Quickcheck: Property testing in Haskell

Paloma Pedregal Helft

Contents

1 Introduction

This document is the report for the final proyect of the course *Analysis of Concurrent Systems*. The proyect's objective was to test a program using **Quickcheck**, a library for property testing.

The first part of the proyect was writiting a Haskell program. In this case, the program is a model of computer microarchitecture for studying different aspects of side-channel cache attacks.

After writing the program, the second step was listing the properties of the program, and the third implementing them in Quickcheck and testing the program.

This document consists of a first section with a background on caches, side channel attacks and the implementation of the program, a second with the properties and implementation of the tests, and finally a section with the results of the tests and a conclusion.

2 A cache model in Haskell

The program on which the tests were performed is a model of some parts of the microarchitecture of computers: Caches, different replacement policies, translation lookaside buffer (TLB). The model's objective is to study and reason about aspects of side-channel cache attacks.

2.1 A primer on caches

Caches are very small and very fast memories that work as buffer between the slower memory of a computer and the very fast processor. Data 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 very expensive timewise.

Caches are partitioned in equally sized *cache sets*. The number of lines in each cache set is called *associativity*.

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 has to be a *cache replacement policy*.

The most common are least-recently used (**LRU**), first-in first-out (**FIFO**), or pseudo-lru (**PLRU**).

Nowadays, replacement policies are increasingly complex, usually undocumented, and sometimes variable depending on the cache set [5, 7].

Where a memory block is mapped to a certain region of the cache because of the **physical address** of its contents: 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.

The translation between virtual and physical addresses is performed by the *Memory Management Unit* (*MMU*). Within this MMU, there is a small

cache for address translation called the *Translation Lookaside Buffer*, the **TLB**. When a virtual address is accessed, the MMU has to find the translation. If it is not in the TLB, then there is a TLB miss, and the MMU has to do a *page walk* to fetch the new address. This process causes some blocks to be introduced in the cache.

2.2 A primer on side-channel cache attacks

There are a series of effective attacks that exploit shared CPU caches [6, 8]

A very simplified idea of an attack would be the following:

- The adversary fills a cache with their own data
- A victim that shares the same cache access a secret
- The attacker accesses their data again, measuring the access times. If the access is slow, then there has been a cache miss. This means that the victim has accessed that part of the cache. A fast access implies that the data is still in the cache and therefore has not been evicted by the victim

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**. In this case, we are defining an eviction set as a set of virtual addresses, of size at least associativity, and that are **congruent**. Congruent addresses are addresses that are mapped to the same cache set.

Associativity is the size needed, because that is the number of blocks that can be stored in each single cache 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.

One of the main goals of the Haskell model is to study different *reduction algorithms*. These are algorithms that take a large set of addresses and reduce the set to a *minimal eviction set*, an eviction set of size associativity.

2.3 A Haskell microachitecture model

The goal of the model is to study different security aspects of caches.

The sets of addresses are represented in two different ways:

- For distribution of addresses across multiple cache sets, addresses are represented with the type `Address`. Each of those is an Int that represent the cache set number of the address.

  A set of addresses is represented as a list of Addresses: `[Address]`

- When there is only one objective set, like in the reduction algorithm, the only information needed is the total number of addresses in the set, and how many of them are congruent with the objective set.

  The representation of those set is done with the type `CacheState` that represents a tuple of Ints for the number of congruent addresses and the total number of addresses: `CacheState(Congruent, Total)`

There are generators of random sets of addresses of a certain length as well as of CacheStates.

There are two different implementation of reductions, as well as different replacement policies and the TLB.

3 Property testing

Quickcheck for Haskell is a library for random testing properties of Haskell programs [4, 2, 1, 3]

3.1 Defining the properties of the program

In order to test the program, it is first necessary to define its properties:

- 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 CacheState with n addresses are of the form `CacheState(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 `CacheState(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
- The group reduction ends with exactly associativity addresses
- The group reduction ends with associativity congruent addresses
- The linear reduction suceeds when there are associativity many addresses or more
- The linear reduction fails when there are less than associativity addresses
- The linear reduction ends with exactly associativity addresses
- The linear reduction ends with associativity congruent addresses
- 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
- An eviction test with associativity or more addresses suceeds in LRU
- An eviction test executed with a trace of only two identical addresses is evicting for any replacement policy

3.2 Implementation of the tests

3.2.1 Generators

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

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

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

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

3.2.2 Properties

The properties described above were tested with different functions. Some of them are here, the rest are in the file `Quickcheck.hs`.

The following property tests that the generator of addresses returns a set of address 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 CacheState 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
  CacheState(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 (CacheState) 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 CacheState.

```
prop_evicts :: CacheState -> Property
prop_evicts cacheState@(CacheState(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 :: CacheState -> Property
prop_group_reduction_bool cacheState@(CacheState(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 CacheState(associativity, associativity) if it succeeds or else its the input.

```
prop_group_reduction :: CacheState -> Property
prop_group_reduction cacheState@(CacheState(congr, total)) = monadicIO $ do
  CacheState(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

.
.
.
(etc)
```

4 Results and conclusion

The properties were a very good way of not only testing the already implemented program but also an interesting way of reasoning about the implementation.

There was a mistake in the code, found thanks to one of the tests. Sometimes, the group reduction finished with a CacheSet that had associativity congruent elements but more total elements.

That would make one of the tests fail with the message `*** Failed! Assertion failed (after 32 test): CacheState(16, 17)`.

This was corrected, and at this moment, the implementation of the program succeeds on all the tests.

Quickcheck has definitely proven its usefulness, and testing properties of a program is a much better solution than other kinds of testing, like unit testing. Its easy to use, and defining the properties of a program, independently of the actual testing, is a way of reasoning about what the implementation is doing vs. how it should be.

References

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

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

[3] Test.quickcheck haskell package. `http://hackage.haskell.org/package/QuickCheck-2.11.3/docs/Test-QuickCheck.html`.

[4] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[5] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

[6] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[7] Henry Wong. Intel ivy bridge cache replacement policy. *Retrieved on July*, 16:149, 2015.

[8] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014.