UPPSALA
UNIVERSITET

Cache Pirating

Understanding Resource Contention

Ragnar Hägg

Project in Computational Science: Report

March 2014

PROJECT REPORT

Institutionen för informationsteknologi

Abstract

Shared cache contention can significantly change the performance of co-running applications on a multicore processor. There are methods to predict the impact of this contention, but in order to use them the applications need to be profiled. One common performance metric used to characterize the application is the fetch ratio as a function of the amount of cache available. A method for obtaining such information is Cache Pirating, which steals a known amount of cache from the target application while various performance events are sampled with hardware performance counters. Cache Pirating is a low-overhead method for measuring performance at runtime on real systems. Earlier implementations of Cache Pirating have been limited to a specific architecture, and the main aim of this project is to make a more portable and easy-to-use implementation of the Pirate. This has been achieved by using `libpfm4` for easier hardware counter handling and `Google Protobuf` for better data management. The Pirate also automatically discovers needed system hardware parameter from the Linux OS. The Pirate needs to self validate that it steals the right amount of cache. Prior implementations have been limited to a small number of microarchitectures. We introduce a new self-validation method that is portable to most modern architectures. The Pirate has been validated on Intel Nehalem, Westmere and Sandy Bridge, and also tested on an AMD machine.

Contents

1 Introduction

According to Moore's law, the number of transistors (and in a simplified way, the number of calculations per second) on a CPU chip doubles every 18 months. This has basically been true since 1965 when it was observed by Gordon E. Moore, co-founder of Intel. When the single core performance started to level off in 2007 we started putting more cores on the chips, and hence Moore's law was still preserved. In other words the number of calculations per second are roughly doubling every 1.5 years.

On the other side, the bandwidth from memory to the CPU is not increasing at the same rate and access to data in main memory becomes an increasing bottleneck. The access time to the memory is also very high relative to the calculation time needed on a single piece of data. To deal with this increasing gap between calculation power and memory bandwidth we have cache memory.

Cache performance is one of the most important issues in modern computers. Since the use of multicore CPUs is a common practice today, it is important to understand how sharing resources between cores affect performance. This can help us make our programs, and the hardware on which the programs run, more effective when co-running with shared resources. For example, Sandberg *et. al* have proposed methods to predict performance variability due to resource sharing effects imposed by co-running applications [1, 2]. For these methods per-application profile data need to be acquired, such as miss rates and hit rates as a function of cache size. Such profiles can be generated through simulations, but these are very time consuming and simulators are difficult to build for modern processors and memory systems.

To acquire performance profiles several techniques have been proposed by the UART team at Uppsala University. StatCache proposed by Berg and Hagersten [3] and StatStack proposed by Eklov and Hagersten [4] are statistical models to model miss ratios of fully-associative random and LRU cache hierarchies from sparse runtime sampling. These are theoretical models based on available knowledge on how the system behaves.

Eklov *et. al* proposed Cache Pirating [5] as a way of measuring performance metrics on a target application as a function of the available shared cache on real hardware at runtime. This is done by having the Pirate co-run with the target while stealing cache without affecting the target noticeably in any other aspects.

Another method also proposed by Eklov *et. al* is the Bandwidth Bandit [6] which uses the same technique but steals memory bandwidth instead. This is hard because we need to know in detail how the memory in the system is configured.

The purpose of this project is to enhance the previous implementation of the Cache Pirate. The previous Cache Pirate implementation is limited to Nehalem machines, and it is also designed for an obsolete performance counter interface. The focus for this project is making the Pirate more portable to different architectures and easier to use. The main changes done to the Cache Pirate are:

- Use of the pfm4 library to encode performance counter information and control performance counters using the native Linux kernel interfaces.

- Use of CPI instead of fetch ratio to validate that the Pirate is stealing the desired amount of cache. This makes it more portable since many architectures lack the performance counters for measuring core-specific fetch ratios.

- Use of the Google Protobuf library for simple and efficient data handling, which enables meta data to be included in the output file. The data can easily be exported to C++, Java and Python.

- The Pirate automatically discovers its runtime parameters from the `sysfs` filesystem on Linux.

2 Background

2.1 Cache memory

Cache memory is a fast memory that is placed on the CPU-chip close to the core. This memory reduces the average time needed to access data in the main memory when data is reused. It also helps to reduce the off-chip bandwidth when data is reused or shared between different cores on the same chip. This is done by storing a copy of the data in the cache.

To make data handling more efficient, data is split into cachelines, each handled as one data unit. This increases the performance by exploiting spatial locality, which helps under the assumption that if a piece of data is used, the data next to it is very likely to be used soon.

In a similar way data in the memory is also split into pages, with the most common size being 4 kB. This is the smallest memory unit the operating system can allocate for a program. Pages are important for translating virtual memory to physical memory. Virtual memory is used in order for programs to be able to use the memory independently of what other programs are running at the same time. A table with one entry for each active page is used to map virtual to physical memory address.

Usually cache memory can be made faster the smaller it is, so the size of the cache is a trade-off between latency and size. To be able to get a bit of both worlds CPUs usually have a multi-level cache-hierarchy. Most commonly a two- or three-level cache hierarchy is used. Figure 1 shows the setup of a two-core CPU with a two-level cache hierarchy. There is a small but fast L1 (Level 1) cache closest to the core. It is then connected to the bigger and shared L2 cache. The L2 cache is slower than the L1 cache but still provides data much faster than the main memory, and it helps performance by holding data that is more sparsely reused or used by both cores. With this setup we get the size of the L2 cache at the same time as we increase average performance by keeping frequently used data close to the cores in the faster L1 caches. The cost for this is physical space on the CPU and extra power for the caches.

2.1.1 Associativity

A simple type of cache is a *direct mapped* one. It uses part of a memory address as an index to decide where in the cache to put the cacheline. The rest of the memory address is stored in the cache as a tag, along with the data, so that

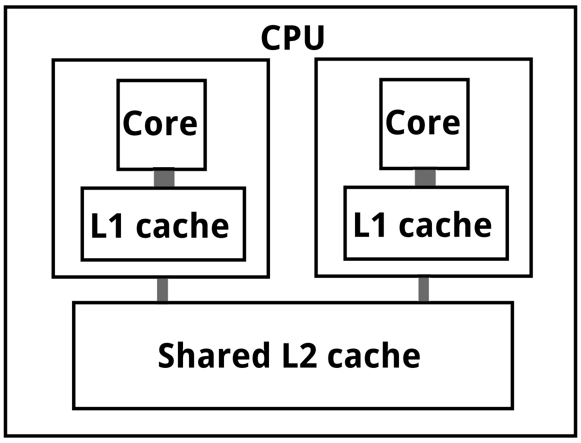


Figure 1: Setup of a CPU with two cores and a 2-level cache-hierarchy.

we can check if the cacheline we are looking for is present in the cache. Each address maps to exactly one cache-slot. So if a cache has 16 cache-slots, then four bits of the memory address will be used as index into the cache and every 16th cacheline will compete for the same slot. The least significant bits are used for the index, since these change more frequently, so that neighbouring cachelines do not compete for the same slot.

If two cachelines that index into the same slot are both reused frequently these will conflict and cause evictions. This is known as conflict misses. To deal with this contention the cache-slots can be divided into sets of two or more *ways*, where each *way* is a choice of slot in which the cacheline can be placed. The address index is then used to decide which set a cacheline goes into, and it can then be placed in either slot in the set. So if we have two slots in each set, called a *two-way set-associative* cache, we can have two frequently reused cachelines that index to the same set without any contention. The *way size* of a cache is the total cache size divided by the number of ways.

The number of ways in a cache, called the associativity, is a trade-off between performance and power consumption. The more ways the cache has, the fewer conflict misses there will be since there will be less chance that two more frequently used cachelines have to compete for the same cache-slot. But the cache has to be searched every time a cacheline is accessed and the more ways there are in a set the more address tags has to be compared with the address tag of the accessed data. Each compare costs power and also time if the compares are not done in parallel.

If the associativity of the cache is increased (while keeping size constant) until the cache has only one set, then this is called a *fully associative* cache. Here any cacheline can be in any cache-slot.

2.1.2 Replacement policies

Each time a new cache-line is brought into the cache a cache-line already in the cache has to be overwritten. In a direct mapped cache this is easy since the cache-line can only go in one slot decided by the index bits in the memory address. As soon as there is associativity in the cache there is a need for a replacement policy, a way to decide which cache-line to overwrite when bringing in new a new one.

The purpose of the replacement policy is to predict which cacheline has the lowest probability of being reused in the near future. Two simple replacement policies are *random* replacement where a random cacheline is chosen to be evicted and *least-recently used* (*LRU*) replacement which replaces the least-recently accessed cacheline. Keeping track of which cache entry was least-recently used can be very costly and there exist several *pseudo-LRU* replacement protocols that approximate a true LRU protocol. An example of this is the *pseudo-LRU* protocol used in the Intel Nehalem architecture. For every entry in the cache it has an *accessed bit*, which is set when the entry is accessed. When data needs to be evicted the accessed bits are searched and the first entry with an *unset* bit is chosen for eviction. There will always be at least one unset bit. When the last unset *accessed bit* is set all other are *unset.*

2.2 Hardware performance counters

Hardware performance counters are special-purpose registers built into the CPU that store the count of hardware-related events in the computer. They are originally built into the processor by the manufacturer to enable easier debugging and performance evaluation. For example, a new branch-predictor can be evaluated by counting the ratio of branch-prediction misses in an application and compare this to a previous branch-predictor. Nowadays the performance counters are also used for low-level performance analysis and tuning of applications. The set of available performance counters vary with different architectures, and are documented in the manuals, provided by the manufacturers.

2.3 Understanding cache performance

There are some common metrics used when analysing cache performance:

- *Miss ratio*: This is the proportion of the total data accesses that result in a cache miss, i.e., the data accessed is not in the cache and will have to be fetched from memory when the access is executed.
- *Fetch ratio*: This is the proportion of the total data accesses that result in a memory fetch, i.e., the data has to be fetched from memory. This includes fetches done by the prefetchers[1]. Is always larger or equal to the miss ratio.
- *Cycles per instruction (CPI)*: The average amount of cycles required for the core to execute an instruction. Depends on for example pipeline efficiency, memory access time and cache miss rate.

If the cache size is increased both the miss an fetch ratio should decrease in most applications since more data can be cached hence more data can be reused. Increasing the associativity of the cache also decreases miss and fetch ratio since there will be less conflict misses. Decreasing the miss ratio will most probably also decrease the CPI since the average access time to data will be lower. On the other hand, increasing both size and associativity will increase the power consumption as mentioned before, so this is the trade-off.

[1]A piece of built-in hardware that predicts what data will be used next and prefetches it to the cache.

2.4 Cache Pirating

Cache Pirating was proposed by Eklov *et. al* [5] as a method to analyse cache performance of a target application. Cache Pirating accomplishes this by "stealing" (occupying) a given amount of the last-level cache (LLC) from the target while co-running the Pirate and the target on different cores that share LLC. This should be done without having any other substantial impact on the target than the amount of shared cache reduced. By doing this we can use hardware performance counters to measure how different performance metrics for the target depend on the amount of shared cache space available. These tests are done on real hardware and therefore account for all effects of the memory hierarchy while it has a low overhead and saves a lot of time compared to running the system in a complex simulator.

The Pirate in no way restricts the target's access to the part of the cache that it is stealing. The target and Pirate are *competing* for the shared cache. The Pirate has to keep its dataset "hot" to make sure it does not get evicted. In an LRU cache this is done by accessing its cache-lines often enough so that they are never the least-recently used ones. As long as the Pirate's data never gets evicted we know that it is stealing all the cache it is supposed to.

Occasional loss of cache-lines for the Pirate can be acceptable since the impact of this will be marginal. If the Pirate will have to fetch data from memory too often this will start affecting the target in two ways; the target will be using more shared cache than it is supposed to and it will have less memory bandwidth since the Pirate is stealing some of that to. This argument that some loss of data is accepted is also the reason why the Pirate works with *pseudo-LRU* replacement. Even if one of the Pirate's cache-line is evicted this is okay as long as we ensure that it is infrequent.

The Pirate needs to be monitored to make sure that it is keeping its dataset in the cache. There have been several solutions to this problem. Eklov *et. al* [5] looked at the fetch ratio of the Pirate. The Pirate is constantly accessing its data, and as long as those accesses do not require a fetch from memory the whole dataset must be in the cache. A problem in this approach is that it can be hard to measure the fetch ratio from a certain core, depending on the hardware.

Another method used by Mechri [7] is based on measuring the time it takes for the Pirate to go through its whole dataset. If the Pirate starts missing in the cache it will start waiting for memory fetches. This will make a noticeable change in the time it takes for the Pirate to go through its whole dataset.

The amount of cache the Pirate can steal is limited by the frequency at which it can touch data. To increase this it is possible to run several Pirate threads. The Pirate dataset can then be divided between Pirate threads, which allows them to touch each piece of data more often. This can increase the amount of cache the Pirate can steal linearly with the number threads. It will also increase the LLC bandwidth used by the Pirate, which we have to make sure is not saturated since the target should not be affected in any other way than the loss of shared cache.

One way to implement the Pirate is to steal a fixed amount of cache for each run, and then do multiple runs with different sizes to generate a performance curve. This results in a large overhead since creating a curve for 8 cache sizes requires the target to be run 8 times.

Another way is to implement online size adjustment where the Pirate changes the size of its dataset during the execution. Figure 2 shows the scheduling for the online size adjustment. Between each (performance counter) sample the Pirate increases its dataset size. The target's execution is paused to enable faster warming of the Pirate. Since the target is not running during this warming it will not throw out any of the Pirate's data from the cache. Therefore the warming of the Pirate will be done when the Pirate has iterated through its whole dataset once.

When the largest size has been sampled the Pirate is set to steal no cache, and the target warms its cache without the Pirate stealing any cache. Since the target is doing an arbitrary task it is difficult to estimate how long the warmup will take. A simple solution is to wait a "long time" to be sure that it has warmed the cache. This measurement cycle is then repeated until the target has finished executing.

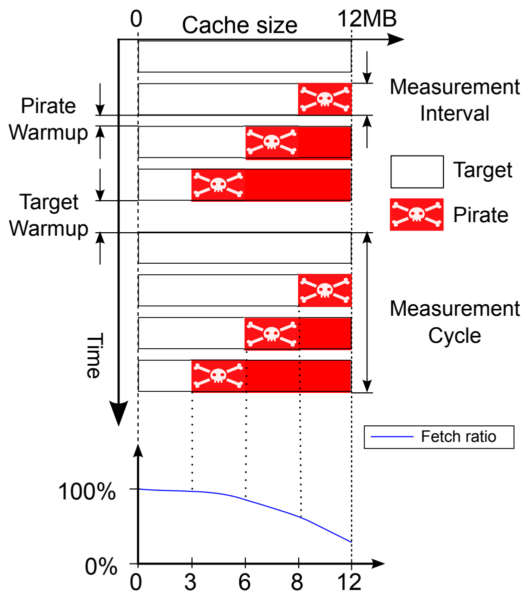


Figure 2: Time flow of Pirate with online size adjustment to get a complete performance metric curve in one run.

In order to make sure that the cache space available to the target acts just like a real cache it is important that the Pirate steals cache evenly in all sets. Therefore the Pirate should steal an amount of data that is a multiple of the way size[2]. The Pirate's dataset should also index evenly over the sets in the LLC.

With the normal size for a memory page (a couple of kilobytes) it can be hard to make sure that a dataset is spread evenly over a all sets in the cache, since a cacheline's place in the cache is defined by its *physical address*, whereas

[2]The number of sets multiplied by the size of a cacheline.

the application uses *virtual adresses.* A typical way size can be 512 kB (8 MB cache with 16 ways) and with 4 kB pages (address mapping granularity) this requires 128 pages. Because the OS will place these as it likes in the memory, there is no telling what physical address the cachelines in the data will have. This can result in unintended conflict misses since the Pirate's dataset might steal more of some sets then others. This is not acceptable since it would result in a non-uniform cache memory space for the target. It can be solved by using huge pages (2 MB). They will always be 2 MB aligned and therefore we can know all bits of the physical memory that will be used for the cache index function. However, we need to explicitly request huge pages when allocating memory.

3 The new Pirate

3.1 Automatic parameter detection

In order to run the Pirate, several parameters have to be set, specifying the hardware it's running on. This information is available on Linux in the `sysfs` filesystem. The Pirate takes this information and automatically configures itself, which makes it more easy to use.

3.2 Performance counter attachment

To attach the counters to the processes the Pirate uses the pfm4 library. The method for configuring these counters is complicated and pfm4 simplifies the process. The events that are available on a machine depends on the hardware (and the OS) and are named differently and pfm4 also helps with this by standardizing names of performance events.

3.3 CPI validation

The CPI of the Pirate is sampled to validate if the Pirate is doing what it's supposed to. Relying on fetch ratio can be inconvenient to since all architectures do not have the per-core performance counters needed to measure it. The performance counters for cycles and instructions on the other hand have standardized names. Using CPI is similar to looking at the time it takes for the Pirate to go through its dataset which Mechri [7] did. The difference is that the CPI does not depend on the size of the dataset which is convenient since this Pirate changes the size of the dataset many times during its execution.

The Pirate does a calibration run before the target is started to get a reference CPI. While the target is sampled the Pirate's CPI is also sampled, and as long as the CPI is within a small range of the reference CPI we know that the Pirate is doing what it is supposed to. If the Pirate would start missing in the cache, it would start having to wait for memory fetches and the CPI would increase.

When looking at the CPI we can also see if the Pirate is being affected in other ways, e.g. through LLC bandwidth contention. If the Pirate is being slowed down by contention from the other cores, this means that the other cores will also be slowed down by the Pirate's LLC traffic. Then the target is being affected in more ways than just a decreased amount of shared cache, and the measurement is not reliable any more. This is likelier to be the case when Pirate threads are used.

So by looking at the CPI we can see both when the Pirate starts losing its data from the cache and when it starts to compete with the target for other shared resources.

3.4 Data management

The Pirate uses Google Protobuf, an automated mechanism for serializing structured data, to make data management easier. This enables metadata to be included in the output file, and all data can easily be accessed with C++, Python and Java using generated source code for reading (and writing) the Protobuf file.

The output is divided into the header with all metadata and then the samples. The output file starts with the magic string `PIRATEv1` and then there is a `uint32_t` with the length of the header data in bytes. After the header follows a list of samples. Each sample is preceded by a `uint32_t` with the sample length in bytes. The content and format of the output file can be seen in the appendix or the source file `perf_pb.proto`.

Along with the Pirate, the Python script `pirate2csv.py` is provided which converts the Protobuf output file to a plain text csv-output with the metadata and the sample summed up per each sample-cache-size. Below is an example of a print-out from running `./pirate2csv.py pirate_output.pb`

```
# 1: Target cache size
#
# Target:
#       Command: pwd -P
#       CPU: 0
#       Sample period: 10000
#       Counters:
#               2: PERF_COUNT_HW_INSTRUCTIONS
#               3: PERF_COUNT_HW_CPU_CYCLES
#               4: OFFCORE_RESPONSE_0:ANY_LLC_MISS:ANY_DATA
#               5: MEM_INST_RETIRED:LOADS
#               6: MEM_INST_RETIRED:STORES
# Pirate:
#       Ways: 16
#       Cache size: 8388608
#       Way size: 524288
#       Stride: 64
#       CPU: 2
#       Counters:
#               7: PERF_COUNT_HW_INSTRUCTIONS
#               8: PERF_COUNT_HW_CPU_CYCLES
#       Reference size: 4194304
#       Reference:      462506 386019
524288 9963 71313 344 2176 709 445811 435001
1048576 10017 14949 188 2361 2 203331 168047
1572864 10018 14357 126 2347 2 193167 154807
```

```
2097152 9976 14073 136 2333 2 190640 152070
2621440 10025 13898 152 2349 2 185852 148955
3145728 9987 13818 144 2339 2 189054 151420
3670016 9972 51267 58 2444 995 276165 220793
4194304 10004 36018 38 2534 1234 272840 217919
4718592 10017 31861 65 2776 1080 390784 311293
5242880 10007 14702 83 2714 1111 200168 160266
5767168 9968 15688 80 2588 1019 212649 170221
6291456 10011 20810 104 2724 1099 446930 357255
6815744 10013 26225 107 2703 1037 461202 368373
7340032 9995 32235 279 2899 1028 548693 438187
7864320 10024 32270 258 3061 1037 558467 449034
8388608 10005 98156 120 1989 1331 2377032 793347
```

3.5 Workaround for non-power-of-2 way sizes

The purpose of the cache indexing function is to map all the memory addresses equally over all the sets in the cache. When the number of sets in the cache is a power of two this is easily done by using the least significant bits as an index. If a cache has N sets then $log_2(N)$ bits are needed for the index. When the number of sets is not a power of two it gets more complicated.

One of the reference machines used for this project has a 16-way set-associative 12 MB LLC, which results in a 768 kB way size and 12288 sets. While only 20 bits are needed to index into the cache, the hardware seems to use a hash function that sometimes uses higher bits to spread cache lines evenly across the sets.

The workaround for this is based on setting the higher bits of the huge-page-address to zero. For each way in the cache a huge page was allocated. Only the first 768 kB of each huge page is used, so the Pirate uses one page per way being stolen.

4 Evaluation methodology

The Pirate is meant to run with any application, and therefore there is no specific application that the Pirate should be tested with. To validate the Pirate we have use a set of benchmarks; two microbenchmarks and three benchmarks from the SPEC CPU2006 benchmark suite. The microbenchmarks are very simple applications where we know their behaviour, and therefore we know the expected result. The SPEC CPU2006 benchmarks are more complex benchmarks, and these results we can only compare with what others have got when using Pirating with the same benchmarks.

4.1 Random access microbenchmark

The random access microbenchmark is an application that given a dataset size accesses all cache-lines in the dataset in a random order. The one used here is written in x86-assembler. It has a tight loop with one memory access in each loop iteration, and uses a linear congruential generator (LCG) [8] to generate random addresses.

Since the data is accessed in a random order the result of this benchmark is independent of the replacement policy used in the architecture since the chance of a certain cacheline being in the cache does not depend on its history. The resulting fetch ratio curve[3] should be linear to the cache size. With a cache size of zero the fetch ratio is one since no data is in the cache and with cache size larger or equal to the dataset size the fetch ratio is one since the whole dataset fits in the cache.

Figure 3 shows simulated fetch ratio curves for the random access benchmark with different replacement policies.

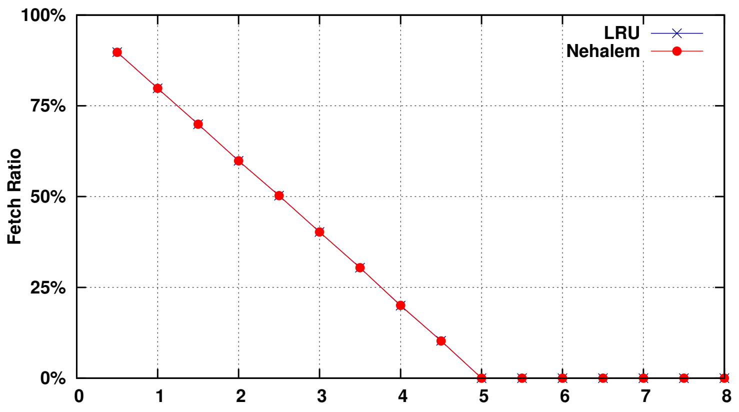


Figure 3: Simulated fetch ratio curve for the random access microbenchmark with 5 MB dataset and with different replacement policies.

4.2 Sequential access microbenchmark

The sequential access microbenchmark is an application that given a dataset size accesses all cache-lines in the dataset in a sequential order. Here the fetch and miss ratio curves will depend on the replacement policy used. With a LRU eviction protocol the fetch ratio will be one for all cache sizes smaller than the dataset size. When iterating over the dataset each cacheline will be accessed in order, and since the whole dataset can not fit in the cache each cacheline will become the least recently used one and be evicted before it is accessed again. When the cache size is larger or equal to the dataset size the fetch ratio will be zero since the whole dataset can fit in the cache.

[3]This is true for both fetch and miss ratio, since prefetching is the difference between them and prefetching will not help because the accesses are random.

With the Nehalem pseudo-LRU cache it is harder to predict the result, but it can be simulated and Figure 4 shows the result of this simulation together with the LRU cache.

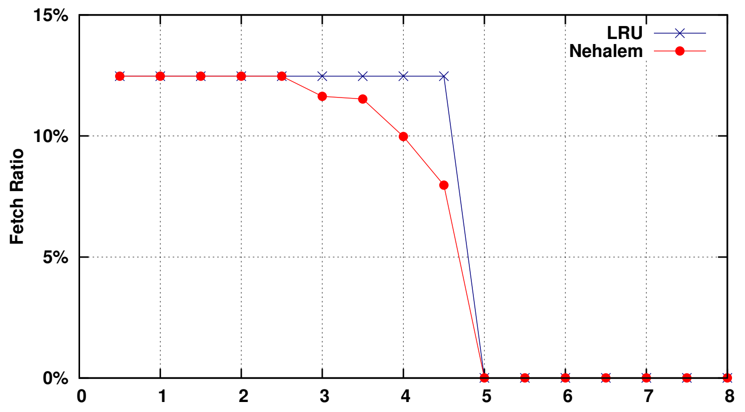


Figure 4: Simulated fetch ratio curve for the sequential access microbenchmark with 5 MB dataset and with different replacement policies.

4.3 SPEC CPU2006

SPEC CPU2006 is a standardized set of benchmarks from real life applications meant to measure the CPU and memory performance in ”real time”. These benchmarks are mostly run to see that nothing unexpected happens. Three benchmarks from SPEC CPU2006 are used in this report:

- **429.mcf:** Integer benchmark solving single-depot vehicle scheduling in public mass transportation problems.
- **450.soplex:** Floating point benchmark solving a linear program using the Simplex algorithm.
- **482.sphinx3:** Floating point benchmark doing speech recognition.

4.4 Reference machines

Four reference machines with different setup (Table 1) are used in this study. All machines have `libpfm4-4.4.0` and Google Protobuf 2.5.0 installed. The hardware performance counters used for the measurements can be found in the appendix.

Architecture	Linux distro	Kernel	CPU	LLC-size	No. ways
Intel Nehalem	Ubuntu	3.11.6	2 x Intel Xeon E5520	8 MB	16
Intel Westmere	Gentoo	3.10.19	Intel Xeon E5620	12 MB	16
Intel Sandy Bridge	Ubuntu	3.8.0	Intel Core i7-2600K	8 MB	16
AMD Phenom	Ubuntu	3.12.3	AMD Phenom II X4 920	6 MB	48

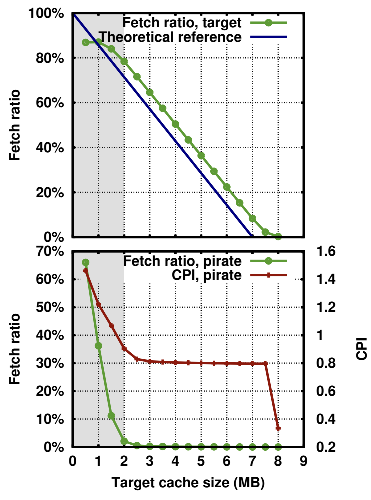
Table 1: Reference hardware setups

5 Results

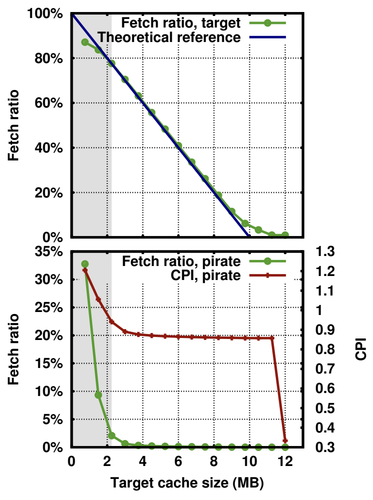
The different benchmarks was run on the reference machines and the plots will be presented here. The grey marked area in the plots are where the Pirate is no longer stealing the cache that it should be, and the limit used for this is that when the CPI has increased 10% from the reference. We can see that the fetch ratio is kept under 10% in the trusted region (white background) of all graphs with this limit.

5.1 Intel

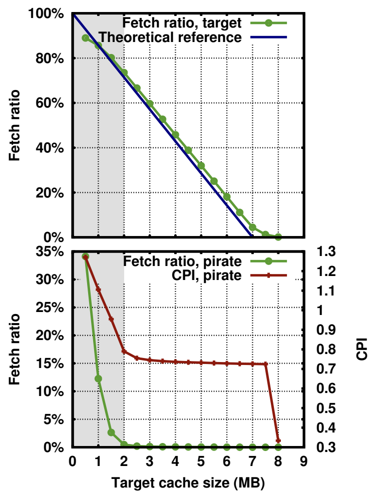
Figure 5 shows the result of the random access microbenchmark for the Intel machines. The fetch ratio curves follow the expected line, but on the Nehalem machine it looks like the Pirate is stealing one way (512 kB) too much. We tested running the random access microbenchmark with a dataset size of 8 MB (the whole cache) without the Pirate, and the fetch ratio corresponded to 512 kB of the cache missing, so the reason is probably that the OS is stealing cache. The curves stray from the line close to zero cache size and this is because the Pirate is stealing less cache than it should in this region, and hence the fetch ratio for the target can be expected not to change linearly. The curve starts to level of when the target cache size get close to the dataset size, and this is likely due to the OS using some cache.



(a) Nehalem, 7MB dataset.



(b) Westmere, 10MB dataset.

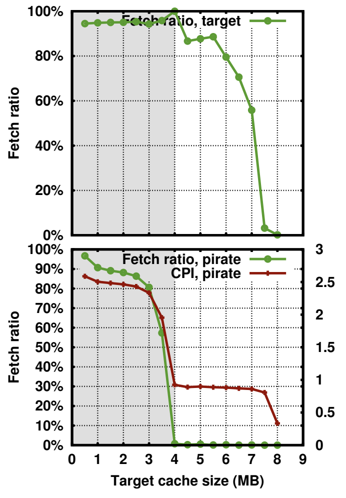


(c) Sandy Bridge, 7MB dataset.

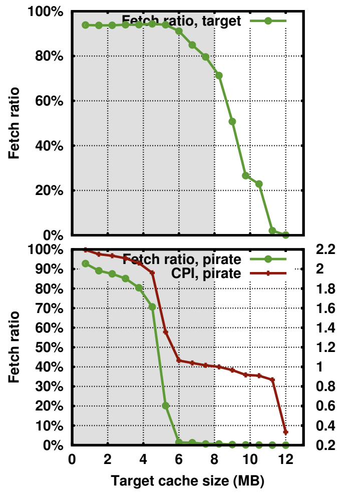
Figure 5: Random access microbenchmark results for Intel machines.

Figure 6 shows the result of the sequential access microbenchmark. This is a more aggressive benchmark and therefore the amount of cache the Pirate can steal decreases. This is because the target accesses its data sequentially and therefore gets help by prefetchers. In the trusted region, the fetch ratio curves fit well with the simulated curve for the Nehalem replacement policy in Figure 4.

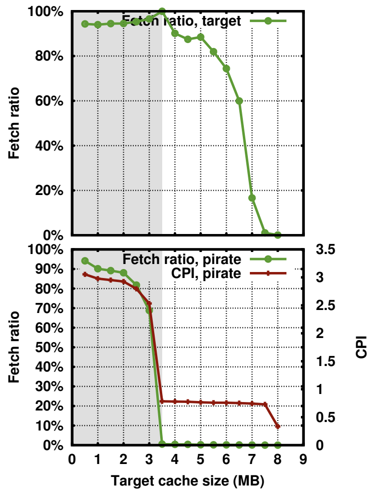
The result from the SPEC CPU2006 benchmarks (Figures 7, 8 and 9) correspond well with the result Eklov *et. al* got with their Pirate [5].



(a) Nehalem, 7MB dataset.

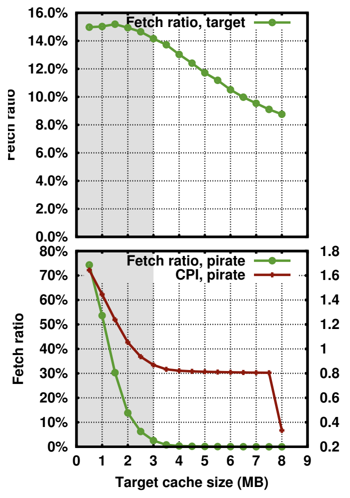


(b) Westmere, 10MB dataset.

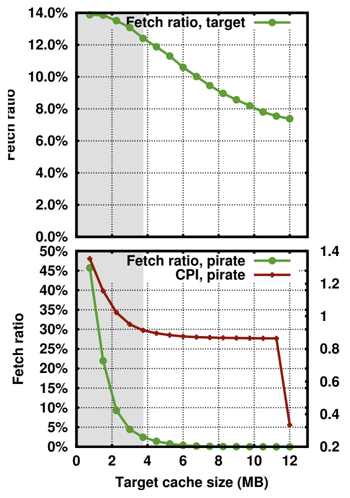


(c) Sandy Bridge, 7MB dataset.

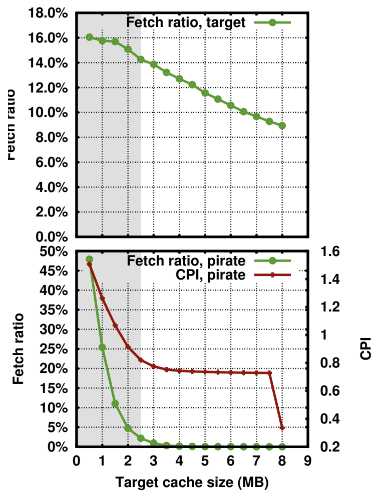
Figure 6: Sequential access microbenchmark results for Intel machines.



(a) Nehalem

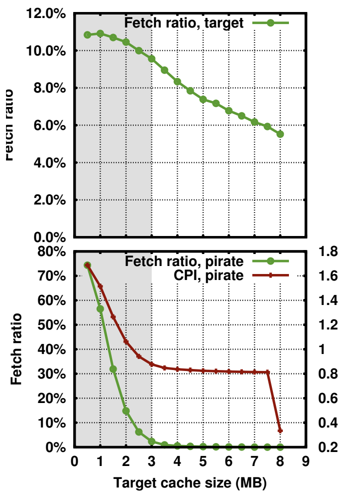


(b) Westmere

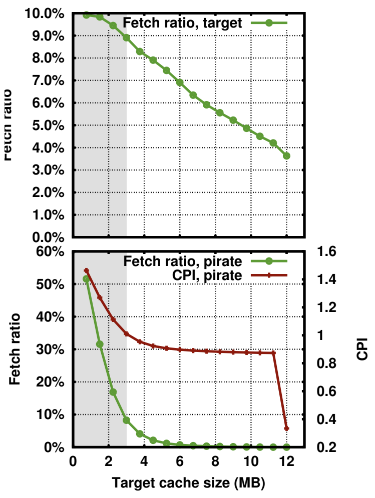


(c) Sandy Bridge

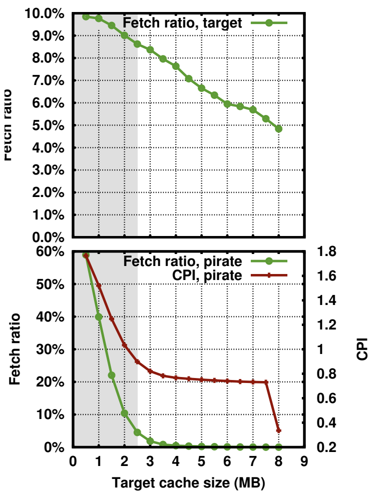
Figure 7: SPEC CPU2006, 429.mcf benchmark results for Intel machines.



(a) Nehalem

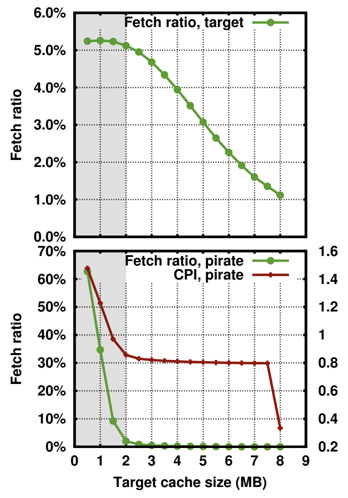


(b) Westmere

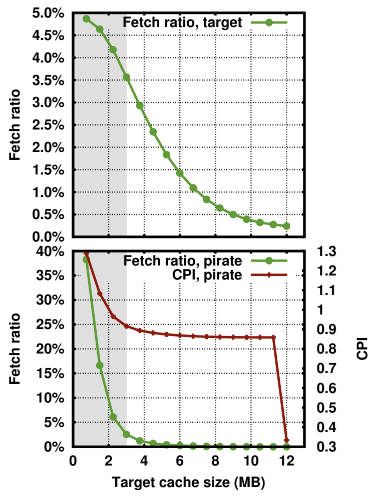


(c) Sandy Bridge

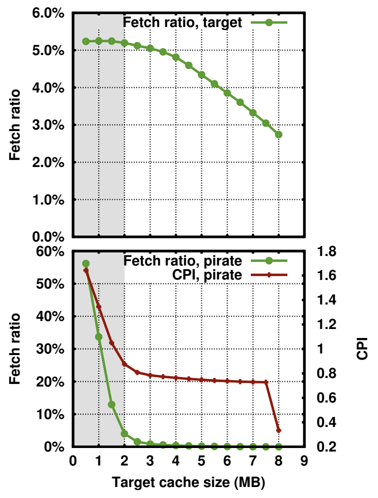
Figure 8: SPEC CPU2006, 450.soplex benchmark results for Intel machines.



(a) Nehalem



(b) Westmere



(c) Sandy Bridge

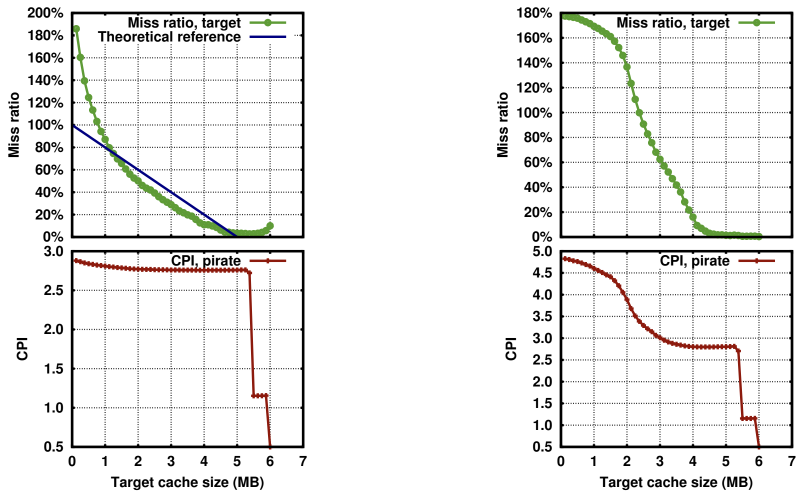
Figure 9: SPEC CPU2006, 482.sphinx3 benchmark results for Intel machines.

5.2 AMD

The results for the AMD machine are more difficult to interpret. A problem with the AMD system is that it does not have any core-specific counters for shared resources, which makes it hard to measure fetch ratio or miss ratio for the target since fetches and misses from the Pirate are counted to. Figure 10 shows the result of the microbenchmarks for the AMD system and we can see that the miss ratio goes up to almost 200% which is a sign that misses from both target and Pirate were counted. In principle this should work since we know that as long as the CPI of the Pirate is within its limit the Pirate is not missing in the cache. Our interpretation of the curves are that this AMD architecture probably uses another replacement policy. The counters used are seen in Table 2. Another thing that the Pirate application does not take into account is that this cache is *exclusive*, and that the size of the L1 and L2 cache should be withdrawn from the amount of cache that the Pirate is stealing, and the available cache to the target should be increased by the same amount.

Event	pfm4 name
Instructions	PERF_COUNT_HW_INSTRUCTIONS
Cycles	PERF_COUNT_HW_CPU_CYCLES
LLC Read Misses	L3_CACHE_MISSES:ANY_READ
Mem Accesses	DATA_CACHE_ACCESSES

Table 2: Counters used when running the Pirate on the AMD system.



(a) Random access microbenchmark

(b) Sequential access microbenchmark

Figure 10: Microbenchmark results with a 5MB dataset for the AMD machine.

6 Enhancements

The rate of how often the Pirate can touch each cacheline is what limits how much cache the Pirate can steal. One way to increase the access rate is to divide the dataset across several Pirate threads as stated above. Another way to increase single thread access rate would be to rewrite the Pirate loop in x86 assembler, to make the loop as tight as possible. This would slightly increase the amount of cache that the pirate can steal. The limit of how much this could help is the LLC bandwidth since the target should not be noticeably affected by the Pirate's memory accesses.

One way to deal with this problem is to write a Pirate loop with a variable amount of `nop` operations, so that the tightness of the loop can be adjusted. This could then be used to ensure that the Pirate's access rate does not affect the bandwidth of the target. However, if the tightness of the Pirate loop would be changed at runtime it would be hard to use CPI for validating the Pirate, and fetch ratio would have to be used.

As mentioned in the introduction, David Eklov who proposed Cache Pirating [5] also proposed the Bandwidth Bandit [6]. The Bandit is based on the same idea as the Pirate but instead of stealing shared cache it steals memory bandwidth. For the Bandit to be effective more needs to be known about the memory system, and this is usually poorly documented, and therefore the Bandit is hard to make portable between different machines.

The Bandit can be implemented using random accesses, which makes it less dependent on the specific memory it is running and more portable across machines, at the cost of effectiveness. This could potentially be implemented alongside with the Pirate. This would enable us to either steal cache and memory bandwidth one at a time, or both at the same time by running the Bandit and the Pirate on separate cores[4].

7 Usage

7.1 Installation

Make sure these two packages are installed:

- **libpfm4** [`http://perfmon2.sourceforge.net/`]
- **Google Protobuf** [`https://developers.google.com/protocol-buffers/`]

Download the Pirate from my git repository and compile:

```
git clone git@github.com:fraghag/pirate.git pirate
cd pirate
make
```

Activate huge pages in your Linux system:
Five-step process found here `https://wiki.debian.org/Hugepages`.

[4]This is under the assumption that we have at least 4 cores in the CPU. One each for the target, Pirate, Bandit and the monitoring application that handles the sampling.

7.2 Running the Pirate

The Pirate is run through the Linux terminal command with:
`./perfpirate [arguments] -- [target_command target_arguments]`

Example

```
./perfpirate -c 0 -C 1 --sample-period=100000 -e BRANCH_INSTRUCTIONS_RETIRED
-e MISPREDICTED_BRANCH_RETIRED -o result.pb -- my_target -s 4096
```

Arguments

`-c, --target-cpu=CPU`
Pin target process to CPU, default is 0.

`-C, --pirate-cpu=CPU` Pin pirate to CPU. Repeat this option for more pirates several Pirate threads. It is recommended that you set this by yourself, since a working default depends on the hardware.

`-e, --target-event=EVENT`
Events to measure on the target. EVENT given with the name used in *libpfm4*.

`-E, --pirate-event=EVENT`
Events to measure on the target. EVENT given with the name used in *libpfm4*.

`-o, --output=FILE`
Filename and path of Protobuf output file. Default is `perfpirate.pb`.

`-r, --target-raw-event=EVENT`
Raw events to measure on the target. EVENT given in the form of a string beginning with '`raw:`' and then the raw event mask (if hexadecimal mask start with '`raw:0x`').

`-s, --pirate-size=SIZE`
Pirate data set size. This disables the online size adjustment, and just samples the given SIZE.

`--sample-freq=N`
Set event sample frequency for the instruction counter on the target. Do not use together with the `--sample-period` argument.

`--sample-period=N`
Set event sample period for the instruction counter on the target. Default value is 1,000,000. Do not use together with the `--sample-freq` argument.

`-?, --help`
Gives a help list.

`--usage`
Give a short usage message.

7.3 Performance counters

With the **libpfm-4.4.0** package there is a application `examples/showevtinfo` which shows all the available events for the current architecture and OS with their libpfm4 names and available unit-masks. It also shows the counters' code which can be used to find the counter in the Intel developer manual [9] where they are documented in the *PERFORMANCE-MONITORING EVENTS* chapter.

To use unit-masks with a counter just add them after the counter name separated with colons: `PFM4_EVENT_NAME:UMASK1:UMASK2`

7.4 Common pitfalls

Cache architecture

If the cache has non-uniform access times the target and the Pirate could have different access times to different parts of the cache and the same expected result as above might not be applicable. It is difficult to predict what the result should be if this is causing problems.

LLC bandwidth contention

If the combined traffic to the LLC gets too big the applications on all cores will start waiting for LLC accesses and memory fetches. This is probably the case when the Pirate's CPI increases without an increase in miss ratio. This is more likely to happen when running several Pirate threads, and can also happen if the target has a high memory access rate. An example of this can be seen in Figure 6b.

Pin processes to the right cores

If the Pirate's CPI does not increase a lot when the target's cache size goes toward zero then make sure that the Pirate is pinned to a core that share LLC with the target. The risk of this is high if the computer has more than one CPU, and the Pirate and target are pinned to different CPUs.

Also be sure that the Pirate and target are not pinned to the same core. The risk of this is high if each core has several threads. On Linux systems, different threads are usually named `cpuX` where `X` is the thread id number.

In the folder `/sys/devices/system/cpu/cpuX/cache` where `X` is the thread id number you can find out which caches are shared with which threads. If threads share LLC cache they are on the same CPU, and if they share L1 cache they threads on the same core.

DVFS (Dynamic Volt and Frequency Scaling)

If the Pirate's CPI suddenly decreases when the target's cache size gets smaller the OS might have used DVFS to clock down the core that the Pirate is running on. If you suspect that this might be the case you can disable DVFS in Linux or the BIOS.

Unpredictable performance counters

If you get very unexpected results, like a 200% fetch ratio (see Figure 10), then the counters you are using do probably not measure what you expect. A counter for LLC read misses for example might count the LLC read misses from all cores on that CPU even though the counter is pinned to a specific core. Check the documentation on the counter used, and see if there are more suitable counters.

Caches that are not inclusive

If the target seems to have access to more cache that it should have, then the cache might be *non-inclusive* or *exclusive.* If the cache is *exclusive* there will be no copy in the L3 cache of the data kept in the L1 and L2. Therefore the size of the available cache to an application is the sum of the L1, L2 and L3 cache size. The Pirate will also be stealing less of the shared cache since part of its dataset will reside in the L1 and L2 cache. If the cache is *non-inclusive* it is more difficult to say, but the result should be correct as long as the Pirate's dataset doesn't fit in the L2 cache.

OS using cache

If the Pirate seems to steal more cache than it is supposed to, then it might be the OS that is using some of the cache due to kernel activity. You can test this by running the random access microbenchmark with a dataset equal to the cache size, and without the Pirate stealing any cache. Since the accesses in the microbenchmark are random the percent miss ratio will also be the proportion of the cache that is missing.

Acknowledgments

I would like to thank the UART team at Uppsala University for involving me in their work and giving me this opportunity to do this project, and Maya Neytcheva for supervising the project course. The biggest thanks goes to my supervisor Andreas Sandberg who has had the patience to help and teach me so much.

References

[1] Andreas Sandberg, David Black-Schaffer, and Erik Hagersten. “Efficient Techniques for Predicting Cache Sharing and Throughput”. In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT).* 2012, pp. 305–314. DOI: 10.1145/2370816.2370861.

[2] Andreas Sandberg et al. “Modeling Performance Variation Due to Cache Sharing”. In: *Proc. International Symposium on High-Performance Computer Architecture (HPCA).* 2013, pp. 155–166. DOI: 10.1109/HPCA.2013.6522315.

[3] Erik Berg and Erik Hagersten. “StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis”. In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS).* 2004, pp. 20–27. DOI: 10.1109/ISPASS.2004.1291352.

[4] David Eklöv and Erik Hagersten. “StatStack: Efficient Modeling of LRU Caches”. In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS).* Mar. 2010, pp. 55–65. DOI: 10.1109/ISPASS.2010.5452069.

[5] David Eklöv et al. “Cache Pirating: Measuring the Curse of the Shared Cache”. In: *Proc. International Conference on Parallel Processing (ICPP).* 2011, pp. 165–175. DOI: 10.1109/ICPP.2011.15.

[6] David Eklöv et al. “Bandwidth Bandit: Understanding Memory Contention”. In: *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS).* 2012, pp. 116–117. DOI: 10.1109/ISPASS.2012.6189214.

[7] Moncef Mechri. “Stealing the shared cache for fun and profit”. MA thesis. Uppsala University, 2013.

[8] Wikipedia. *Linear congruential generator.* English. Feb. 1. URL: http://en.wikipedia.org/wiki/Linear_congruential_generator.

[9] *Intel 64 and IA-32 Architectures Software Developer’s Manual.* Volume 3B: System Programming Guide. 30.4.4 Precise Event Based Sampling (PEBS). Intel Corporation. 2010.

A Tables of counters used

Event	pfm4	Type	Code	Unit mask
Instructions	**PERF_COUNT_HW_INSTRUCTIONS**	**0**	**0x1**	**-**
Cycles	PERF_COUNT_HW_CPU_CYCLES	0	0x0	-
Mem Fetches	**OFFCORE_RESPONSE_0:ANY_LLC_MISS:ANY_DATA**	**4**	**0x1b7**	**0xf833**
Mem Loads	MEM_INST_RETIRED:LOADS	4	0xb	0x01
Mem Stores	**MEM_INST_RETIRED:STORES**	**4**	**0xb**	**0x02**

Table 4: Counters used when running the Pirate on the Nehalem system.

Event	pfm4	Type	Code	Unit mask
Instructions	**PERF_COUNT_HW_INSTRUCTIONS**	**0**	**0x1**	**-**
Cycles	PERF_COUNT_HW_CPU_CYCLES	0	0x0	-
Mem Fetches	**OFFCORE_RESPONSE_0:ANY_LLC_MISS:ANY_DATA**	**4**	**0x1b7**	**0xf833**
LLC Misses	OFFCORE_RESPONSE_1:ANY_LLC_MISS:DMND_DATA_RD:DMND_RFO	4	0x1bb	0xf803
Mem Loads	**MEM_INST_RETIRED:LOADS**	**4**	**0xb**	**0x01**
Mem Stores	MEM_INST_RETIRED:STORES	4	0xb	0x02

Table 5: Counters used when running the Pirate on the Westmere system.

Event	pfm4	Type	Code	Unit mask
Instructions	**PERF_COUNT_HW_INSTRUCTIONS**	**0**	**0x1**	**-**
Cycles	PERF_COUNT_HW_CPU_CYCLES	0	0x0	-
Mem Fetches	**OFFCORE_RESPONSE_0:MISS_DRAM:ANY_DATA**	**4**	**0x1b7**	**0x400091**
Mem Loads	MEM_UOPS_RETIRED:ALL_LOADS	4	0xd0	0x81
Mem Stores	**MEM_UOPS_RETIRED:ALL_STORES**	**4**	**0xd0**	**0x82**

Table 6: Counters used when running the Pirate on the Sandy Bridge system.

B Protobuf output structure

```
/* Info about each performance counter */
message PerfCtrInfo
{
    optional int32 id = 1;
    /* Name in pfm4 library */
    optional string name = 2;
    optional uint64 config = 3;
    optional uint64 config1 = 4;
    optional uint64 config2 = 5;
    optional uint32 type = 6;
}

message PerfCtrSample
{
    /* Target cache size for which the sample was taken */
    optional uint32 size = 1;
    /* Value for each counter in same sample */
    repeated uint64 ctr = 2 [packed=true];
}

message PerfCtrDump
{
    /* Samples for target */
    optional PerfCtrSample t_sample = 1;
    /* Samples for each pirates-thread */
    repeated PerfCtrSample p_sample = 2;
}

message PerfHeader
{
    message TargetSetup
    {
        /* CPU that the target ran on */
        optional uint32 cpu = 1;
        /* Number of instructions er sample */
        optional uint64 sample_period = 3;
        /* Number of counters on the target */
        optional uint32 n_ctrs = 4;
        /* Target run command */
        optional string command = 5;
        /* List of used counter on target */
        repeated PerfCtrInfo ctr = 6;
    }

    message PirateSetup
    {
        /* Number of ways in the LLC */
        optional uint32 ways = 1;
        optional uint32 cache_size = 2;
        /* Cache-line size for LLC */
        optional uint32 stride = 3;
        optional uint32 way_size = 4;
        /* Pirate steal same size whole simulation */
        optional bool no_sweep = 5;
        /* Number of Pirate threads used */
        optional uint32 n_pirates = 6;
        /* Number of counters on each Pirate thread */
        optional uint32 n_ctrs = 7;
        /* List of used counter on Pirate threads */
        repeated PerfCtrInfo ctr = 8;
        /* List of CPUs for Pirate threads */
        repeated uint32 cpu = 9 [packed=true];
    }

    /* Target header */
    optional TargetSetup t_setup = 1;
    /* Pirate header */
    optional PirateSetup p_setup = 2;
    optional bool no_reference = 3;
    /* Sample for reference run of Pirate */
    optional PerfCtrSample reference = 4;
}
```