: https://github.com/wg/wrk.

[169] *WRK2 - a HTTP Benchmarking Tool*. https://github.com/giltene/wrk2, Retrieved 09.09.2018.

[170] Jidong Xiao et al. "A Covert Channel Construction in a Virtualized Environment". In: CCS. 2012.

[171] Yuan Xiao et al. "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation". In: SEC. 2016.

[172] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack". In: SEC. 2014.

[173] Peter Zatko. *How to write Buffer Overflows*.

[174] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. "A Software Approach to Defeating Side Channels in Last-Level Caches". In: CCS. 2016.