Server Performance of Newcache: A Novel Cache Design for Security

Project 4: Sam Payne, ELE 386

Introduction

In a growing digital market, the relevant features of computer design are changing. The memory hierarchy, in particular, has become a bottleneck in many computing systems, and cache performance is extremely important. As computing devices become more mobile, optimizing caches for power efficiency has become more important as well. Though it is less often thought of, another crucial issue in cache design is security. Since, today, such a large amount of confidential information is stored and exchanged on computers, it is important that these machines remain secure and resistant to attacks.

Side Channel Attacks on caches capable of obtaining guarded encryption keys pose many threats to computing systems. Past method of guarding against these attacks have come at a significant cost, resulting in reduced performance speed and power [1]. However, an alternative architecture in cache design, Newcache, has not only proven secure, but also higher-performing and more power efficient than past solutions [2]. This project aims to analyze the performance of this secure cache on a server.

Caches and Definition of Terms

A cache is a subset of memory used by programs. In theory, a program uses a small amount of memory for most of its calculations, so a cache can provide quick access to this memory rather than forcing a program to access slow main memory on every load. Caches can be built on chip with the processor, as is typically done with the first level (L1) cache. Some architectures include room for an L2 and L3 cache, which are larger, to reduce their miss rate. The miss rate in a cache represents the ratio of loads that do not find their data in the cache, and therefore the next level in memory must be queried. Alternatively, the hit rate of a cache represents the ratio of loads that find the data they need in the cache.

Caches possess a variety of features to improve their hit rates (and decrease their miss rates). Every cache is divided into “lines,” which are locations where blocks of data can be held. A cache can be associative or direct mapped. In a direct mapped cache, each piece of data can reside in a particular line; if that line is occupied, the data currently residing there is usually evicted from the cache. In an associative cache, there are several lines where data may reside, and the line evicted must be chosen by a particular method (in many cases, Least Recently Used or LRU). In this kind of cache, if a data block can reside in one of four lines, the cache is said to be “four way set associative”; if a data block can reside in any line of the cache, that cache is “fully associative.” We can think of a direct mapped cache as sort of one-way associative.

Pieces of data are “indexed” into a line, and their “tag” uniquely identifies them (both the tag and index are taken from some upper bits of the data’s address in main memory, and the offset within the line

makes up the remaining bits in the address). Direct mapped caches have an advantage: Only one tag check needs to be performed if the index matches. A set associative cache must perform more tag checks, and a fully associative cache must perform many checks, which will use a more significant amount of power. The tradeoff here is that higher associativity requires more power while reducing the number of conflict misses.

Cache Attacks

In any side-channel attack, an attacker attempts to gain intelligence based on information that inherently results from performance [1]. For example, to determine how hard a machine is working, one could observe the power consumption of the machine or the amount of heat it generates. From there, an attacker may even be able to determine what program the machine is likely running. In cache side channel attacks, cache miss access times are used to determine information about encryption keys [2].

Description of Attacks

Two cache attacks serve as good examples of the range of ways in which side channel attacks can obtain information from caches:

Percival's Attack relies on the ability to share a cache with another thread [1]. In modern multi-core and multi-threaded systems, programs running concurrently can share caches. Percival's Attack aims to directly observe other threads' cache accesses [1]. The method used allows the attacker to perform a set of loads that will fill the cache with its own data. This will cause misses by the victim thread that will then load its data into the cache after accessing the next level of memory. The attacker then attempts to access the data it loaded into the cache. By using timing techniques, the attacker can determine if a cache miss occurred on its own loads. In this way, the attacker is capable of determining which line of the cache was accessed by the victim – since whichever line the attacker misses on must have been the line accessed by the victim [1].

This attack is dangerous, since many encryption methods, including RSA, employ lookup tables. Because the table is stored in memory and will be loaded into the cache, the index in the table can be determined based on the line of the cache accessed. From this information, the attacker can infer which index was used in the table and can determine a segment of the encryption key. This attack uses what is called *external interference*, since a thread outside of the victim is interfering with cache accesses [1]. Next, we will look at an attack that uses *internal interference.*

Bernstein's Attack on AES does not have to run locally on a machine, though it can. This attack sends requests to be encrypted by the victim. Based on the execution time of the victim, the AES key can be extracted. First, the attacker sends a large number of random plaintexts to the victim and records the

time it takes to encrypt them with a known key. The attacker then repeats the first with new random input and with an unknown key. Finally, the attacker is able to use a correlation algorithm to discover the unknown key based on execution time characteristics of the random plaintext. These timing characteristics are based on cache misses that occur during the encryption of the plaintext. This attack uses *internal interference* since cache misses are internal to the machine's functioning and are not influenced by the attacking program (unlike in Percival's Attack, where misses are a direct result of the attack running on the same cache) [1].

These two attacks are just examples of possible side channel attacks, and they present two extremes between external and internal interference. There are a variety of other attacks whose methods range between these two.

Root Causes for Attacks

The primary weakness in each side channel cache attack is that timing characteristics can be extracted from cache interference patterns [2]. In one attack, evictions resulting from conflict misses reveal what data is being used, and in the other, timing characteristics resulting from conflict evictions reveal important data to the attacker. These features are not unique to a particular kind of cache; almost any system using traditional cache line replacement methods is vulnerable to these kinds of attacks.

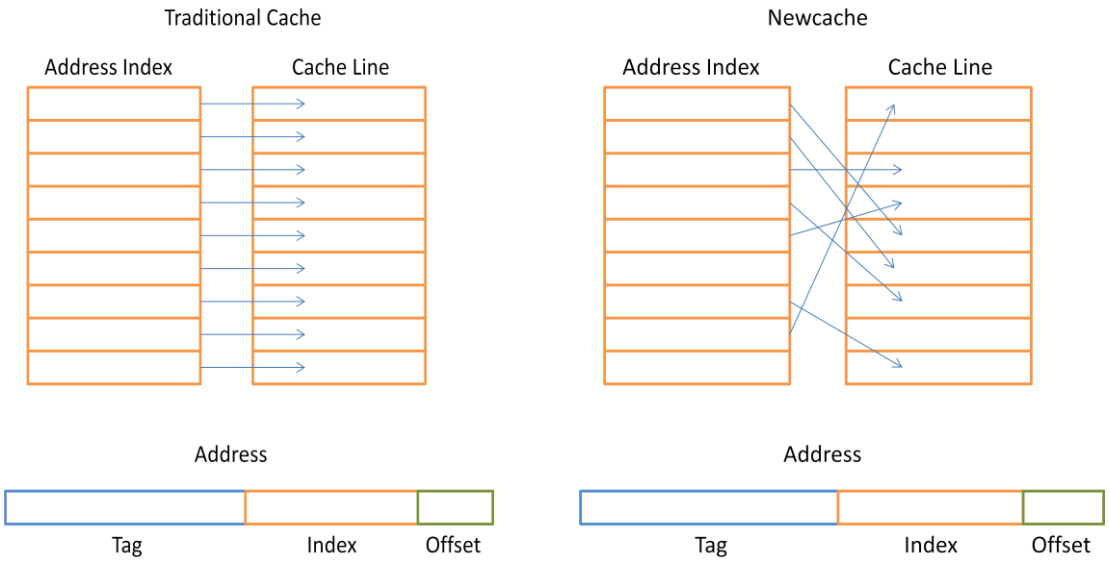
Existing solutions

Unfortunately, existing solutions in software are attack-specific. Software solutions require the rewriting of programs and do not address the fundamental problems behind these attacks. Additionally, these solutions typically result in significant performance degradation [1].

On the hardware side, there are two possible solutions that show promising security, performance, and power-efficiency: the Partition-Locked Cache (PLcache) and the Random Permutation Cache (RPcache) [1]. This paper studies an improvement on the RPcache called Newcache.

Description of Newcache

Newcache uses a unique line mapping design to maintain performance and protect against attacks. The key design feature is a remapping of indexes in the cache to cache lines. Each index can be mapped to a different physical cache line, independent of the index. This design is similar to a direct mapped cache in that only one tag is checked if the index is found in the cache. This saves power on tag checking compared to a fully associative cache, and it maintains performance by the same method [2].



The mapping feature also presents a unique opportunity to virtualize the cache and gain performance benefit similar to a fully associative cache. By increasing the size of the index, more indexes can be simultaneously mapped into the cache, while the number of lines actually present remains the same. In normal caches, the size of the index is determined by the size of the cache; a cache's size must double to increase the index length by a single bit. A larger index allows for fewer conflict evictions; however, it comes at the cost of exponential growth in cache size. In Newcache's mapping scheme, the index can be increased without a significant cost in hardware [2]. If the index is expanded by 1 bit, twice as many indexes can be mapped to unique cache lines, while only half of these indexes can be mapped at the same time. The addition of k bits to an index of size n can be called a growth of k "nebits." So to map the index of the cache line requires the matching of $k + n$ bits, while the tag comparison in mapped lines after index matching is reduced to $t - k$ bits, where t is the size of the original tag. This saves power tag checking, but at the cost of a data addressed lookup on indexes in the mapping table [2]. In general, this method still saves power compared to a fully associative cache since the size of the indexes compared in new cache will be smaller than the size of the tags compared in a fully associative cache.

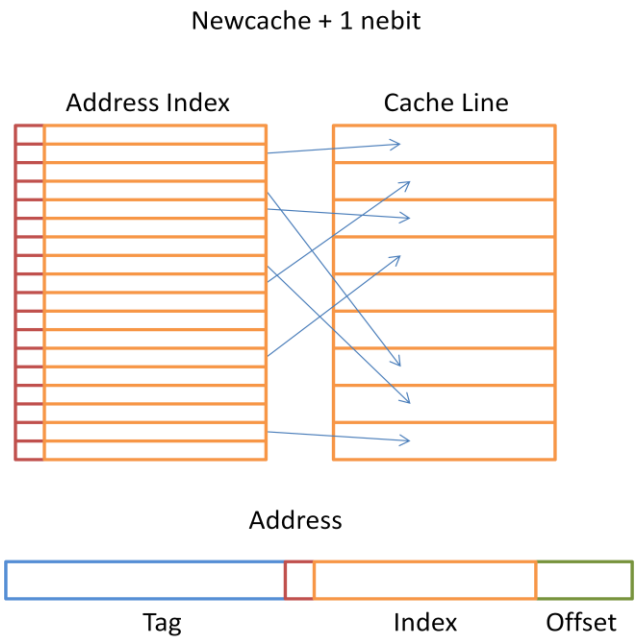


Illustration of Newcache mapping with one additional nebit

This mapping also provides security against attacks that use cache interference to gain information. If a random replacement algorithm is implemented, then any incoming line to the cache may replace any other line. If this is the case, then is extremely hard for side channel attacks to gain any information about what information has entered the cache based on which line was swapped out [2]. This protects Newcache from the threats we have mentioned while maintaining good performance and low power usage. The rest of this report will focus on the performance of Newcache while running benchmarks alongside the performance of a standard cache using LRU replacement.

Testing Methodology

This project aims to measure the performance of Newcache in a server environment. Since servers hold and transmit so much important information that users trust is confidential, it is extremely important that servers be secure.

Testing was carried out on Gem5, a clock cycle accurate simulator. The original benchmarks to be run to measure Newcache performance included the multi-threaded netapps from the PARSEC benchmark. However, due to difficulties running PARSEC on Gem5 (see Appendix for details), select tests from a supplementary single-threaded benchmark, SPEC, were run to measure cache performance.

To characterize Newcache's performance on a server, 8 of 18 SPEC benchmarks were run on a cache using Least Recently Used (LRU) replacement and Newcache on Gem5. The benchmarks include bzip2, milc, libquantum, lbm, gcc, mcf, astar, and 264ref. More information about these benchmarks can

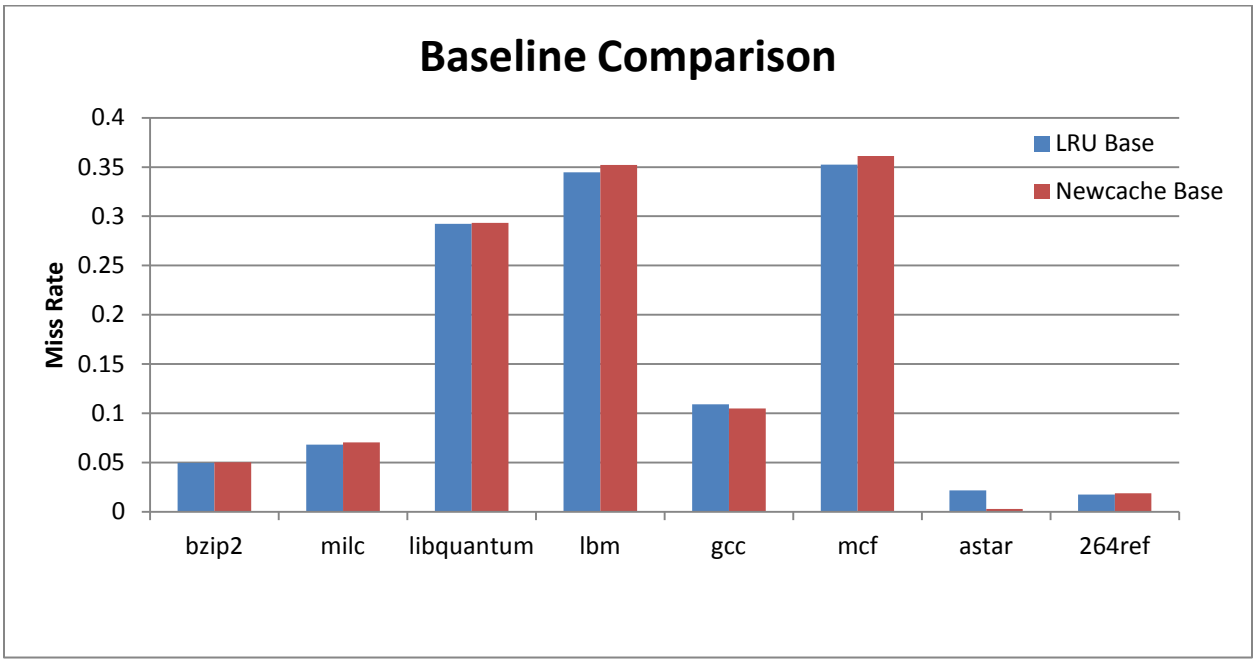
be found in the SPEC2006 documentation. These benchmarks cover both integer and floating point operations.

The performance of the L1 data cache was measured, since this cache is typically targeted in side-channel attacks. The L1 instruction cache was kept constant with 32 KB of space and with 4-way set associativity, and the L2 cache was kept constant with 2 MB of space and 8-way set associative. Across each test, the L1 data cache was varied in size, associativity, line size, nebits, and type (LRU vs. Newcache). Each test was run for 2 billion instructions after a fast-forward of 1 billion cycles.

Results

LRU vs. Newcache

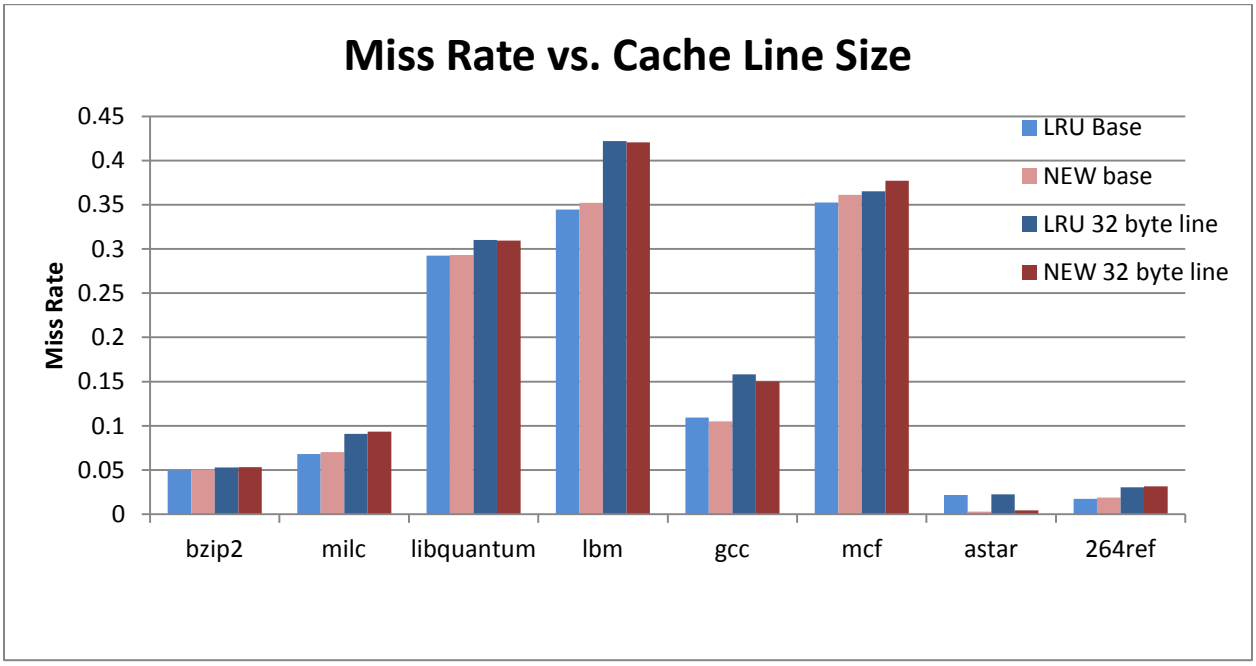
At baseline configuration: LRU cache and Newcache are of equal size (32KB) and equal cache line size (64 KB), with LRU set up with 8-way set associativity and Newcache with 4 nebits. In all of the benchmarks, Newcache performs comparably well, achieving a lower miss rate in several tests, and it never increases the miss rate by more than 1%. Newcache may perform better in many cases becacuse the addition of nebits allows a more beneficial replacement of cache lines, yielding better performance. Also, the random replacement algorithm may result in better or worse performance, depending on luck.



The two parameters common to both the LRU cache and Newcache are the cache size and cache line size. The following sections analyze the performance of each cache as these parameters are varied.

Cache Line Variation

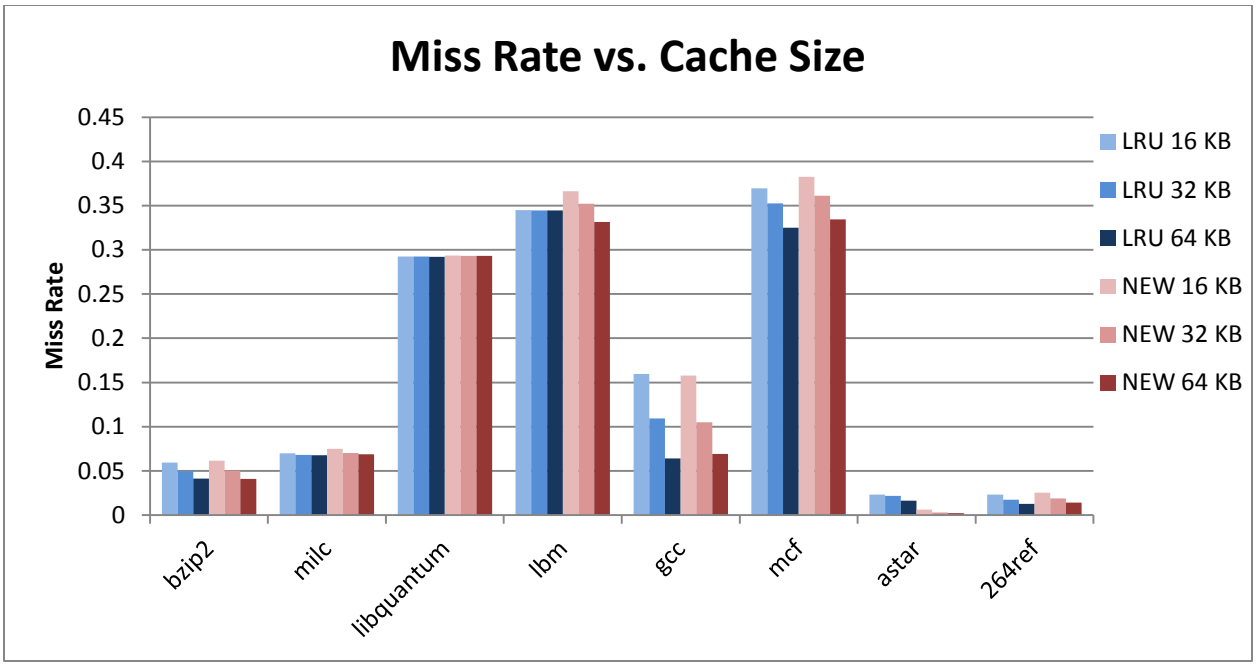
The relative performance of Newcache changes with cache line size in a manner similar to an LRU cache's performance.



In each case, the performance of the two is nearly identical when their cache line sizes match, revealing that a change in cache line size results in similar changes for both caches. In no cases does a change in cache line size have a different effect on LRU than on Newcache, though the magnitude of change differs slightly. We expect that a decrease in cache line size would cause in increase in miss rate since a smaller cache line size cannot leverage special locality as well as a larger line. Though, in some cases (not seen here), conflict misses can be avoided since cache lines take up less room in the cache.

Cache Size Variation

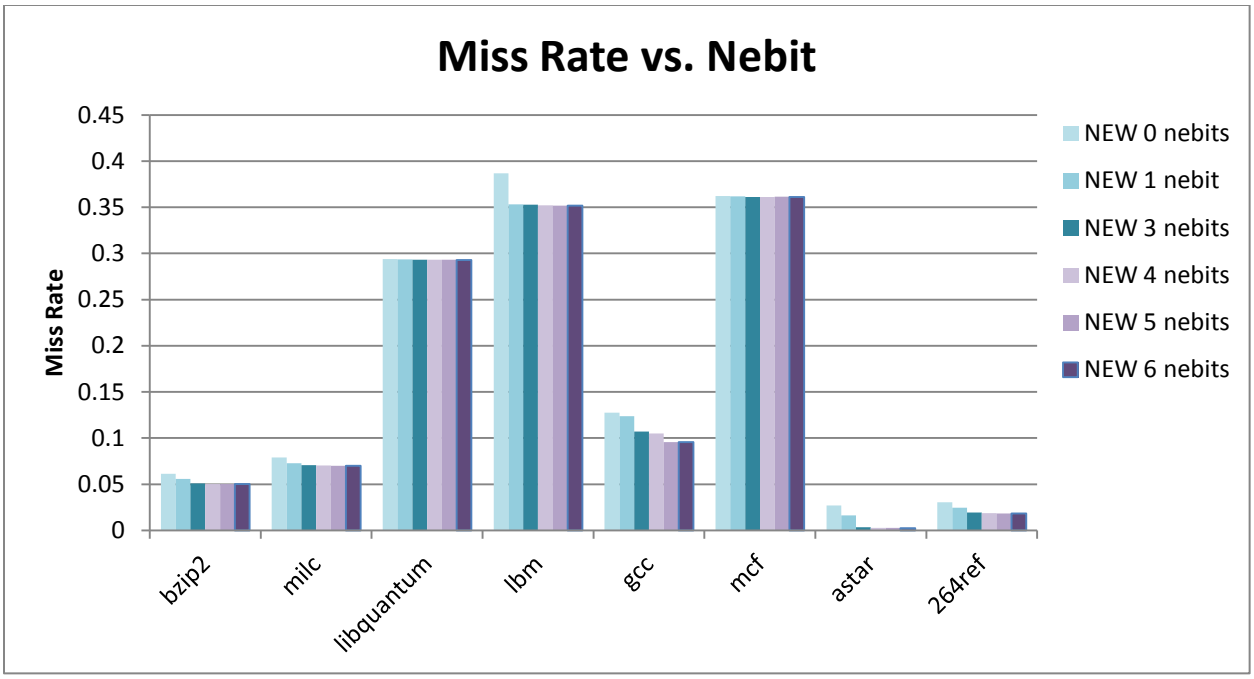
The two baselines caches were varied between 16, 32, and 64 KB sizes. Both caches display nearly consistent improvement in cache miss rate as cache size increases.

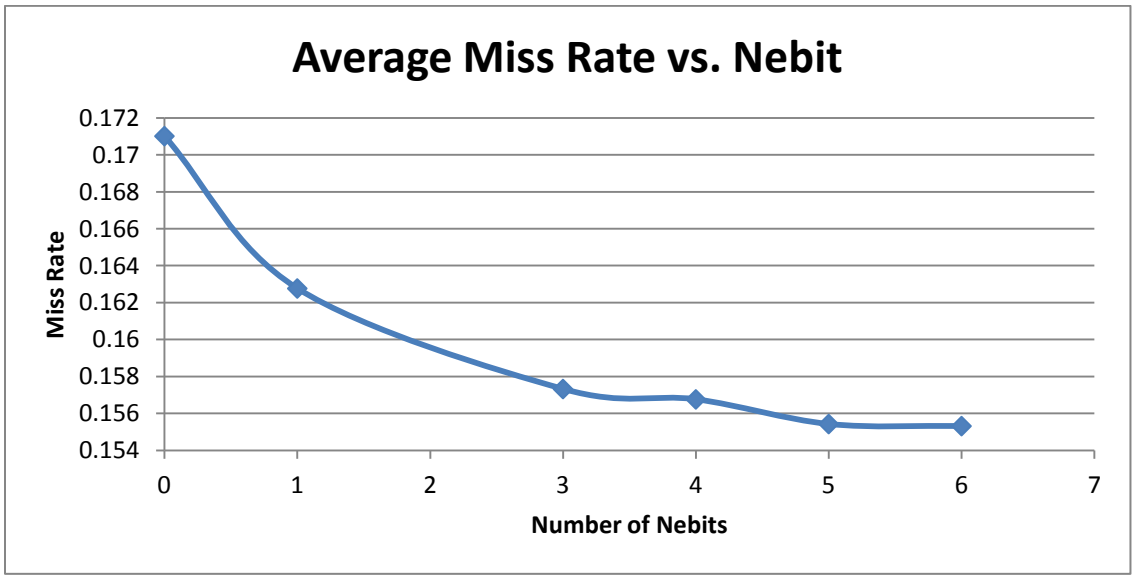


We expect that results would change in this manner as cache size becomes larger. It is important to note a special case revealed by the lbm benchmark. In this benchmark, the dynamic matching at larger cache sizes allowed for better cache use than did a standard LRU replacement algorithm. Since dynamic mapping allows for any cache line to hold any data, it is likely that the cache could be better utilized to increase performance at larger cache sizes. This would imply that the LRU cache was not utilizing all of its cache lines to their full extent.

Newcache – nebit influence

Within Newcache, it is important to observe the miss rates that result from increasing the number of nebits.





The most significant performance benefit occurs between 0 and 1 nebits, and increasing the number of nebits consistently decreases the miss rate, but with diminishing returns. Some may find this surprising, since addition of one nebit decreases the chance of two indexes matching by 100%. However, this does not result in a significant reduction of cache misses at higher number of nebits, since replacement is not based on index matching but instead relies on available cache mappings. Each cache miss will result in replacement of a cache line if no empty mappings are present. The improvement in performance results from the avoidance of matching index cases.

Conclusions

Newcache provides a unique and secure architecture without significant loss in performance. This paper has touched the surface of Newcache's performance in a single-threaded environment. However, significant expansion can be done to explore speed and the soundness of Newcache's security (see George Touloumes' paper). Additionally, tests should be performed to characterize the power consumption of this improved cache, since a data addressed lookup is required to determine mappings in the cache (an operation that is not free of cost). As evidenced by the results of this paper, Newcache provides a secure solution to cache side-channel attacks without a significant increase in cache miss rates in a single-threaded environment.

References

1. Z. Wang and R. B. Lee, “New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks,” in Proceedings of 34th Annual International Symposium on Computer Architecture, ISCA 07, New York, pp. 494-505.
2. Z. Wang and R. Lee, “A Novel Cache Architecture with Enhanced Performance and Security,” in proceeding of Microarchitecture, 2008. MICRO-41, pp. 83-93
3. "Gem5 Repository." *Gem5*. Web. 26 Apr. 2013. [http://www.m5sim.org/]
4. M. Gebhart, J. Hestness, E. Fatehi, P. Gratz, S. W. Keckler, “Running PARSEC 2.1 on M5 Version 1.0,” Technical Report TR-09-32. [http://www.cs.utexas.edu/~parsec_m5/TR-09-32.pdf]
5. "SPEC CPU2006 Documentation." *SPEC CPU2006 Documentation*. Web. 28 Apr. 2013. [http://www.spec.org/cpu2006/Docs/]

Appendix

Running PARSEC 2.1 benchmark on Gem5

The PARSEC 3.0 net benchmarks netferret, netstreamcluster, and netdedup all have non-net counterparts that can be used as a substitute if a client-server relationship is too hard to set up. The challenge of running PARSEC 2.1 on Gem5 is that the tests themselves do not fit in a standard disk image. Instead, a larger disk image must be created. The following steps outline the processes of making a larger disk image.

The University of Texas has done previous work to get PARSEC running on x86. The following resources can be used:

- Pre-compiled X86 kernel: [http://www.cs.utexas.edu/~parsec_m5/x86_64-vmlinux-2.6.28.4-smp]
- linux-bigswap2.img: [http://www.m5sim.org/dist/current/m5_system_2.0b3.tar.bz2]
- X86 full system image: [http://www.m5sim.org/dist/current/x86/x86-system.tar.bz2].
- Parsec pre-compiled binaries: [http://parsec.cs.princeton.edu/download/2.1/binaries/parsec-2.1-amd64-linux.tar.gz]

It is important to note that the full system image is not large enough to hold the benchmarks of interest, so to create a larger disk image, Gem5 supplies an application: gem5img.py in gem5/util/. Using the "init" option, the application will create a mountable directory with a specified size. For example, a 2GB disk image can be created with the following command:

```
./gem5img.py init x86-parsec.img 2048
```

Transferring the x86 system image to this larger disk image is simple (note that these steps must be performed with root access):

1. Create two mount points

   ```
   sudo mkdir –p /mnt/large
   sudo mkdir –p /mnt/linux-x86
   ```

2. Mount the large image and the linux-x86 image

   ```
   sudo mount –o loop,offset=32256 large.img /mnt/large
   sudo mount –o loop,offset=32256 linux-x86.img /mnt/linux-x86
   ```

3. Copy the content of /mnt/linux-x86/ to /mnt/large/

   ```
   sudo cp –rf /mnt/linux-x86* /mnt/large
   ```

4. Copy the necessary binaries for the desired tests
5. Unmount both images

   ```
   sudo umount /mnt/linux-x86
   sudo umount /mnt/large
   ```

Finally, this disk image can be placed in the Gem5 disks directory and be booted from. However, in our study, we were unable to execute binaries placed on this image due to a dependency issue that could not be solved with the allotted time and resources. In theory, if the proper dependencies were placed on this same disk image, the benchmarks could be run in Gem5.

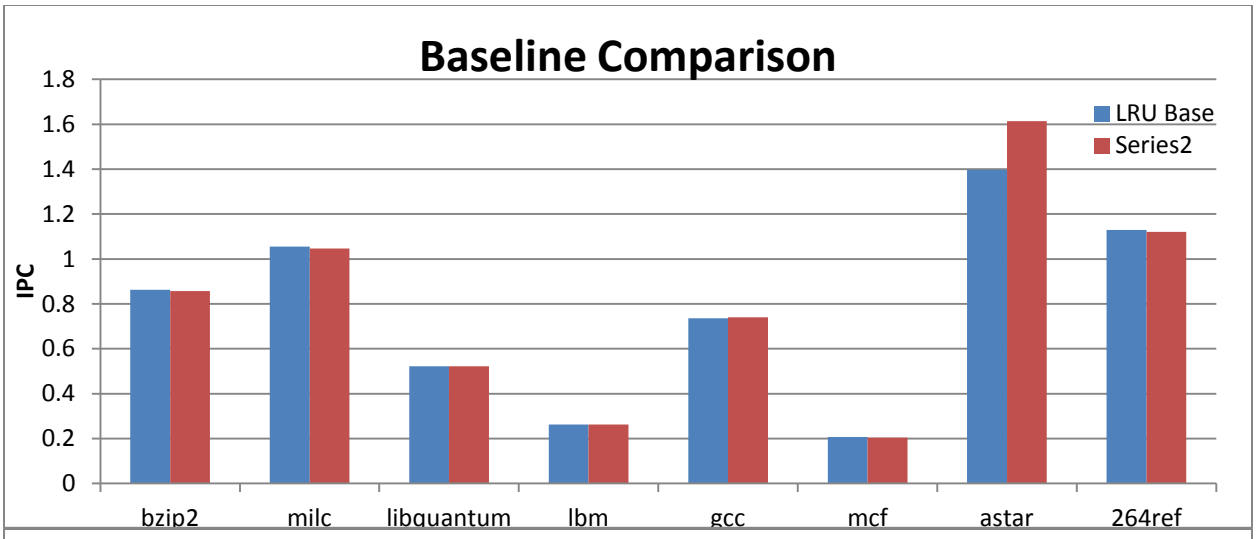
Unfortunately, this route was also abandoned due to lack of time. In theory, the project could resume given a later deadline.

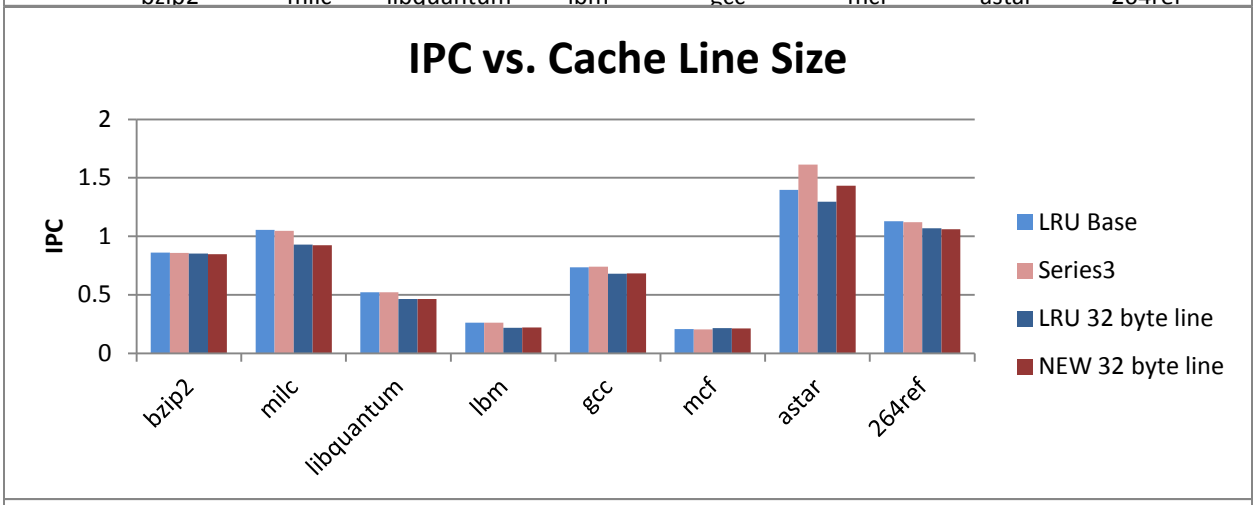
Running PARSEC 3.0 netapps on Gem5

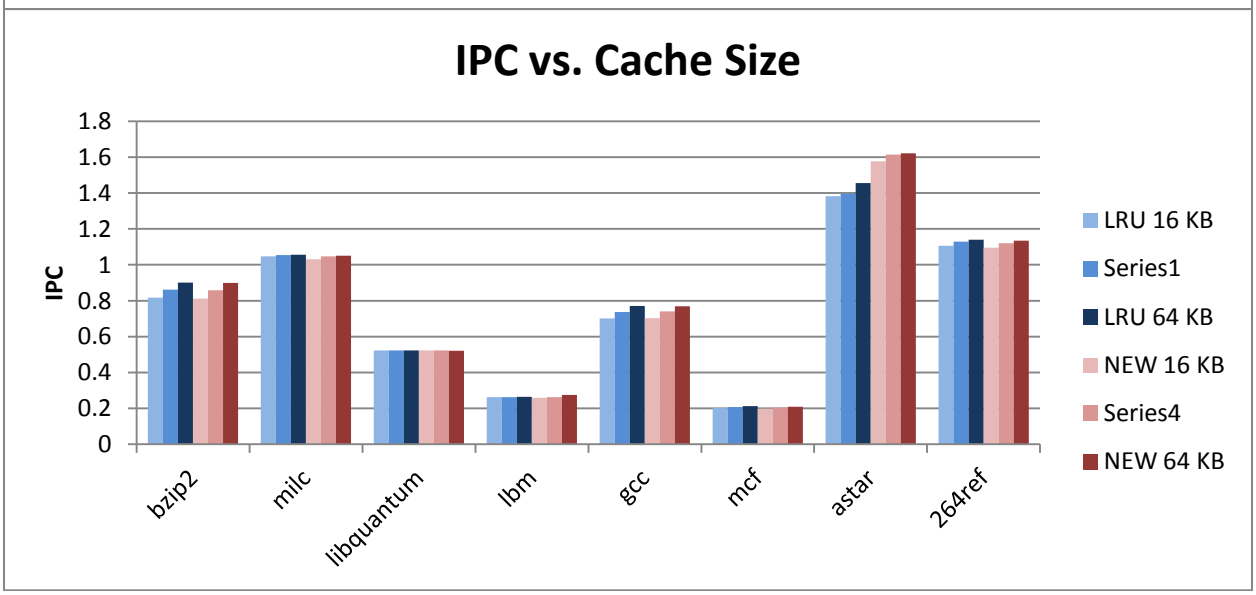
The three netapp benchmarks provided by PARSEC 3.0 include netferret, netdedup, and netstreamcluster. Numerous challenges arose when attempting to run these benchmarks, the most significant of which was our inability to statically build the benchmarks. Due to time constraints and a lack of sufficient computing resources, this route was abandoned in this experiment. In theory, with more time and sufficient resources, this project could resume.

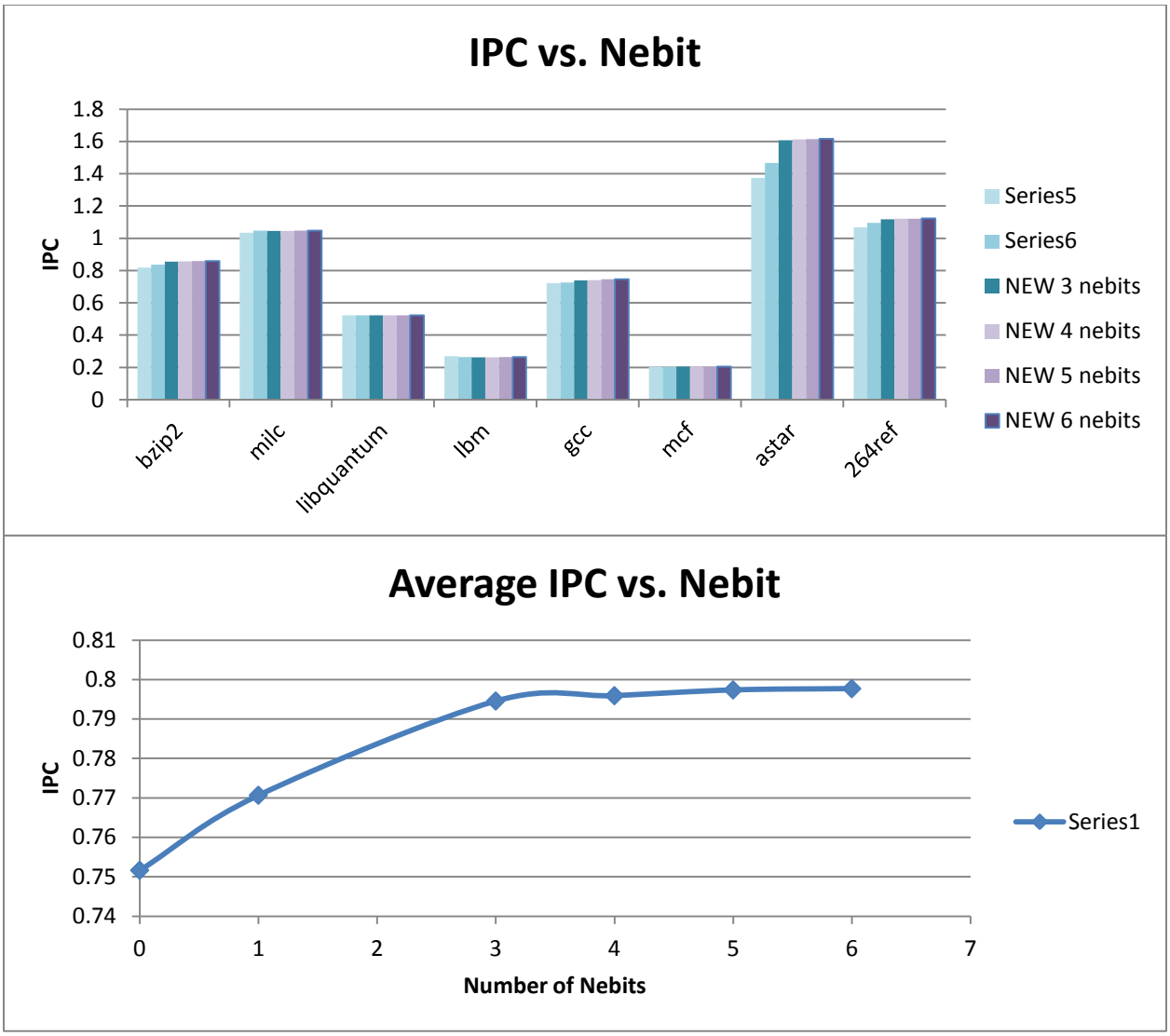
Instruction per Cycle Performance

For interested parties, illustrations of performance as measured by IPC are below. Relative performance is strongly correlated with miss rate.


Baseline Comparison
IPC
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
bzip2
milc
libquantum
lbm
gcc
mcf
astar
264ref
LRU Base
Series2


IPC vs. Cache Line Size
IPC
2
1.5
1
0.5
0
bzip2
milc
libquantum
lbm
gcc
mcf
astar
264ref
LRU Base
Series3
LRU 32 byte line
NEW 32 byte line


IPC vs. Cache Size
IPC
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
bzip2
milc
libquantum
lbm
gcc
mcf
astar
264ref
LRU 16 KB
Series1
LRU 64 KB
NEW 16 KB
Series4
NEW 64 KB


IPC vs. Nebit
IPC
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
bzip2
milc
libquantum
lbm
gcc
mcf
astar
264ref
Series5
Series6
NEW 3 nebits
NEW 4 nebits
NEW 5 nebits
NEW 6 nebits
Average IPC vs. Nebit
IPC
0.81
0.8
0.79
0.78
0.77
0.76
0.75
0.74
0
1
2
3
4
5
6
7
Number of Nebits
Series1