CAMPUS
DE EXCELENCIA
INTERNACIONAL

UNIVERSIDAD POLITÉCNICA DE MADRID

E.T.S.I. INFORMÁTICOS

Master in Software and Systems

MASTER THESIS

Microarchitecture Simulation for Security

Author: Paloma Pedregal Helft

Supervisor: Clara Benac

Co-Supervisor: Boris Köpf

MADRID, JULY 2018

Abstract

Finding *minimal eviction sets*, sets of memory addresses of minimum size that can evict content from the cache, is essential in many cache side-channel attacks.

There is a lack of knowledge about numerous elements of the microarchitecture involved in those side-channel attacks, such as replacement policies, slicing and translation of virtual to physical addresses.

In order to better understand the security performance of caches and to bridge the gap between the results of tests over real hardware, and the theoretical behaviour of caches, we create a model of the microarchitectural aspects of computers relevant to finding minimal eviction sets. The model allows simulating tests on different caches, thanks to the parametrization of the features of the microarchitecture. The parameters include deterministic and probabilistic replacement policies.

We model the algorithms used to evaluate eviction sets and three reduction algorithms, which reduce eviction sets to minimal eviction sets. We use the model to simulate two of the reduction algorithms in caches with different characteristics, and we evaluate the robustness of those algorithms.

Resumen

Encontrar *eviction sets* mínimos, es decir, sets de direcciones de memoria de tamaño mínimo que puedan desalojar contenido de una caché, es esencial en numerosos ataques de canal lateral (side-channel attacks) a cachés.

Existe una desinformación general sobre numerosos elementos de la microarquitectura de los ordenadores relacionados con este tipo de ataque, como las políticas de reemplazamiento, el *slicing* o la traducción de direcciones de memoria virtuales a físicas.

Para alcanzar una mayor comprensión del comportamiento de las cachés en cuestión de seguridad, y para salvar la distancia entre los resultados de tests reales sobre hardware, y el comportamiento teórico de las cachés, creamos un modelo en Haskell de los aspectos de la microarquiectura relevantes para encontrar *eviction sets* mínimos. El modelo permite simular experimentos en cachés con diferentes características, gracias a la parametrización de los diferentes rasgos de la microarquitectura. Los parámetros incluyen políticas de reemplazamiento deterministas y probabilisticas.

Modelamos el algoritmo usados para evaluar *eviction sets* y tres *algoritmos de reducción*, usados para reducir un *eviction set* a un tamaño mínimo. Utilizamos el modelo para simular dos de los algoritmos de reducción en cachés con diferentes características, y evaluamos la robustez de los algoritmos.

INDEX

Chapter 1

Introduction

In this section we introduce the concepts necessary along the report: caches, memory, side-channel attacks, eviction and reduction. We conclude the section with motivation and objectives for the thesis.

Caches

[1]Caches are small and fast memories that sit between the slower memory of a computer and the faster processor, bridging the latency gap between them.

In order to store data in a computer's storage, it is logically partitioned into a set of *blocks*. If a block has to be accessed, it first has to be determined whether that block is already in the cache.

If it is, there is a *cache hit*. In this case it can be accessed without further delay. Otherwise, there is a *cache miss*, and the block has to be fetched from memory, which is expensive timewise.

Caches are partitioned in equally sized *cache sets*.

When there is a cache miss and a new block is fetched, this block has to evict and replace a block already in the cache. In order to choose which block to replace, there is a defined *cache replacement policy*.

Nowadays, replacement policies are increasingly complex, frequently undocumented, and sometimes variable depending on the cache set [2, 3].

Memory

The region of the cache to which a memory block is mapped is determined by a *physical address*: a memory address has a physical address that is known by the operating system, but not by any program on the computer. Usually programs only have access to the *virtual address* space. The virtual address space is a construction that allows an

application to have a linear and infinite address space, isolated from other processes, even if the physical addresses are fragmented, or divided between main memory and storage.

There is a translation from virtual to physical addresses, performed by the Memory Management Unit of the machine.

Side-channel attacks

Side channel attacks recover secret inputs to programs from non-functional characteristics of computation such as time, power or memory consumption. [4], i.e. an attack based on information obtained from the implementation of a computer system (side channel information), rather than weakness in the implemented algorithm itself.

Typical goals of side-channel attacks are the recovery of cryptographic keys and private information about users.

General kinds of side channel attacks include:

- **Cache attack** [5] Based on monitoring and timing cache accesses made by the victim in a shared cache system.
- **Timing attack** [6] Measures how much time a certain cryptographic algorithm takes to finish.
- **Power monitoring attack** [7] Measures the variations in power consumption of hardware during computations
- **Electromagnetic attack** [8] Uses measurements of leaks of magnetic radiation, similarly to acoustic cryptanalysis.
- **Branch prediction attack** [9] Based on the study of processors' branch prediction units.
- **Acoustic cryptanalysis** [10] Studies the sound generated by a computation.
- **Data remanence** [11] Performs readings of sensitive data that is supposed to be deleted.

Side-channel cache attacks

There are a series of effective attacks that exploit shared CPU caches [5, 12].

By knowing wich cache sets have been accessed, the adversary could learn the secret of the victim.

Traditionally, a set of addresses used to replace content in the cache is called an *eviction set*.

Accessing a very large set of addresses can be enough for evicting any content out of the cache, however, it is impractical to use such big sets. The ideal would be to have a set with *only* congruent addresses. The reduction of an eviction set to an eviction set of the minimal size, is performed with *reduction algorithms*.

Motivation and objectives

Cache side-channels have proven to be a real and effective way to attack a system. In order to prevent these attacks it is necessary to understand the security aspects of the microarchitecture of computers.

A key requirement for many of the documented attacks is that the adversary is able to efficiently evict specific content from the CPU's cache [1]. Reduction algorithms are necessary for finding minimal eviction sets, but there is no evaluation of the robustness of the different reduction algorithms under different replacement policies, or different cache characteristics. Pseudo-random replacement policies can be found in ARM processors. Pseudo-random replacement policies difficult attacks on caches [13], and however they have not been systematically analyzed.

There are already a number of models and simulations of caches, like [14] and [15], but these models generally focus on performance, instead of security.

The objective of this thesis was to create a model of the microarchitectural aspects of computers relevant to cache side-channel attacks. In addition, we wanted to use the model to compare the performance and robustness of different reduction algorithms proposed to compute minimal eviction sets. Such a model would act as a bridge between the practical results of the tests over a real cache, and the theory of how caches act:

In order to achieve a clean and readable representation, we decided to use the functional programming language Haskell to implement the model. The purpose was to be able to simulate the behaviour of the microarchitecture of computers with different characteristics, such as cache replacement policies, size of the cache and associativity. Hence the model had to be parametric. Real-life performance and implementation details that do not affect security are not relevant to the model. Therefore, in order to make the simulation faster, and the mechanisms easier to understand, we decided to make an abstraction of the microarchitecture: instead of modeling all the elements, the model had to simulate the behaviour of the relevant features.

As a complement to the implementation of the model, we wanted to analyze the effect of the different replacement policies and cache parameters on eviction rates and

replacement policies, and perform a comparative analysis on the behaviour of two reduction algorithms.

Organization of the report

The rest of this report is organized in 7 parts: Chapter 2 is a more detailed theoretical background, with information about caches, virtual memory, and cache side-channel attacks. Chapter 3 contains an explanation of the different algorithms proposed to perform eviction tests and reductions, and Chapter 4 explains how those algorithms are adapted to the model. It includes also the different data abstractions created for the model. Every algorithm is explained in Section 3.X, and its abstraction is explained in Section 4.X. Chapter 5 discusses how the cache has been modeled. Chapter 6 discusses the implementation aspects and the representation of the microarchitecture. Chapter 7 includes results of testing the model, an analysis and discussion about the outcome. Finally, Chapter 8 is the conclusion of the report.

Chapter 2

Caches and virtual memory

This section provides background on side-channel cache attacks and the microarchitecture involved.

2.1 CPU caches

Nowadays, the speed of processors is higher than the speed of main memory in computers. Caches are small memories in comparison to the main memory of a computer, they are faster than main memory, they are close to a processor core, and they help bridge the latency gap between processors and memory by copying frequently accessed data.

Current CPU caches are organized following a hierarchy of different levels: There are more small and fast caches that are closer to the cores, and a larger and slower *last-level cache* (LLC), shared between all the cores. The relationship between the content of the different levels of the cache is defined by an inclusion policy. We consider it here to be *inclusive*, as is usually the case in Intel caches.

An inclusive policy means that the content of the smaller, lower caches is also in the LLC, and content evicted from the LLC is removed from the others.

The main memory of a computer is logically partitioned into *blocks*. Caches are partitioned into lines, each of them exactly the size of a block.

The most common kind of caches, which is the kind that we studied in this project is *set-associative caches*: Caches are partitioned into equally sized *sets*, each of them with the same number of lines. This number of lines is the *associativity* of the cache.

When a block has to be accessed, first there is an attempt to fetch it from the cache.

If it's successful and the block is already in the cache, that is a *cache hit*. If it's not, there is a *cache miss* and the block is retrieved from main memory and stored in the cache for later availability.

The physical address is what determines to which cache set to map content from memory: in the example of Figure 2.1, bits 6 to 18 of the physical address indicate the cache set. Within that set, the block can be stored in *any* of the associativity lines.

Therefore, when there is a cache miss and a block is inserted in the cache, it has to replace a memory block from the set determined by its physical address.

Replacement policies are in charge of deciding which memory block to *evict* from the cache. There are deterministic replacement policies and probabilistic replacement policies.

2.1.1 Replacement policies

Deterministic replacement policies

- **LRU** Discards first the *least recently used* (LRU) items. The cache must keep track of the age of each line.
- **MRU** Evicts first the *most recently used* (MRU). This algorithm behaves best when there is a scan over a large dataset, as it retains older data.
- **PLRU** Pseudo-LRU is similar to LRU but does not require to store the age of every item. It works as a binary tree with 1-bit pointers that indicate the less recently used subtree.
- **FIFO** Works as a FIFO queue. The first block accessed is discarded, regardless of how many times it has been accessed.
- **LIP** [16] LRU insertion policy or LIP inserts all the lines in the LRU position. They are only moved to the MRU position if they are re-referenced while still in the LRU.

Probabilistic replacement policies

- **RR** Evicts a random element
- **BIP** [16] the bimodal insertion policy is derived from LIP. It behaves like LIP, but has a low probability (usually 1/32 or 1/64) of inserting the new line in the MRU position

– **PLRU-Rand** There are two variants of randomized PLRU. They can be found in some Intel processors, and have been reverse-engineered by [17]. They also work as a binary trees, but with randomized nodes: In one variant, the last node before the leaves of the tree do not have a pointer. Instead, they randomly direct to one leaf or the other. The other variant, it is the first node that randomly points to a subtree. In this second case, this first node can point to more than two subtrees, for example if the associativity of the cache is not a power of 2.

Adaptive replacement policies

It has been proposed to use an *adaptive* policy that selects dynamically the replacement policy [16]. It divides the sets of the cache in *followers* and *leaders*, and keeps a counter that records the number of hits of the leaders, that have a fixed policy. It then sets the followers to use the policy that is more effective.

2.1.2 Slicing

Slicing is a partition of the LLC made by modern Intel CPUs. Usually there are 2^s many slices, typically one or two per core. The slice is determined by an undocumented s-bit hash of the most significant $n-l$ bits of the address, so with slicing, the c bits only determine the cache set within the slice, but the slice itself remains unknown [1].

2.2 Memory organization

Memory addresses are used by CPUs to track and access data stored in the main memory of a computer. A memory address is a unique identifier that represents, a memory location in the computer.

Physical addresses are the real identifiers of a memory location, usually only used by the sy stem's software, like the BIOS or operative systems.

Nowadays, most programs don't have access to physical addresses, but to *virtual addresses*.

Virtual memory allows the computer to compensate physical memory shortages by expanding main memory (RAM) with disk storage. It also hides the fragmentation of physical memory, letting programs use different parts of the memory as if they were contiguous addresses by providing a linear memory space.

With virtual memory, applications do not have to manage shared memory or take

into account the amount of available RAM. They are given a virtual memory space, and the memory management unit (MMU) is in charge of translating the virtual addresses used by the program to the physical addresses used to access the data.

Physical memory is partitioned in *pages* of size 2^p.

The translation from virtual to physical addresses mantains the last p less significant bits, that represent the *page offset*, and maps the rest of the bits, from *virtual page number* to physical frame number. The transformation of this bits is not known, and therefore treated as uniformly distributed. Figure 2.1 shows the translation, as well as the cache's interpretation of the physical address bits.

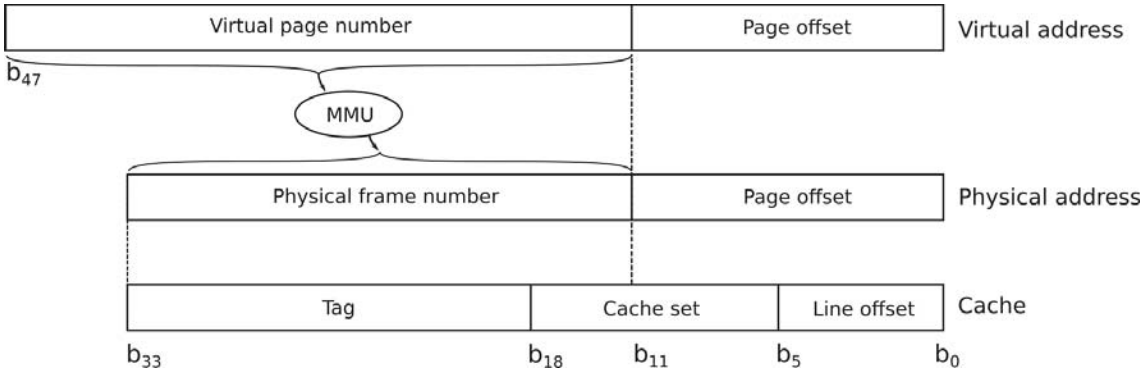

Figure 2.1: Anatomy of a memory address, for virtual addreses of 48 bits, 1024 cache sets ($c = 13$), lines of 64 Bytes ($l = 6$), pages of 4KB ($p = 12$)

2.2.1 TLB

The mapping of virtual to physical addresses is done with a tree structure called *page table*. When a virtual address is accessed, the MMU traverses the page table up to the corresponding physical address. This is called a *page walk*.

The *translation lookaside buffer* (TLB) is a cache of the MMU. It stores recent translations from virtual to physical memory.

Every time a virtual memory address is accessed, the system has to look for the translation to a physical memory address. The TLB is checked at this point. If the address is not found in the TLB, then a page walk is performed.

2.3 Cache-based side-channel attacks

Cache side-channel attacks are enabled by the microarchitectural design of the CPU. There are several ways to monitor what is happening in the cache, making caches an interesting source for side-channel attacks.

Cache side-channel attacks can be time-driven, trace-driven and access-driven. This proyect studies access-driven side-channel attacks.

Access-driven side channel attacks [18–20] are performed in environments where the attacker and victim share a same hardware architecture, and as a consequence, a cache. This happens commonly in the cloud. The last-level cache is shared globally among all users, despite the illusion of isolation, and an attacker can access its own memory region and infer victim's use of the shared cache.

Examples of this kind of attacks are Evict+Time [19], Flush+Reload [12] or Prime+Probe attacks [19].

Prime + Probe

(1) The attacker primes a cache set to contain a known group of addresses.

(2) The attacker waits while the victim executes something.

(3) The attacker accesses again the group of addresses from the first step, measuring the access time for each address. If the access is fast, there has been a cache hit. If it's slow, then there has been a cache miss, meaning that the victim has accessed that cache set, and the attacker address has been evicted.

In order to perform this kind of attacks, an attacker needs a set of addresses that evict all the content from the cache.

We define *congruence* as an equivalence relation between addresses. Congruent addresses are those that map to the same cache set.

Eviction sets

We define an *eviction set* as a set of congruent addresses of size *at least* associativity. When the addresses of the set are accessed sequentially, all the content from the cache set is evicted. As we will see later, this is only true for some replacement policies.

We define an *evicting set* as a set that in practice evicts content from the cache. In LRU, for example, an evicting set is equivalent to an eviction set, but in MRU, a set with only one congruent address can be evicting, but not be an eviction set.

We define a *p-evicting set* as a set that is an evicting set with probability p.

The challenge of finding an eviction set is that the cache set is determined by physical addresses, and an adversary can only have access to the virtual address space. Therefore, an adversary can't examine a virtual addresss and see to which cache set it belongs.

In order to determine whether a set can evict content from a victim cache set, it's necessary to perform an *eviction test*. The test can be found in Section 3.1, and the abstraction for the model in Section 4.1.

Reduction algorithms

A set with a big enough number of addresses can be evicting, even if not all of the addresses are congruent, but performing attacks with such sets would be slow and could introduce unwanted noise in the cache due to the unnecessary memory accesses. The ideal set of addresses for an attack would be a *minimal eviction set*: an eviction set of size exactly associativity.

There are algorithms, called *reduction algorithms* that are used to reduce a set of addresses to a minimal eviction set.

Two reduction algorithms have been studied in this project, a baseline reductions that can be found in Section 3.2, and a threshold group reduction, in Section 3.4. The abstractions and adaptation to the model for these algorithms are in Section 4.2 and Section 4.4 respectively. A second baseline algorithm has been described in Section 3.3 and implemented in Section 4.3, but not studied in comparison with the two others.

Chapter 3

Reduction algorithms: Finding optimal eviction sets

3.1 Eviction test

Eviciting content from the cache is essential in many documented side-channel attacks. For that, an attacker needs an *eviction set*.

As it is not possible for an attacker to compare the bits of the virtual addresses, knowing if a set is evicting requires tests that involve timing side-channels.

This eviction test from [5, 21], requires being able to measure access times, distinguishing between cache hits and misses.

The test reveals if a set of addresses is evicting for a victim address a_v:

(1) The victim a_v is accessed

(2) All the elements of the set are sequentially accessed: $a_1, a_2, a_3, ..., a_n$

(3) The victim a_v is accessed again, and this access is timed.

If the time for the second access to the victim is above a threshold which indicates a cache miss, then the set of addresses has evicted the victim from the cache. Otherwise, the victim has remained in the cache from the first access.

3.2 Baseline reduction

There are baseline reduction algorithms that have been informally described in the literature.

The pseudocode for Algorithm 1 was described in [1].

The baseline algorithm receives an input of S, an eviction set for v, as well as v. It iterates over S, picking each candidate address c and testing if $S' = S \setminus \{c\}$ is still an eviction set for v. If it is, then it continues, but if it is not, then it copies c in R to record it and continues, until R is of size associativity.

Algorithm 1 Reduction algorithm 1

Input: Set of addresses S, victim address v
Output: Minimal eviction set R

$R \leftarrow [\,]$
while $|R| <$ *associativity* **do**
  if $\neg$TEST$(R \cup (S \setminus \{c\}),\ v)$ **then**
    $R \leftarrow R \cup \{c\}$
  $S \leftarrow S \setminus \{c\}$
return R

This algorithm requires $\mathcal{O}(N^2)$ memory accesses, where N is the size of the set, as this size is an upper bound for the number of loop iterations [1].

Some reduction algorithms use extra redundancy, performing a reduction with two steps, like the following reduction algorithm.

3.3 Baseline reduction 2

There are variations of this algorithm. The approach used in this project is adapted from the one proposed in [5], but it finds an eviction set for only one victim set.

The algorithm uses a `Probe` function that tests if a set evicts an address, and a main reduction function that is divided in two parts: first it constructs a *conflict set*, that contains exactly enough addresses to be evicting (associativity if the replacement policy is LRU), but that can also contain addresses from other sets.

Then, it makes an *eviction set*, by eliminating taking from the conflict set all the addresses that make it an eviction set: those that make the conflict set not evicting when they are removed from it.

Algorithm 2 Reduction algorithm 2

Input: Set of addresses S, victim address v
Output: Minimal eviction set R

```
function PROBE (Set S, Address c)
    access c
    for all l in S do
        access l
    access c and measure time
    return time > threshold                ▷ Returns True if the set is evicting for c

function REDUCTION (Set s)
    conflictSet ← [ ]
    for all candidate in S do
        if ¬ probe (conflictSet, candidate) then
            insert candidate in conflictSet
    evictionSet ← [ ]
    if probe (conflictSet, v) then
        for all l in conflictSet do
            if ¬ probe (conflictSet - l, candidate) then
                insert l in evictionSet
    return evictionSet
```

3.4 Threshold group reduction

We present now Algorithm 3, proposed in [1]. It reduces an eviction set to an optimal size in $\mathcal{O}(N)$ memory accesses, where N is the size of the set.

Algorithm 3 Reduction algorithm 3

Input: Set of addresses S, victim address v, associativity a
Output: Minimal eviction set

```
while |S| > a do
    {T_1, ..., T_{a+1}} ← split (S, a+1)
    i ← 1
    while ¬TEST (S \ T_i, v) do
        i ← i+1
    S ← S \ T_i
return S
```

It is based on ideas from threshold group testing. Group testing consists on reduc-

ing the number of tests that have to be performed, by eliminating groups of elements, instead of performing individual tests on all of them.

The algorithm is based on the following: if there are at least a congruent elements in the set, and we partition it in $a+1$ partitions, there is at least one partition that can be eliminated from the set, and it will still be an eviction set.

The algorithm receives as input a set of addresses S and a victim address, and splits the set into $a+1$ partitions $T_1, \ldots, T_{a+1}$. It finds the first partition T_i such that $S' = S \setminus T_i$ is still an eviction set, and repeats the process with S', until the resulting set is of size a.

Chapter 4

Abstracting reduction algorithms

4.0 Data representation

Memory addresses in a computer are composed of a series of bits: zeros and ones.

Those bits carry a lot of information, both for physical and virtual addresses, like the mapping of the address in the cache, the position of a memory location or where to find the translation to the corresponding physical address.

If the model worked with virtual addresses generated as literal representations of bits, most of those bits would be discarded during the translation, and then of the bits representing the corresponding physical addresses, the only part used would be the bits establishing the set of the address. Because of this, the model does not treat addresses as chains of zeros and ones, but instead makes an abstraction that preserves only the relevant information of the addresses.

In this Section, we present the different abstractions, and how the algorithms from Section 3 have been adapted to fit them.

Following this, there are different abstraction of the addresess:

Address abstraction 1

For algorithms tested in this project, eviction and the reduction algorithms, all the addresses are tested against that one target set.

In order to test if a set is evicting, the only characteristic of the addresses of the set that is important is whether they belong to the target set or not.

For this second abstraction, a set of addresses is represented as a *SetState(cn, n)*, where *cn* is the number of addresses that are congruent to the target set, and *n* is the total size of the set of addresses. For example, *SetState(3, 25)* is a set with 25 addresses, of which 3 are congruent.

All the addresses from the same set are interchangeable.

```
SetState :: (Congruent, Total)

Congruent :: Int
Total :: Int
```

Address abstraction 2

The second abstraction is used when all the addresses belong to the same cache set.

An AddressIdentifier is used to identify different addresses from the same set, as a SetState does not allow distinguishing congruent addresses.

```
AddressIdentifier :: Int
```

It's used in two structures:

(1) The state of a cache set. The state of a cache set is represented by the type `CacheSetContent`, which is a list of AddressIdentifier. It must be of size associativity, as a cache set can only contain as many as associativity addresses.

(2) A trace of addresses that is going to be inserted in a set. A Trace is a list of AddressIdentifier, of any length. Each element of the trace is an address that will be inserted into the cache, therefore two repeated elements of the list represent the same address (any address belonging to that cache set), that is inserted twice.

```
Trace :: [AddressIdentifier]

CacheSetContent :: [AddressIdentifier]
```

The identifiers of each address indicate whether an address is already in the cache, or if the trace of addresses contain patterns of repeated addresses, or on the contrary they are all unique.

It is possible to create a trace of addresses from a SetState: A set of addresses SetState(cn, n) that is going to be introduced in the cache, is transformed into a trace [1, 2, ... cn], with all the congruent addresses of the set. We can also build a list of identical elements of any length represent a trace where the same address being introduced multiple times.

4.1 Eviction test

Algorithm 4.1 is the adaptation of the eviction test described in Section 3.1.

Algorithm 4 Reduction algorithm 1: Abstraction

Input: SetState $S(c, n)$, ReplacementPolicy pol
Output: Maybe minimal eviction set

$i \leftarrow$ `initial_cache_state`
$t \leftarrow$ `consecutive_trace`(S)
$(nc, \_) \leftarrow$ INSERT$(pol, i, (\mathit{AddressIdentifier}(c+1) : t), n)$
$(\_, h) \leftarrow$ INSERT$(pol, nc, (\mathit{AddressIdentifier}(c+1), 1)$
return h = 0

Algorithm 4.1 takes as input a tuple (c, n) that represents a set of n addresses, of which c are congruent, and a replacement policy pol. It generates a trace t of AddressIdentifier $[1 \dots c]$.

Then a representative of the victim address, AddressIdentifier $c+1$, is appended to the beginning of the trace. This is because in the real eviction test the victim is accessed before the addresses in the set.

The initial state of the cache i is generated as a list of size associativity, of AddressIdentifier set to 0.

The new trace t' is sent to the cache, along with the replacement policy, the total number of addresses, and the initial state of the cache set.

The model of the cache is in charge of computing the TLB noise, introducing the trace in the cache, and returns a tuple with the new cache state (that can contain some of the addresses of the cache), and the number of hits. In this case, the number of cache hits is always 0, as all the addresses introduced are different.

Finally, the test sends to the cache a trace consisting of only the victim address, AddressIdentifier $c+1$, and gets back a new cache state, and the number of hits for the victim. If the number of hits is 0, then the set S is evicting. If the number of hits is equal to one, the victim was still in the cache from the first access, and the set is not evicting.

Test for different traces of addresses

Algorithm 4 for testing eviction creates a trace made of a sequence of all the congruent addresses in a set. This is only one of multiple combinations that could be tested. For LRU, this approach is straightforward and effective, as new, not visited addresses evict old addresses. In other replacement policies, other patterns might be more optimal.

For instance, there could be tests performed where a same address is accessed multiple times (a trace $[a_d, a_d, a_d, \ldots, a_d]$), or a same pattern of addresses is repeated ($[a_1, a_2, a_3, a_1, a_2, \ldots, a_3]$). This sequential trace of unique elements was chosen because it is the result of testing all the addresses in a set of addresses. This is the approach used when the congruent addresses of the test are not known, and an attacker accesses all the addresses of the set sequentially.

If the congruent addresses are known, for instance after a successful reduction, tests can be performed with different patterns of traces. In order to test different patterns, Algorithm 4 can be modified to create different traces.

Test for different replacement policies

Algorithm 4 can be tested with all the implemented replacement policies: LRU, FIFO, MRU....

Some of this replacement policies are deterministic, such as LRU, MRU, FIFO and LIP. They always return the same result when a same set is tested with them, not counting potential noise from the TLB.

The other policies implemented are probabilistic: a same set can be evicting once, and then in the next test not be evicting. For example BIP is implemented such that new elements are generally inserted in the MRU position of the cache set, but they have a small probability of being inserted in the LRU. This can make a reduction algorithm fail easily. If a test results in a false positive, the reduction can fail.

In order to be able to deal with these probabilistic replacement policies, we defined in Section 2.3, the concept of a *p-evicting* set. A p-evicting set is a set that has a probability of *at least p*% of being evicting. This definition allows to learn more information about probabilistic replacement policies.

Property

Thanks to the definition of p-eviction, we can perform two tests:

First, we can check the p of eviction of a set by testing for eviction multiple times and calculating the probability.

Second, we can fix a probability p, and use a test for p-eviction instead of a simple eviction test during the reduction.

In all the reduction algorithms defined below there is a `Property` test.

In deterministic replacement policies, the property is an eviction test:

Property *(Set s, d_replacement_policy)* = Evicts *(Set s, d_replacement_policy)*

In probabilistic replacement policies, the property is a p-eviction test:

Property *(Set s, p_replacement_policy)* = P-evicts *(Set s, p_replacement_policy)*

Where `P-evicts` is a tests that performs an eviction test multiple times, and is true if the set is evicting at least p% of the times.

4.2 Baseline reduction

Algorithm 5 is the adaptation to the model of Algorithm 3.2.

We have generalized the reduction algorithm: The size of the output set S is not fixed, but can be chosen. The value of this parameter is by default *associativity*.

Algorithm 5 Reduction algorithm 1: Abstraction

Input: Set of addresses $S(c,n)$, replacement policy pol, reduction size r
Output: Maybe minimal eviction set

```
function TAKE(Set of addresses S(c,n))
    f ← RANDOM(c,n)
    return ((c − f, n − 1), f)

function BASELINE_REDUCTION(Set of addresses S, replacement policy pol)
    eviction_set (ec, et) ← (0,0)
    BASELINE(S, pol, eviction_set)

function BASELINE(Set S(c,n), ReplacementPolicy pol, Set eviction_set (ec,en))
    while ec < r ∧ n > 0 do
        (new_state, f) ← TAKE(S)
        if PROPERTY(new_state + eviction_set, pol) then
            BASELINE(new_state, pol, eviction_set)
        else BASELINE(new_state, pol, (ec + f, en + 1))
    if en = r ∧ ec = r then
        return Just(eviction_set)
    else return Nothing
```

The algorithm has an auxiliary method called `Take`. This method takes a set of addresses(c, n),and randomly chooses an address from the set. The probability of choosing a congruent address is c out of n.

The original set (c, n) minus the taken address is S'(c-f, n-1). If the address randomly chosen was congruent, f is 1, otherwise it is 0. The algorithm returns the new set S' and f.

The reduction, `Baseline_reduction` receives as input an eviction set of addresses S and a replacement policy pol. If the set is not evicting, the reduction fails. If the set is evicting, the algorithm calls the auxiliary method `Baseline` with S, pol and a new empty set of addresses.

`Baseline` receives the set of addresses that have to be reduced, S, the replacement policy r, a set of addresses called *eviction_set*.

For each iteration, the algorithm calls the function `Take`. This function returns a tuple (*new_set*, f). *New_set* is the original set minus one element. If the element is congruent, f is 1, and if it's not congruent, it's 0 Then it checks if *new_set* plus *eviction_set* holds a property of being p-evicting 4.1. If it is not, then the element that has been taken out was a congruent address that made the eviction set. The element is copied to *eviction_set*. If after the test the property does not hold, then the address was not congruent. If the address was not congruent, the eviction set does not change.

Finally, the function `Baseline` is called again, with *new_set* and the new eviction set. The recursion will stop once the eviction set has reached the desired size of r, or when all the set has been visited, so when all the addresses have been taken and *new_set* is empty.

4.3 Baseline reduction 2

Algorithm 6 is the adaptation to the model of Algorithm 3.3. This algorithm is an other baseline approach to the reduction. One difference between Algorithm 5 and Algorithm 6 is that the second one does not need a fixed length for the minimal eviction set. This algorithm has more complexity than the other baseline reduction studied, because it needs to construct two lists in order to complete the reduction.

The algorithm can be used to find eviction sets for more than one cache set.

Algorithm 6 Reduction algorithm 2: Abstraction

Input: Set of addresses $St(c,n)$, ReplacementPolicy pol, reduction size r
Output: Maybe minimal eviction set

```
S ← createBinaryList (St)
conflictSet ← [ ]
for all candidate in S do
    if ¬PROPERTY(()conflictSet, candidate) then
        insert candidate in conflictSet
v ← 1                                                    ▷ victim
evictionSet ← [ ]
for all l in conflictSet do
    if ¬PROPERTY(conflictSet - l, pol, candidate) then
        insert l in evictionSet
return evictionSet
```

The algorithm takes as input a set of addresses $St(c,n)$ and a replacement policy pol. It generates a random list S of ones and zeros, with c ones and $n-c$ zeros. The list S represents the set of addresses. Each one represents a congruent address, and each zero is a non-congruent address. Then the algorithm creates an empty list *conflict_set*. For each element of the list, there is a test of `Property` (4.1). If the property does not hold for the candidate and the *conflict_set*, then the candidate is inserted in the *conflict_set*: The candidate was not evicted by the conflict set, and therefore there are not enough addresses congruent with the candidate.

Once all the elements of the set have been visited, the algorithm creates a new empty list: *eviction_set*. The algorithm should have exactly associativity congruent elements in the conflict set. The algorithm iterates over the conflict set and builds the eviction set: if the conflict set does not hold the property when an element is eliminated, that element is copied to the eviction set. The size of the final set is determined by the replacement policy, but in LRU, for instance, the eviction set will have exactly associativity elements.

4.4 Threshold group reduction

Algorithm 7 is the adaptation to the model of Algorithm 3.4.

We have also generalized this reduction algorithm to be parametric for the output size of the reduction. By default this size r is *associativity*.

This threshold group reduction algorithm receives a set of addresses $S(c,n)$, and a replacement policy pol.

While the number of elements of the set is bigger than r, the algorithm reduces the size of the set, in groups. First, it divides the set in $r+1$ partitions. Then, it converts each partition (C_i, N_i) to a partition $(c - C_i, n - N_i)$, so each partition is a subset of the set, minus the corresponding partition. Then, it finds the first of the subset for which the property still holds. This subset is the new subset used to call the reduction again.

Algorithm 7 Reduction algorithm 3: Abstraction

Input: SetState $S(c,n)$, ReplacementPolicy pol, size r
Output: Maybe minimal eviction set

while $(n > r) \wedge (c > r)$ **do**
  $\{(c_1,n_1),\ldots,(c_n,n_n)\} \leftarrow split\ (S, r+1)$
  $\{T_1 = (c-c_1, n-n_1), \ldots, T_{r+1} = (c-c_n, n-n_n)\} \leftarrow \{(c_1,n_1),\ldots,(c_n,n_n)\}$
  $i \leftarrow 1$
  while $\neg$PROPERTY(T_i, pol, v) **do**
    $i \leftarrow i+1$
  $S \leftarrow T_i$
return S

Chapter 5

Abstracting microarchitecture: The cache model

The objective of the model is to represent security aspects of the microarchitecture, meaning it is not necessary to emulate the details that do not affect security.

The model of the cache is composed by four elements: slicing, the different replacement policies, the TLB simulator, and the cache itself.

5.1 Slicing

Slicing is a partition of the LLC. It behaves a an unknown s-bit hash of the most significant $n-l$ bits of the address. This means that knowing a physical address is not enough to know the cache set to which that address belongs. If two addresses have the same cache set bits, they can belong to different slices and therefore, not map to the same cache set.

In order to replicate this in the model, we add two extra bits to the addresses. These bits can't be known even if the adversary controls all the set bits of the addresses. In order to do that, we sum s, the number of slice bits, to the color bits in the model.

5.2 Replacement policies

In a cache, replacement policies are responsible for deciding which blocks are evicted upon a cache miss.

In the model, a series of different functions, corresponding to replacement policies,

determine how the cache saves or evicts addresses, depending on which policy it is using. These functions dictate the behaviour of the cache.

```
RepPol :: CacheSetContent -> Trace -> IO(CacheSetContent, HitNumber)

CacheSetContent :: [SetIdentifier]

SetIdentifier :: Int
```

These algorithms receive as input the `CacheSetContent`, which is the content of the cache set where the addresses are going to be inserted, as well as a trace of addresses, in the form of a list of address identifiers, as described in Section 4.0.

Each algorithm inserts all the elements from the trace sequentially. For each element inserted, it counts if it is already in the cache set (cache hit), or if it's not. If it's not in the cache it inserts it, as established by the replacement policy. After inserting all the elements, it returns the new contents of the cache set, as well as the number of hits.

There are two kinds of replacement policies: deterministic and probabilistic.

The implemented replacement policies are LRU, MRU, FIFO, RR, LIP and BIP 2.1.1.

5.3 TLB misses

Accessing virtual addresses for eviction tests implies that the TLB will have to translate each of them. This can result in TLB misses, that trigger page walks. These page walks may introduce data into the victim cache set, maybe evicting content from the target cache set, regardless of the set that is being tested for eviction. This can result in false positives.

In order to imitate the effect of the TLB in the tests, we compute the TLB misses caused by accessing the n elements of the set of addresses:

When a page walk occurs, we suppose that there is a block of data randomly inserted into one of the cache sets. The affected cache sets are not only the *color* cache sets from where the attacker chooses the addresses, but all of the sets of the cache.

We computed the number of TLB misses that were generated by the access to a set of addresses by associating each address to an identifier, and creating a random list of the addresses for which the TLB had a translation. The addresses of the set that were not in the TLB would cause a TLB miss. Additionally, we added associativity to the TLB.

The average result of counting each time the number of TLB misses showed to be

equivalent to the difference between the capacity of the TLB and the number of TLB misses, as shown in Figure. Because of this, in order to make the rest of the tests faster, the number of TLB misses was estimated with Algorithm 8.

Algorithm 8 Expected TLB misses

Input: Size of set n
Output: Estimated average number of TLB misses
$TLB\_capacity = 2^{TLB\_bits}$ ▷ TLB_bits indicate the position within the TLB

if $n <= TLB\_capacity$ **then**
return 0
else
return $TLB\_capacity - n$

From this total number of TLB misses m, the model of the TLB counts how many belong to the victim cache set: m'. For this it follows a process identical to generating a new SetState, with m total addresses, and the resulting number of congruent addresses, how many TLB misses affect the victim cache set. The only difference is that when generating addresses, they can belong to any cache set, not only to $color$ different ones.

5.4 Cache

The *interface* method represents the cache itself. It organizes and puts the rest of the elements together.

The interface receives three inputs:

(1) rep, the replacement policy, one of the defined in 5.2

(2) $setcontent$, the content of the cache set where the addresses will be inserted

(3) t, the trace of elements that are going to be accessed in the cache, as defined in Section 4.0

(4) n, the total number of addresses

The reason why the cache needs both a trace and the total number of addresses (including the non-congruent ones) is that the TLB noise is determined by the total number of address accessed. At the same time, it was interesting to have a cache where the trace of addresses could be determined from outside the cache (as opposed to the cache creating the trace from the number of congruent addresses of a set of addresses), in order to be able to experiment with different combinations of addresses in the traces.

The interface uses the number of addresses to calculate the number of TLB misses for the victim cache set m'. m' number of elements are then added to the trace t. Then the cache calls replacement policy rep with $setcontent$ and the trace t. The policy returns the new state of the cache set, and the number of hits, and the interface outputs the same information.

Chapter 6

Implementation of the Haskell model

This section discusses various aspects of the implementation of the model.

The code can be found at https://github.com/palomatanis/ideal-memory-model

6.1 Haskell

The model has been implemented in Haskell [22]. Haskell is a functional programming language. We chose this language because we wanted simplicity and clean, readable code, over a more complicated The objective was to write a *mathematical executable model*.

The type system of Haskell was a decisive factor. It is not necessary to read the implementation of the methods to know their function. Haskell data types make the behaviour of the program understandable without needing to read the actual code.

The IO monad was used in all the implementation. In pure functional programming, a function called multiple times with the same arguments should return always the same output. For the model, all the sets of addresses had to be randomly generated. This introduced the IO monad in the address set generators, and in the probabilistic replacement policies, and spread to the rest of the functions.

6.2 Property testing

Quickcheck for Haskell is a library for random testing properties of Haskell programs [23–26]. It allows to define a series of properties and performs random testing to check whether the properties hold in the program.

Property testing with Quickcheck requires first defining the properties of the program, and then generating tests for each properties, as well as data generators for the different data types.

Defining the properties of the program

In order to test the program, it was first necessary to define its properties. The properties have been defined for those replacement policies where the behaviour is deterministic.

- A generated set of Addresses of n elements are of length n
- A generated set of Addresses have only addresses represented by cache sets in the correct range [0, free cache sets - 1]
- A generated SetState with n addresses are of the form `SetState(c, n)` where c is between 0 and n
- The number of tlb misses for n addresses are between 0 and n
- A generated TLBState with n addresses are of the form `SetState(c, m)` where m is the number of estimated cache misses for n and c is between 0 and m
- The group reduction suceeds when there are associativity many addresses or more
- The group reduction fails when there are less than associativity addresses for LRU and FIFO
- The group reduction ends with exactly associativity addresses for LRU and FIFO
- The group reduction ends with associativity congruent addresses for LRU and FIFO
- The linear reduction suceeds when there are associativity many addresses or more for LRU and FIFO
- The linear reduction fails when there are less than associativity addresses for LRU and FIFO
- The linear reduction ends with exactly associativity addresses for LRU and FIFO
- The linear reduction ends with associativity congruent addresses for LRU and FIFO
- A replacement policy executed with an empty trace does not change the cache state, for all replacement policies
- A replacement policy executed with a trace of one address on a cache state with that address produces a hit
- An eviction test with less than associativity does not succeed in LRU and FIFO

- An eviction test with associativity or more addresses suceeds in LRU and FIFO
- An eviction test executed with a trace of only two identical addresses is evicting for any replacement policy

Implementation of the tests: Generators and Properties

The generator of addresses generates random addresses of type `Address` within the range of sets

```
instance Arbitrary Address where
  arbitrary = do
    a <- choose(0, free_cache - 1)
    return $ Address a
```

The cache state generator creates structures of type `SetState` with a number of congruent addresses equal or smaller to number of total addresses (which must be a positive number or 0)

```
instance Arbitrary SetState where
  arbitrary = do
    NonNegative total <- arbitrary
    congruent <- choose (0, total)
    return $ SetState(congruent, total)
```

The properties described above were tested with different functions.

The following property tests that the generator of addresses returns a set of addresses of the correct size

```
prop_address_list_size :: (NonNegative Int) -> Property
prop_address_list_size (NonNegative n) = monadicIO $ do
  l <- run $ list_random_sets n random_set
  let lenl = length l
  assert $ lenl == n
```

The parameter, the length of the list has to be an Integer equal or greater than 0. Then the property asserts that the generator does create a list of that size.

This next test test that the generators only creates sets where the addresses are in the correct range of cache sets

```
prop_address_list_range :: (Positive Int) -> Property
prop_address_list_range (Positive n) = monadicIO $ do
  l <- run $ list_random_sets n random_set
  let max = maximum $ map (\(Address i) -> i) l
  let min = minimum $ map (\(Address i) -> i) l
  assert $ max < (2^cacheSet) && (min >= 0)
```

This first two tests are repeated for other generators.

Following, the calculated number of TLB misses for a trace of length n must be in the range of [0, n]

```
prop_tlb_misses :: (NonNegative Int) -> Property
prop_tlb_misses (NonNegative n) = monadicIO $ do
  r <- run $ list_random_tlb n
  let t = tlb_misses r
  assert $ (t <= n) && (t >= 0)
```

This test verifies that when a SetState is randomly generated, the values are within correct ranges

```
prop_make_cache_state_size :: (NonNegative Int) -> Property
prop_make_cache_state_size (NonNegative n) = monadicIO $ do
  SetState(c, t) <- run $ random_cacheState n
  assert $ (t == n) && (c <= t) && (c >= 0)
```

The tests continue with the eviction test. This test tries to introduce victim address in the cache, then accesses all the set of addresses (SetState) and tries to access the victim again. If the set is evicting, then the eviction is successful.

It should be successful when there are at least associativity many addresses in SetState.

```
prop_evicts :: SetState -> Property
prop_evicts cacheState@(SetState(congr, total)) = monadicIO $ do
  e <- run $ evicts cacheState lru
  if (congr >= associativity)
    then assert e
    else assert $ not e
```

The following test tests that when an empty trace is inserted in a cache set, the content remains the same.

```
prop_rep_empty_trace :: Set -> Property
prop_rep_empty_trace set = monadicIO $ do
  (s, h) <- cacheInsert lru set (Trace 0)
  assert $ s == set
```

These last two properties are for reduction and are for all replacement policies.

The first checks that when the replacement policy receives as imput a cache state where there are at least associativity many addresses, the reduction succeeds.

```
prop_group_reduction_bool :: SetState -> Property
prop_group_reduction_bool cacheState@(SetState(congr, total)) = monadicIO $ do
  r <- run $ reduction cacheState lru
  if (congr >= associativity)
    then assert r
```

```
    else assert $ not r
```

The second tests that the output of the reduction is SetState(associativity, associativity) if it succeeds or else its the input.

```
prop_group_reduction :: SetState -> Property
prop_group_reduction cacheState@(SetState(congr, total)) = monadicIO $ do
  SetState(c, t) <- run $ reduction_b cacheState lru
  if (congr >= associativity)
    then assert $ (c == t) && (t == associativity)
    else assert $ (c == congr) && (t == total)
```

Finally all the tests are run from the main, trying 100 tests for each of the implemented properties.

```
main :: IO ()
main = do
  putStrLn "Created list of addresses is correct size 1"
  quickCheck prop_address_list_size_partial

  putStrLn "Created list of addresses is correct size 2"
  quickCheck prop_address_list_size
```

Quickcheck results

Quickcheck was useful. The properties were a way of not only testing the already implemented program but also a way of reasoning about the implementation.

There was an error in the code, found as a result of the testing: In occasions, the group reduction finished with a CacheSet that had associativity congruent elements but more total elements.

An example of the error message `*** Failed! Assertion failed (after 32 test): CacheState(16, 17).`

This was error was corrected, now the implementation of the program succeeds on all the tests.

6.3 Address generation

Sets of addresses are needed in order to execute the model and test the algorithms.

When a set of addresses is chosen by an adversary in preparation for an attack, the adversary can select addresses that have the same page offset. The set of addresses will be of size n.

The selection of addresses with the same offset allows to control partially the set of the corresponding virtual addresses:

Color

A user without root privileges can only control the virtual address space, which means that only a part of the virtual addresses will remain unchanged when translated to physical addresses: the $color$ bits where the cache set overlaps the page offset, as the page offset bits p are identical for virtual and physical addresses.

$$color = c + l - p$$

In the case of the example of Figure 2.1, $color = 7$ bits.

Therefore, if an adversary chooses a set where all the addresses have the same page offset, the addresses will belong to 2^{color} sets instead of 2^c sets, where $color \leq c$, increasing the probability for the addresses of the set to be congruent.

We consider the translation of addresses to behave as a uniform distribution. Hence from the set of n virtual addresses chosen by the attacker, we consider a resulting set of n uniformly distributed physical addresses that can belong to 2^{color} different sets.

Random generation

In order to generate random addresses we use the method `random_setOfAddresses`

```
random_SetOfAddresses :: Int -> IO(SetState)
```

The method receives an Int n and creates a random set of addresses of n elements SetState(c, n).

In order to calculate the number of congruent addresses c, the algorithm randomly assigns a set number n times, and sums all the congruent addresses. The set number assigned is a uniform distribution of discrete values, between 0 and $2^{color + s}$

When we refer to generate a set of random addresses we refer to addresses within this range and with the same offset

6.4 Implementation of the model

The code is divided in 6 modules:

Module ***Tests.hs*** contains all the tests. 6.4

Module ***Base.hs*** contains the declaration of the data types and parameters. 6.4

Module ***AddressCreation.hs*** contains the methods for generating addresses. 6.4

Module ***Quickcheck.hs***: contains all the Quickcheck properties, tests and generators. 6.4

Module ***Cache.hs***: is the model of the cache. 6.4

Module ***Algorithms.hs***: contains the implementations of all the algorithms. 6.4

Hereunder, in the rest of the section, we go over each of them in detail.

Tests.hs

All the tests and simulations of the algorithms were performed from this module.

The following are the parameters for the tests:

`numberAddrToTest\_From :: Int`
The starting point for the size of the set used in the tests.

`numberAddrToTest\_To :: Int`
The ending point for the size of the set used in the tests.

`rangeTests :: Int`
Interval for the range of set sizes.

`iterations :: Int`
Number of iterations for the test.

`path :: String`
Path used to save the results of the tests.

For example, parameters *numberAddrToTest_From* = 0, *numberAddrToTest_To* = 4000, *rangeTests* = (2*associativity) and *iterations* = 1000 mean that there will be 1000 iterations of the test, for sets of addresses of sizes [0:2*associativity:4000].

The **main** method executes the tests, according to the parameters. It executes the test *iterations* times and computes the mean. Finally it saves in the file specified by *paths* the list with the average result of the tests for each size of set.

The following tests can be performed from the *main*:

```
test_reduction :: ReductionAlgorithm -> RepPol -> Int -> IO (Int)
```
The reduction test takes as input a reduction algorithm of the type ReductionAlgorithm, a replacement policy of the type RepPol and the size of the set of addresses.

The test generates a set of addresses of type SetState, and calls the reduction algorithm with the replacement policy and the set. Then it converts the output of the reduction to a 1 if the reduction succeeds and a 0 if it does not. The output is returned as a number to allow the main to compute the average result.

```
test_eviction :: RepPol -> Int -> IO(Int)
```
The eviction test receives as input a replacement policy of type RepPol and a number for the size of the set of addresses. It generates a set of type SetState with the given size of addresses and performs an eviction test (4.1) on the set. Then it converts the output to a 1 if the eviction is successful and a 0 if it does not.

```
count_tlb_misses :: Int -> IO(Int)
```
This last test counts the TLB misses. It receives a set size, generates a corresponding set of type SetState and sends the set to the cache. It receives and outputs the number of TLB misses.

Base.hs

This module contains the data definitions, as well as the parameters of the cache.

The data types are the following:

```
data Address = Address Int
```
Type Address, represented by an Int.

```
data SetState = SetState(Int, Int)
```
Type SetState, a tuple of Int. This data representation is explained in Section 4.0.

```
data AddressIdentifier = AddressIdentifier Int
```
Type AddressIdentifier. An address is represented as an Int to distinguish it from other congruent addresses. Explained in 4.0.

```
data Trace = Trace [AddressIdentifier]
```
Type trace. This is used to create lists of addresses that are going to be introduced in the cache. Explained in 4.0.

```
data CacheSetContent = CacheSetContent [AddressIdentifier]
```
CacheSetContent is a list of AddressIdentifier. It is of size associativity and represents the sate of a cache set. Explained in 4.0.

```
data HitNumber = Hit Int
```
HitNumber is an integer returned by the replacement policies.

The **parameters** of the cache are described below. Changing the value of these parameters changes the characteristics of the cache where the model is executed.

```
virtual_address_length :: Int
```
Number of bits of the virtual addresses.

```
physical_address_length :: Int
```
Number of bits of the physical addresses.

```
pageOffset :: Int
```
Bits of page offset.

```
cacheOffset :: Int
```
Number of bits that indicate cache offset. This means n bits for blocks of size 2^n.

```
cacheSet :: Int
```
Number of bits that indicate the cache set. This means c bits for 2^c sets.

```
slice_bits :: Int
```
Number of bits for the slicing hash. s bits indicate 2^s slices.

```
associativity :: Int
```
Associativity of the cache.

```
tlb_assoc_bits :: Int
```

Number of bits for TLB associativity. t bits indicate associativity of 2^t.

`tlb_size :: Int`
Number of entries in the TLB. It has to be a multiple of tlb_assoc_bits

AddressCreation.hs

This module contains the methods that are used to generate new data structures. It is used to create the random sets of addresses used in the tests, as well as the empty cache states and the traces of addresses.

`initialSet :: CacheSetContent`
Generates an empty CacheSetContent, a list of AddressIdentifier of size associativity where the value of all the addresses is 0.

`list_random_sets :: Int -> (IO (Address)) -> IO ([Address])`
Receives a size n and a seed and generates a list of addresses of size n.

`random_set_partial :: IO (Address)`
Generates an Address, that is, the identifier of a cache set. The Address is one in the range that can be chosen by an attacker. Therefore, the set number ranges between 0 and $2^{color} - 1$.

`random_set :: IO (Address)`
Generates an Address, that is, the identifier of a cache set. The address can belong to any cache set. The set number ranges between 0 and $2^c - 1$.

`list_random_tlb :: Int -> IO ([Int])`
Receives a size n, and generates a list of addresses of size n belonging to any set.

`new_tlb_list :: Int -> IO(SetState)`
Receives as input a size n and generates a SetState(c, n). The addresses can belong to any set, not just the *color* sets.

`random_SetOfAddresses :: Int -> IO(SetState)`
Receives as input a size n and returns a randomly generated SetState(c, n). The addresses can belong to 2^{color} different sets.

`consecutive_trace :: Int -> Trace`

Receives a number of congruent addresses c and generates a trace with all the addresses $[1 \dots c]$.

Quickcheck.hs

This module is explained in detail in Section 6.2

Cache.hs

In this module is model of the cache itself. The function of each implemented method is explained in detail in Section 5.

```
noise :: Bool
```

The parameter of the cache *noise* allows choosing whether the TLB thrashing affects the eviction or not. If the value is *False*, then the cache inserts the congruent addresses in the cache without adding extra addresses due to TLB misses. If the value is *True*, then the cache computes an estimation of TLB misses and inserts additional addresses in the trace, that may cause false positives in an eviction test.

```
tlb_misses :: [Int] -> Int
```

This function, *tlb_misses* takes a list of numbers representing a set of addresses and computes the number of TLB misses for that set, as detailed in Section 5.3

```
cacheInsert :: RepPol -> CacheSetContent -> Trace -> Int
                                                  -> IO(CacheSetContent, HitNumber)
```

cacheInsert is the main function of the model. It organizes the rest of elements and acts as the interface with the algorithms. It receives as input a replacement policy of type RepPol, the content of the cache where the addresses will be inserted as an element of type CacheSetContent, a trace of addresses to be inserted and the total number of addresses from the set. For instance, if an algorithm has a set of c congruent addresses and n total addresses that is going to be inserted in the cache, *cacheInsert* would receive a trace $[1 \dots c]$ or $[1, 1, \dots c, c]$ but the total number of different addresses (necessary in order to compute the TLB misses) is always n.

```
expected_tlb_misses :: Int -> Int
```

The function *expected_tlb_misses* receives a number of addresses and estimates the number of TLB misses caused by accessing those addresses.

```
type RepPol = CacheSetContent -> Trace -> IO(CacheSetContent, HitNumber)
```

The type *RepPol* is a synonym for the type of all the implemented replacement policies of the model. The replacement policies receive the content of the cache set as an element of type *CacheSetContent* and a Trace of addresses, and they output a tuple with the new CacheSetContent, once all the addresses have been inserted, as well as the number of hits occurred while inserting all the addresses.

The replacement policies MRU, LIP and BIP are modeled assuming that the cache set is not empty: The addresses marked as 0 are unknown, not nonexistent.

```
lru :: RepPol
```

We will observe in detail the implementation of the replacement policy LRU. The other implemented policies are listed in 5.2. They have all the same type *RepPol*.

All the replacement polices R have an auxiliary function R'.

The function *lru* receives a cache state (the state of the cache set), and a trace of addresses. The function calls *lru'* with the same arguments in addition to an empty counter for the number of hits.

```
lru :: RepPol
lru set trace = do
  (t, s, h) <- lru'(trace, set, Hit 0)
  return (s, h)
```

The auxiliary function *lru'* inserts the address from the trace one by one.

The LRU replacement policy in a real cache works with a counter for each slot of the cache. Each time a block has to be evicted, it checks the counters and finds the *least recently used* block. This block is replaced by the new block, and the counter set to 0.

This implementation uses the CacheSetContent as a queue, where the head of the list is the most recently used, and the tail the least recently used. When the function receives the arguments, it checks if the head of the trace is in the cache set. If the head is in the cache set, then it removes it from its position and puts it at the beginning of the list. The resulting state of the cache set is a permutation of the elements of the previous state, but with the inserted address as the head of the list. The number of hits is augmented in 1. If the head of the trace is not in the cache already, then the last element is deleted, as it is the oldest one, and the head of the trace is inserted as the head of the list. Finally, the function calls itself tuple with the trace without the head, the new cache set state and the number of hits. If the trace is empty, the result is returned to *lru*, which returns a tuple with the new cache set state and the number of hits.

```
lru' :: (Trace, CacheSetContent, HitNumber) -> IO(Trace, CacheSetContent, HitNumber)
lru' i@(Trace [], _, _) = do return i
```

```
lru' (Trace trace, CacheSetContent set, Hit hit) =
  case (elemIndex h set) of
    Just elem -> do
      r <- lru'(Trace (tail trace), CacheSetContent(h : (deleteN elem set)), Hit (hit + 1))
      return r
    Nothing -> do
      r <- lru'(Trace (tail trace), CacheSetContent(h : init set), Hit hit)
      return r
  where h = head trace
```

Algorithms.hs

This module contains the implementations of the algorithms adapted to the model, as well as their auxiliary functions. We now discuss the implementation of the main functions

```
evicts :: SetState -> RepPol -> IO(Bool)
```

The *evicts* function performs the eviction test explained in Section 4.1. The function receives as input a set of addresses as a SetSate, and a replacement policy with the type RepPol, and performs an eviction test with the set and the policy. It generates a trace of addresses for the congruent addresses of the set. It adds a victim address to the beginning of the trace. It then calls the function *cacheInsert*. The cache function will return the new cache set state after inserting the trace and the number of hits (which will be 0, as the function inserted a list of unique addresses into an empty cache). Then *evicts* will call *cacheInsert* again. From the tuple of new state and number of hits, the function will learn if the set of addresses was evicting or not.

```
evictschance :: SetState -> RepPol -> Int -> Int -> IO(Bool)
```

This function checks if a set of addresses is *p-evicting*. This property is explained in section 4.1. It receives the set of addresses, a replacement policies, and two Int, a and b. The function performs an eviction test, and returns True if the set of addresses was evicting at least a out of b times.

```
type ReductionAlgorithm = SetState -> RepPol -> IO(Maybe(SetState))
```

The type declaration for the reduction algorithm is a synonym for the type of all the implemented reduction algorithms. A ReductionAlgorithm takes as input a set of addresses as a SetState and a replacement policy, and performs a reduction on the set of addresses. The algorithms return Nothing if the reduction has failed, and Just R, where R is a minimal eviction set, if the reduction has succeeded.

```
  reduction_size :: Int
```

The value of *reduction_size* defines the size of the output of the reduction algorithms.

```
baseline_reduction :: ReductionAlgorithm
```

The *baseline_reduction* method is the implementation of the baseline reduction algorithm as explained in Section 4.2.

```
baseline2_reduction :: ReductionAlgorithm
```

The *baseline_reduction* method is the implementation of the second baseline reduction algorithm as explained in Section 4.3.

```
reduction :: ReductionAlgorithm
```

The *baseline_reduction* method is the implementation of the threshold group reduction algorithm as explained in Section 4.4.

Chapter 7

Results, analysis and discussion

In this Section we use the model to simulate the reduction algorithms and make a comparison between the baseline reduction 3.2 and the threshold group reduction 3.4.

The default parameters for the cache are: Physical addresses of 34 bits, 1024 cache sets ($c = 13$), lines of 64 Bytes ($l = 6$), pages of 4KB ($p = 12$), TLB of size 1536, and no slicing. These are the parameters used in the tests unless otherwise specified.

7.1 Eviction rate

The eviction rate is the relative frequency of the tests returning true on a randomly generated set of addresses of a fixed size (from [1]).

First, we analyze the eviction rates for the different replacement policies. Figure 7.1 shows the eviction rates for the implemented replacement policies.

The eviction test is important because both reduction algorithms use this test in order to know if a set is evicting. The reduction rate will be heavily influenced by the eviction rate.

The eviction rate for LRU (Figure 7.1a) follows a cumulative binomial distribution curve. As soon as there are *associativity* congruent addresses the eviction test succeeds. The likelihood of generating a set with *associativity* congruent addresses increases with the size of the set. The eviction rate for FIFO (not shown) is the same as the eviction rate for LRU because the two replacement policies behave in the same way with traces of sequential non-repeated addresses.

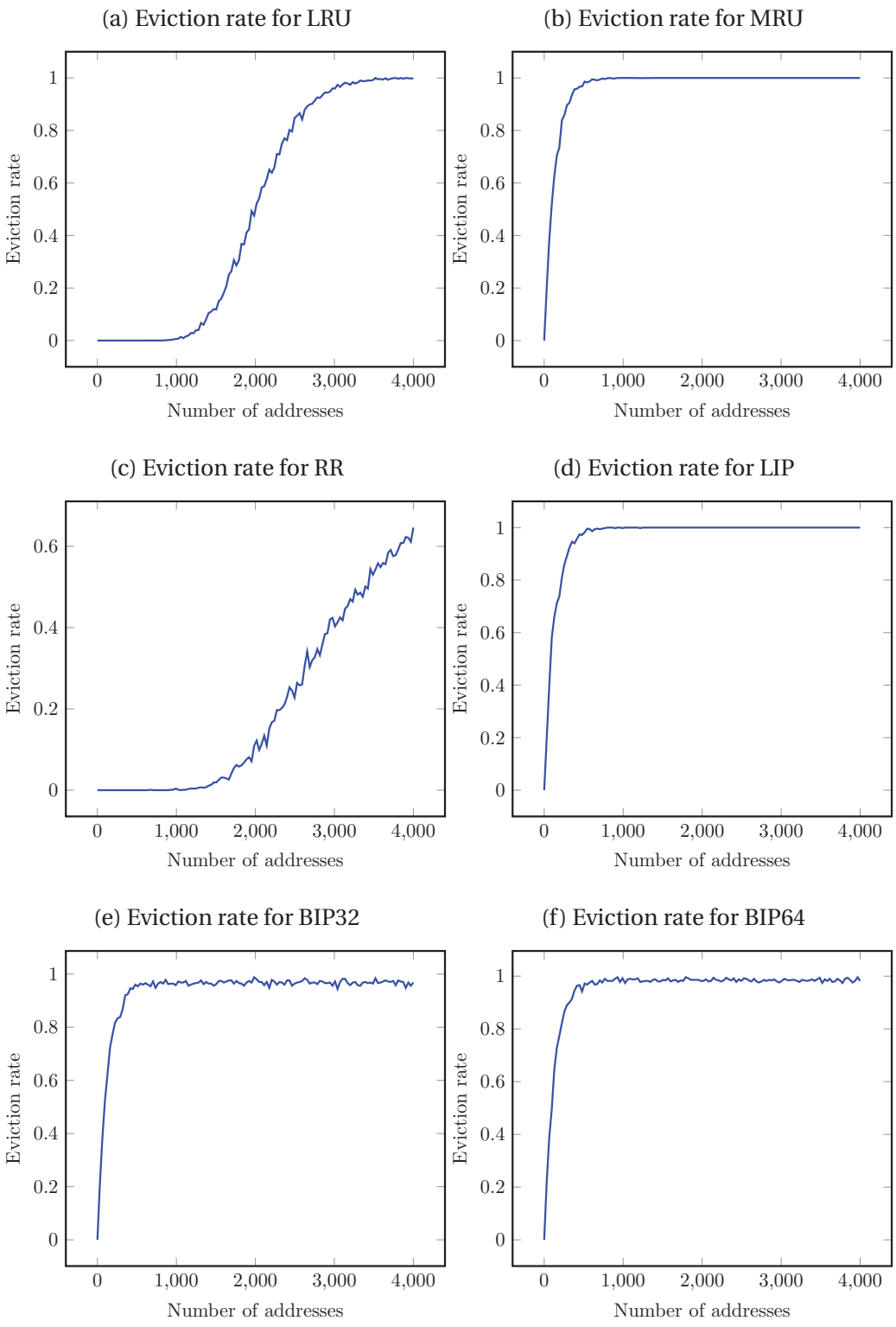

Figure 7.1: Eviction rates for different replacement policies. Average of 1000 iterations.

The same happens with the eviction rate of MRU (Figure 7.1b) and LIP (Figure

7.1d). They present the same behaviour under this pattern of trace. For this two replacement policies, one congruent address is enough for eviction. Therefore, as soon as the set of addresses has one congruent address, the test succeeds.

The eviction rate of BIP is similar to the eviction rate of LIP, as it is a variation of it. It introduces the addresses in the most-recently-used position, but it has a chance of putting the address in the least-recently-used position, i.e. the head of the list, in this case. Because of this, there is a possibility for the victim to be inserted in the least-recently-used position of the cache set. If that happens, the rest of the congruent addresses will not be able to evict it unless enough addresses are put in the least-recently used position, and the victim returns to the most recently used position again, where it could be evicted. This is an unlikely situation, so an evicting cache set has a certain possibility to not be evicting with this replacement policy.

We tested two variations of BIP: BIP with 1 possibility over 32 of introducing addresses in the least recently used position (BIP32), and BIP with 1 over 64 possibilities of introducing addresses in the least recently used position (BIP64), the two recommended variations for this policy [16].

The eviction rate for BIP 32 (Figure 7.1e) and BIP 64 (Figure 7.1f) are very similar, but the eviction rate of BIP64 is predictably closer to LIP, as it is more likely to behave is the same way as it. For most of the rest of the tests we will use the variation BIP32, that we will call BIP, as BIP32 is a worst-case version of BIP64, which is going to be a version closer to LIP.

The eviction rate of RR 7.1c shows that the probability of evicting increases with the probability of having more congruent addresses in the set. As this replacement policy randomly evicts an element of the cache set, the more congruent addresses, the more likely it is to evict the victim from the cache.

We tested eviction with RR in sets with exactly associativity congruent addresses (eviction sets). The average eviction rate for 10000 iterations was $6.01e-2$. This means that in RR an eviction test is not always evicting, but it has a 60.1 % chance of being evicting. An eviction set in RR is 60-evicting.

Generally, the eviction rate for LRU is an indicator of the chance for a set of a certain size to be an *eviction set*. The eviction rate for each policy indicates the probability of this set to be an *evicting set*.

7.2 Reduction

We now compare the two reduction algorithms. We simulate both algorithm with and without TLB thrashing, for all the different replacement policies tested above. We also test some modifications and compare the results.

We define reduction rate as the relative frequency of the success of a reduction test on a randomly generated set of addresses of a fixed size. A test is successful if the output is a set of r congruent addresses, where r is the fixed output size of the reduction.

In the reduction tests, the defined size for the output of the reduction is *associativity* unless otherwise specified.

7.2.1 Baseline reduction

We first simulate the first baseline reduction (from Section 3.2) for all the replacement policies.

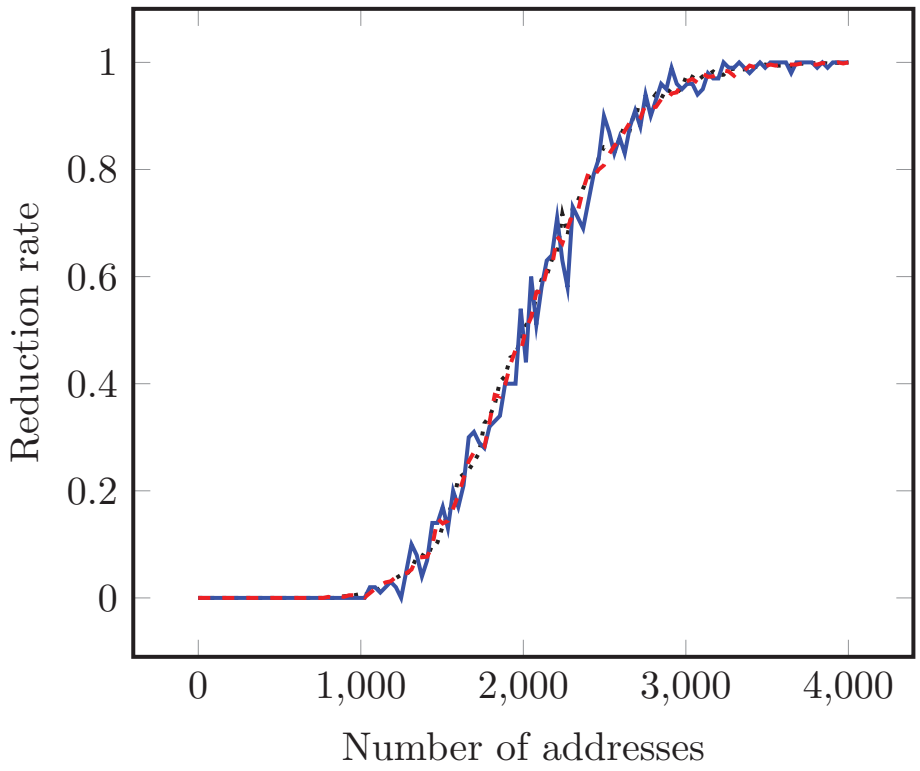

Figure 7.2: Reduction rate for LRU with the baseline reduction algorithm. Average of 500 iterations.

The results of performing the baseline reduction on LRU can be seen in Figure 7.2. The eviction rate is the theoretical prediction for the reduction rate. The reduction rate for LRU is the same as the eviction rate, which means that every time that the set of addresses was evicting, the reduction succeeded to make it minimal. The introduction of TLB thrashing introduced some noise, but did not make a significant difference.

The reduction rate of FIFO is the same than the eviction rate of LRU. As for the rest of the policies, MRU, RR, LIP and BIP, the eviction rate was 0 for all of them, for all sets of addresses.

The cause of this failure of the reduction rate for RR is that a reduction algorithm performs multiple times the eviction test, with increasingly small sets of addresses. As the RR policy randomly selects which address to evict upon a cache miss, a same set can be evicting in an eviction test and not evicting in an other. Hence it is unlikely for the reduction algorithm to achieve a set of exactly associativity addresses.

As an alternative, we tested the RR policy again, but instead of testing for *eviction* during the reduction, the algorithm tested for *80-eviction*. That is, a p-eviction property, where the set of addresses was tested to check if it is evicting 80 % of the time. This did not work. The result of this test was equal to the previous one.

The other three policies, MRU, LIP and BIP, fail all for the same reason: One congruent address is enough to evict a victim address from the cache. Therefore, the algorithm cannot find exactly associativity congruent addresses, as one single congruent address will make the whole set of addresses evicting. During the reduction, addresses are eliminated one by one, until an address is taken which makes the set no longer evicting. After, all the addresses that make the set no longer evicting are inserted into an eviction set. With these policies, all the addresses will be removed from the set until only one congruent address is left, and after that it is not longer possible to construct an eviction set.

We test the baseline reduction for LIP and BIP (MRU in this case behaves like LIP, evicting the last address inserted) with the baseline reduction algorithm, but changing the fixed output size: the size of the final set of addresses is fixed to 1 instead of associativity (which is 16 in this case).

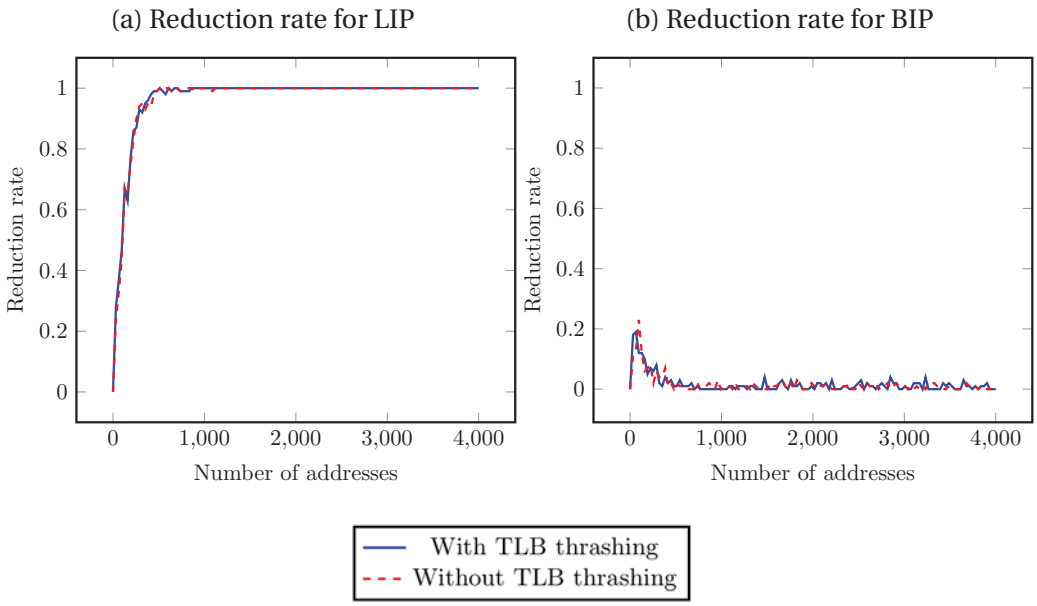

Figure 7.3: Reduction rates for different replacement policies. Average of 500 iterations.

Figure 7.3 shows the result of this experiment.

With the new size for the resulting eviction set, the reduction rate for LIP improves drastically (Figure 7.3a), making it reach its eviction rate. TLB thrashing does not affect the reduction.

Meanwhile, the reduction rate for BIP (Figure 7.3b) increases for small sets of addresses but is close to 0 after that.

We test the baseline reduction again for BIP, changing the test performed by the algorithm. In the previous tests, the algorithm tested the set of addresses for *eviction* in order to make the reduction. We now change this property to *80-eviction*, so for a set of addresses to be considered valid, it must be evicting 80% of the time.

The result (Figure 7.4) shows that the eviction rate has improved but is still substantially lower than the eviction rate. More tests of p-eviction with different p values could reveal if this property can make the reduction rate closer to the eviction rate.

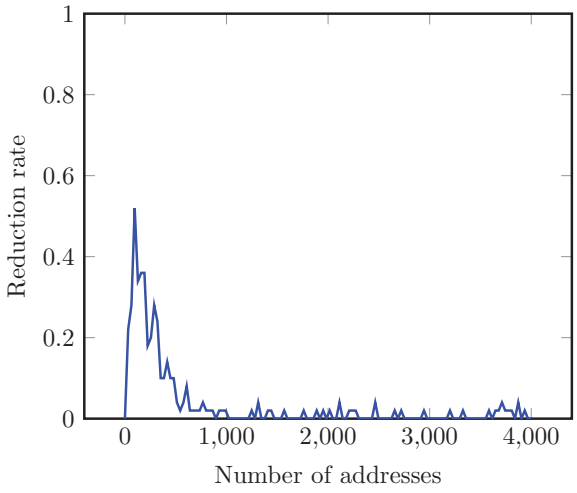

Figure 7.4: Reduction rate for BIP with the threshold group 1-reduction algorithm. Average of 100 iterations.

7.2.2 Threshold group reduction

We simulate the threshold group reduction for all the replacement policies:

Figure 7.5 shows the result of the threshold group reduction for LRU.

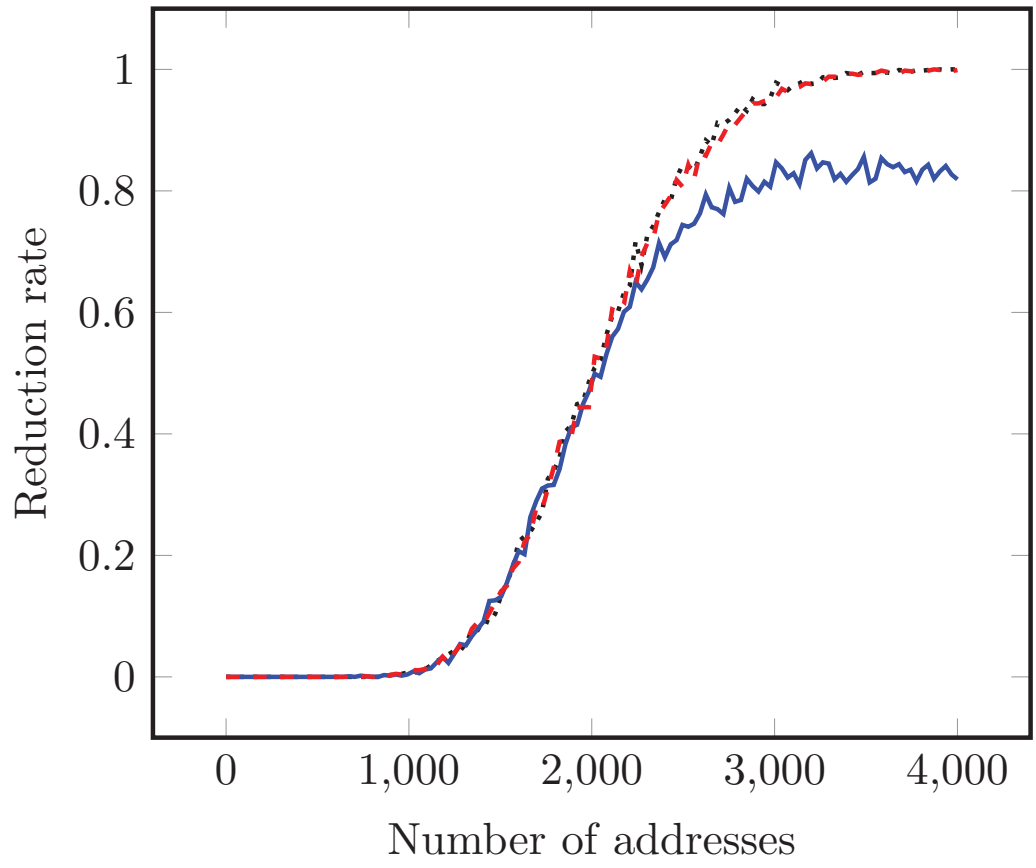

Figure 7.5: Reduction rate for LRU with the threshold group reduction algorithm. Average of 1000 iterations.

The reduction without TLB thrashing is close to the upper-bound of the eviction rate. The noise created by the TLB thrashing does have an effect, reducing the maximum reduction rate. TLB misses cause page walks that introduce noise in the cache. This may cause false positives. During the reduction, the algorithms might get a false positive when testing a subset of the initial set for eviction. A false positive during the reduction can cause the algorithm to take the wrong subset, find and in the following iteration that there are not enough congruent addresses.

The reduction rate for FIFO is the same as the reduction rate for LRU in this case.

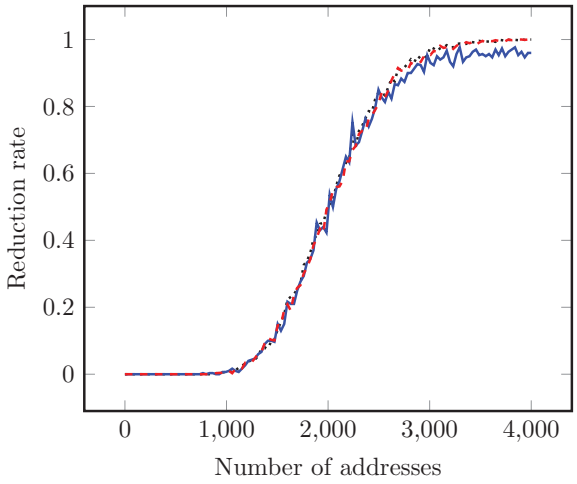

Figure 7.6: Reduction rate for LRU with the threshold group reduction algorithm, with TLB thrashing, testing for 80-eviction. Average of 200 iterations.

We test the reduction algorithm again for LRU with TLB thrashing, but instead of testing eviction in order to decide if a partition has to be discarded, we test for 80-eviction, that is, a set must be evicting 80% of the time.

Figure 7.6 shows the result for LRU (and FIFO).

With the property, the TLB thrashing does not affect the reduction anymore, and the reduction rate raises to the eviction rate. This is because the property prevents false positives generated by TLB misses.

The reduction for RR fails in all the tests. Testing for 80-eviction and 60-eviction returns the same results.

The reduction for MRU, LIP and BIP is 0 for all the sizes of sets. The reason is the same as in the previous reduction: the algorithms is only successful if the output is a set of congruent addresses of size associativity, but this is only possible if the replacement policy is not evicting with less than associativity addresses.

We test the threshold group 1-reduction, where the size of the output is 1, for MRU, LIP and BIP. The results can be seen in Figure 7.7. The reduction rate for MRU is the same as LIP. With this 1-reduction, the reduction rates are equal to the eviction rates.

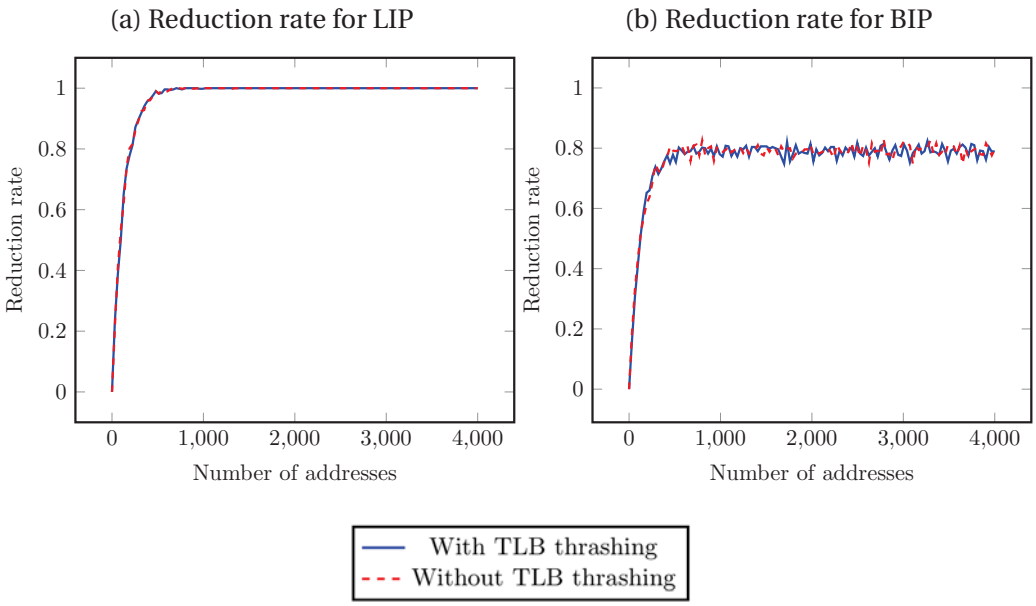

Figure 7.7: Reduction rates for different replacement policies. Average of 500 iterations.

7.2.3 Tests with different cache parameters

Test with heavy TLB thrashing

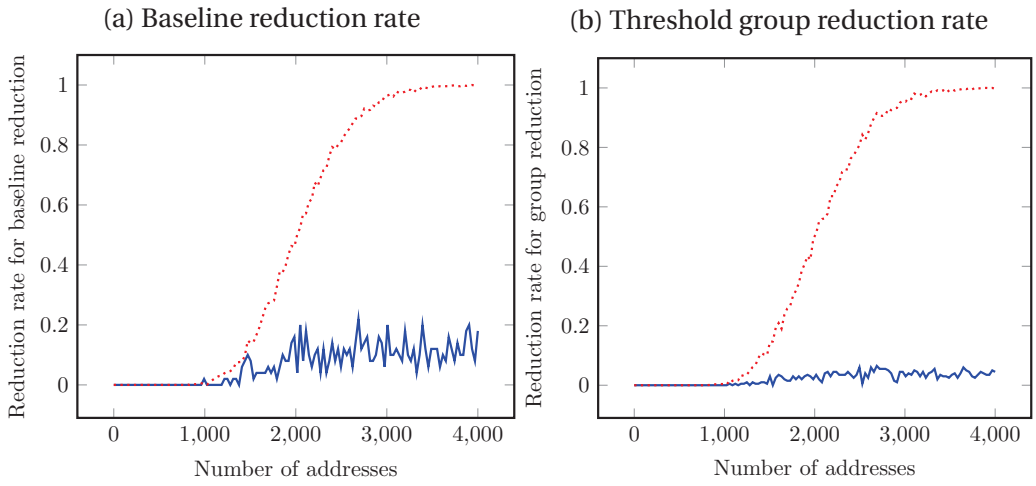

Figure 7.8: Reduction rates for LRU, with TLB size = 0. Average of 200 iterations.

We test the reduction rate with TLB thrashing, for a cache where the size of the TLB is 0. This means that there are as many TLB misses as addresses are accessed: for a set of size n, there are n TLB misses.

The results can be seen in Figure 7.8. Under this circumstance, the baseline re-

duction algorithm is still more resistant to the effect of the TLB, but the reduction rates decrease significantly for both algorithms.

This can be compared to the effect of the TLB in Figures 7.2 and 7.5.

Different number of *color* bits

The number of *color* bits for the previous tests is 7. We test the eviction rate for different values of *color*.

We test the eviction rate for LRU for 4, 5, 6 and 8 *color* bits.

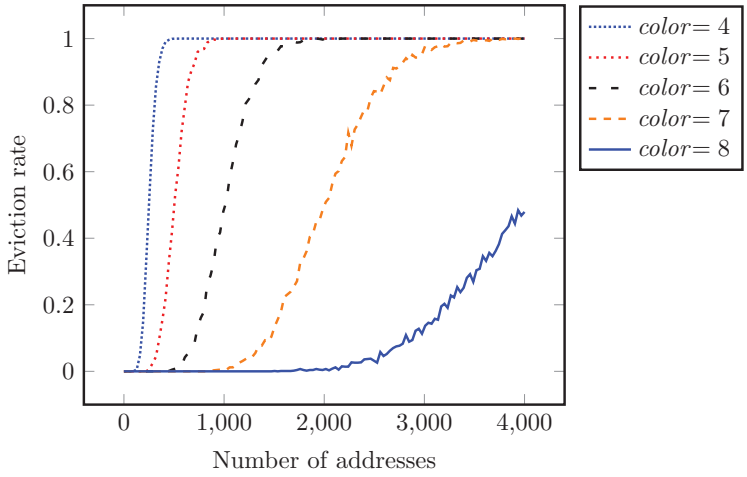

Figure 7.9: Eviction rate for different *color* bits with LRU. Average of 500 iterations.

We see in Figure 7.9 how the number of *color* bits affect the eviction rate. As the number of bits increase, the eviction rate decreases. This is because the number of cache sets is 2^{color}. As this number increases, there are more possible cache sets, and the probability of having at least *associativity* congruent addresses decreases.

We test with *color* = 10. For this value, the eviction rate is 0 for all sets.

Different associativity

We test with *associativity* 8 and 32. Figure 7.10 shows the eviction rates for associativity of 8, 16 (the value used in the rest of the tests) and 32. The eviction rate changes with the probability of having at least *associativity* congruent addresses. When the associativity is smaller, the probability of randomly generating a set with *associativity* congruent addresses increases.

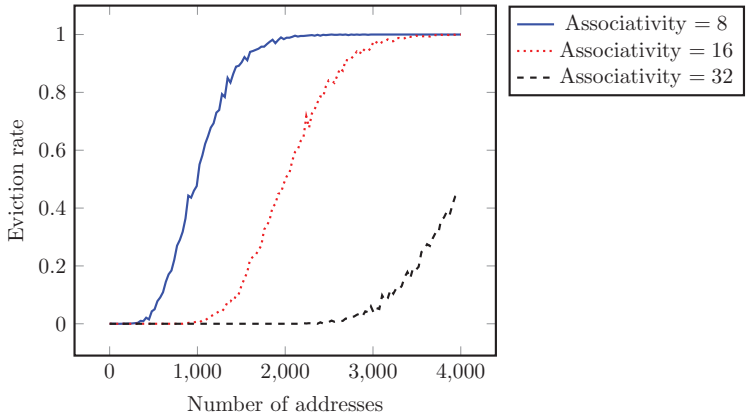

Figure 7.10: Eviction rate for different *color* bits. Average of 500 iterations.

7.3 Discussion

The tests show the strengths and weaknesses of each reduction algorithm.

First, on the reduction to a set of size associativity, both reductions fail with RR, MRU, LIP and BIP. For LRU and FIFO, the reduction rate without TLB thrashing is similar, but the baseline algorithm has shown to be more resistant to the noise caused by the TLB than the threshold group reduction algorithm. This can be seen in Figures 7.2 and 7.5.

Regarding LIP, changing the size of the output of the reduction makes LIP replacement's 1-reduction rate raise to the eviction rate for both algorithms, but the case is not the same for BIP. As the baseline algorithm performs significantly more accesses to addresses than the group reduction, BIP policy's 1-reduction rate fails still considerably. Meanwhile, the 1-reduction rate of BIP for the group reduction is equivalent to the upper bound of the eviction rate, Figures 7.3 and 7.7.. It is important to note that these thrashing-resistant policies behave like LRU if a different eviction strategy is used, like accessing each address twice.

The conclusion of these tests is that the baseline reduction is more robust against false positives, that can make the group reduction fail. The group reduction is more robust for some cases of probabilistic replacement policies, as it performs fewer accesses that the baseline reduction.

The group reduction algorithm is not as robust against TLB thrashing, but during the reduction, replacing the eviction test for a 80-evicting test makes the algorithm as robust as the baseline algorithm (Figure 7.6). Given the difference in complexity between both algorithms, the group reduction with the property is still significantly faster than the baseline reduction.

The replacement policies have a clear effect on the result of the simulations. The noise created by the probabilistic replacement policies (except the more extreme example of RR) can be mitigated, at least for some cases, by replacing the eviction test by a p-eviction test. This property can also be used to lessen the effect of the TLB.

The experiments also show that a very small probability of failure (for example introduced by a probabilistic replacement policy) can lead to a total break down of the reduction algorithm, as they repeat this test a high number of times.

Chapter 8

Conclusion

We have introduced a Haskell model of the relevant aspects of the microarchitecture of computers, relevant to finding and reducing eviction sets.

We have adapted an eviction test and three reduction algorithms to the model, and we have performed an analysis of the results of simulating two reduction algorithms in caches with different characteristics.

Our results highlight strengts and weaknesses of both algorithms, as well as the effect that different parameters of the cache have on those reduction algorithms.

The results obtained with the model seem to be comparable to the results obtained in real tests over hardware. Tests for the simulated cache can be observed in [1]. They have been useful to understand the results of real hardware experiments, to understand the effect of the TLB and the probabilistic and adaptive replacement policies.

Next steps include adding new replacement policies to the model, especially adaptive replacement policies. They also include performing more tests, with different cache parameters, including tests with slicing, and with traces of addresses with different patterns.

Bibliography

[1] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," 2018. Paper under submission.

[2] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 60–71, ACM, 2010.

[3] H. Wong, "Intel ivy bridge cache replacement policy," *Retrieved on July*, vol. 16, p. 149, 2015.

[4] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, p. 4, 2015.

[5] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 605–622, IEEE, 2015.

[6] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*, pp. 104–113, Springer, 1996.

[7] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual International Cryptology Conference*, pp. 388–397, Springer, 1999.

[8] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 251–261, Springer, 2001.

[9] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*, pp. 225–242, Springer, 2007.

[10] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers.," in *USENIX Security symposium*, pp. 307–322, 2010.

[11] P. Gutmann, "Data remanence in semiconductor devices," in *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, p. 4, USENIX Association, 2001.

[12] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack.," in *USENIX Security Symposium*, pp. 719–732, 2014.

[13] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices.," in *USENIX Security Symposium*, pp. 549–564, 2016.

[14] E. S. Tam, J. A. Rivers, G. S. Tyson, and E. S. Davidson, "mlcache: A flexible multi-lateral cache simulator," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1998. Proceedings. Sixth International Symposium on*, pp. 19–26, IEEE, 1998.

[15] J. Edler, "Dinero iv trace-driven uniprocessor cache simulator," *http://www. cs. wisc. edu/markhill/DineroIV/*, 1998.

[16] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 381–391, ACM, 2007.

[17] A. Abel and J. Reineke, "Reverse engineering of cache replacement policies in intel microprocessors and their evaluation," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 141–142, IEEE, 2014.

[18] C. Percival, "Cache missing for fun and profit," 2005.

[19] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' Track at the RSA Conference*, pp. 1–20, Springer, 2006.

[20] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 490–505, IEEE, 2011.

[21] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S $ a: a shared cache attack that works across cores and defies vm sandboxing–and its application to aes," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 591–604, IEEE, 2015.

[22] R. Bird *et al.*, *Introduction to functional programming using Haskell*, vol. 2. Prentice Hall Europe Hemel Hempstead, UK, 1998.

[23] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.

[24] "Quickcheck: Automatic testing of haskell programs." `http://hackage.haskell.org/package/QuickCheck`.

[25] "Introduction to quickcheck." `https://wiki.haskell.org/Introduction_to_QuickCheck1`.

[26] “Test.quickcheck haskell package.” http://hackage.haskell.org/package/QuickCheck-2.11.3/docs/Test-QuickCheck.html.