Performance Measurement and Security Testing of a Secure Cache Design

Hao Wu

Master's Thesis
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Master of Science in Engineering

Recommended for Acceptance
by the Department of
Electrical Engineering
Adviser: Professor Ruby B. Lee

June 2015

Abstract

A side channel attack exploits information leaked out by the physical implementation of a cryptosystem. Cache side channel attacks are attacks based on the processor's cache access mechanisms, from which the attacker can infer extra information (e.g. secret key bits etc.) to break the cryptosystem. Software solutions to mitigate information leakage of cache side channels have been proposed, but they alone are not enough to defend against this kind of attack. Adding security properties into cache design can inhibit the root causes of cache side channels. For instance, Newcache is a secure cache design that is reported to have performance and power efficiency comparable to regular set-associative caches through the implementation of dynamic memory-cache remapping and eviction randomization. However, these cache designs have not had their performance verified experimentally under full system architectural simulation for current cloud computing environments.

In this thesis, we carefully selected some commonly-used cloud server benchmarks, and did a thorough performance measurement (e.g. IPC, Cache Miss Rate etc.) of Newcache used as a data cache, an L2 cache and an instruction cache for these cloud server benchmarks on the gem5 simulator. The results show that for the L1 data cache and L1 instruction cache, Newcache has performance comparable with normal set-associative caches. Newcache as L2 cache has higher local and global L2 miss rates compared with a set-associative cache as L2 cache. However, the overall performance of the program execution, in terms of Instructions Per Cycle (IPC), is not impacted.

Furthermore, previous research had shown that Newcache can mitigate data cache side channel attacks. However, no experiment has ever been done to show Newcache's ability in defending against instruction cache side channel attacks. In this thesis, we also did security testing experiments on Newcache used as an instruction cache. The results show that Newcache can thwart side-channel attacks targeting set-associative instruction caches. In addition, we did a detailed security analysis on Newcache as instruction cache.

Acknowledgements

First and foremost, I would like to thank my Thesis Advisor, Prof. Ruby B. Lee, for her continuous assistance and support during the semester just after I switched my research group, and in the following year. Without her support, I would not even have a chance to continue my graduate study in Princeton, nor would I be able to finish this Master Thesis. I would like to thank her for giving me the opportunity to study and do research in her PALMS group. Even on sabbatical, Prof. Lee still spent a lot of time helping me gain a decent understanding of different research topics and steering my research in the right direction. I would also like to thank current members of the PALMS research group (Fangfei, Tianwei and Pramod) for the time they spent discussing the research. In particular, Fangfei was a great mentor, who helped me through the starting phase in PALMS group and filled in the voids I lack in system and security research. Moreover, Prof. Lee and all group members showed great concerns for my health after I went through an acute appendicitis surgery the first semester here.

I am really grateful to Gang Hu, my former high school classmate and now a graduate student in Columbia, for his insightful discussions about the implementation details in my research. I also want to express my gratitude to Yinqian Zhang from UNC-Chapel Hill for his generous sharing of research codes, Maggie Ye for bringing me happiness and enthusiasm, and Liang Dai, my roommate, for just being my roommate since my 2nd year graduate study.

I would not have achieved any of this without the support of my parents. They tried their best to provide their only child with all the opportunities I could ever ask for. I would like to express my gratitude again for their unconditional love and encouragement.

Contents

Abstract . . . iii

Acknowledgements . . . iv

List of Tables . . . viii

List of Figures . . . xi

1 Introduction **1**

2 Background and Past Work **4**

2.1 Cache Side Channel Attacks . . . 4

2.1.1 Access-Driven Attack: Prime and Probe . . . 4

2.1.2 Trace-Driven Attack and Timing-driven Attack . . . 8

2.2 Software and Hardware Solutions . . . 11

2.3 Newcache . . . 11

2.3.1 The Original Newcache [42] . . . 12

2.3.2 The Modified F. Liu's Newcache [35] . . . 15

2.3.3 Security Analysis . . . 16

3 NewCache Performance Measurement and Analysis for Cloud Server Benchmarks **18**

3.1 Gem5 Simulation Methodology . . . 18

3.2 Cloud Server Benchmarks Description and Selection . . . 20

3.2.1 Web Server and Client . . . 20

3.2.2 Database Server and Client . . . 21

3.2.3 Mail Server and Client . . . 21

3.2.4 File Server and Client . . . 22

3.2.5 Streaming Server and Client . . . 23

3.2.6 Application Server and Client 24
3.2.7 Benchmark Summary 25
3.3 Testing Results: Newcache as L1 Data Cache 25
3.3.1 IPC 25
3.3.2 DCache Miss Rate 28
3.3.3 Global L2 Miss Rate 30
3.4 Testing Results: Newcache as L2 Cache 31
3.4.1 IPC 31
3.4.2 DCache Miss Rate 33
3.4.3 Local L2 Miss Rate 34
3.4.4 Global L2 Miss Rate 37
3.5 Testing Results: Newcache as Both L1 Data Cache and L2 Cache 38
3.5.1 IPC 38
3.5.2 DCache Miss Rate 39
3.5.3 Local L2 Miss Rate 40
3.5.4 Global L2 Miss Rate 40
3.6 Testing Results: Newcache as L1 Instruction Cache (Part A) 41
3.6.1 IPC 42
3.6.2 ICache Miss Rate 42
3.6.3 Local L2 Miss Rate for Instructions 43
3.6.4 Global L2 Miss Rate for Instructions 44
3.7 Testing Results: Newcache as L1 Instruction Cache (Part B) 44
3.7.1 IPC 45
3.7.2 ICache Miss Rate 45
3.7.3 Local L2 Miss Rate for Instructions 46
3.7.4 Global L2 Miss Rate for Instructions 47
3.8 Testing Results: Newcache as L1 Instruction Cache (Part C) 48
3.8.1 IPC 48
3.8.2 ICache Miss Rate 48
3.8.3 Local L2 Miss Rate for Instructions 49
3.8.4 Global L2 Miss Rate for Instructions 50
3.9 Chapter Summary 51

4 Security Analysis of Newcache as Instruction Cache 53

4.1 Reconstruction of Instruction Cache Side-Channel Attack 53

4.1.1 Instruction Cache PRIME and PROBE 53

4.1.2 Clean PROBE-ing Result 55

4.1.3 Unique ICache Footprints Left by Square, Mult and Reduce 57

4.1.4 Modular Exponentiation Experiments 61

4.2 Newcache in Defending against Instruction Cache Side-Channel Attack 64

4.2.1 Clean PROBE-ing Result 64

4.2.2 ICache footprint left by Square, Mult and Reduce 67

4.2.3 More Analysis 71

5 Closing Words 76

A Data for Newcache Performance Measurement 77

A.1 Data for Newcache as L1 Data Cache 77

A.2 Data for Newcache as L2 Cache 81

A.3 Data for Newcache as Both L1 Data Cache and L2 Cache 86

A.4 Data for Newcache as L1 Instruction Cache (Part A) 88

A.5 Data for Newcache as L1 Instruction Cache (Part B) 91

A.6 Data for Newcache as L1 Instruction Cache (Part C) 94

B Performance Measuring Scripts and Data Collecting Scripts 98

B.1 Sample Performance Measuring Script for Dbench_Write 98

B.2 Python Script to Generate Performance Measuring Scripts 99

B.3 Sample Server-Side and Client-Side Configuration Scripts for Dbench_Write on Gem5 103

B.4 Sample Performance Stats Collecting Script 106

Bibliography 109

List of Tables

3.1 Metrics Definitions 19

3.2 Gem5 Base CPU and Cache Configurations 20

3.3 Summary of Cloud Server Benchmarks 25

3.4 Newcache as L1-DCache: Different Cache Sizes' IPC, and IPC Increase Relative to SA 27

3.5 Newcache as L1-DCache: Different Associativity and Nebit's IPC, and IPC Increase Relative to SA-8way 28

3.6 Newcache as L1-DCache: Different Cache Sizes' DCache Miss Rate Increase Relative to SA 29

3.7 Newcache as L1-DCache: Different Associativity and Nebit's DCache Miss Rate Increase Relative to SA-8way 29

3.8 Newcache as L1-DCache: Different Cache Sizes' Global L2 Miss Rate Increase Relative to SA 30

3.9 Newcache as L1-DCache: Different Associativity and Nebit's Global L2 Miss Rate Increase Relative to SA-8way 31

3.10 Newcache as L2 Cache: Different Cache Sizes' IPC, and IPC Increase Relative to SA 32

3.11 Newcache as L2 Cache: Different Associativity and Nebit's IPC Increase Relative to SA-8way 32

3.12 Newcache as L2 Cache: Different Cache Sizes' DCache Miss Rate Increase Relative to SA 34

3.13 Newcache as L2 Cache: Different Associativity and Nebit's DCache Miss Rate Increase Relative to SA-8way 34

3.14 Newcache as L2 Cache: Different Cache Sizes' Local L2 Miss Rate Increase Relative to SA 35

3.15 Newcache as L2 Cache: Different Associativity and Nebit's Local L2 Miss Rate Increase Relative to SA-8way 36

3.16 Newcache as L2 Cache: Different Cache Sizes' Global L2 Miss Rate Increase Relative to SA 37
3.17 Newcache as L2 Cache: Different Associativity and Nebit's Global L2 Miss Rate Increase Relative to SA-8way 38
3.18 Newcache as Both L1-DCache and L2 Cache: IPC Increase Relative to l1d.SA-l2.SA 38
3.19 Newcache as Both L1-DCache and L2 Cache: DCache Miss Rate Increase Relative to l1d.SA-l2.SA 39
3.20 Newcache as Both L1-DCache and L2 Cache: Local L2 Miss Rate Increase Relative to l1d.SA-l2.SA 40
3.21 Newcache as Both L1-DCache and L2 Cache: Global L2 Miss Rate Increase Relative to l1d.SA-l2.SA 41
3.22 Newcache as L1-ICache: IPC, and IPC Increase Relative to all.SA.base 42
3.23 Newcache as L1-ICache: ICache Miss Rate Increase Relative to all.SA.base 42
3.24 Newcache as L1-ICache: Local L2 Instruction Miss Rate Increase Relative to all.SA.base 43
3.25 Newcache as L1-ICache: Global L2 Instruction Miss Rate Increase Relative to all.SA.base 44
3.26 Newcache as L1-ICache: IPC, and IPC Increase Relative to all.SA.base 45
3.27 Newcache as L1-ICache: ICache Miss Rate Increase Relative to all.SA.base 46
3.28 Newcache as L1-ICache: Local L2 Instruction Miss Rate Increase Relative to all.SA.base 46
3.29 Newcache as L1-ICache: Global L2 Instruction Miss Rate Increase Relative to all.SA.base 48
3.30 Newcache as L1-ICache: IPC, and IPC Increase Relative to all.SA.base 48
3.31 Newcache as L1-ICache: ICache Miss Rate Increase Relative to all.SA.base 50
3.32 Newcache as L1-ICache: Local L2 Instruction Miss Rate Increase Relative to all.SA.base 50
3.33 Newcache as L1-ICache: Global L2 Instruction Miss Rate Increase Relative to all.SA.base 51

4.1 Simulator Configurations 64
4.2 Classification Matrix for 8-way SA Cache with LRU Replacement Algorithm used as ICache , with Fixed Interval for Victim to Execute, between Prime-Probe Phases . . 74
4.3 Classification Matrix for Newcache with Nebit k=4 used as ICache , with Complete S, M or R Operation Performed, between Prime-Probe Phases 75

4.4 Classification Matrix for Newcache with Nebit k=4 used as ICache, with Fixed Interval for Victim to Execute, between Prime-Probe Phases 75

A.1 Newcache as L1-DCache: IPC for Different Cache Sizes 77
A.2 Newcache as L1-DCache: IPC for Different Associativity and Nebit 78
A.3 Newcache as L1-DCache: DCache Miss Rate for Different Cache Sizes 78
A.4 Newcache as L1-DCache: DCache Miss Rate for Different Associativity and Nebit . 79
A.5 Newcache as L1-DCache: Global L2 Miss Rate for Different Cache Sizes 79
A.6 Newcache as L1-DCache: Global L2 Miss Rate for Different Associativity and Nebit 80
A.7 Newcache as L2 Cache: IPC for Different Cache Sizes 81
A.8 Newcache as L2 Cache: IPC for Different Associativity and Nebit 82
A.9 Newcache as L2 Cache: DCache Miss Rate for Different Cache Sizes 82
A.10 Newcache as L2 Cache: DCache Miss Rate for Different Associativity and Nebit . . 83
A.11 Newcache as L2 Cache: Local L2 Miss Rate for Different Cache Sizes 83
A.12 Newcache as L2 Cache: Local L2 Miss Rate for Different Associativity and Nebit . . 84
A.13 Newcache as L2 Cache: Global L2 Miss Rate for Different Cache Sizes 84
A.14 Newcache as L2 Cache: Global L2 Miss Rate for Different Associativity and Nebit . 85
A.15 Newcache as Both L1-DCache and L2 Cache: IPC 86
A.16 Newcache as Both L1-DCache and L2 Cache: DCache Miss Rate 86
A.17 Newcache as Both L1-DCache and L2 Cache: Local L2 Miss Rate 87
A.18 Newcache as Both L1-DCache and L2 Cache: Global L2 Miss Rate 87
A.19 Newcache as L1-ICache: IPC 88
A.20 Newcache as L1-ICache: ICache Miss Rate 89
A.21 Newcache as L1-ICache: Local L2 Instruction Miss Rate 89
A.22 Newcache as L1-ICache: Global L2 Instruction Miss Rate 90
A.23 Newcache as L1-ICache: IPC 91
A.24 Newcache as L1-ICache: ICache Miss Rate 92
A.25 Newcache as L1-ICache: Local L2 Instruction Miss Rate 92
A.26 Newcache as L1-ICache: Global L2 Instruction Miss Rate 93
A.27 Newcache as L1-ICache: IPC 94
A.28 Newcache as L1-ICache: ICache Miss Rate 95
A.29 Newcache as L1-ICache: Local L2 Instruction Miss Rate 96
A.30 Newcache as L1-ICache: Global L2 Instruction Miss Rate 97

List of Figures

2.1 Prime and Probe 4

2.2 Instruction Cache Prime and Probe 7

2.3 Timing Characteristic Charts for Byte 0 for Known Key K 10

2.4 Timing Characteristic Charts for Byte 0 for Unknown Key K' 10

2.5 Newcache Block Diagram 1 12

2.6 Cache Access Handling Process 1 (SecRAND) 14

2.7 Newcache Block Diagram 2 15

2.8 Cache Access Handling Process 2 16

3.1 x86 Dual System 19

3.2 Newcache as L1-DCache: IPC for Different Cache Sizes 26

3.3 Newcache as L1-DCache: IPC for Different Associativity and Nebit 27

3.4 Newcache as L1-DCache: DCache Miss Rate for Different Cache Sizes 28

3.5 Newcache as L1-DCache: DCache Miss Rate for Different Associativity and Nebit . 29

3.6 Newcache as L1-DCache: Global L2 Miss Rate for Different Cache Sizes 30

3.7 Newcache as L1-DCache: Global L2 Miss Rate for Different Associativity and Nebit 31

3.8 Newcache as L2 Cache: IPC for Different Cache Sizes 32

3.9 Newcache as L2 Cache: IPC for Different Associativity and Nebit 33

3.10 Newcache as L2 Cache: DCache Miss Rate for Different Cache Sizes 33

3.11 Newcache as L2 Cache: DCache Miss Rate for Different Associativity and Nebit . . 34

3.12 Newcache as L2 Cache: Local L2 Miss Rate for Different Cache Sizes 35

3.13 Newcache as L2 Cache: Local L2 Miss Rate for Different Associativity and Nebit . . 36

3.14 Newcache as L2 Cache: Global L2 Miss Rate for Different Cache Sizes 37

3.15 Newcache as L2 Cache: Global L2 Miss Rate for Different Associativity and Nebit . 38

3.16 Newcache as Both L1-DCache and L2 Cache: IPC 39

3.17 Newcache as Both L1-DCache and L2 Cache: DCache Miss Rate 39
3.18 Newcache as Both L1-DCache and L2 Cache: Local L2 Miss Rate 40
3.19 Newcache as Both L1-DCache and L2 Cache: Global L2 Miss Rate 41
3.20 Newcache as L1-ICache: IPC 42
3.21 Newcache as L1-ICache: ICache Miss Rate 43
3.22 Newcache as L1-ICache: Local L2 Instruction Miss Rate 43
3.23 Newcache as L1-ICache: Global L2 Instruction Miss Rate 44
3.24 Newcache as L1-ICache: IPC 45
3.25 Newcache as L1-ICache: ICache Miss Rate 46
3.26 Newcache as L1-ICache: Local L2 Instruction Miss Rate 47
3.27 Newcache as L1-ICache: Global L2 Instruction Miss Rate 47
3.28 Newcache as L1-ICache: IPC 49
3.29 Newcache as L1-ICache: ICache Miss Rate 49
3.30 Newcache as L1-ICache: Local L2 Instruction Miss Rate 50
3.31 Newcache as L1-ICache: Global L2 Instruction Miss Rate 51

4.1 Typical Memory Address 53
4.2 32kB Contiguous Memory Chunk for PRIME-ing 55
4.3 Clean PROBE-ing Result for 32kB 8-way Set-Associative Cache 56
4.4 Wrong PROBE-ing Result for 32kB 8-way Set-Associative Cache 56
4.5 ICache Footprint Left by Square 57
4.6 Square's Footprint after Subtracting the Clean Prime-Probe 58
4.7 ICache Footprint Left by Mult 59
4.8 Mult's Footprint After Subtracting the Clean Prime-Probe 59
4.9 ICache Footprint Left by Reduce 60
4.10 Reduce's Footprint after Subtracting the Clean Prime-Probe 60
4.11 Gray Scale Matrix I 61
4.12 Gray Scale Matrix II 62
4.13 Gray Scale Matrix III 63
4.14 Clean PROBE-ing Result I for 32kB Newcache 65
4.15 Clean PROBE-ing Result II for 32kB Newcache 66
4.16 Averge PROBE Time for 800 Clean Prime-Probe 66
4.17 Number of PRIME-ed Cache-lines 67

4.18 PROBE Result I for Square 68
4.19 PROBE Result II for Square 68
4.20 Average PROBE Result for Square 68
4.21 PROBE Result I for Mult 69
4.22 PROBE Result II for Mult 69
4.23 Average PROBE Result for Mult 69
4.24 PROBE Result I for Reduce 70
4.25 PROBE Result II for Reduce 70
4.26 Average PROBE Result for Reduce 70
4.27 Illustration of the Attack (a) 72
4.28 Illustration of the Attack (b) 72

Chapter 1

Introduction

Cryptography uses ciphers to encrypt our data so as to protect our secret information. Strong cryptography makes it computationally impossible for an attacker to infer the key bits by using a brute-force attack or exploiting the weakness in the algorithm. However, in the real world, attacks can involve multiple parts of the system: algorithm level, software level, hardware level etc. Security problems may occur when components at different levels interact with each other. Side channel attacks on hardware can exploit information leaked out by the physical implementation of the cryptosystem. For example, timing, power consumption, and electromagnetic radiation can provide attackers with extra information to break the cryptosystem [32, 27, 31].

Cache side channel attacks are side channel attacks based on the processor's cache access mechanisms [26, 25, 44, 38, 23, 29, 43, 45, 22]. One class of cache side channel attacks is the timing attack. Some cryptographic operations frequently look up data in precomputed tables in memory, and the addresses of memory accesses are dependent on the secret key. For traditional caches and memory mechanisms, the timing difference between cache misses and cache hits is large. The attacker can thus use some techniques to learn the addresses of the accessed memories, and infer secret key bits. Bernstein's AES attack [26] and Bonneau's AES attack [23] are timing attacks. Bernstein's attack is mainly due to deterministic cache contention, while Bonneau's attack is based on reuse of data across victim and attacker processes [36]. Another cache side channel attack is the access-driven attack, the representative techniques of which are *Prime-Probe* [25, 44] and *Flush-Reload* [29, 43, 45]. I mainly focus on *Prime-Probe* attacks in this thesis. *Prime-Probe* involves an attacker's exploitation of distinguished cache footprints left by different cryptographic operations. In this attack, cryptographic operations' sequence contains information related to secret key bits. Different cryptographic operations will access different data and use different instructions residing in different memory regions, which may leave a distinguished cache footprint after the operation is executed. Percival's RSA attack [25] and Zhang's ElGamal attack [44] are this kind of cache side channel attack. Percival's RSA attack is based on the data cache, while Zhang's attack is based on the instruction cache. Cache side channel attacks are a serious security consideration; they can be formed on platforms ranging from smart cards to cloud servers.

Different solutions to mitigate information leakage of cache side channels have been proposed, and most of them are software-based techniques. Earlier software solutions (mostly on AES) improve security but may lead to significant performance degradation [24, 38]. Recently, by optimizing the implementation of the AES algorithm [33], by extending existing Instruction Set Architecture (ISA) [37] to support dedicated AES instructions, or by adding general-purpose parallel table lookup instructions that can accelerate AES (and other algorithms) [34], people have achieved both high performance and security for AES. However, these solutions are either specific to AES or have not been widely deployed. Cache based hardware techniques have also been proposed. Some cache approaches use private partitions or locked regions for security data [39, 41], which prevents undesired cache evictions by other cache lines. However, these methods can result in cache under-utilization problems. Some other solutions use a randomized mapping approach [41, 42], which randomizes cache interference by doing random permutations of the memory-cache mapping or randomizing cache evictions. Furthermore, the randomization strategies are reported to have little or no performance degradation.

Newcache is a representative cache design using the randomization approach; it was first designed by Z. Wang and R. Lee [42] and then improved by F. Liu and R. Lee [35]. It employs the techniques of dynamic memory-cache remapping and eviction randomization. According to Z. Wang's analysis, due to Newcache's mechanism, it can prevent existing AES attacks and RSA attacks [42]. Moreover, F.Liu's improved version of Newcache [35] can prevent redesigned AES and RSA attacks targeting the original Newcache. However, these AES attacks and RSA attacks both target the data cache side channel; no one has ever done experiments to see if Newcache can mitigate instruction cache side channel attacks, a representative of which is Zhang's Elgamal attack [44]. Also, a full system architectural simulation under gem5 [20] needs to be done to see if Newcache can achieve performance comparable to the conventional set-associative (SA) cache (e.g. IPC, Cache Miss Rate etc.) by running representative and commonly used benchmarks on the simulator. The main contributions of this work include:

- A careful selection of the commonly-used cloud server benchmarks that are suitable for testing under the full system architecture simulator gem5 [20].

- A thorough measurement of the performance (e.g. IPC, Cache Miss Rate etc.) of Newcache used as a data cache, an L2 cache and an instruction cache, and a detailed performance analysis and comparison with a conventional SA cache for these cloud server benchmarks on gem5.

- Doing experiments and security analysis on Newcache as an instruction cache to see if Newcache can prevent Zhang's side channel attack [44] on the instruction cache.

Chapter 2 provides necessary background information and gives a brief introduction to some representative cache side channel attacks, mitigation approaches and Newcache's cache design and mechanism. In Chapter 3, we use Newcache as a data cache, an L2 cache and an instruction cache, and evaluate the performance of Newcache for cloud server benchmarks under gem5. In Chapter 4, we first reconstruct Zhang's attack by modifying the necessary operations in libgcrypt 1.5.3 library [11] under a conventional SA cache configured memory system. We then do experiments to see if by replacing the instruction cache with Newcache, we can prevent Zhang's side channel attacks targetting conventional SA instruction caches. Finally we analyze more security issues of Newcache used as an instruction cache. Chapter 5 provides conclusions and future work.

Chapter 2

Background and Past Work

2.1 Cache Side Channel Attacks

Cache side channel attacks can be categorized into access-driven, trace-driven and timing-driven attacks. They will be described in detail in the following sub-sections.

2.1.1 Access-Driven Attack: Prime and Probe

For an access-driven attack, an attacker can use a technique called *Prime and Probe* shown in Figure 2.1.

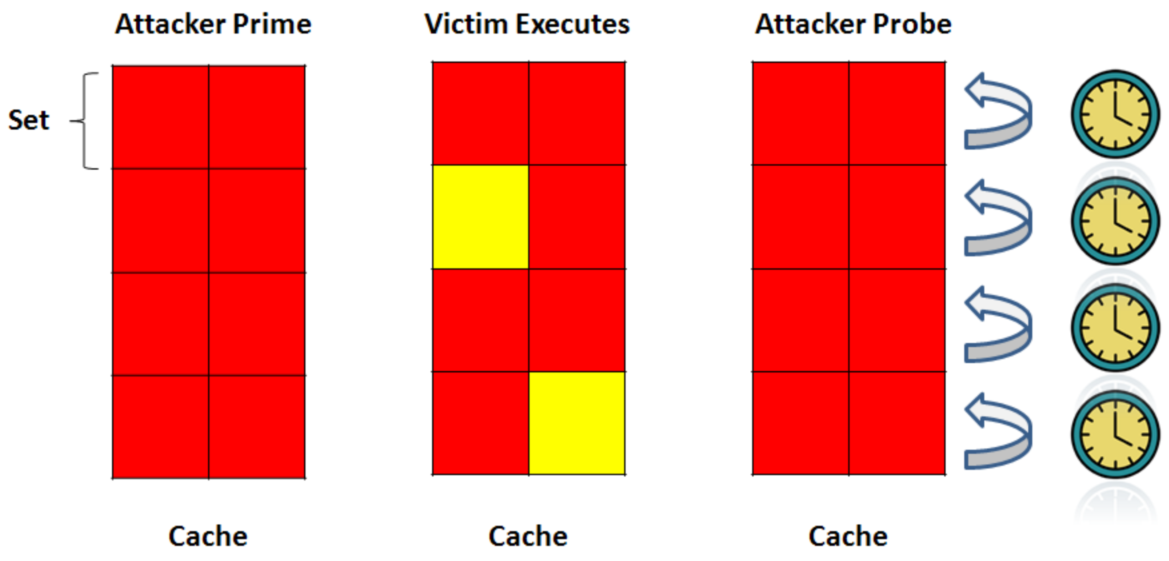


Figure 2.1: Prime and Probe

The figure shows a cache with 4 sets, and each set has 2 cache blocks (i.e., 2-way set-associative cache). We have an attacker process and a victim process running on the same processor. The attacker process fills the cache with its own data, which is called a PRIME phase. Then the victim process gains control of the processor and runs. After a certain time interval, the attacker process gains control of the CPU again, and measures the time taken to load its data into the cache for each cache set, which is called a PROBE phase; the PROBE phase actually primes the cache for subsequent observations. If the victim process uses some cache sets during this time interval, some of the attacker's cache lines in these cache sets will be evicted, which causes longer load time during the PROBE phase. The timing information of accessing all the cache sets during the PROBE phase can be represented as a cache footprint. Different cryptographic operations from the victim process will access different data and use different instructions residing in different memory regions, which may leave a distinguished cache footprint after the operation is executed. More importantly, a sequence of cryptographic operations may contain information of secret key bits. Thus an attacker can exploit the *Prime and Probe* technique to learn information about key bits. Percival's RSA attack [25] and Zhang's ElGamal attack [44] are this kind of cache side channel attack. Percival's RSA attack is based on the data cache, while Zhang's attack is based on the instruction cache.

- Percival's RSA attack:

Percival's Attack [25] demonstrates an L1 data cache side-channel attack on the OpenSSL 0.9.7c's [15] implementation of the RSA algorithm. The attack is based on the fact that multiple threads can run simultaneously on modern Simultaneous Multi-Threading (SMT) processors, and thus they can share part of the cache subsystem. The spy process is thus able to observe other concurrent threads' L1 cache accesses.

- Attack Description: In the attack demonstration, the victim process performs an RSA operation, and the spy process runs simultaneously with the victim process. The spy process uses the *Prime and Probe* technique described above to repeatedly load its own data into each cache line. Meanwhile it uses the *rdtsc* instruction to measure the total time for PROBE-ing each cache set. The victim's cache accesses will evict the attacker's data from the cache, causing different time measurements for each set from the attacker's perspective.

- RSA Description: To fully understand how the attack reveals RSA's secret key, we need to understand the RSA algorithm first. RSA is an algorithm for public-key cryptosystems. The method to generate RSA private and public key pairs is briefly described as follow:

- We choose two primes p and q, where $p \neq q$.
- Calculate: $N = p \cdot q$, and $\phi(N) = (p-1)(q-1)$, where $\phi()$ is the Euler function ($\phi(N) =$ the number of positive integers less than N that are relatively prime to N).
- Randomly choose integer e ($1 < e < \phi(N)$), such that e and $\phi(N)$ are coprime.
- Calculate the modular multiplicative inverse of integer e modulo $\phi(N)$ to get d.
- (N, e) is the public key, while (N, d) is the private key.

The encryption and decryption process are straight-forward:

- Encryption: given plaintext m, we do $m^e \equiv c \pmod N$ to get encrypted message c.
- Decryption: given encrypted message c, we do $c^d \equiv m \pmod N$ to get decrypted message m.

RSA's key generation scheme ensures that e and d have this kind of property to do encryption and decryption.

- Attack Analysis: From the previous RSA descryption, we know that the core operation in RSA is modular exponentiation and the exponent is the key. For OpenSSL, it computes a 1024-bit modular exponentiation using two 512-bit modular exponentiations. The goal of the attacker is to get every bit of the private key d used by the victim.

- The main reason that the attacker can extract information from the data cache side-channel is that the modular exponentiation in OpenSSL is implemented with a series of squarings and multiplications (decomposing $x := a^d \ mod \ p$ into a series of $x := a^2 \ mod \ p$ and $x := x \cdot a^{(2k+1)} \ mod \ p$; a here can be either the plaintext m or the encrypted message c). From attacker's perspective, the modular squaring and modular multiplication are easily distinguishable from their unique cache footprints. Thus, from the unique sequence of squaring and multiplications, about 200 bits out of each 512-bit exponent can be obtained [25].

- Moreover, different cache footprints are left behind by the multiplication $x := x \cdot a^{(2k+1)} \ mod \ p$ for different ks. As described above, a cache footprint is the timing information of accessing all the cache sets during the PROBE phase. The multipliers $a^{(2k+1)}$ $(k = 0, 1.., 15)$ are pre-computed at the start of the modular exponentiation. When accessing the multiplier data, the CPU fetches the data into the data cache. Because different multipliers reside in different memory locations, different multiplier data may belong to different data cache sets. So after each multiplication is executed, a distinguished data cache footprint is left. By examining the footprint, we can identify the locations where these multipliers are stored. By oberserving a sequence of these footprints, we may get the information of the sequence of multipliers that are used, from which we can get additional key bits.

- Zhang's Elgamal attack:

Another paper demonstrated the utilization of Cross-VM Side Channels to extract private ElGamal keys [44]. Their work also exploits an access-driven side channel. However this side channel is constructed on the L1 instruction cache, which is originally described by Aciiçmez [22]. In Zhang's attack, the victim process performs Elgamal decryptions using the libgcrypt v1.5.0 cryptographic library [11], and the spy process schedules itself on the same physical CPU so that it can use the *Prime and Probe* technique to collect information leaked out by the victim process.

Prime and Probe for the instruction cache is a bit different from that for the data cache, but has the same idea (See Figure 2.2). Assume the cache is a w-way SA cache, and it has n cache sets, and the cache block is b bytes. Then the cache has size $b \cdot w \cdot n$ bytes. Also memory addresses that are $b \cdot n$ bytes apart will be mapped to the same cache set.

To prime one cache set, we just need to allocate w cache blocks (the red (or dark) blocks in Figure 2.2), and starting from the second cache block, each cache block is $b \cdot n$ bytes away from the previous block. Inside each cache block is the instruction **jmp b · n**, which jumps to the next block, except for the last block, which is a **ret** (return) instruction. When priming the cache set, the attacker's main program jumps (using **call** instruction) to the address of the first block. Then the w cache blocks will be fetched into the cache set one by one, and the last block **ret** returns to the main program. We can use the same mechanism to prime the remaining $n-1$ sets. Basically, we allocate a contiguous memory chunk that has the same size as the instruction cache to prime the whole cache. PROBE-ing is just like PRIME-ing, except the *rdtsc* instruction is used to measure the time to probe each set of the instruction cache.

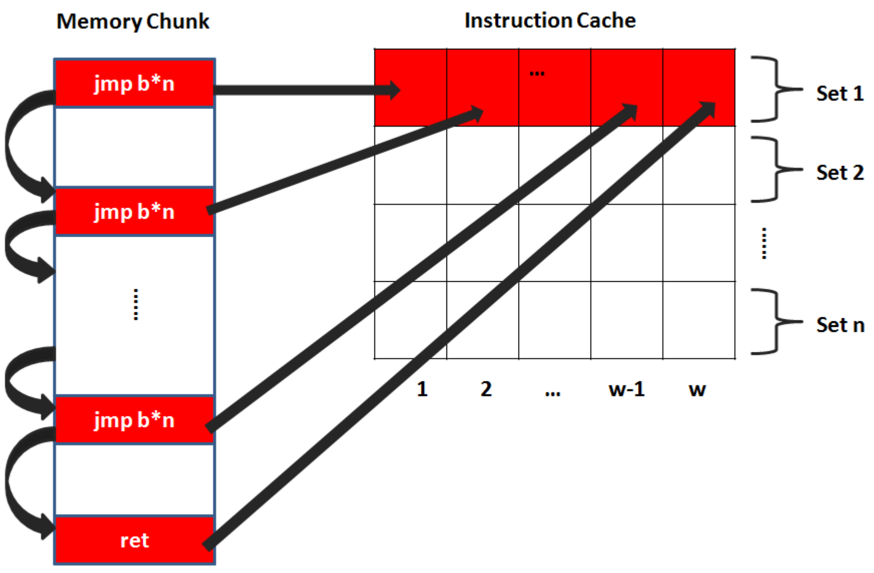


Figure 2.2: Instruction Cache Prime and Probe

In Elgamal decryption, we have the encrypted message (a, b), and we need to use secret key x to do $m = decode(a, b) = \frac{b}{a^x (mod\ p)}$ to get the decrypted message m. The decryption computes the modular exponentiation $a^x\ (mod\ p)$. The modular exponentiation computed in libgcrypt v1.5.3 also uses the square and multiply algorithm. The square and multiply algorithm is depicted in Algorithm 1. They let S, R, M stand for calls to functions Square, ModReduce, Mult, respectively, inside SquareMult.

By observation, the sequence of function calls in one execution of SquareMult will leak information about the exponent *exp*, which is the secret key x here. For example, the sequence

(SRMR)(SR) corresponds to $exp = 110_b = 6$. **SRMR** leaks information $e_2 = 1$, and **SR** gives $e_1 = 0$ (note that the most siginificant bit, here e_3, is always 1; the sequence leaks information from the second most significant bit e_{n-1} to the least siginificant bit e_1). Unfortunately the attacker can use the *Prime and Probe* technique to know the S, R, M sequence, and thus infer the secret key. Because each of the S, R and M functions have their instructions located in different memory addresses; performing these 3 calls will leave the instruction cache with different unique footprints. The attacker can detect which function has been executed through the cache footprint derived from the PROBE-ing phase.

Algorithm 1 SquareMult Algorithm

```
procedure SquareMult($base, exp, mod$)
    Let $e_n, ..., e_1$ be the bits of $exp$, and $e_n = 1$
    $y \leftarrow base$
    for $i = (n-1) \rightarrow 1$ do
        $y \leftarrow$ Square($y$)                    (S)
        $y \leftarrow$ ModReduce($y, mod$)            (R)
        if $e_i = 1$ then
            $y \leftarrow$ Mult($y, base$)            (M)
            $y \leftarrow$ ModReduce($y, mod$)        (R)
        end if
    end for
end procedure
```

In Zhang's work, they use a support vector machine (SVM) to classify the instruction cache PROBE-ed footprints into the corresponding function calls (S, M or R). To make the attack actually work, they have to deal with issues like observation granularity, observation noise etc. Especially when their work elevates the side-channel attack from process-level to virtual machine level, they will have to consider more issues, e.g. core migration, error correction, operation sequence reassembly etc.

2.1.2 Trace-Driven Attack and Timing-driven Attack

For trace-driven attacks and timing-driven attacks, an attacker can get information related to the cache hits and cache misses for the victim process's memory accesses. In trace-driven attacks, the profile of all the cache activities in terms of hit or miss for each memory access can be captured by the attacker. While in timing-driven attacks, an attacker only gets the aggregate profile of the cache activities (e.g. the total execution time for a cipher to do a block encryption), from which the attacker can know approximate cache hits and misses.

In the timing attacks, an attacker can use a technique called *Evict and Time* [38] to introduce some interference to the victim process and infer critical information from the latter timing results. The attacker can frequently trigger a victim block encryption. Before each encryption, the attacker evicts one specific cache set with its own data items, and then measures the encryption time. If the victim process accesses the evicted set and replaces at least one of the attacker's cache blocks, the block encryption time tends to be higher. Bernstein's AES attack [26] is this kind of timing attack. However, instead of the attacker's eviction of a cache set, some data components inside the victim AES encryption process will always evict some fixed cache sets before each encryption.

- Bernstein's AES attack: Bernstein's attack presents a successful extraction of a complete AES key from a networked server computer. The victim is a server-side service program that runs AES encryption. The attacker can be a process requesting encryption service; he can choose different inputs to the service AES encryption software and measure the time of each encryption. The attacker process can perform requests either from a remote machine or in the same machine that runs AES encryption. The victim uses OpenSSL v0.9.7a library [15] to do the AES encryption. To increase encryption speed, four 1024-byte tables T0, T1, T2, T3 are pre-computed from 256-byte tables S and S'. All the subsequent operations, such as sbox-lookup, shift-rows, mix-columns etc., can be accomplished by directly indexing T0, T1, T2, T3. However, the time for these table lookups depends on whether the accessed table entries are present in the cache or not, which means the time for the whole AES computation is correlated with these table lookups. An attacker can detect key-byte values from the distribution of AES encryption timings.

- Attack Summary:

Take AES128 for example, we have key length of 128 bits (16 bytes) and block length of 128 bits (16 bytes). In the preparation phase, the attacker lets the victim use a known key K. He gives a lot of different plaintext block inputs n to the encryption process (victim) and records the time for each AES encryption. Suppose we just consider one byte $n[0]$ (input n has 16 bytes $n[0]...n[15]$). We record all the encryption times for $n[0]$, and get the average time for each $n[0]$ value (byte $n[0]$ has 8 bits, thus 256 possible values) (Figure 2.3 [41]).

Then in the attacking phase, the attacker repeats the previous phase with an unknown key K', and gets the timing characteristics of byte $n[0]$ for K' (Figure 2.4 [41]). The inputs are not necessarily the same as that used in the preparation phase. Figure 2.3 and Figure 2.4 are just one pair of timing characteristic charts; there are 15 other pairs for bytes $n[1]$ to $n[15]$.

During the final key recovery phase, we may first observe from Figure 2.3 that the encryption time for the known key is maximum when $n[0] = 223 = 11011111_b$, and from Figure 2.4 that the time for the unknown key is maximum when $n'[0] = 5 = 00000101_b$. Suppose the known key byte $K[0] = 33 = 00100001_b$, then the attacker can be almost sure that the AES time is maximum when $K[0] \oplus n[0] = 33 \oplus 223 = 00100001_b \oplus 11011111_b = 11111110_b = 254$. Since in AES, $K[0] \oplus n[0] = K'[0] \oplus n'[0]$, the attacker can conclude that the victim's $K'[0] = 254 \oplus n'[0] =$

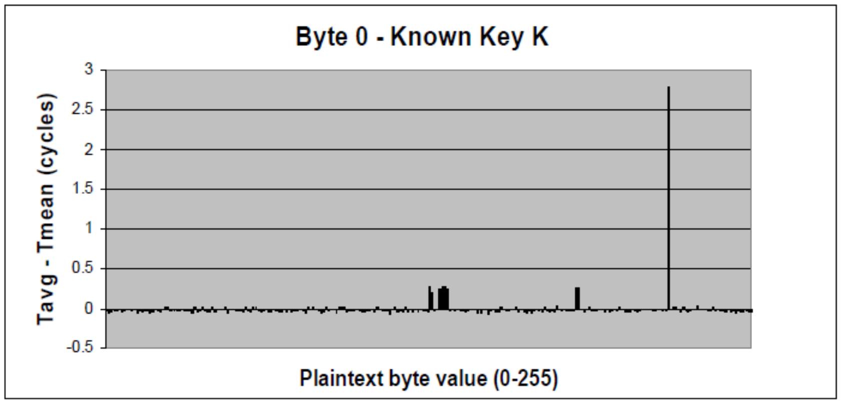


Figure 2.3: Timing Characteristic Charts for Byte 0 for Known Key K

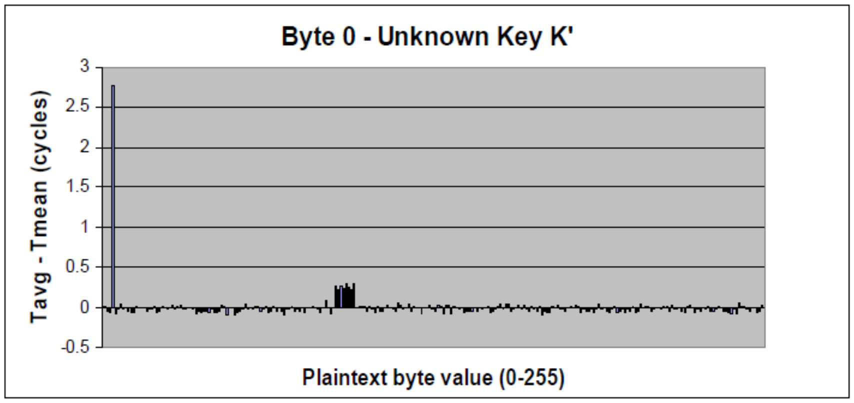


Figure 2.4: Timing Characteristic Charts for Byte 0 for Unknown Key K'

$11111110_b \oplus 00000101_b = 11111011_b = 251$. The attacker can also get the rest of the key bytes from the other 15 pairs of timing characteristic charts.

- Attack Analysis:

People may get confused about the final key recovery phase. Table lookups are intensively used in the AES implementation of OpenSSL v0.9.7a [15] library and other cryptographic libraries to get high performance. During the first round of each encryption, for each byte $n[i]$ of the input plaintext, $K[i] \oplus n[i]$ is used to index one of the tables ($K[i]$ is the $i'th$ byte of the known key K). Similarly, during the attack phase, $K'[i] \oplus n'[i]$ is also used to index that table. If $K[i] \oplus n[i] = K'[i] \oplus n'[i]$, then they are actually indexing the same entry in the table. Ideally, if the cache is large enough, these table lookups will always hit in the cache. However, in the real case, just before the encryption starts, some fixed AES table entries in the cache have always been evicted by some memory accesses from the victim software itself. These memory accesses regularly contend for cache lines at fixed locations. If given an index, the corresponding table entry is mapped to these ”hot” cache locations, a cache miss occurs, and the average encryption time will be long (the high bar in Figure 2.3 and Figure 2.4). Thus there are always some high bars in each of these 16 pairs of timing characteristic charts. If index $K[i] \oplus n[i]$ and $K'[i] \oplus n'[i]$ correspond to the high bars in Figure 2.3 and Figure 2.4, then they should be the same index. From $K[i] \oplus n[i] = K'[i] \oplus n'[i]$, we can get the unknown key byte $K'[i]$.

2.2 Software and Hardware Solutions

A lot of solutions to mitigate information leakage of cache side channels have been proposed, and most of them are software-based techniques. Earlier software solutions (mostly on AES) improve security but may lead to significant performance degradation [24, 38]. Recently, progress has been made in high performance AES implementations. Könighofer's bitslices implementation [33], Intel's introduction of AES-NI instructions [37] as an extension to the x86 ISA, and Lee's proposition of a general-purpose parallel table lookup instruction [34] all speed up AES and mitigate cache side channel attacks. Cache based hardware techniques have also been proposed. Some cache micro-architecture designs allow security data to be put into the private partition or locked region in the cache [39, 41], which prevents undesired cache evictions by other cache lines. However, cache lines that are locked or belong to the private partition cannot be used by other processes' cache line data even if the partition is not fully used or the locked lines are not used, which cause a cache under-utilization problem. Some other solutions use randomization-based approach [41, 42], which randomizes cache interference by doing random permutation of the memory-cache mapping or randomizing cache evictions.

2.3 Newcache

Newcache is a representative cache design using the randomization approach; it is first designed by Wang and Lee [42] and then improved by Liu and Lee [35]. It employs the techniques of dynamic memory-cache remapping and randomized eviction.

For a conventional SA cache, the index-bits from the memory address are used to index the cache set. n index bits indicate 2^n sets in the cache. For example, the data cache and instruction cache we use in our lab have 64 sets, and bit 11 to bit 6 of the physical memory address are used to index the cache set. For a 32KB SA cache with 64 byte cache-line size, we have $32kB/64B/64 = 8$ way set-associative cache. Our Intel cpu uses virtually-indexed physically-tagged (VIPT) caches, thus cache indexing and TLB lookup are done in parallel. The virtual tag is translated into physical tag, and then compared with the 8 physical tags inside the indexed cache set. If the physical tag matches one of the physical tags inside the set, then the corresponding cache block contains the desired data; otherwise a new cache block is fetched from higher level caches or main memory into the cache. Some cache block is replaced, and the corresponding physical tag in the cache set is updated.

2.3.1 The Original Newcache [42]

Newcache is quite different from a conventional SA cache. A 32KB Newcache with 64Byte cache-line size has $32kB/64B = 512$ entries. Thus we need $n = log_2 512 = 9$ bits to index these 512 entries. However, As shown in Figure 2.5 [42], it uses $n + k$ index bits. In this case the $n + k$ index bits are no longer used to index a cache set to compare the corresponding physical tags, instead the $n + k$ bits themselves are used for comparison first. The $n + k$ bits are stored in one of the Line Number Registers (LNreg), along with a d-bit RMT_ID. These $d+n+k$ bits form the Line Number Register (LNreg) for one entry. When the CPU needs data from a typical address, the $d+n+k$ bits are first compared with all the 2^n LNregs (at most one LNreg will match). If one LNReg matches, the tag comparison will be performed in the corresponding tag entry in the tag array. If the tag matches, it means that the corresponding cache block contains the desired data. Otherwise, either an index miss or a tag miss causes cache line replacement.

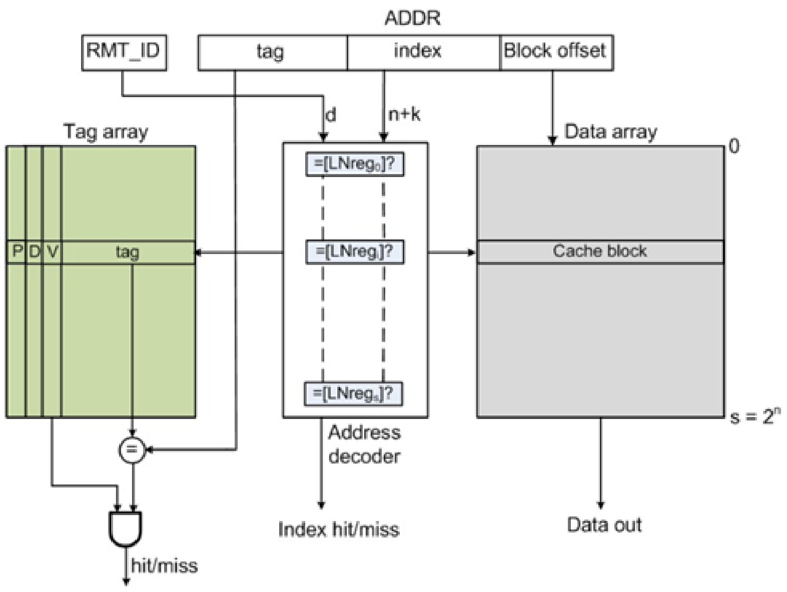


Figure 2.5: Newcache Block Diagram 1

- Newcache as PIPT cache:

Also, Newcache is implemented as a physically indexed physically tagged (PIPT) cache in gem5; so the virtual address is first translated to the physical address via the TLB, and then the physical address is used to fetch data in Newcache. In x86, a memory chunk is allocated as 4kB pages, and the lower 12 bits of the memory address are used as offset inside the page. For a 32KB 8-way set-associative cache, bit 11 to bit 6 are the index bits, and bit 5 to bit 0 are the cache block offset; thus there is no aliasing problem, and the cache indexing and TLB lookup can be done in parallel, because the lower 12 bits of the address will not change at all. However for a 32kB Newcache with

$k = 4$, the memory address uses $n + k = 9 + 4 = 13$ index bits; along with the 6 bits for block offset, these 19 bits will cause aliasing problem with 4kB page's 12 bits, which is why Newcache is implemented as a PIPT cache.

- LDM Cache:

As described above, Newcache contains 2^n physical cache lines, while it uses $n+k$ index bits, which means that 2^{n+k} cache lines can be mapped into 2^n physical cache lines. However, the cache block with the same $n + k$ bits cannot appear more than once in Newcache. It is equivalent to mapping the memory space to a larger logical direct-mapped (LDM) cache with 2^{n+k} lines before mapping to Newcache. The LDM cache does not physically exist; it is only used to facilitate the analysis. The LDM cache is just a direct-mapped cache with $n + k$ index bits. So the memory-cache mapping is a fixed-mapping from the memory to the LDM cache, and is a fully-associative mapping from the LDM cache to the Newcache.

- RMT_ID:

For x86, the TLB simply maps virtual addresses to physical addresses; thus for x86, we have to do a full TLB flush on each context switch. Some other platforms (e.g. MIPS, some ARM core etc.) map (ASID, virtual address) pairs to physical addresses. ASID (Address Space Identifier) is basically an identifier for a process. In the Newcache design, Z. Wang used something similar to ASID called RMT_ID. The RMT_ID identifies a hardware context, and each process is attached to a context RMT_ID. A process that needs to be protected against information leak from other processes should use a different RMT_ID. RMT_ID's d bits and the $n + k$ index bits are stored inside the LNregs, so actually the LDM cache has 2^{d+n+k} cache lines that can be mapped into the 2^n physical cache lines.

- Protection Bit:

As shown in the above block diagram (Figure 2.5), each cache line also has a protection bit P, indicating if the corresponding cache line is protected or not. Z. Wang puts the protection bit in the tag array, which may cause some security problem as analyzed by F. Liu [35]. F. Liu puts the protection bit P inside the LNreg. Also, each page table entry and page directory entry should be implemented with a Protection Flag bit, indicating a protected page.

- Cache access handling process:

The cache access handling process for Z. Wang's Newcache is shown in Figure 2.6 [42], which is called SecRAND. When the CPU needs to fetch data of memory block D from Newcache, it first uses $d + n + k$ (RMT_ID + index-bits) bits of D to compare with the contents in each 2^n LNreg. If no matched LNreg is found, an index miss occurs (4th column). For this case, a random victim from the 2^n cache line is selected for replacement, thus the interference caused by an index miss is always randomized. If the LNreg of a cache line C matches the RMT_ID + index-bits of D, but the tag of D does not match the corresponding tag in the tag array; in this case, a tag miss occurs. A tag miss always indicates a matching RMT_ID and index-bits, thus C and D have the same RMT_ID, which usually means C and D belong to the same process. This is internal interference of the same process or processes in the same security group (with the same RMT_ID). Under the tag miss, if it involves protected memory lines (colume 3), meaning that the interference may leak out critical information, a random victim is evicted, and D is accessed directly from higher level cache or memory without being put into the cache. The protection bit introduced here is used to randomize internal cache interference. If the tag miss does not involve protected memory lines (column 2), the miss is handled normally like in a traditional cache: the cache block corresponding to the LNreg is evicted, and the new cache block is fetched into this location. Cache hits (both index-bits and tag match) (column 1) are also handled as for a traditional cache.

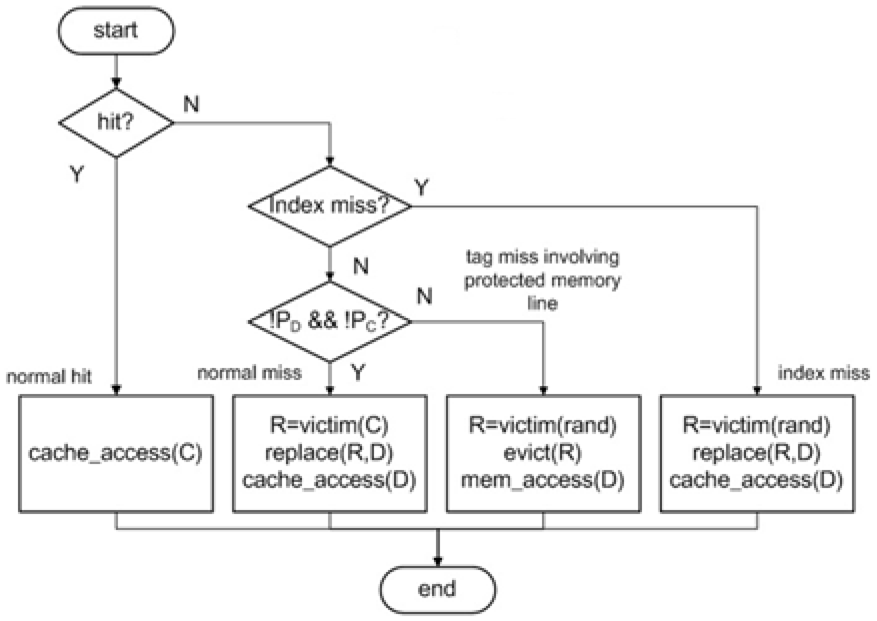

start
hit?
N
Y
Index miss?
Y
N
tag miss involving protected memory line
!P_D && !P_C?
N
Y
normal hit
normal miss
index miss
cache_access(C)
R=victim(C) replace(R,D) cache_access(D)
R=victim(rand) evict(R) mem_access(D)
R=victim(rand) replace(R,D) cache_access(D)
end

Figure 2.6: Cache Access Handling Process 1 (SecRAND)

However, Z.Wang's design has some security problems. By carefully redesigning attacks targeting newcache, F.Liu found that the original Newcache cannot 100% prevent the evict and time attack (e.g. Bernstein's AES attack) [35]. We noticed that although the mapping from the LDM cache to the physical cache is fully associative, the mapping from the memory to the LDM is fixed. Thus a memory line only has a fixed cache slot to be placed in if the mapping from the LDM cache to the physical cache has been established (i.e. index hit but tag miss). The redesigned attack involves an attacker's memory line A that has the same mapping (same $d + n + k$ LNreg bits) as a memory line

E that contains some of the AES table entries. Before each of the consecutive measurements, the attacker accesses A and then triggers a block encryption. If line A occupies the cache slot first, the following access to E will be a tag miss involving protected memory line (E has its P-bit set). Thus E cannot be fetched into the cache slot and replace A, but is accessed from memory. The SecRAND algorithm cannot prevent the attacker's memory line A from occupying the conflicting cache slot first when neither A nor E is in the cache. Thus the following accesses to E are all memory accesses (taken a longer time like a cache miss), which helps the attacker to achieve a time difference of AES table accesses between E and other AES table entries (other AES table entries in the physical cache).

2.3.2 The Modified F. Liu's Newcache [35]

F. Liu puts the protection-bit (P-bit) into the LNregs instead of the tag array (Figure 2.7 [35]). The new Newcache access handling process will consider a P-bit difference as an index miss (Figure 2.8 [35] column 3). So if line A occupies the cache slot first, the following access to E will be an index miss, and a random cache line is chosen to be replaced. The new Newcache access handling process provides a true randomized memory-cache mapping, and the internal cache interference through the tag miss of the original Newcache is eliminated. The resulting replacement algorithm is also simpler (see Figure 2.8).

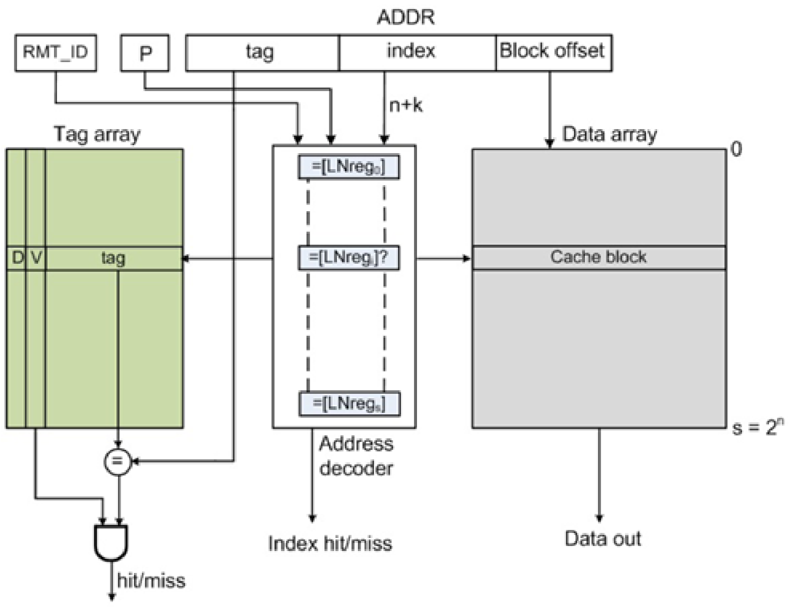


Figure 2.7: Newcache Block Diagram 2

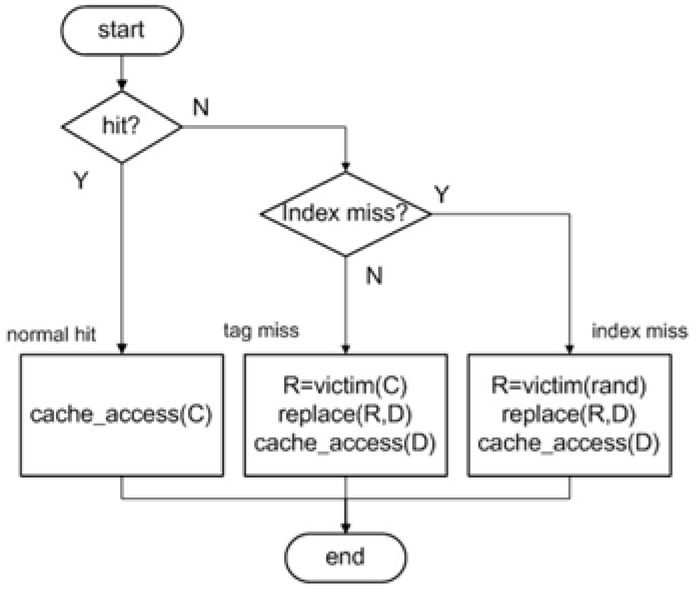


Figure 2.8: Cache Access Handling Process 2

2.3.3 Security Analysis

RMT_ID is used to distinguish different processes, or processes from different security regions. The index miss caused by RMT_ID will randomize the eviction, thus the attacker can get no useful information about his evicted cache lines in the PRIME-PROBE attacks (e.g. Percival's RSA attack, Zhang's ElGamal attack etc.). Basically, RMT_ID is used to eliminate external cache interference from the attacker's process.

P-bit is used to set protected memory regions inside a process. The index miss caused by P-bit mismatch will also randomize the eviction. Thus for attacks concerning internal cache interference (e.g. Bernstein's AES attack), the evicted cache lines will no longer be some fixed entries of the AES table. Instead, the randomized evicted cache lines will make the obtained timing information useless.

One more thing to notice: F.Liu's redesigned attack targeting the original Newcache actually involves an attacker process's memory cache lines A occupying some AES table's cache lines E. Actually, if the attacker's process and the victim process (AES encryption program) are assigned different RMT_IDs, the following access to E will be considered as an index miss in the first place. To make the attack work, the victim process's binary should be modified, so the malicious data will be assigned with the same RMT_ID. The attack model here is unrealistic, in my opinion.

After some careful thoughts, I think that under this attack model, F.Liu's new Newcache still does not necessarily prevent the redesigned Bernstein's AES attack. If the attacker has the power to modify the binary, then modifying the P-bit of his own cache lines seems even easier. As long as the attacker allocates himself memory locations that have the same $n + k$ index bits, RMT_ID bits and the P-bit as those of the victim's cache lines, he can always bypass the index miss and force a tag miss (See Figure 2.8 column 2). The attacker is thus still able to occupy cache lines in the LDM cache that should be occupied by the protected AES table entries before each encryption, which will then cause a longer encryption time every time the user tries to access that portion of the AES table.

In summary, if we use Fangfei's attack assumption that the attacker can modify RMT_ID and the P-bit, then neither the old nor the new Newcache can prevent the redesigned attack. However, in the real world, RMT_ID is deliberately designed to mitigate Fangfei's redesigned attack. Two cache lines with different RMT_ID are considered as index miss in the first place, and the attacker's cache line A can never occupy the cache before victim's cache line E does.

There is actually another question: do we actually need the P-bit in a real implementation to prevent Bernstein's attack? Is the P-bit really necessary to prevent internal cache interference? Note that the success of Bernstein's attack relies on its internal contention (from some program wrapper module) with some AES entries for the same cache lines (which means they have the same set number (6-bit) in the conventional SA cache). However, in Newcache, we have 9-bit index bits, plus k=4 Nebit. Will the cache lines that originally contended for the same sets still have the same 13 bits as those entries in the AES tables? Probably not, as the index bits increase, the chance of a program wrapper module and the AES table entries contending for the same physical cache lines is already greatly reduced.

These analysis will give system designers useful information to judge how to implement the P-bit and RMT_ID bits.

Chapter 3

NewCache Performance Measurement and Analysis for Cloud Server Benchmarks

In this chapter, we use Newcache as a (1) L1 data cache, (2) L2 cache, (3) L1 instruction cache, and (4) mix of the previous 3 caches, we measure the performance difference between Newcache and conventional set-associative (SA) caches. To see the performance difference, we use different nebits (nebits are the k bits of the $n+k$ index bits described in Chapter 2.3.1) and cache size for Newcache, and different associativity and cache size for the SA cache. Then we simulate server and client communication under gem5 [20] (a cycle accurate architectural simulator). We use these different cache configurations on the server side, and measure the cache performances for different cloud server benchmarks on the server side. These server applications are driven by testing tools from the client side. This experiment will help us gain a deeper understanding of Newcache performance under real world common server platforms and applications.

Table 3.1 shows the definitions of all the parameters we measured and analyzed for each benchmark.

3.1 Gem5 Simulation Methodology

To simulate both server side and client side, we need to do a dual-system configuration under gem5. Since most current programs run under x86 architecture, we want to ensure that x86's dual-system works under gem5. The configuration was successfully set by F.Liu in our lab. She assigned I/O Advanced Programmable Interrupt Controller (APIC) PIN17 to the Ethernet device and configured I/O APIC via Multiprocessor Configuration Table (MP Table, a data structure in BIOS). As shown

Table 3.1: Metrics Definitions

IPC	Instructions per cycle: the average number of instructions executed for each clock cycle for each benchmark
L1 Data Cache Miss Rate	The overall misses of L1 Data cache over the overall accesses to L1 Data cache
L1 Instruction Cache Miss Rate	The overall misses of L1 Instruction cache over the overall accesses to L1 Instruction cache
Local L2 Cache Miss Rate for Data	The overall data misses of L2 cache over the overall data accesses to L2 cache
Local L2 Cache Miss Rate for Instructions	The overall instruction misses of L2 cache over the overall instruction accesses to L2 cache
Global L2 Cache Miss Rate for Data	The overall data misses of L2 cache over the overall accesses to L1 Data cache
Global L2 Cache Miss Rate for Instructions	The overall instruction misses of L2 cache over the overall accesses to L1 Instruction cache

in Figure 3.1, after configuration, the server side (Test system) and the client side (Drive system) should be able to communicate via the Ethernet link. The server has an IP address 10.0.0.1, and the client has an IP address 10.0.0.2. The server has different server-side applications installed. The drive system is used to send requests to the server, and the server handles the requests. The purpose of the experiment is to see the performance of different detailed cache models (SA cache and Newcache) on the server side.

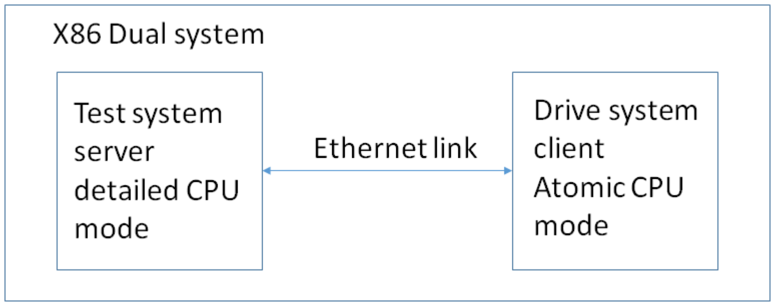

X86 Dual system
Test system server detailed CPU mode
Ethernet link
Drive system client Atomic CPU mode

Figure 3.1: x86 Dual System

It's not easy to install server benchmarks under the gem5 simulator system. So we install the software package on a real machine, and then copy the installed package and all the required dynamic libraries to gem5's image disk. Both server side and client side use a basic Linux system installed on gem5's image disk. When booting the dual system in gem5, both server and client configure their network interface. The server starts the required network services, sets the necessary server side configuration, and sends a 'ready' signal to the client. The client waits for the 'ready' signal, and after that, drives the server side with requests.

Table 3.2 shows the detailed base CPU and cache configurations used on the server side of gem5. The configurations are collected from Hennessy and Patterson's latest architecture book [30] for Intel's Core i7 hierarchy. Gem5 simulates one of the 4 cores of Intel i7. All the caches are normal SA Cache. Since latest i7 has very high cpu frequency, 4 cycles latency for L1-ICache and L1-Dcache hits are not too much penalty. Based on the base configurations, we change different levels' cache type to Newcache and measure the performance differences. The baseline Newcache configuration has k=4.

Table 3.2: Gem5 Base CPU and Cache Configurations

Single-core out-of-order X86 processor	L1-ICache (private)	L1-DCache (private)	L2 Cache (private)	L3 Cache (shared)	Memory (shared)
	32 kB 4-way 4-cycle latency	32 kB 8-way 4-cycle latency	256 kB 8-way 10-cycle latency	2MB 16-way 35-cycle latency	2GB 100-cycle latency
Cache line size	64B				
Clock freq	3 GHz				
Evaluate the performance of Newcache as L1 data cache, L2 cache and L1 instruction cache, k=4 for baseline Newcache configuration					

3.2 Cloud Server Benchmarks Description and Selection

We divide cloud server benchmarks into several categories based on D. Perez-Botero's MSE thesis [40]: Web Server, Database Server, Mail Server, File Server, Streaming Server, and Application Server. We describe the potential possible benchmarks for each category and the way we do the performance measurements.

3.2.1 Web Server and Client

When people want to host web sites (e.g., a small company's homepage, a blog, etc.), a software web server is required to handle clients' Hypertext Transfer Protocol (HTTP) requests, and to process and deliver web pages to clients. Sometimes we also deploy a database server (MySQL [13], etc.) to store data and a server-side scripting engine (PHP [16], etc.) to generate dynamic web pages if necessary. The Web Server and Client are basically chosen by F. Liu.

Server:

- **Apache HTTP server (httpd)** [2]: We basically choose httpd as our web server. Apache httpd has been the most popular web server on the Internet since April 1996. The Apache HTTP Server Project is an effort to develop and maintain an open-source HTTP server for modern operating systems including UNIX and Windows NT. On the server side, we write a script called *apachectl*, which uses the binary *httpd* to start the Apache HTTP server.

Client:

- Apache Benchmark Tool (ab) [1]: ApacheBench (ab) is a single-threaded command line computer program for measuring the performance of HTTP web servers. It is also well suited for the non-GUI testing environment under gem5. It is originally designed to test the Apache HTTP Server, but actually it is generic enough to test any web server, which especially shows how many requests per second the Apache installation is capable of serving. *ab* allows picking the number of total requests and the number of concurrent requests. For example, to send 1000 HTTP requests to our Apache server with a concurrency of 10 requests at the same time, we type *ab -n 1000 -c 10 http://10.0.0.1:8080/*.

3.2.2 Database Server and Client

The Database Server and Client are also chosen by F.Liu.

Server:

- MySQL [13]: MySQL is the database server we select for the test. It is a famous open-source relational database management system (DBMS). It is a popular choice of database for use in web applications, and is a central component of the widely used LAMP open source web application software stack. To start MySQL server, we need to do some post-installation setup. We need to create a user and group for the main program *mysqld* to run. Then we run script *mysql_install_db* to set up initial grant tables, which determine how users are permitted to connect to the server. Finally we start *mysqld* service and explicitly allow the client to remotely connect to the server.

- IBM DB2 [8]: IBM DB2 is an alternative server. It is a commercialized family of database server products developed by IBM. It is standardized but closed source and expensive.

Client:

- SysBench [19]: SysBench is a modular, cross-platform and multi-threaded benchmark tool for evaluating OS parameters that are important for a system running a database under intensive load. The idea of this benchmark suite is to quickly get an impression about system performance without setting up complex database benchmarks. In SysBench, we use OLTP test mode, which benchmarks a real database's performance. In the preparation phase, we create test tables with 100 records, while for the running phase, we do 200 advanced transactions.

3.2.3 Mail Server and Client

We can categorize mail servers into outgoing mail servers and incoming mail servers. The outgoing mail server is known as Simple Mail Transfer Protocol (SMTP), while the incoming mail server can be Post Office Protocol v3 (POP3) or Internet Message Access Protocol (IMAP). We mainly focus on SMTP under gem5.

Server:

- Bhm [17]: We choose bhm to act as the SMTP server. A sender SMTP server would receive messages from email clients (Gmail, Outlook Express, etc.), and a message usually contains the sender's and recipients' email addresses, the message body, attachments, etc. The sender SMTP server would put the message into a FIFO queue and send them to other servers, which are finally routed to the recipient's POP3 or IMAP server. The bhm program is a simple SMTP sink program, which monitors TCP:25 on the server side and dumps the messages to */dev/null*. Typing *bhm t X* in the command line means using X threads to receive incoming messages.

Client:

- Postal [17]: We use postal to send messages to the server. Postal aims at benchmarking mail server performance. It shows how fast the system can process incoming email. We can set thread number, message size on command line when using postal.

Testing:

Bhm is fixed to use 2 threads to receive incoming messages. In the results shown in the next few sections, Mail_tx means using x threads in Postal to send messages to server. In gem5 dual system, Mail_t1 sends 521KB/minute, Mail_t2 sends 1043KB/minute, and Mail_t5 sends 2245KB/minute. All three tests last 20 seconds.

3.2.4 File Server and Client

A client can interact with the remote server file system via serveral protocols: the File Transfer Protocol (FTP), the SSH File Transfer Protocol (SFTP), the Server Message Block (SMB) protocol, the Network File System (NFS) protocol, etc. FTP is widely used, but it only transfers (upload-/download) files between servers and clients. In SFTP and SMB, the client can also mount and interact (view, edit, zip, etc.) with files and directories located on a remote server. In addition, SMB provides file sharing and printing services to Windows clients as well as Linux clients.

Server:

- Samba smbd [18]: We use smbd as the file server in our test. A session is created whenever a client requests one. Each client gets a copy of the server for each session. This copy then services all connections made by the client during that session. We type *smbd start* to start the smbd service.

Client:

Dbench [5]: We choose dbench as the workload generator. It can generate different I/O workloads to stress either a file system or a networked server. We can first choose dbench's stressing backend (smb, nfs, iscsi, socketio, etc.) by specifying the -B option. Since the server is Samba smbd, we choose the backend to be smb. Then we need to specify the shared file server folder and the user-password pair through the –smb-share option and the –smb-user option. The shared folder and the user-password pair are already set up by the smbd server. However in our experiments, we don't use a user-password pair. Moreover, dbench has a key concept of a "loadfile", which is a sequence of operations to be performed on the file server's shared folder. The operations could be "Open file 1.txt", "Read XX bytes from offset XX in file 2.txt", "Close the file", etc. In the experiment, we generate two different "loadfiles", one is a write-intensive load (smb-writefiles.txt), and another is a read-intensive load (smb-readfiles.txt). Finally, we can add a number **n** at the end of the dbench command to specify the total clients simultaneously performing the load.

Testing:

We use smbd and dbench to form file server and client. In the results shown in the next few sections, we type *./dbench -B smb –smb-share=//10.0.0.1/share –smb-user=% –loadfile=smb-writefiles.txt –run-once –skip-cleanup 3* to generate Dbench_write, which means launching 3 clients (simulated as processes), and each client opens and writes five 64kB files. By replacing loadfile with smb-readfiles.txt, we generate Dbench_read, which launches 3 clients and each client opens and reads five 64kB files.

3.2.5 Streaming Server and Client

There are different kinds of streaming protocols such as RTSP/RTP, MMS, HTTP etc. RTSP allows states, and streaming can be controlled and given feedback by the client side, while HTTP is a stateless protocol. We choose RTSP protocol. Most RTSP servers use the Real-time Transport Protocol (RTP) in conjunction with Real-time Control Protocol (RTCP) for media stream delivery. In a word, RTP is used for data transmission while RTCP is used to control the transmission.

Server:

- ffserver [6]: We use ffserver as our streaming server. It is a streaming server for both audio and video, which can stream mp3, mpg, wav etc. ffserver is part of the ffmpeg package. It is small and robust. Before starting the server, we need to register server side media-files at ffserver.conf file.

- LIVE555 Media Server [12]: The "LIVE555 Media Server" is a complete RTSP server application. It can also stream several kinds of media files. Unlike ffserver, LIVE555 Media Server does not need to register all the media files during configuration. Instead, we just need to set the current working directory which contains all the media files to make these files stream-able.

- VLC [21]: VLC is a free and open source cross-platform multimedia player and framework, and it is the favorite multimedia player for a lot of people. By doing simple configurations on VLC, we can also make it a useful streaming server. All the other computers that have VLC installed on them can stream video or audio located on a remote server. However, it is not a good option for gem5's testing purpose. Basically VLC uses LIVE555 Media Server as its streaming component, but VLC also contains a lot of other internal packages that cannot be disabled (e.g. encoder/decoder packages etc.). A lot of internal packages require too many dynamic libraries, and installing all these libraries one by one would be quite inconvenient. Most importantly, loading all the libraries and running the whole VLC as a test server on gem5 would be really slow.

Client:

- openRTSP [14]: We use openRTSP as our streaming client. openRTSP is an RTSP client, which is also part of the LIVE555 streaming media package. RTSP (Real Time Streaming Protocol) is a network control protocol designed to control streaming media servers. Clients can issue VCR-like commands, such as play and pause, to facilitate real-time control of playback of media files from the server.

In the experimental results shown in the next few sections, Rtsp_sX means sending X different remote connection requests to the ffserver for media file streaming. We basically generate 3 different streaming workloads (X=1, X=3 and X=30) to test the streaming server.

3.2.6 Application Server and Client

An application server [4] is either a software framework that provides a generalized approach for creating an application-server implementation or the server portion of a specific implementation. An application server acts as a set of components accessible to the software developer through an API defined by the platform itself. For web applications, these components' main job is to support the construction of dynamic pages [4]. However, many application servers also provide services like clustering, fail-over, and load-balancing [4]. The currently most used Java application server is actually an extension of Java virtual machine for running applications. The server handles connection to database on one side and connections to the web client on the other side [4]. Application servers differ from web servers by dynamically generating html pages each time a request is received, while most http servers just fetch static web pages. Application servers can utilize server-side scripting languages (PHP, ASP, JSP, etc.) and Servlets to generate dynamic contents.

Server:

- Tomcat [3]: We use Tomcat on the server side. Tomcat, Jetty [10], Jboss [9] and Glassfish [7] are all popular Java application servers. Tomcat is more popular, because it is small, robust, and supports the required JSP and Servlet. Most of the application servers work above the Java runtime environment; they are pretty slow when launching under gem5. Tomcat would take an hour, Glassfish4 would take more than 8 hours. That's the main reason we use Tomcat as the application server.

Client and Testing:

For the testing, we use **Apache ab** [1] to send HTTP requests to **Tomcat** (Apache ab is described previously as a web server client). Tomcat provides us with a lot of small JSP and Servlet examples, which are quite useful. We use command: *ab -n Y -c Z http://10.0.0.1:8080/(X URLs)*, where X represents X different URLs, Y is the number of requests to perform for each URL, and Z is the requesting concurrency for each URL. These different URLs contain different JSP and Servlet examples provided by **Tomcat**. In our experiments, we fix Y=10 and Z=2, and choose X to be 1, 3 and 11, respectively, for three different workloads, ranging from light to heavy work.

3.2.7 Benchmark Summary

Table 3.3 summarizes the testing benchmarks and the driving tools we use for the Web Server, Database Server, Mail Server, File Server, Streaming Server and Application Server.

Table 3.3: Summary of Cloud Server Benchmarks

	Server-side Benchmarks	Client-side Driving Tool
Web Server	Apache httpd	Apache ab
Database Server	MySQL	SysBench
Mail Server	Bhm	Postal
File Server	Samba smbd	DBench
Streaming Server	ffserver	openRTSP
Application Server	Tomcat	Apache ab

We should also notice that some Server-side benchmarks require some time to start up before their services are available to the client under gem5. I actually use a daemon program to test periodically if the service ports are opened up by the benchmark. Only until that time will the server send a 'ready' signal to the client.

3.3 Testing Results: Newcache as L1 Data Cache

Based on the base CPU and cache configurations in Table 3.2, we replace the L1 data cache with Newcache. We run the benchmark services on the server side and driving tools on the client side. We then collect the required data and get IPC (Instructions per cycle), Data Cache Miss Rate and Global L2 Miss Rate for all the benchmarks. These results are compared with those we got from the base configurations, where L1 data cache is a conventional set-associative (SA) Cache.

3.3.1 IPC

Figure 3.2 shows the IPC for different cache sizes for all the benchmarks. The results include a conventional SA cache (size 16kB, 32kB, and 64kB) and Newcache (size 16kB, 32kB and 64kB).

From the results, we see that:

- IPCs for these server benchmarks are far below 1, which means there is not a lot of instruction-level parallelism that can be exploited by the Out-Of-Order processor. Server applications tend to have very low IPCs [28], because they spend a lot of time in the kernel. Accessing the network stack, the disk subsystem, handling the user connections and requests, syncing large

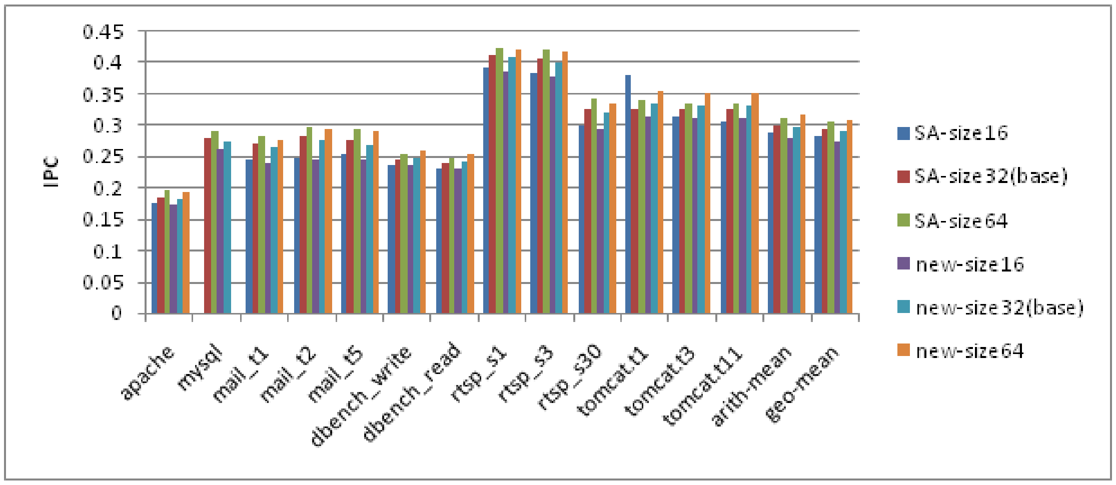


Figure 3.2: Newcache as L1-DCache: IPC for Different Cache Sizes

amounts of threads all require the program trapping into the kernel, which also induces a lot of context switches. Kernel codes have a lot of dependencies, thus causing low IPCs.

- For both SA cache and Newcache, as the cache size increases, the IPC also increases for each benchmark. Larger cache tends to have more immediate data available for instructions' use, thus more instructions that have no dependencies can execute simultaneously in this case, which leads to larger IPC.

- From the Streaming Server test rtsp_s1, rtsp_s3, rtsp_s30 and Mail Server test mail_t1, mail_t2, mail_t5, we can see that as the client requests increase, the IPC decreases, which agrees with our expectation. As the server program handles more user connections and requests, and sync more threads etc., more kernel traps are incurred. More context switches and a lot of dependencies inside kernel codes causes lower IPCs.

- There is an abnormal value for tomcat.t1 under SA-size16. From time to time, gem5 may cause certain glitch on a certain running of a benchmark. We also cannot get any result for mysql under SA-size16. Under a small cache size, some benchmark service cannot start normally under gem5 or the results cannot be dumped out, which might be caused by gem5's internal implementations.

Table 3.4 summarizes the Newcache's IPC increase relative to SA for different cache sizes on average. As we described in Table 3.2, the baseline Newcache has Nebit $k = 4$, and the baseline SA cache has associativity 8 for L1-DCache. We can see that basically Newcache performs as well as SA cache. As the size increases, Newcache tends to perform a little better than SA cache. Newcache's fully associative mapping from LDM to physical cache and it's random eviction scheme tend to benefit more from larger cache sizes.

Table 3.4: Newcache as L1-DCache: Different Cache Sizes' IPC, and IPC Increase Relative to SA

	16KB		32KB		64KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	0.2890	0.2791	0.2999	0.2984	0.3125	0.3167
Increase relative to SA		-3.45%		-0.52%		1.36%

Figure 3.3 shows the IPCs for different SA-Associativity and Newcache-Nebit for all the benchmarks. The larger the SA-Associativity, the more cache lines with the same index can stay in the cache at the same time. Similarly, the bigger the Newcache-Nebit, the longer the index $(n+k)$ bits, thus the larger amount of cache lines that can be mapped into the physical cache at the same time.

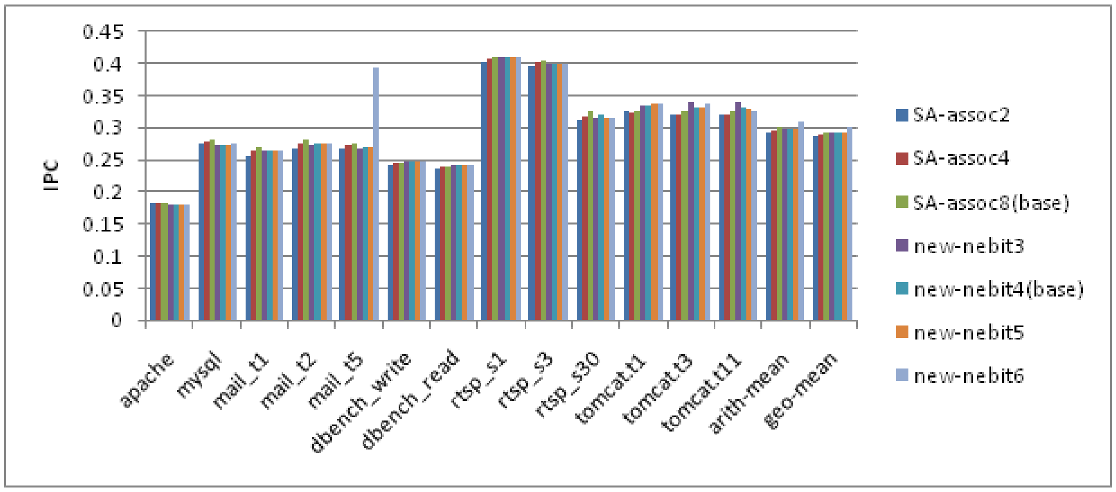


Figure 3.3: Newcache as L1-DCache: IPC for Different Associativity and Nebit

Table 3.5 shows all these configurations' IPC Increase Relative to SA-8way on average of all the benchmarks. As we described in Table 3.2, the baseline Newcache and the baseline SA cache have 32kB cache size. Again, we can see that Newcache performs as well as SA cache. The IPCs tend to be better with larger SA-Associativity. Table A.2 in Appendix A shows that for most of the benchmarks, the IPCs tend to be better with bigger Newcache-Nebit. A 32kB Newcache with Nebit $k = 3$ has a 256kB LDM cache, while a 32kB Newcache with Nebit $k = 6$ has a 2048kB LDM cache. Unlike desktop applications, server applications tend to fetch data from a wider range of memory space, because server applications may need to handle a lot of user connections and requests, and sync large amounts of threads. That's why server applications tend to benefit more from bigger Newcache-Nebit. However, in Table 3.5, the average IPC seems to decrease for larger nebit, because some of the benchmarks in Table A.2 do have lower IPC for $k = 4$ and $k = 5$. Also, the glitch of mail_t5 in Figure 3.3 causes the average (arith-mean) IPC of $k = 6$ to be a bit higher. Thus, some benchmarks benifit from larger nebit, some may not.

Table 3.5: Newcache as L1-DCache: Different Associativity and Nebit's IPC, and IPC Increase Relative to SA-8way

	SA 2way	SA 4way	SA 8way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
Average	0.2929	0.2962	0.2999	0.2991	0.2984	0.2983	0.3082
Increase relative to SA-8way	-2.35%	-1.26%	0.00%	-0.28%	-0.52%	-0.53%	2.77%

3.3.2 DCache Miss Rate

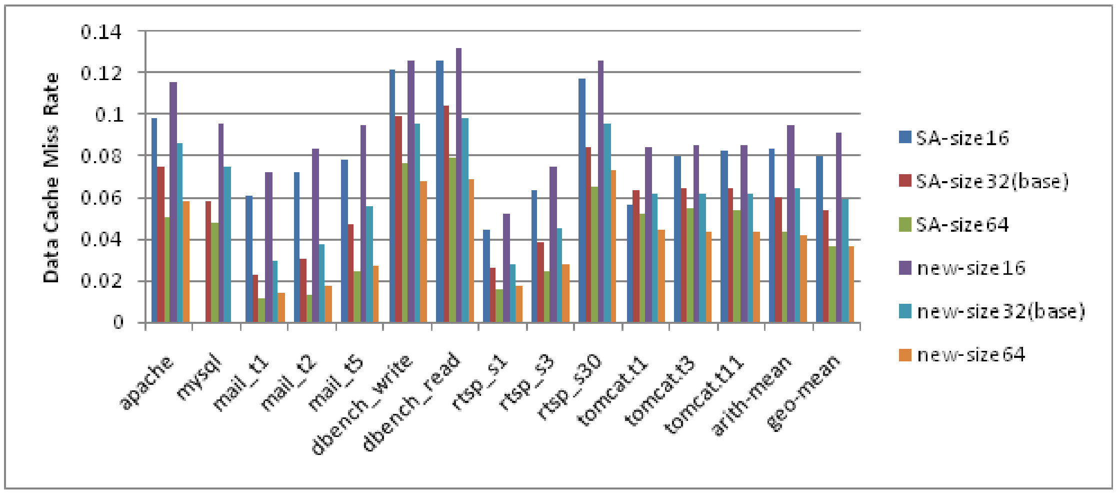


Figure 3.4: Newcache as L1-DCache: DCache Miss Rate for Different Cache Sizes

Figure 3.4 shows the Data Cache Miss Rate under different cache sizes for all the benchmarks. We can find something similar to the IPC results:

- For both SA cache and Newcache, as the cache size increases, the DCache Miss Rate decreases for each benchmark. Larger cache tends to have more data available for immediate use, which leads to lower DCache Miss Rate.
- From the Streaming Server test rtsp_s1, rtsp_s3 and rtsp_s30 and Mail Server mail_t1, mail_t2, mail_t5, we can see that as the client requests increase, the DCache Miss Rate increases a lot. Because as the server handles more user requests and more threads, the data from different threads tend to contend for the limited slots in the cache, thus evicting each others' data and causing larger DCache Miss Rate.

Table 3.6 summarizes the Newcache's DCache Miss Rate increase relative to SA for different cache sizes on average. Still, Newcache basically performs as well as SA cache. 16kB cache size really limits the Newcache performance, while as the size increases, Newcache tends to perform much better than SA cache. As we described above, Newcache tends to benefit more from larger cache sizes.

Table 3.6: Newcache as L1-DCache: Different Cache Sizes' DCache Miss Rate Increase Relative to SA

	16KB		32KB		64KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	0.0836	0.0945	0.0598	0.0640	0.0437	0.0420
Increase relative to SA		13.05%		6.97%		-3.91%

Figure 3.5 shows the DCache Miss Rate for Different Associativity and Nebit, and Table 3.7 shows the different configurations' DCache Miss Rate Increase Relative to SA-8way.

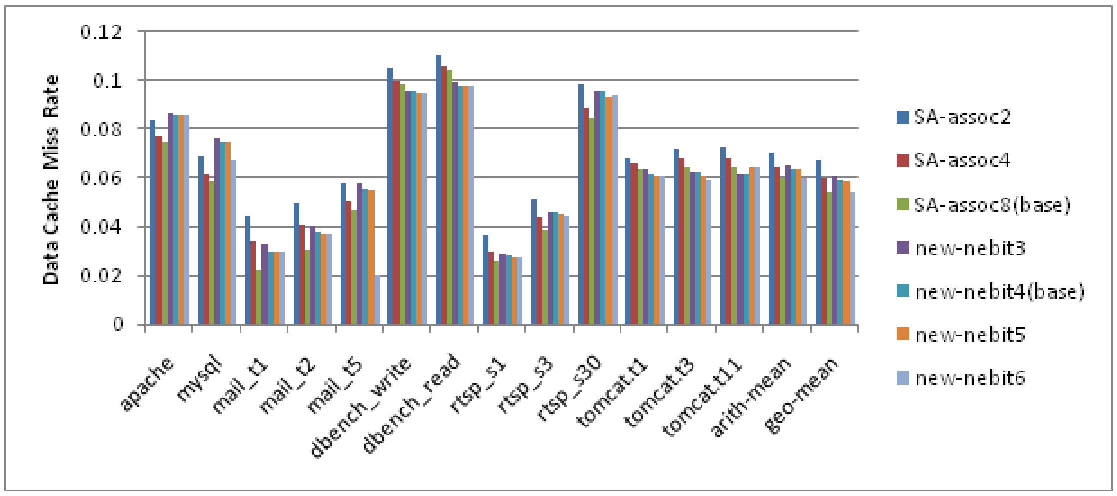


Figure 3.5: Newcache as L1-DCache: DCache Miss Rate for Different Associativity and Nebit

Table 3.7: Newcache as L1-DCache: Different Associativity and Nebit's DCache Miss Rate Increase Relative to SA-8way

	SA 2way	SA 4way	SA 8way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
Average	0.0706	0.0641	0.0598	0.0650	0.0640	0.0637	0.0602
Increase relative to SA-8way	18.12%	7.22%	0.00%	8.68%	6.97%	6.45%	0.70%

From the results, we can see that SA-8way performs best for DCache Miss Rate under different DCache configurations. Smaller SA-Associativity and smaller Newcache-Nebit have large negative impacts on performance, causing high miss rate. From the results, we can also see that only when Nebit reaches 6 can Newcache reach almost the same miss rate as that of the SA-8way (base case).

3.3.3 Global L2 Miss Rate

Here L2 Cache is the base a conventional 256kB 8-way SA cache. The Global L2 Cache Miss Rate is the miss rate for data access. Here

$$Global\ L2\ Miss\ Rate = \frac{l2.overall\_misses}{cpu.dcache.overall\_accesses}$$

Figure 3.6 shows the Global L2 Miss Rate under different cache sizes for all the benchmarks.

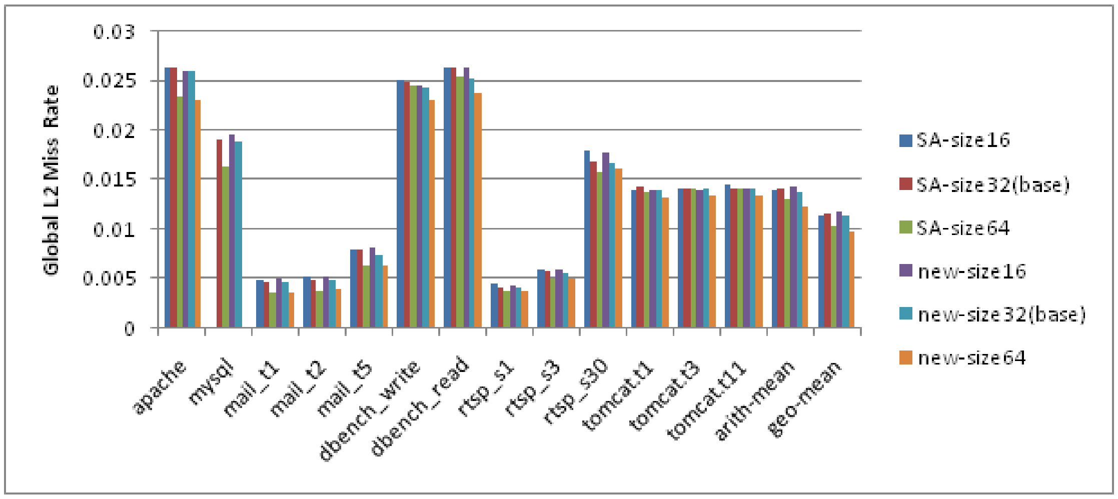


Figure 3.6: Newcache as L1-DCache: Global L2 Miss Rate for Different Cache Sizes

Table 3.8: Newcache as L1-DCache: Different Cache Sizes' Global L2 Miss Rate Increase Relative to SA

	16KB		32KB		64KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	0.0139	0.0142	0.0141	0.0138	0.0130	0.0123
Increase relative to SA		2.40%		-1.81%		-5.17%

We can find something similar to the DCache Miss Rate results:

- As analyzed above, for both SA cache and Newcache, as the cache size increases, the DCache Miss Rate decreases for each benchmark. Because larger cache tends to have more data available for immediate use, which leads to lower DCache Miss Rate. So the number of data requests from DCache to L2 Cache also shrinks, which is why Global L2 Miss Rate also decreases.

- The test results for the Streaming Server series rtsp_s1, rtsp_s3, rtsp_s30 and the Mail Server series mail_t1, mail_t2, mail_t5 also meet our expectations. The more the client's requests, the larger the Global L2 Miss Rate.

Table 3.8 shows the different cache sizes' Global L2 Miss Rate Increase Relative to SA. Still, Newcache basically performs as well as SA cache. It's interesting to find that after changing DCache to Newcache, Global L2 Miss Rate benefits a lot for 64kB cache sizes.

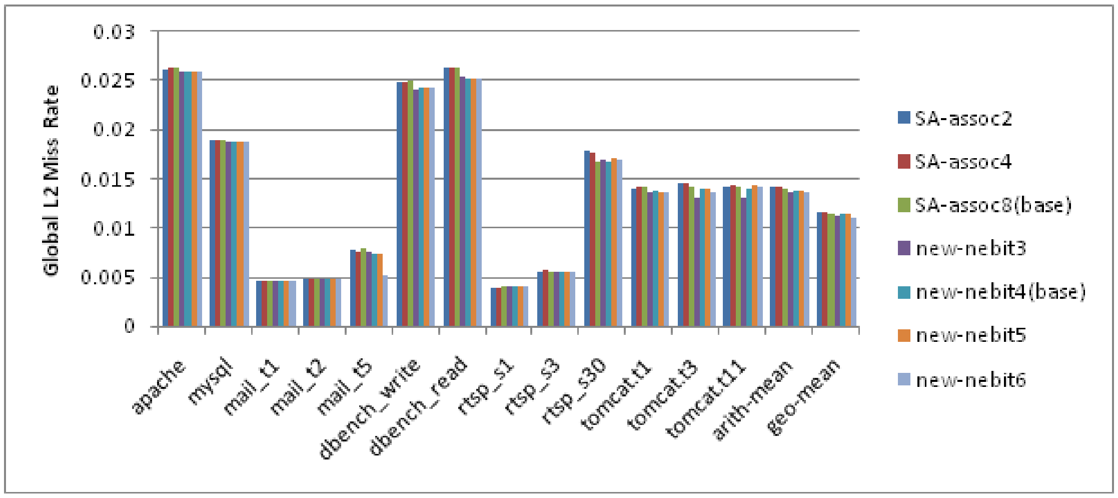


Figure 3.7: Newcache as L1-DCache: Global L2 Miss Rate for Different Associativity and Nebit

Table 3.9: Newcache as L1-DCache: Different Associativity and Nebit's Global L2 Miss Rate Increase Relative to SA-8way

	SA 2way	SA 4way	SA 8way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
Average	0.0141	0.0141	0.0141	0.0137	0.0138	0.0138	0.0136
Increase relative to SA-8way	0.42%	0.60%	0.00%	-2.82%	-1.81%	-1.67%	-3.12%

Figure 3.7 shows the Global L2 Miss Rate for Different Associativity and Nebit, and Table 3.9 shows the different configurations' Global L2 Miss Rate Increase Relative to SA-8way. When DCache is Newcache, Global L2 Miss Rates are even better than those of a conventional SA cache.

3.4 Testing Results: Newcache as L2 Cache

Similar to the previous section, based on the base CPU and cache configurations in Table 3.2, we replace L2 cache with Newcache in this test. We run the benchmark services on the server side and driving tools on the client side. We then collect the required data and get Instructions per cycle (IPC), Data Cache Miss Rate, Local L2 Miss Rate and Global L2 Miss Rate for all the benchmarks.

3.4.1 IPC

Figure 3.8 shows the IPC for different L2 cache sizes for all the benchmarks. The results include conventional SA cache (size 128kB, 256kB and 512kB) and Newcache (size 128kB, 256kB, and

512kB). As we described in Table 3.2, the baseline Newcache for L2 cache has Nebit $k = 4$, and the baseline SA cache has associativity 8 for L2 Cache.

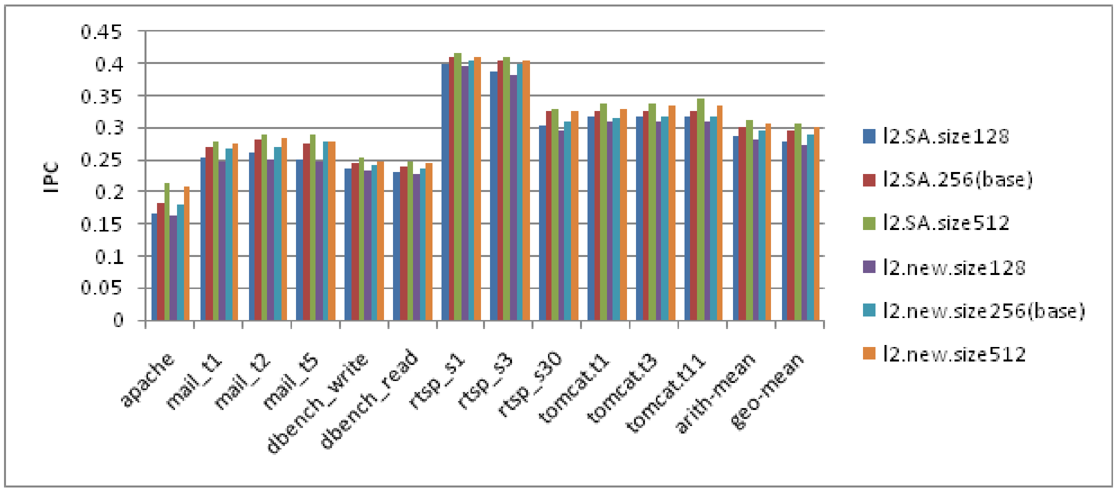


Figure 3.8: Newcache as L2 Cache: IPC for Different Cache Sizes

Table 3.10: Newcache as L2 Cache: Different Cache Sizes' IPC, and IPC Increase Relative to SA

	128KB		256KB		512KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	0.2864	0.2812	0.3016	0.2951	0.3123	0.3064
Increase relative to SA		-1.81%		-2.16%		-1.89%

Table 3.10 summarizes the Newcache's IPC increase relative to SA for different L2 cache sizes on average. From the results, we can see that Newcache performs almost as well as SA cache for different L2 cache sizes. On average, it performs only slightly worse than SA cache by about 2%.

Figure 3.9 shows the IPCs for different L2 SA-Associativity and Newcache-Nebit for all the benchmarks, and Table 3.11 shows all these L2 configurations' IPC increase relative to SA-8way on average of all the benchmarks. As we described in Table 3.2, the baseline Newcache and the baseline SA cache have 256kB cache size for L2 cache. Again, L2 as Newcache performs almost as well as SA cache. Also, the IPC increases slowly as Newcache-Nebit increases from 3 to 6.

Table 3.11: Newcache as L2 Cache: Different Associativity and Nebit's IPC Increase Relative to SA-8way

	SA 4way	SA 8way	SA 16way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
Average	0.2997	0.3016	0.3040	0.2938	0.2951	0.2945	0.3053
Increase relative to SA-8way	-0.61%	0.00%	0.81%	-2.56%	-2.16%	-2.36%	1.22%

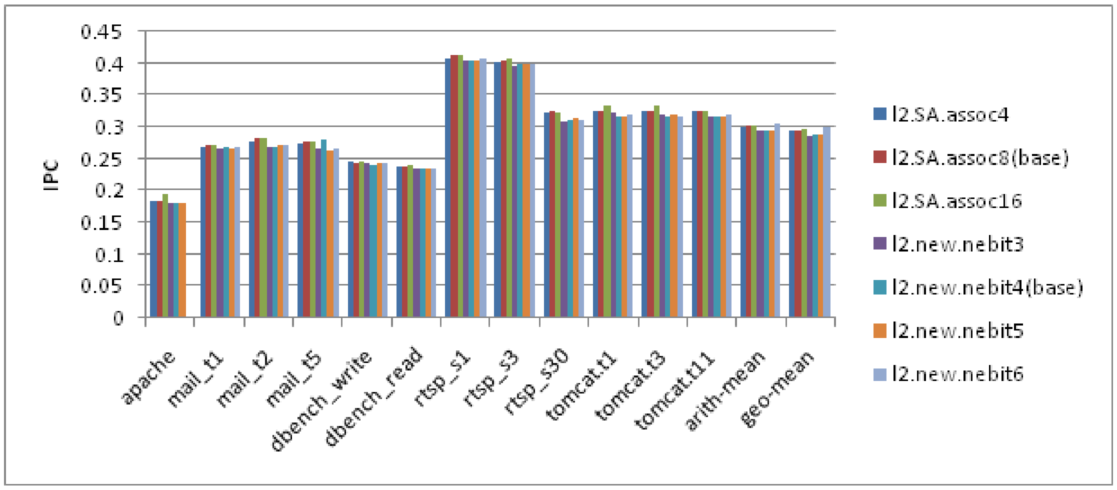


Figure 3.9: Newcache as L2 Cache: IPC for Different Associativity and Nebit

3.4.2 DCache Miss Rate

Figure 3.10 shows the Data Cache Miss Rate under different L2 cache sizes for all the benchmarks, and Table 3.12 summarizes the Newcache's DCache Miss Rate increase relative to SA for different cache sizes on average. Basically Data Cache Miss Rate increases a little bit by changing L2 cache from normal SA cache to Newcache with the same cache size.

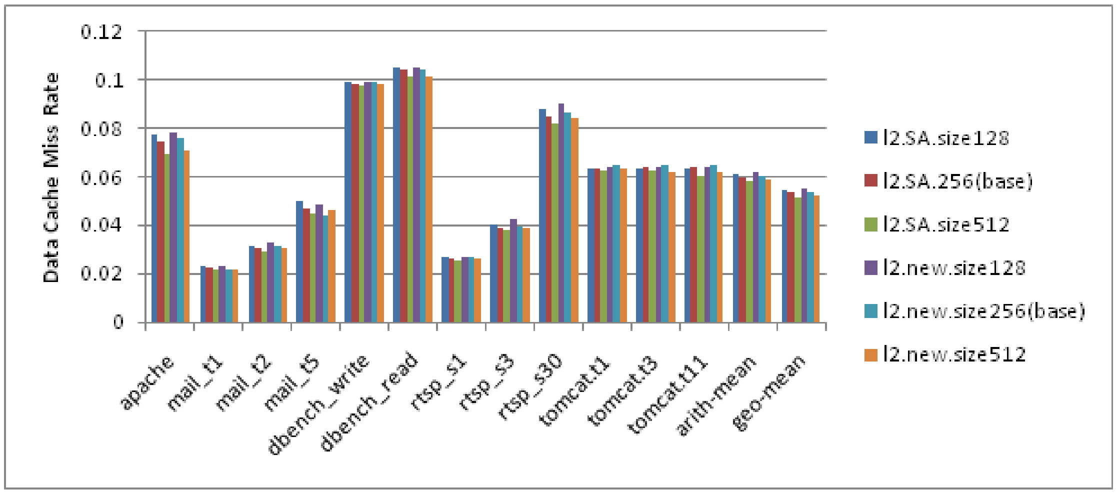


Figure 3.10: Newcache as L2 Cache: DCache Miss Rate for Different Cache Sizes

Figure 3.11 shows the DCache Miss Rate for Different Associativity and Nebit, and Table 3.13 shows the different configurations' DCache Miss Rate increase relative to SA-8way. We can see that for L2 as Newcache results, the DCache Miss Rate tends to decrease as we increase Nebit from 3 to 6.

Table 3.12: Newcache as L2 Cache: Different Cache Sizes' DCache Miss Rate Increase Relative to SA

	128KB		256KB		512KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	0.0609	0.0616	0.0599	0.0602	0.0578	0.0587
Increase relative to SA		1.08%		0.54%		1.47%

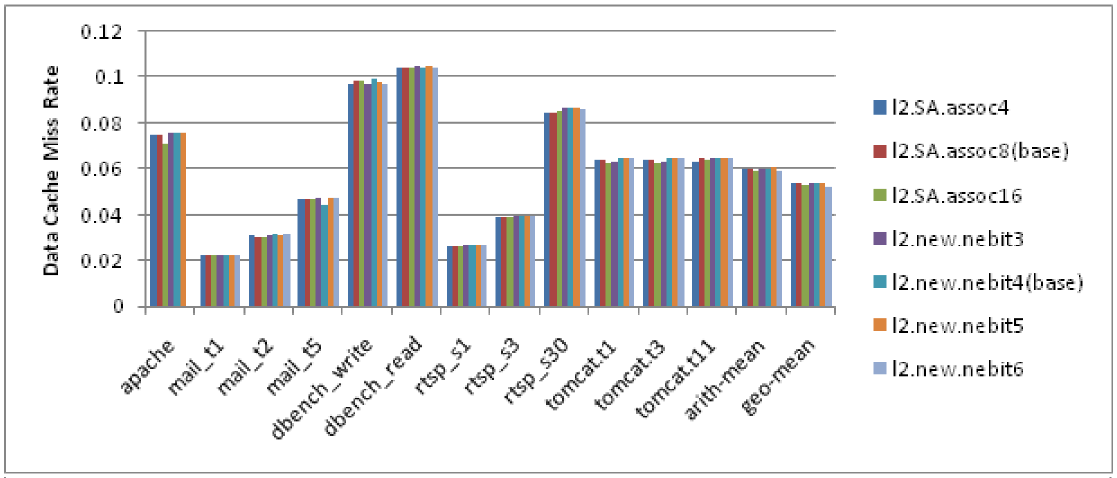


Figure 3.11: Newcache as L2 Cache: DCache Miss Rate for Different Associativity and Nebit

3.4.3 Local L2 Miss Rate

The Local L2 Cache Miss Rate here is the miss rate for data access, and we have the equation:

$$Local\ L2\ Miss\ Rate = \frac{l2.overall\_misses}{dcache.overall\_mshr\_misses}$$

Here MSHR stands for Miss Status Handling Register for Dcache. It is used to store the fetch request of an outstanding miss for a non-blocking cache. So actually, two data misses inside the same cache block are stored as one outstanding miss in MSHR.

Figure 3.12 shows the Local L2 Miss Rate under different cache sizes for all the benchmarks, and Table 3.14 shows the different cache sizes' Local L2 Miss Rate increase relative to SA.

Table 3.13: Newcache as L2 Cache: Different Associativity and Nebit's DCache Miss Rate Increase Relative to SA-8way

	SA 4way	SA 8way	SA 16way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
Average	0.0596	0.0599	0.0592	0.0603	0.0602	0.0605	0.0589
Increase relative to SA-8way	-0.48%	0.00%	-1.13%	0.60%	0.54%	0.92%	-1.64%

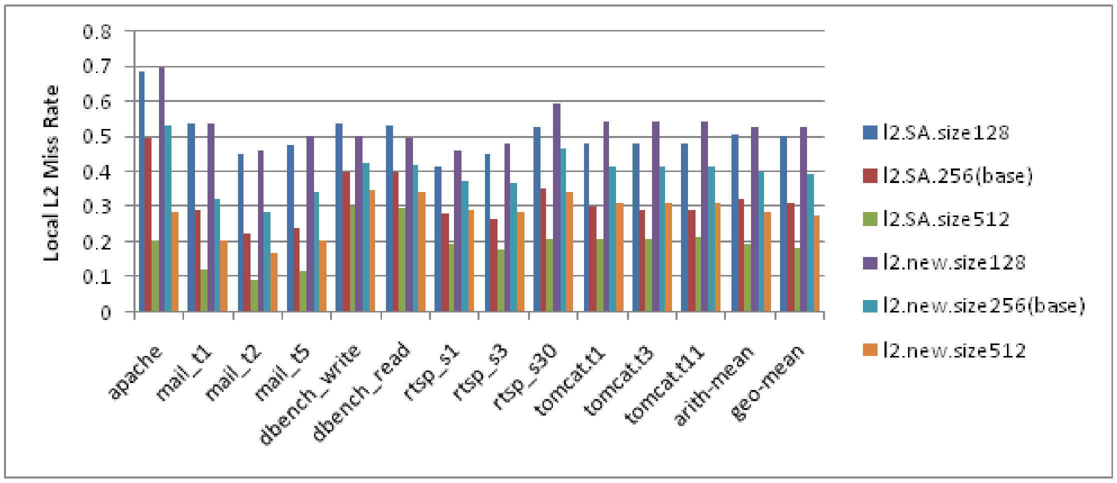


Figure 3.12: Newcache as L2 Cache: Local L2 Miss Rate for Different Cache Sizes

Table 3.14: Newcache as L2 Cache: Different Cache Sizes' Local L2 Miss Rate Increase Relative to SA

	128KB		256KB		512KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	0.5053	0.5307	0.3212	0.3991	0.1961	0.2852
Increase relative to SA		5.05%		24.26%		45.46%

From the results, we may find that on average, as the L2 cache size increases, the L2 Local Miss Rate decreases a lot for both SA cache and Newcache. However, compared with SA cache with the same size, Newcache performs worse, especially for larger cache sizes. Nevertheless, the Newcache's L2 local miss rate results do not influence the performance of program execution too much, as we can see from the above results of IPC and data cache miss rate. We give some reasons:

1. The average data cache miss rate from Figure 3.10 and Figure 3.11 are already very low; under this case, even Newcache as L2 cache has larger local L2 miss rate than SA cache as L2 cache, the performance of program execution, which is directly reflected by the IPC, is not influenced too much.
2. For the base configurations shown in Table 3.2, both L1 Dcache and L2 cache are 8-way SA cache. If SA cache uses LRU or LRU-like replacement policies, the data in lower level cache may very likely reside in higher level caches. However, Newcache as L2 cache is different. Due to Newcache's random eviction scheme, it's more likely to evict cache lines which still have their copies in L1 data cache. So next time when these cache lines need to be evicted from L1 Data cache, L2 cache miss will be incurred if cache lines are dirty and data need to be written back. Currently, we just model a single core under gem5; things will get more complicated in a real-world multi-core case, which is common for servers and even today's desktops and laptops. Due to the data sharing problem, some implementation requires data to be written-through at least to the lowest-level shared cache (L3 cache in Intel i7 case). Thus more cache coherence issues need to be considered when we implement

Newcache on large memory hierarchies.

3. Server applications tend to do a lot of stuff, like accessing network stack, the disk subsystem, handling the user connections and requests, syncing large amounts of threads etc., which induce a lot of context switches and kernel trapping. L2 Newcache's random eviction scheme may cause those constantly-needed data and instructions to be replaced, which cause a lot of penalties on local L2 Cache Miss. In constract, SA cache tends to keep these cache lines for a long time.

4. For the two tests dbench_read and dbench_write, we can see that Newcache only performs a little worse on size 512kB, while on size 128kB, Newcache even has smaller L2 Miss Rate than SA cache. Perhaps dbench_read only focuses on reading files and dbench_write only focuses on writing files on the server, there are not much other work that needs to be done. For other testing series like rtsp and tomcat which require a lot of server side controls and actions, Newcache performs worse than SA cache on all cache sizes.

Figure 3.13 shows the Local L2 Miss Rate under different SA-Associativity and Newcache-Nebit for all the benchmarks, and Table 3.15 shows the different configurations' Local L2 Miss Rate increase relative to SA-8way. Similarly, Newcache basically performs worse than SA cache with the same cache size.

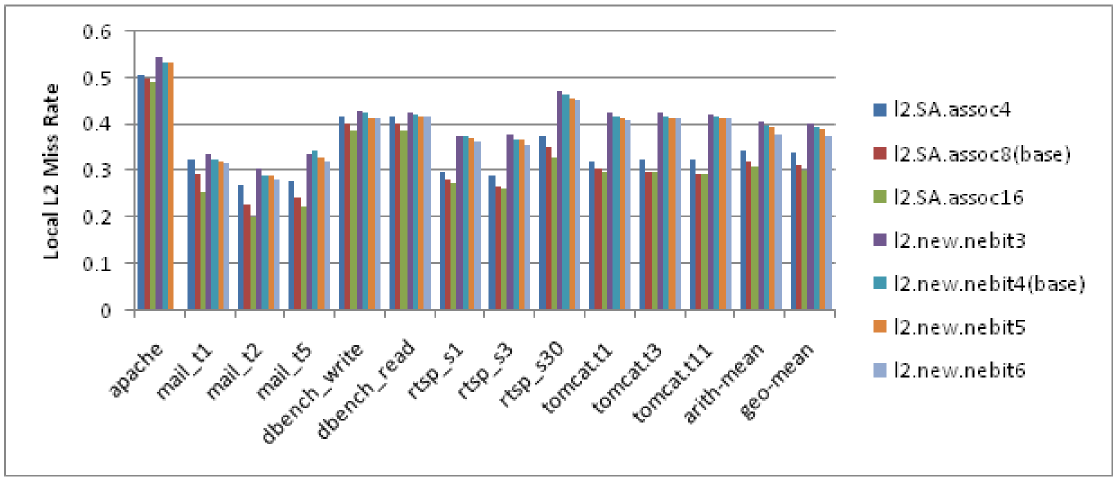


Figure 3.13: Newcache as L2 Cache: Local L2 Miss Rate for Different Associativity and Nebit

Table 3.15: Newcache as L2 Cache: Different Associativity and Nebit's Local L2 Miss Rate Increase Relative to SA-8way

	SA 4way	SA 8way	SA 16way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
Average	0.3445	0.3212	0.3071	0.4053	0.3991	0.3944	0.3771
Increase relative to SA-8way	7.25%	0.00%	-4.40%	26.19%	24.26%	22.78%	17.40%

3.4.4 Global L2 Miss Rate

We list the equation again: *Global L2 Miss Rate* $= \frac{l2.overall\_misses}{cpu.dcache.overall\_accesses}$. Figure 3.14 shows the Global L2 Miss Rate under different cache sizes for all the benchmarks, and Table 3.16 shows the different cache sizes' Global L2 Miss Rate increase relative to SA.

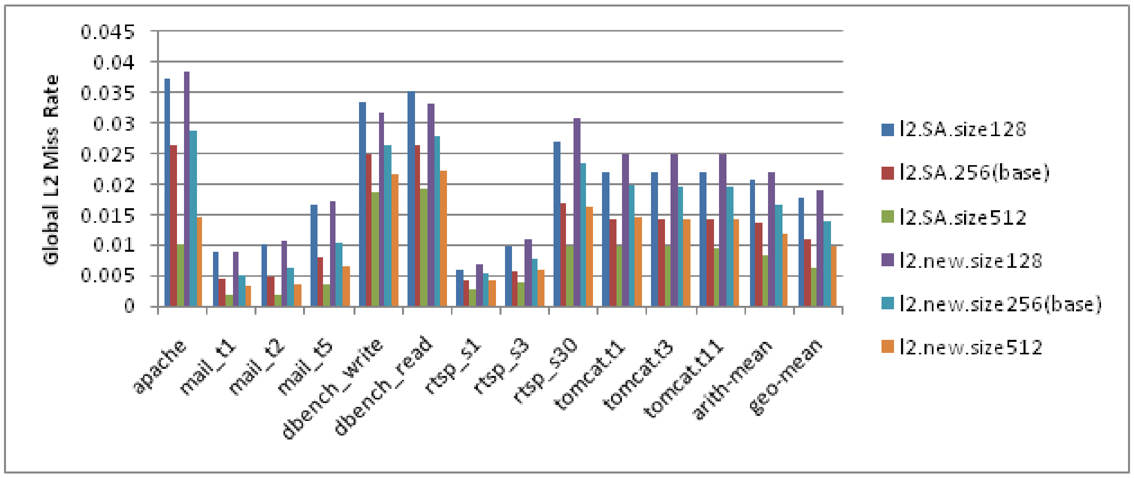


Figure 3.14: Newcache as L2 Cache: Global L2 Miss Rate for Different Cache Sizes

Table 3.16: Newcache as L2 Cache: Different Cache Sizes' Global L2 Miss Rate Increase Relative to SA

	128KB		256KB		512KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	0.0208	0.0219	0.0136	0.0167	0.0084	0.0118
Increase relative to SA		5.25%		22.15%		40.48%

The results are similar to what we have got from the previous Local L2 Miss Rate section. On average, as the L2 cache size increases, the L2 Global Miss Rate decreases a lot for both SA cache and Newcache. However, compared with SA cache with the same size, Newcache performs worse, especially for larger cache sizes.

Figure 3.15 shows the Global L2 Miss Rate under different SA-Associativity and Newcache-Nebit for all the benchmarks, and Table 3.17 shows the different configurations' Global L2 Miss Rate increase relative to SA-8way. Similarly, Newcache basically performs worse than SA cache with the same cache size.

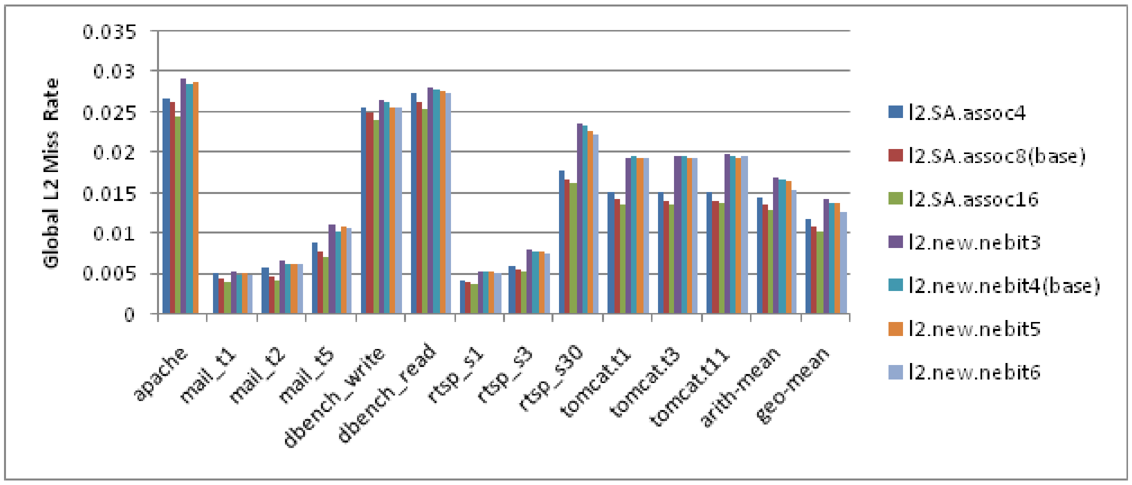


Figure 3.15: Newcache as L2 Cache: Global L2 Miss Rate for Different Associativity and Nebit

Table 3.17: Newcache as L2 Cache: Different Associativity and Nebit's Global L2 Miss Rate Increase Relative to SA-8way

	SA 4way	SA 8way	SA 16way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
Average	0.0144	0.0136	0.0130	0.0169	0.0167	0.0165	0.0153
Increase relative to SA-8way	5.68%	0.00%	-5.08%	23.83%	22.15%	21.14%	12.04%

3.5 Testing Results: Newcache as Both L1 Data Cache and L2 Cache

In this test, I set both L1 Data Cache and L2 Cache as Newcache, and measure IPC, Data Cache Miss Rate, Local L2 Miss Rate and Global L2 Miss Rate for all the benchmarks. The l1d.new-l2.new results are then compared with l1d.SA-l2.SA, l1d.new-l2.SA and l1d.SA-l2.new. The settings are all base configurations (base size, base associativity and base nebit in Table 3.2).

3.5.1 IPC

From Figure 3.16 and Table 3.18, on average, the IPC for l1d.new-l2.new is just slightly smaller than that for l1d.SA-l2.SA. Basically, even if both L1 DCache and L2 Cache are Newcache, IPC does not change a lot.

Table 3.18: Newcache as Both L1-DCache and L2 Cache: IPC Increase Relative to l1d.SA-l2.SA

	l1d.SA-l2.SA	l1d.new-l2.SA	l1d.SA-l2.new	l1d.new-l2.new
Average	0.3016	0.3004	0.2951	0.2933
Increase relative to l1d.SA-l2.SA	0.00%	-0.37%	-2.16%	-2.73%

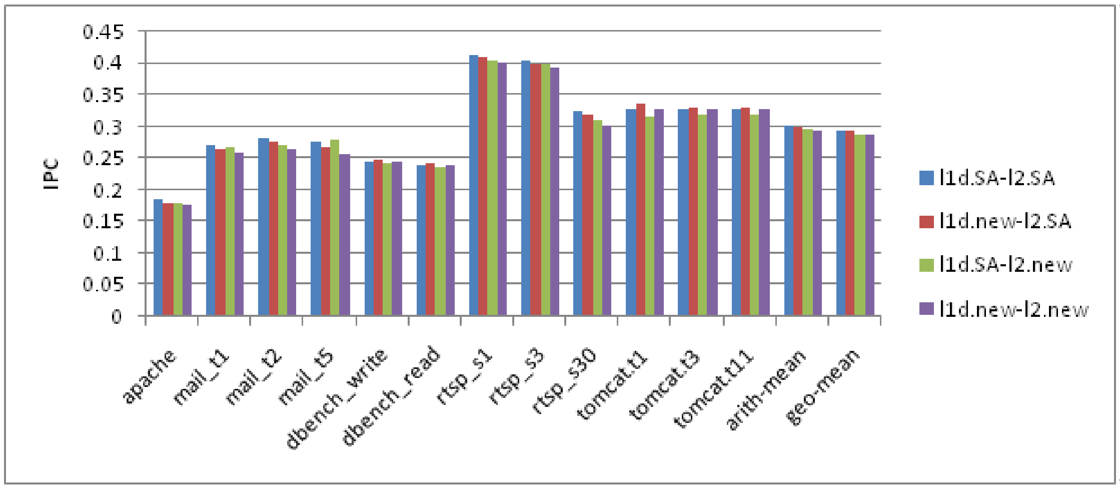


Figure 3.16: Newcache as Both L1-DCache and L2 Cache: IPC

3.5.2 DCache Miss Rate

Figure 3.17 shows the Data Cache Miss Rate of the four configurations for all the benchmarks, and Table 3.19 summarizes on average the DCache Miss Rate Increase of the four configurations relative to l1d.SA-l2.SA. l1d.new-l2.new induces only slightly more DCache Miss Rate than that of l1d.SA-l2.SA. Basically, even if both L1 DCache and L2 Cache are Newcache, DCache Miss Rate does not increase a lot.

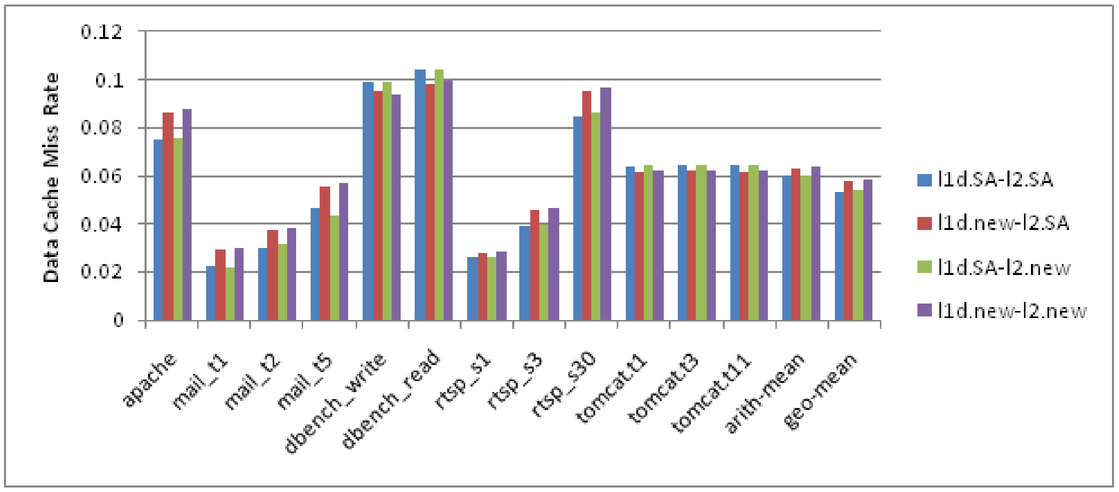


Figure 3.17: Newcache as Both L1-DCache and L2 Cache: DCache Miss Rate

Table 3.19: Newcache as Both L1-DCache and L2 Cache: DCache Miss Rate Increase Relative to l1d.SA-l2.SA

	l1d.SA-l2.SA	l1d.new-l2.SA	l1d.SA-l2.new	l1d.new-l2.new
Average	0.0599	0.0631	0.0602	0.0639
Increase relative to l1d.SA-l2.SA	0.00%	5.25%	0.54%	6.57%

3.5.3 Local L2 Miss Rate

Figure 3.18 shows the Local L2 Miss Rate of the four configurations for all the benchmarks, and Table 3.20 summarizes on average the Local L2 Miss Rate Increase of the four configurations relative to l1d.SA-l2.SA. Still similar to the results we get in previous Section 3.4.3, Newcache as L2 Cache performs worse on Local L2 Miss Rate than SA cache. However, Newcache as L1-DCache performs better, on average, than conventional SA cache as L1-DCache. The reasonings are similar to what I have analyzed in Section 3.3.3.

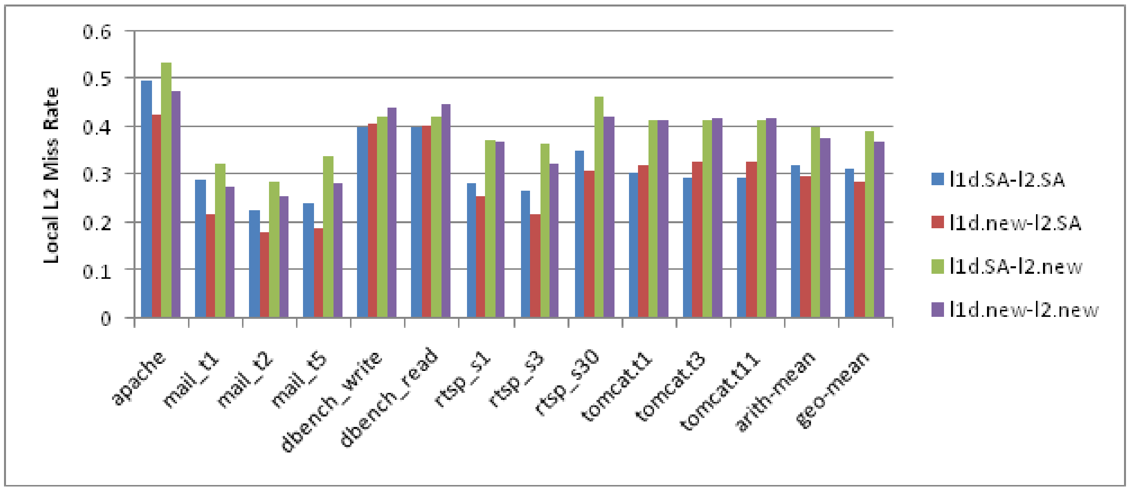


Figure 3.18: Newcache as Both L1-DCache and L2 Cache: Local L2 Miss Rate

Table 3.20: Newcache as Both L1-DCache and L2 Cache: Local L2 Miss Rate Increase Relative to l1d.SA-l2.SA

	l1d.SA-l2.SA	l1d.new-l2.SA	l1d.SA-l2.new	l1d.new-l2.new
Average	0.3212	0.2992	0.3991	0.3788
Increase relative to l1d.SA-l2.SA	0.00%	-6.86%	24.26%	17.92%

3.5.4 Global L2 Miss Rate

Figure 3.19 shows the Global L2 Miss Rate of the four configurations for all the benchmarks, and Table 3.21 summarizes on average the Global L2 Miss Rate Increase of the four configurations relative to l1d.SA-l2.SA. Still similar to the results we get in previous Section 3.4.4, Newcache as L2 Cache performs worse on Global L2 Miss Rate than SA cache. However, Newcache as L1-DCache performs better (by 6.86%) than SA cache used as L1-DCache. Note also that the Global L2 miss rates are an order of magnitude smaller than the local L2 miss rates. The reasonings are similar to what I have analyzed in Section 3.3.3.

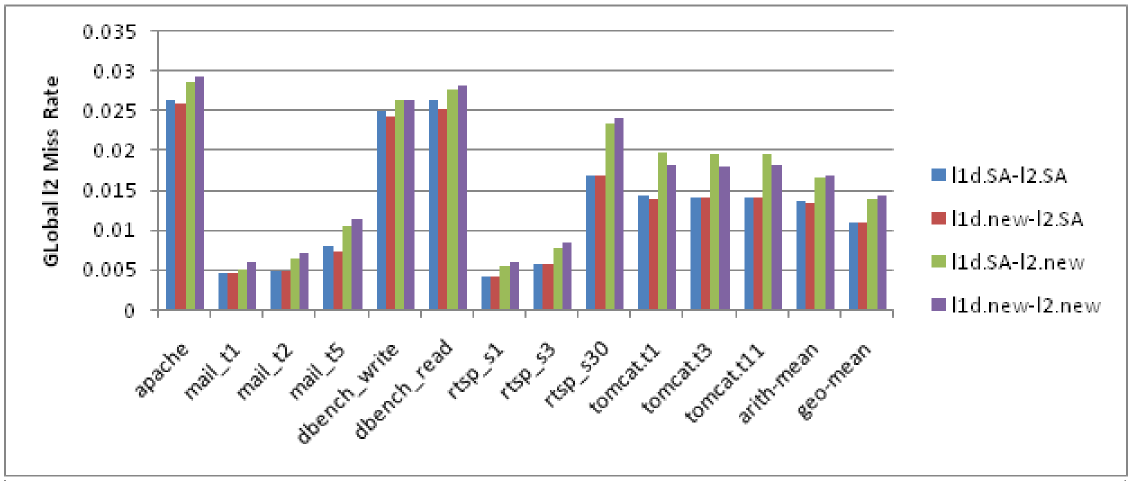


Figure 3.19: Newcache as Both L1-DCache and L2 Cache: Global L2 Miss Rate

Table 3.21: Newcache as Both L1-DCache and L2 Cache: Global L2 Miss Rate Increase Relative to l1d.SA-l2.SA

	l1d.SA-l2.SA	l1d.new-l2.SA	l1d.SA-l2.new	l1d.new-l2.new
Average	0.0136	0.0134	0.0167	0.0167
Increase relative to l1d.SA-l2.SA	0.00%	-1.92%	22.15%	22.44%

3.6 Testing Results: Newcache as L1 Instruction Cache (Part A)

In this experiment, we test the server side's IPC, ICache Miss Rate, Local L2 Instruction Miss Rate and Global L2 Instruction Miss Rate for all the benchmarks under 6 configurations:

- All L1-ICache, L1-DCache and L2 Cache are SA-base (Table 3.2)
- L1-ICache as SA-base, L1-DCache and L2 Cache as Newcache-base (same size as SA cache and nebit k=4)
- L1-ICache as Newcache (k=3), L1-DCache and L2 Cache as Newcache-base
- L1-ICache as Newcache (k=4), L1-DCache and L2 Cache as Newcache-base
- L1-ICache as Newcache (k=5), L1-DCache and L2 Cache as Newcache-base
- L1-ICache as Newcache (k=6), L1-DCache and L2 Cache as Newcache-base

L1 Instruction Cache is similar to L1 Data Cache. Instruction cache also fetches data in cache-blocks, except the data are x86-instructions (machine codes). When the program runs, the instructions are stored in the process's text section. So instructions tend to be fetched from a fixed memory region. Also, programs tend to run the instruction right after the current executing instruction, if the current executing instruction is not a taken jump. For DCache, data can be fetched from a much wider range of memory space: stack, heap and initialized region etc. However, the ways the ICache and DCache mechanisms work are similar.

3.6.1 IPC

So according to our analysis, IPC should not change a lot if we use Newcache as L1 Instruction cache. Figure 3.20 and Table 3.22 below show the results. Basically, IPC only decreases by a small amount by replacing SA cache with Newcache as L1-ICache.

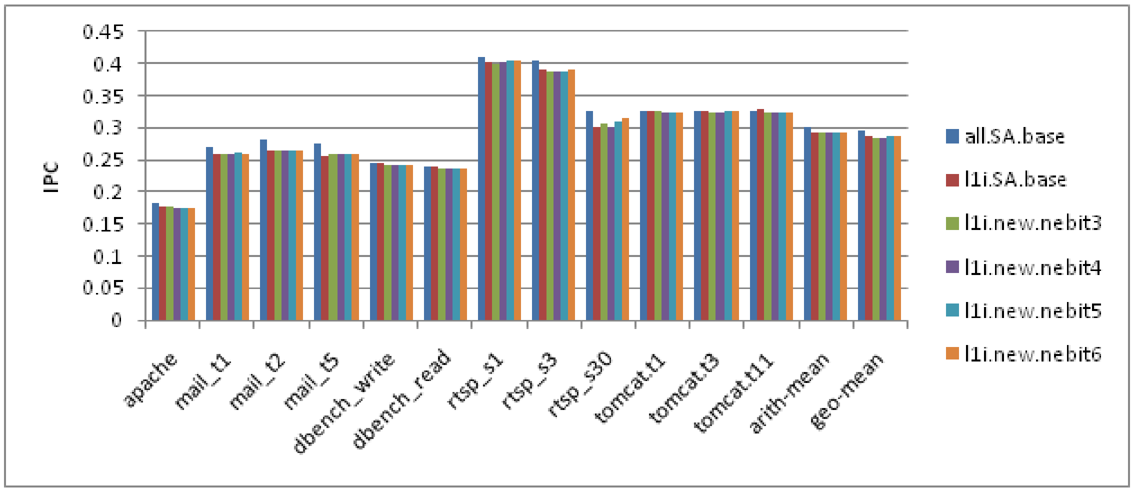


Figure 3.20: Newcache as L1-ICache: IPC

Table 3.22: Newcache as L1-ICache: IPC, and IPC Increase Relative to all.SA.base

	all.SA .base	l1i.SA .base	l1i.new .nebit3	l1i.new .nebit4	l1i.new .nebit5	l1i.new .nebit6
Average	0.3016	0.2933	0.2921	0.2914	0.2927	0.2930
Increase relative to all.SA.base	0.00%	-2.73%	-3.15%	-3.39%	-2.95%	-2.84%

3.6.2 ICache Miss Rate

Figure 3.21 and Table 3.23 show the ICache Miss Rate results. Similar to the analysis we did in Section 3.3.1 and 3.3.2, Newcache should perform almost the same as SA cache on L1-ICache, and ICache Miss Rate slightly decreases as Newcache nebit increases.

Table 3.23: Newcache as L1-ICache: ICache Miss Rate Increase Relative to all.SA.base

	all.SA .base	l1i.SA .base	l1i.new .nebit3	l1i.new .nebit4	l1i.new .nebit5	l1i.new .nebit6
Average	0.1309	0.1326	0.1395	0.1377	0.1362	0.1364
Increase relative to all.SA.base	0.00%	1.32%	6.60%	5.21%	4.06%	4.25%

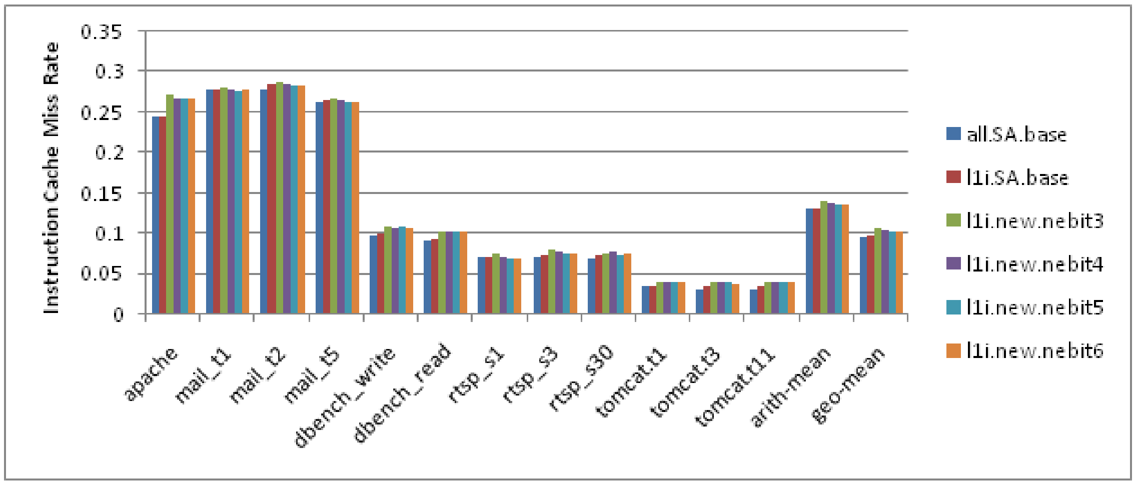


Figure 3.21: Newcache as L1-ICache: ICache Miss Rate

3.6.3 Local L2 Miss Rate for Instructions

Figure 3.22 and Table 3.24 show the local L2 Instruction Miss Rate results. Similar to what we have analyzed in Section 3.4.3, on average, L2 cache as Newcache increases the Local L2 instruction Miss Rate by about 40% compared with L2 cache as a conventional SA cache. Newcache used as ICache does not induce more local L2 miss rate. However, setting L2 cache as a conventional SA cache, and getting the performance results of ICache as Newcache would be necessary in the future work.

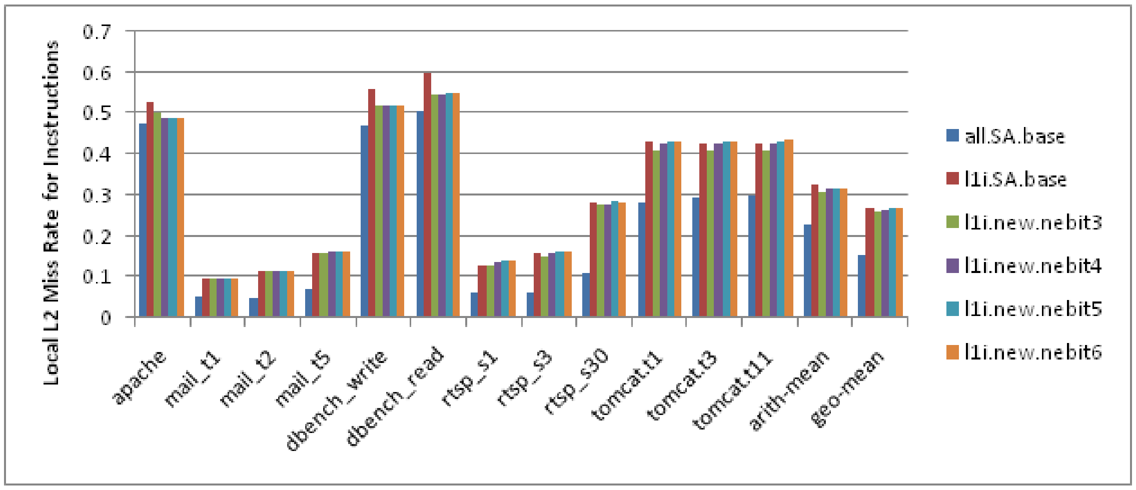


Figure 3.22: Newcache as L1-ICache: Local L2 Instruction Miss Rate

Table 3.24: Newcache as L1-ICache: Local L2 Instruction Miss Rate Increase Relative to all.SA.base

	all.SA .base	l1i.SA .base	l1i.new .nebit3	l1i.new .nebit4	l1i.new .nebit5	l1i.new .nebit6
Average	0.2276	0.3261	0.3101	0.3152	0.3177	0.3181
Increase relative to all.SA.base	0.00%	43.30%	36.26%	38.51%	39.62%	39.80%

3.6.4 Global L2 Miss Rate for Instructions

Figure 3.23 and Table 3.25 show the global L2 Instruction Miss Rate results. Similar to what we have analyzed in Section 3.4.3 and 3.4.4, on average, L2 cache as Newcache increases the Global L2 instruction Miss Rate by about 40% compared with L2 cache as a conventional SA cache. Note, however, that this is a large percentage of a very small global L2 miss rate of on average 2-3%. Also, the performance of program execution, which is directly reflected by the IPC, is not influenced too much.

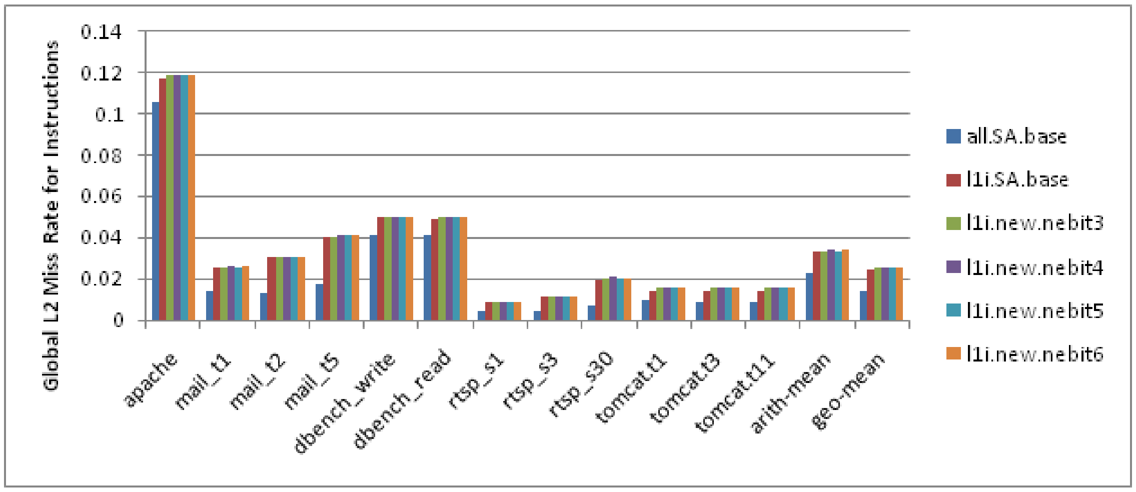


Figure 3.23: Newcache as L1-ICache: Global L2 Instruction Miss Rate

Table 3.25: Newcache as L1-ICache: Global L2 Instruction Miss Rate Increase Relative to all.SA.base

	all.SA .base	l1i.SA .base	l1i.new .nebit3	l1i.new .nebit4	l1i.new .nebit5	l1i.new .nebit6
Average	0.0230	0.0329	0.0336	0.0337	0.0337	0.0337
Increase relative to all.SA.base	0.00%	42.98%	46.26%	46.70%	46.42%	46.53%

3.7 Testing Results: Newcache as L1 Instruction Cache (Part B)

In this experiment, we fix L1-DCache and L2 Cache to be SA-base (Table 3.2), and test the server side's IPC, ICache Miss Rate, Local L2 Instruction Miss Rate and Global L2 Instruction Miss Rate for all the benchmarks under the below 5 configurations:

- L1-ICache as SA-base
- L1-ICache as Newcache (k=3)
- L1-ICache as Newcache (k=4)

- L1-ICache as Newcache (k=5)

- L1-ICache as Newcache (k=6)

It's important to see the the performance of Newcache as ICache alone, with DCache and L2 Cache fixed as the conventional SA cache.

3.7.1 IPC

According to Figure 3.24 and Table 3.26 below, IPC almost stays the same if we use Newcache as L1 Instruction cache, compared with the all.SA.base case. Also, IPC cannot benefit from increasing the nebit value of ICache. Since instructions tend to be fetched from the same page (virtual and physical), increasing the LDM cache size does not influence IPC.

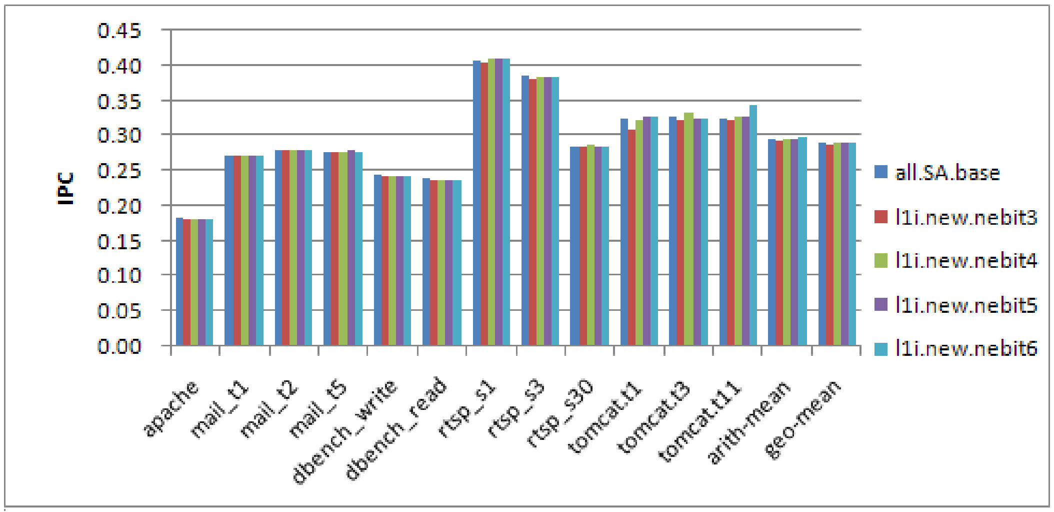


Figure 3.24: Newcache as L1-ICache: IPC

Table 3.26: Newcache as L1-ICache: IPC, and IPC Increase Relative to all.SA.base

	all.SA .base	l1i.new .nebit3	l1i.new .nebit4	l1i.new .nebit5	l1i.new .nebit6
Average	0.2958	0.2928	0.2963	0.2960	0.2974
Increase relative to all.SA.base	0.00%	-1.01%	0.17%	0.08%	0.55%

3.7.2 ICache Miss Rate

Figure 3.25 and Table 3.27 show the ICache Miss Rate results. As expected, on average Newcache induces about 3% more ICache Miss Rate than that of the all.SA.base case. I would say that the random eviction policy does pose some negative effects on the ICache Miss Rate, destroying some conventional advantages of SA cache (e.g. temporal/spatial locality of instructions, etc.). However, for some benchmark (e.g. tomcat, etc.), Newcache as ICache almost induces no penalty on ICache Miss Rate.

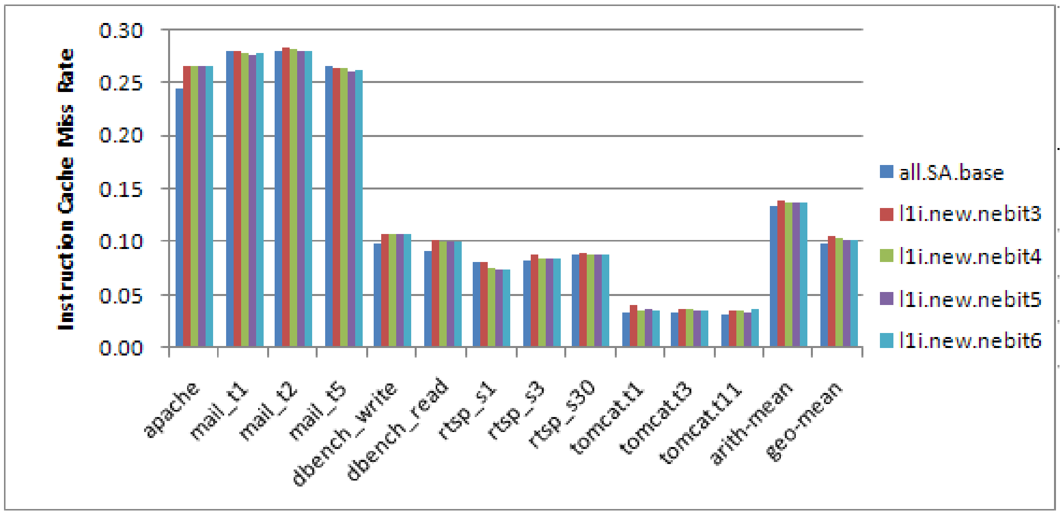


Figure 3.25: Newcache as L1-ICache: ICache Miss Rate

Table 3.27: Newcache as L1-ICache: ICache Miss Rate Increase Relative to all.SA.base

	all.SA .base	l1i.new .nebit3	l1i.new .nebit4	l1i.new .nebit5	l1i.new .nebit6
Average	0.1344	0.1398	0.1381	0.1372	0.1374
Increase relative to all.SA.base	0.00%	4.00%	2.71%	2.06%	2.23%

3.7.3 Local L2 Miss Rate for Instructions

Figure 3.26 and Table 3.28 show the local L2 Instruction Miss Rate results. It's interesting to see the Local L2 Miss Rates for Instructions for most benchmarks are much lower if we use Newcache as ICache alone. Actually, the high L1 ICache Miss Rate causes more outstanding MSHR accesses to the L2 Cache. However, these instructions tend to stay in the L2 Cache, which means that the amount of total L2 misses does not increase. That is why we have lower L2 Miss Rate for l1i.new.nebit3-6 cases.

Table 3.28: Newcache as L1-ICache: Local L2 Instruction Miss Rate Increase Relative to all.SA.base

	all.SA .base	l1i.new .nebit3	l1i.new .nebit4	l1i.new .nebit5	l1i.new .nebit6
Average	0.2251	0.2097	0.2128	0.2088	0.2142
Increase relative to all.SA.base	0.00%	-6.83%	-5.46%	-7.26%	-4.87%

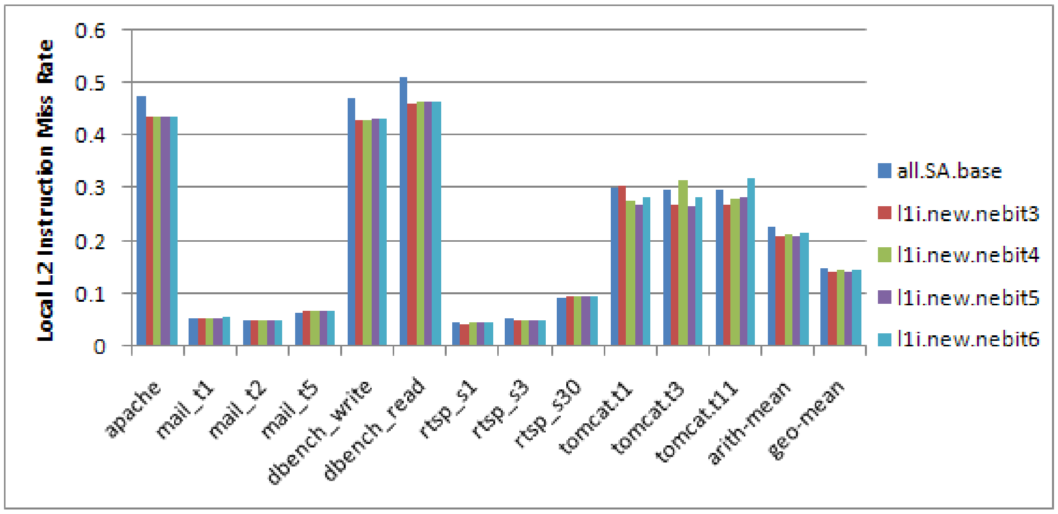


Figure 3.26: Newcache as L1-ICache: Local L2 Instruction Miss Rate

3.7.4 Global L2 Miss Rate for Instructions

Figure 3.27 and Table 3.29 show the global L2 Instruction Miss Rate results. The average global L2 Miss Rate almost shows no difference between Newcache with different nebits and conventional SA cache, which indicates the fact that the total L2 cache misses do not increase when L1 ICache changes from SA cache to Newcache.

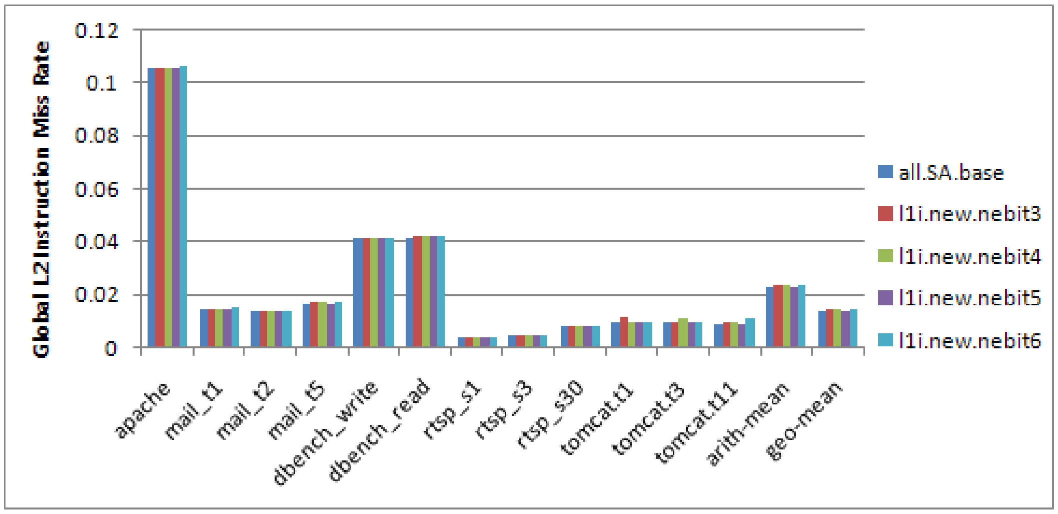


Figure 3.27: Newcache as L1-ICache: Global L2 Instruction Miss Rate

Table 3.29: Newcache as L1-ICache: Global L2 Instruction Miss Rate Increase Relative to all.SA.base

	all.SA .base	l1i.new .nebit3	l1i.new .nebit4	l1i.new .nebit5	l1i.new .nebit6
Average	0.0229	0.0233	0.0232	0.0230	0.0232
Increase relative to all.SA.base	0.00%	1.69%	1.44%	0.43%	1.47%

3.8 Testing Results: Newcache as L1 Instruction Cache (Part C)

In this experiment, we still fix L2 Cache to be SA-base (Table 3.2), and test the server side's IPC, ICache Miss Rate, Local L2 Instruction Miss Rate and Global L2 Instruction Miss Rate for all the benchmarks under the below 8 configurations:

- L1-ICache as Newcache (k=3, 4, 5, 6), and L1-DCache as Newcache (k=4)

- L1-ICache as Newcache (k=3, 4, 5, 6), and L1-DCache as Newcache (k=6)

It's also important to see the the performance of Newcache as ICache, with DCache set to be Newcache, but L2 Cache still fixed as the conventional SA cache.

3.8.1 IPC

According to Figure 3.28 and Table 3.30 below, similar to section 3.7.1, IPC almost stays the same if we use Newcache as L1 Instruction cache, compared with the all.SA.base case, except for the rtsp_s3 and rtsp_s30 case, which indicates that when the streaming server side tries to handle many media streaming requests, the system performance might be degraded. Also, IPC cannot benefit from increasing the nebit value of ICache. Since instructions tend to be fetched from the same page (virtual and physical), increasing the LDM cache size does not influence IPC.

Table 3.30: Newcache as L1-ICache: IPC, and IPC Increase Relative to all.SA.base

	all.SA .base	l1i.new .nebit3- l1d.new .nebit4	l1i.new .nebit4- l1d.new .nebit4	l1i.new .nebit5- l1d.new .nebit4	l1i.new .nebit6- l1d.new .nebit4	l1i.new .nebit3- l1d.new .nebit6	l1i.new .nebit4- l1d.new .nebit6	l1i.new .nebit5- l1d.new .nebit6	l1i.new .nebit6- l1d.new .nebit6
Average	0.2958	0.2955	0.2940	0.2951	0.2951	0.2927	0.2953	0.2951	0.2950
Increase relative to all.SA.base	0.00%	-0.76%	-0.60%	-0.24%	-0.23%	-1.06%	-0.16%	-0.24%	-0.25%

3.8.2 ICache Miss Rate

Figure 3.29 and Table 3.31 show the ICache Miss Rate results. On average Newcache as both ICache and Dcache induces about 3% more ICache Miss Rate than that of the all.SA.base case. Still, I would say that the random eviction policy does pose some negative effects on the ICache

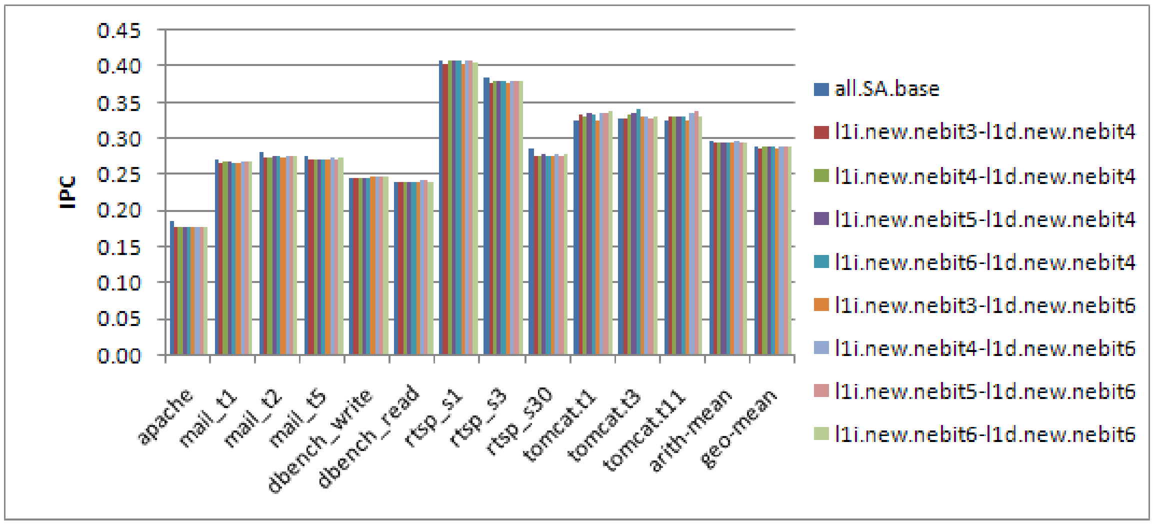


Figure 3.28: Newcache as L1-ICache: IPC

Miss Rate, destroying some conventional advantages of SA cache (e.g. temporal/spatial locality of instructions, etc.). However, for some benchmark (e.g. tomcat, etc.), Newcache as ICache almost induces no penalty on ICache Miss Rate.

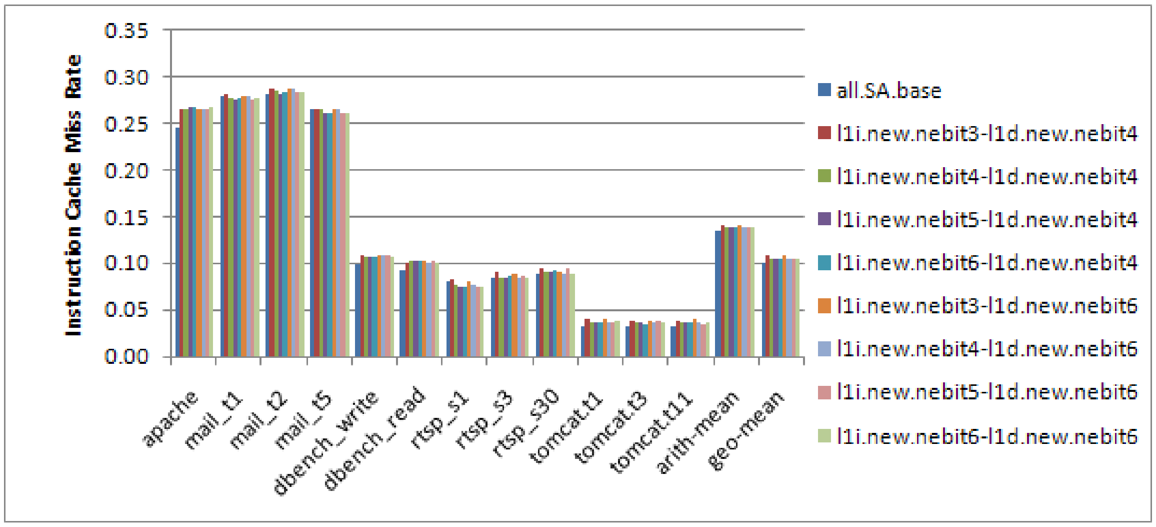


Figure 3.29: Newcache as L1-ICache: ICache Miss Rate

3.8.3 Local L2 Miss Rate for Instructions

Figure 3.30 and Table 3.32 show the local L2 Instruction Miss Rate results. Similar to section 3.7.3, the Local L2 Miss Rates for Instructions for most benchmarks are much lower. Actually, the high L1 ICache Miss Rate causes more outstanding MSHR accesses to the L2 Cache. However, these

Table 3.31: Newcache as L1-ICache: ICache Miss Rate Increase Relative to all.SA.base

	all.SA .base	l1i.new .nebit3- l1d.new .nebit4	l1i.new .nebit4- l1d.new .nebit4	l1i.new .nebit5- l1d.new .nebit4	l1i.new .nebit6- l1d.new .nebit4	l1i.new .nebit3- l1d.new .nebit6	l1i.new .nebit4- l1d.new .nebit6	l1i.new .nebit5- l1d.new .nebit6	l1i.new .nebit6- l1d.new .nebit6
Average	0.1344	0.1409	0.1389	0.1378	0.1382	0.1407	0.1387	0.1383	0.1380
Increase relative to all.SA.base	0.00%	4.78%	3.33%	2.50%	2.78%	4.68%	3.17%	2.90%	2.68%

instructions tend to stay in the L2 Cache, which means that the amount of total L2 misses does not increase. That is why we have lower L2 Miss Rate for l1i.new.nebit3-6 cases.

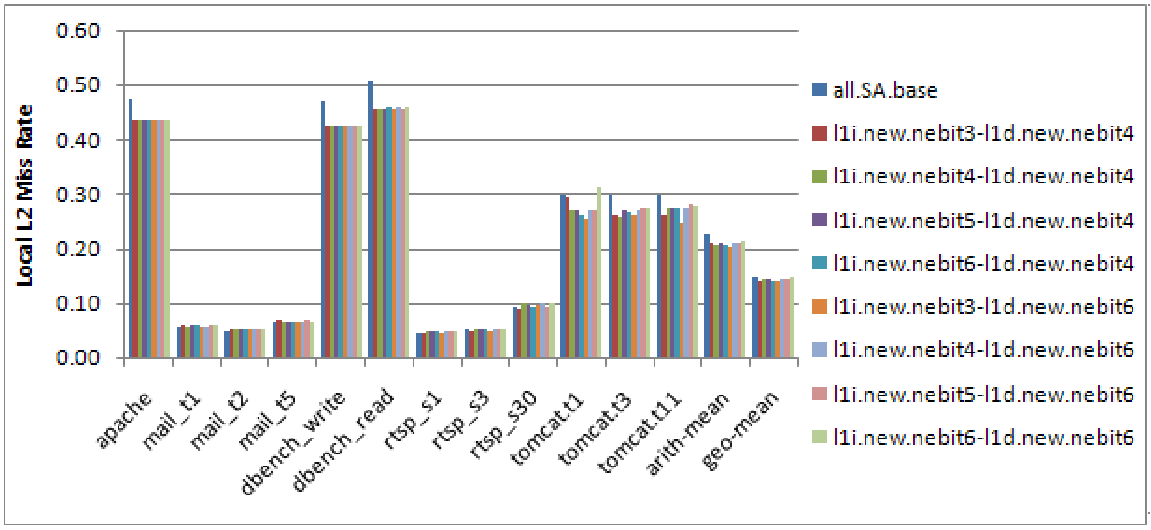


Figure 3.30: Newcache as L1-ICache: Local L2 Instruction Miss Rate

Table 3.32: Newcache as L1-ICache: Local L2 Instruction Miss Rate Increase Relative to all.SA.base

	all.SA .base	l1i.new .nebit3- l1d.new .nebit4	l1i.new .nebit4- l1d.new .nebit4	l1i.new .nebit5- l1d.new .nebit4	l1i.new .nebit6- l1d.new .nebit4	l1i.new .nebit3- l1d.new .nebit6	l1i.new .nebit4- l1d.new .nebit6	l1i.new .nebit5- l1d.new .nebit6	l1i.new .nebit6- l1d.new .nebit6
Average	0.2251	0.2079	0.2073	0.2089	0.2070	0.2033	0.2086	0.2092	0.2133
Increase relative to all.SA.base	0.00%	-7.66%	-7.90%	-7.19%	-8.04%	-9.71%	-7.35%	-7.09%	-5.25%

3.8.4 Global L2 Miss Rate for Instructions

Figure 3.31 and Table 3.33 show the global L2 Instruction Miss Rate results. The average global L2 Miss Rate almost shows no difference between Newcache with different nebits and conventional SA cache, which indicates the fact that the total L2 cache misses do not increase when L1 ICache changes from SA cache to Newcache.

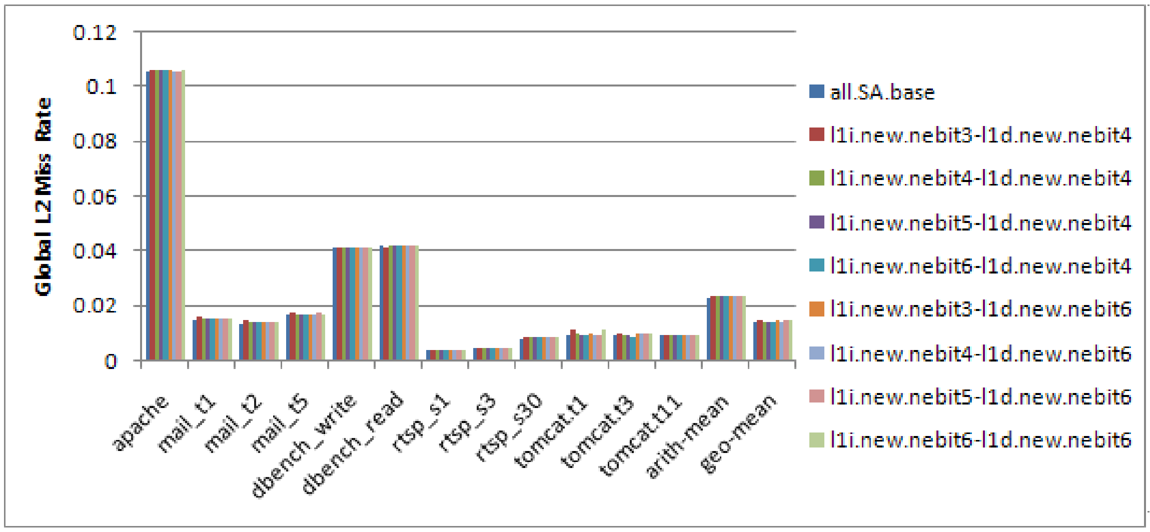


Figure 3.31: Newcache as L1-ICache: Global L2 Instruction Miss Rate

Table 3.33: Newcache as L1-ICache: Global L2 Instruction Miss Rate Increase Relative to all.SA.base

	all.SA .base	l1i.new .nebit3- l1d.new .nebit4	l1i.new .nebit4- l1d.new .nebit4	l1i.new .nebit5- l1d.new .nebit4	l1i.new .nebit6- l1d.new .nebit4	l1i.new .nebit3- l1d.new .nebit6	l1i.new .nebit4- l1d.new .nebit6	l1i.new .nebit5- l1d.new .nebit6	l1i.new .nebit6- l1d.new .nebit6
Average	0.0229	0.0234	0.0231	0.0231	0.0231	0.0232	0.0231	0.0232	0.0233
Increase relative to all.SA.base	0.00%	2.06%	1.11%	1.06%	0.76%	1.48%	1.03%	1.39%	1.80%

3.9 Chapter Summary

In this chapter, we introduce gem5's configuration methodology and how we set the server side and client side under gem5. Then we describe the way we choose cloud server benchmarks. Afterwards, by using different cache configurations, we do the following tests and performance measurements for all the cloud server benchmarks under gem5:

- L1-DCache performance measurement
- L2 Cache performance measurement
- L1-DCache and L2 Cache performance measurement
- L1-ICache performance measurement

The results basically show that L1-DCache as Newcache and L1-ICache as Newcache have comparable performance results with L1-DCache as SA cache and L1-ICache as SA cache. Newcache as L2 cache has higher global and local l2 miss rate compared with SA cache as L2 cache. However, the performance of program execution, which is directly reflected by the IPC, is not influenced too much (only 2-3%).

In section 3.6, we studied configurations where when Newcache is used as the L1-ICache, it is also used as the L1-DCache and L2 cache. In section 3.7, we considered the performance of Newcache used as L1-ICache (k = 3 to 6), with the L1-DCache and L2 cache remaining as SA caches, to see the impact of just changing the L1-ICache. In section 3.8, we also considered the configurations where both Level 1 caches (L1-ICache and L1-DCache) are Newcache (L1-ICache: k = 3 to 6; L1-DCache: k = 4 and 6), but L2 cache remains as a conventional SA cache, which would improve the local and global L2 cache miss rates, without compromising the security provided to the L1 caches.

Chapter 4

Security Analysis of Newcache as Instruction Cache

4.1 Reconstruction of Instruction Cache Side-Channel Attack

4.1.1 Instruction Cache PRIME and PROBE

We have Intel(R) Xeon(R) CPU in our lab testbed, of which the Instruction Cache (ICache) is a 32kB, 8-way set-associative (SA) cache. The block size is 64 bytes, thus there are 32k/64/8 = 64 sets. For a typical virtual memory address (Figure 4.1), the last 6 bits (bit 5 to bit 0) are the offset for the 64-byte block, while bit 11 to bit 6 are used to index the cache sets.

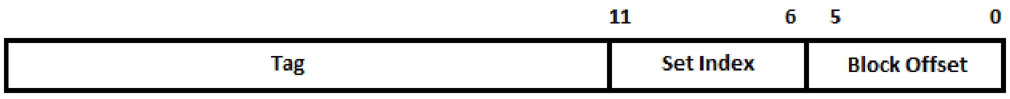


Figure 4.1: Typical Memory Address

When the cpu needs to execute instructions from some virtual address, it will first check if these instructions are already in the ICache, if not, these instructions will be fetched from higher-level caches or physical memory. A virtual memory address will always be mapped into the same set location inside the ICache. For example, virtual address 0xFFFFFABF has set index $101010_2 = 42_{10}$; thus the memory block corresponding to this address will be fetched to set 42. Moreover, the system usually allocates memory in a 4kB page size. Thus there are 4kB/64B = 64 cache-line-sized memory blocks inside each page.

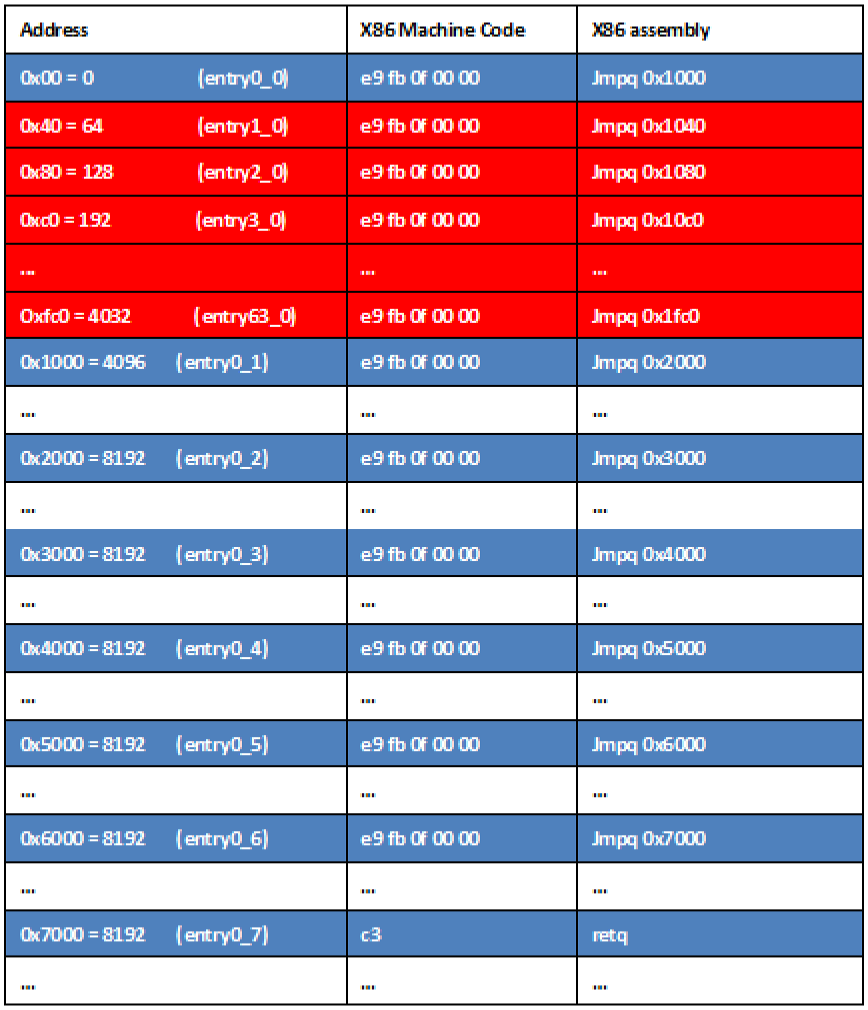
- Instruction Cache PRIME-ing and PROBE-ing:

Aciiçmez [22] described a ICache side-channel: for each individual cache set, timing how long it takes to read data to occupy this whole set. To do this, we need to first allocate a chunk of memory, the size of which equals the size of the ICache (32kB). 32kB is actually 8 contiguous 4kB memory pages. Within each page, we divide the page into 64 64-byte blocks (Figure 4.2). Within each block, only the first 5 bytes are used as machine codes, the rest can be anything (e.g. nop etc.). The 5 bytes of machine code are "e9 fb 0f 00 00", which jumps to an address 0x1000 (4kB) higher. For example, see the blue (light shaded) blocks, the code jumps from entry0_0 to entry0_1, then to entry0_2, ..., and finally to entry0_7. These blue blocks are in the same location in their corresponding memory pages, which means the addresses in these blue blocks have the same 12 least significant bits, so these blue blocks will be mapped to the same cache set (in this case, set0). Thus, doing an indirect call to entry0_0's address will fill cache set 0 (notice that entry0_7 is actually a retq instruction that returns to the caller). Filling a cache set is called PRIME-ing. The rest of the 63 entries from entry1_0 to entry63_0 in the first page perform similarly: doing indirect call to entry1_0 primes cache set 1, doing indirect call to entry2_0 primes cache set 2 etc.

We also want to measure the time it takes to prime each cache set, which is called PROBE-ing. X86 provides an instruction *rdtsc*, which stores the current time_stamp (cpu-cycles) in two registers. In the main function, we can do *rdtsc* just before we call entry0_0, and do *rdtsc* again just after entry0_7 returns. We then store the difference of the two measurements, which is the elapsed time of filling cache set 0. This is repeated for each of the 64 cache sets to produce a vector of elapsed time.

- The Prime-Probe Protocol:

Osvik et al. [38] described this common prime-probe protocol. A lot of access-driven cache attacks are achieved by first PRIME-ing and later PROBE-ing the cache. Specifically, the attacker needs to bind the prime-probe process to the same cpu used by the victim's process. The attacker's process first fills all the ICache sets as described above (PRIME-ing), then it waits for a prespecified PRIME-PROBE interval while the cpu and caches are utilized by the victim's process. After this interval, the attacker's process times the duration to refill the ICache sets (PROBE-ing) to learn the victim's cache activity. The victim's activity during the PRIME-PROBE interval will evict the attacker's instructions from the cache sets, which causes higher timing measurements for these sets during the attacker's PROBE phase. We also need to notice that the attacker's PROBE-ing actually PRIMEs the whole cache; so repeatedly PROBE-ing the cache with the PRIME-PROBE interval eliminates the need to separately do a PRIME.



Address	X86 Machine Code	X86 assembly
0x00 = 0 (entry0_0)	e9 fb 0f 00 00	Jmpq 0x1000
0x40 = 64 (entry1_0)	e9 fb 0f 00 00	Jmpq 0x1040
0x80 = 128 (entry2_0)	e9 fb 0f 00 00	Jmpq 0x1080
0xc0 = 192 (entry3_0)	e9 fb 0f 00 00	Jmpq 0x10c0
...	...	...
0xfc0 = 4032 (entry63_0)	e9 fb 0f 00 00	Jmpq 0x1fc0
0x1000 = 4096 (entry0_1)	e9 fb 0f 00 00	Jmpq 0x2000
...	...	...
0x2000 = 8192 (entry0_2)	e9 fb 0f 00 00	Jmpq 0x3000
...	...	...
0x3000 = 8192 (entry0_3)	e9 fb 0f 00 00	Jmpq 0x4000
...	...	...
0x4000 = 8192 (entry0_4)	e9 fb 0f 00 00	Jmpq 0x5000
...	...	...
0x5000 = 8192 (entry0_5)	e9 fb 0f 00 00	Jmpq 0x6000
...	...	...
0x6000 = 8192 (entry0_6)	e9 fb 0f 00 00	Jmpq 0x7000
...	...	...
0x7000 = 8192 (entry0_7)	c3	retq
...	...	...

Figure 4.2: 32kB Contiguous Memory Chunk for PRIME-ing

4.1.2 Clean PROBE-ing Result

Since the test-bed server has 8 CPUs, measurements on different CPUs may result in slightly different results, so I bind the main prime-probe program to CPU5.

A clean PROBE-ing means PROBE-ing happens directly after PRIME-ing. For this case, before probing, the 32kB cache has already been occupied with the 32kB allocated instructions. We want to see if the resulting PROBE-ing time for each of the 64 sets is minimum or not. We did 30000 probings for all the 64 sets and averaged the results (Figure 4.3).

From the results, we can see that most of the sets are around 100 cycles. Only set0, set1, set32 and set33 are distinctly larger than 100 cycles. First, PRIME() function has virtual addresses 0x52030 to 0x5203d, the index bits (bit 11 to bit 6 of the address) of which show index 0. These addresses are in set0, which means that after PRIME-ing the last set, set0 is again occupied by PRIME()'s codes before PROBE-ing; thus PROBE-ing set0 will take a lot longer. Also, the function PROBE()

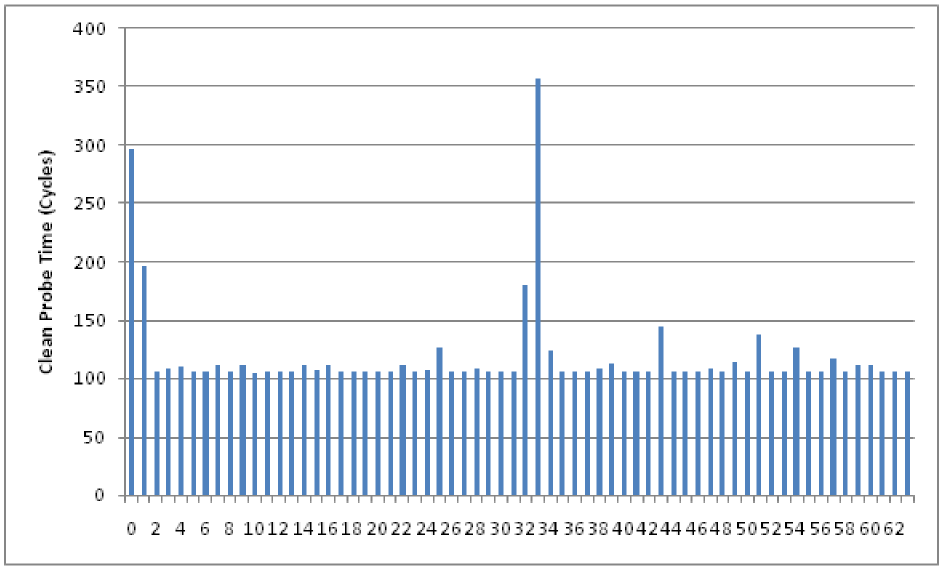


Figure 4.3: Clean PROBE-ing Result for 32kB 8-way Set-Associative Cache

is located in the address range 0x51820 to 0x51879 in the final compiled library. These addresses correspond to set32 and set33 in the instruction cache, thus probing set32 and set33 will take longer time.

Moreover, using printf() function to print the probing time directly inside the PROBE() function is not a good idea. Because printf() function itself occupies a lot of cache sets before PROBE-ing each set (Figure 4.4), which totally ruins the purpose of PROBE-ing. Instead, we could store the PROBE-ing time in a global array. After PROBE() finishes for all the 64 cache sets, we could use another function printtime() to print the global array.

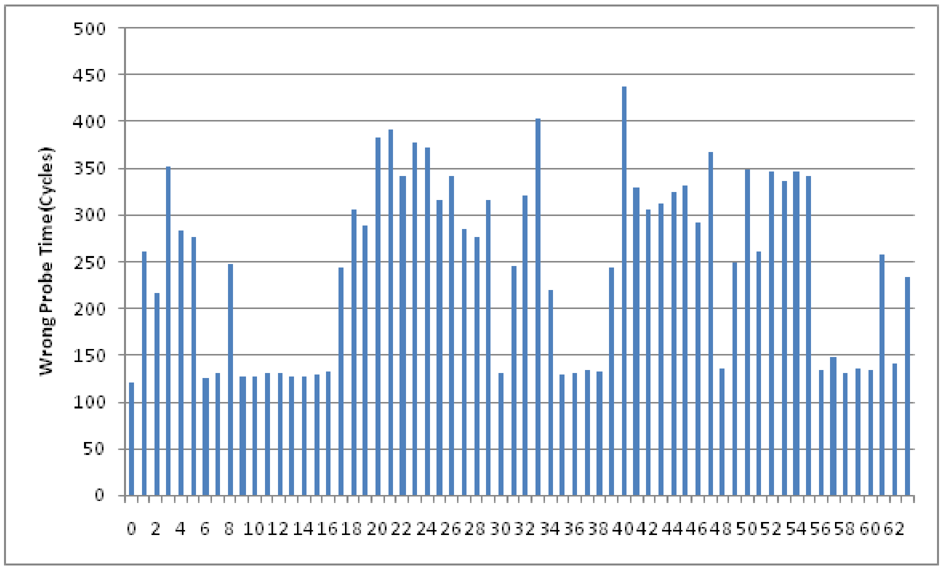


Figure 4.4: Wrong PROBE-ing Result for 32kB 8-way Set-Associative Cache

4.1.3 Unique ICache Footprints Left by Square, Mult and Reduce

As described in Algorithm 1 in Chapter 2.1.1's Zhang's Elgamal Attack, the modular exponentiation computed in libgcrypt v1.5.3 uses the square and multiply algorithm. They let S, R, M stand for calls to functions Square, ModReduce and Mult, respectively, inside SquareMult. As described previously, the sequence of function calls in one execution of SquareMult will leak information about the exponent exp, which is the secret key x here. For example, the sequence **(SRMR)(SR)** corresponds to $exp = 110_b = 6$. **SRMR** leaks information $e_2 = 1$, and **SR** gives $e_1 = 0$ (the most siginificant bit here is always 1; the sequence leaks information from the second most significant bit e_{n-1} to the least siginificant bit e_1). Because each of the S, R and M functions have their instructions located in different memory addresses; performing these 3 calls will leave different unique footprints in the instruction cache. The attacker can detect which function has been executed through the cache footprint he got from the PROBE-ing phase. Afterwards, the attacker can infer the secret key from the S,R,M sequence.

In my experiment, to get the clear unique ICache footprint left by each of these functions, a PRIME() is done just before calling to that function, and a PROBE() is done just after the function returns.

- Square:

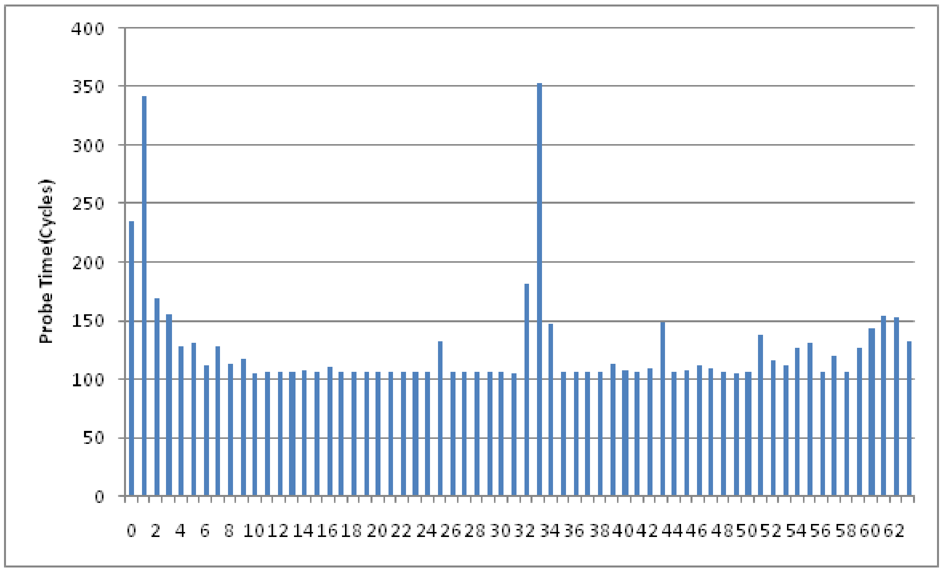


Figure 4.5: ICache Footprint Left by Square

Figure 4.5 is the probing time for each cache set after the Square function returns. We need to understand that set0,set1, set32, set33 are the sets we want to ignore (as described above), because they are not related to the Square function. We get Figure 4.6 by doing a subtraction of Figure 4.5

and the clean prime-probe figure, so that we can clearly get how much more time (cycles) is required to PROBE each set after the Square function.

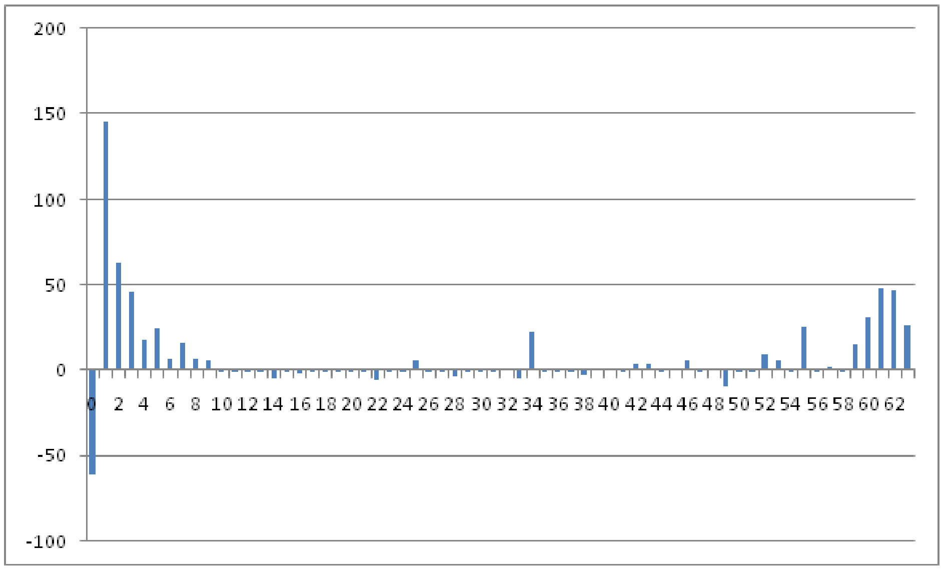


Figure 4.6: Square's Footprint after Subtracting the Clean Prime-Probe

In the final compiled dynamic library libgcrypt.so.11.8.2, the Square operation: _gcry_mpih_sqr_n() has virtual address ranging from 0x54e50 to 0x5521d, which is about 973 bytes. These instructions occupy set57 to set63, and set0 to set8. From Figure 4.6, we can see that probing these sets takes longer time. Moreover, _gcry_mpih_sqr_n() also calls other functions: _gcry_mpih_addmul_1 at address 0x57040 to 0x5707f (set1), _gcry_mpih_sub_n at address 0x56fb0 to 0x56fea (set62 to set63), _gcry_mpih_add_n at address 0x56f70 to 0x56faa (set61 to set62), _gcry_mpih_sqr_n_basecase at address 0x54d30 to 0x54e4f (set52 to set57). These information are also reflected in Figure 4.6.

- Mult:

Similarly, Figure 4.7 is the probing time for each cache set after Mult function returns. Figure 4.8 is done by doing a subtraction of Figure 4.7 and the clean prime-probe figure.

The Mult operation _gcry_mpih_mul_karatsuba_case() has virtual address ranging from 0x555a0 to 0x558ff, which is about 863 bytes. These instructions occupy set22 to set35. From Figure 4.8, we can see that probing these sets takes longer time. Moreover, _gcry_mpih_mul_karatsuba_case() also calls other functions: mul_n_basecase at address 0x54750 to 0x5486f (set29 to set33), _gcry_mpi_free_limb_space at address 0x55a30 to 0x55a69 (set40 to set41), _gcry_mpih_add_n at address 0x56f70 (set61), mul_n at 0x54870 (set33), _gcry_mpi_alloc_limb_space at 0x55930 (set36), _gcry_mpih_mul at 0x553f0 (set15) etc. I will not list how much virtual memory all these functions take, but basically, the probing pattern is reflected in Figure 4.8.

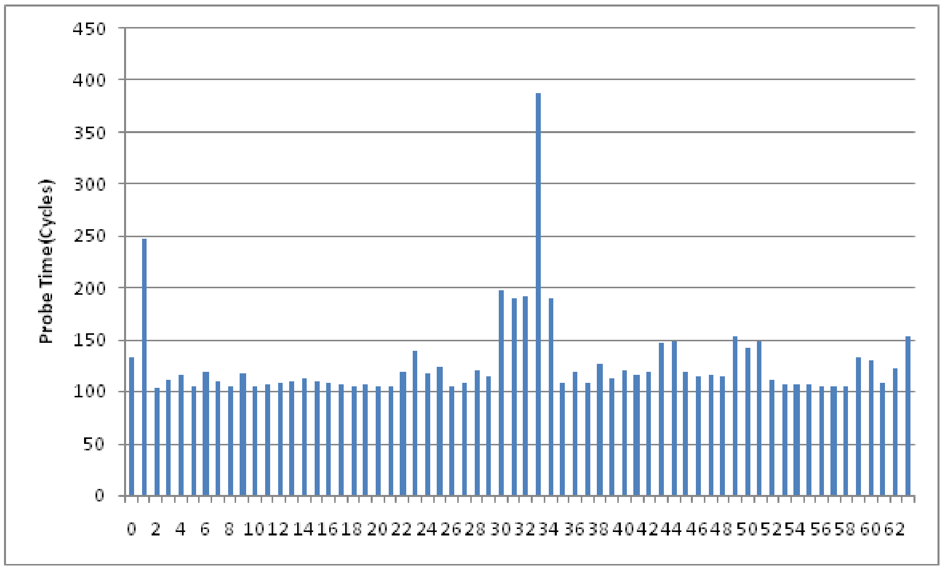


Figure 4.7: ICache Footprint Left by Mult

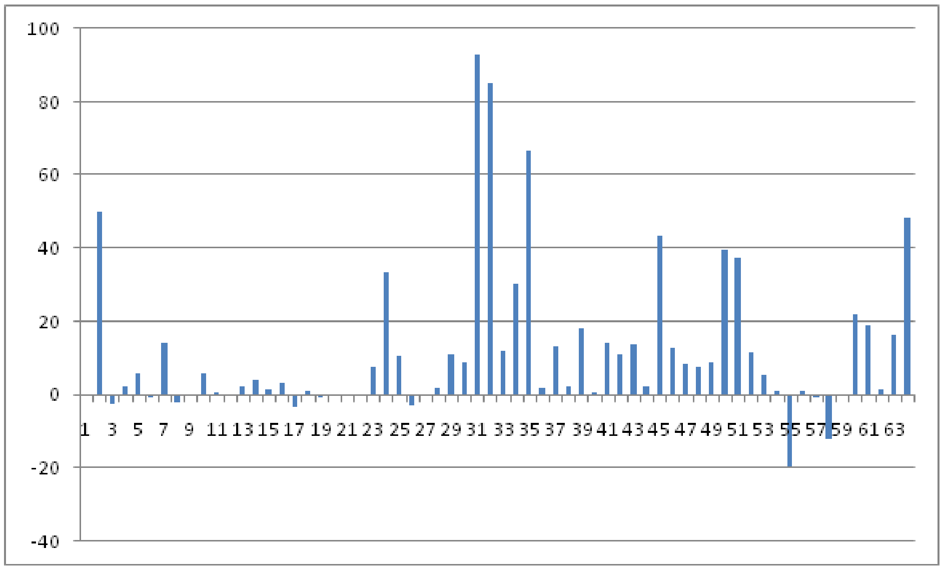


Figure 4.8: Mult's Footprint After Subtracting the Clean Prime-Probe

- Reduce:

Figure 4.9 is the probing time for each cache set after the Reduce function returns. Figure 4.10 is done by doing a subtraction of 4.9 and the clean prime-probe figure.

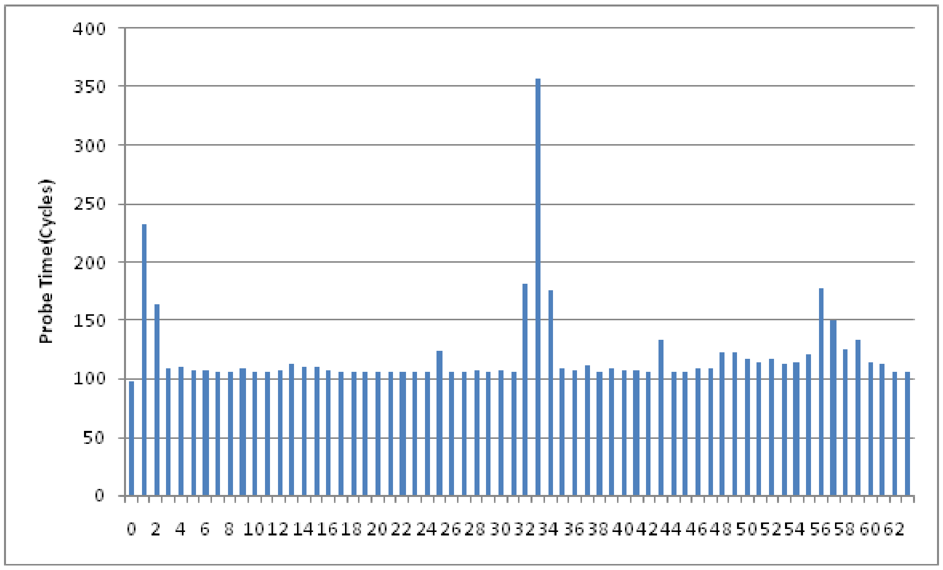


Figure 4.9: ICache Footprint Left by Reduce

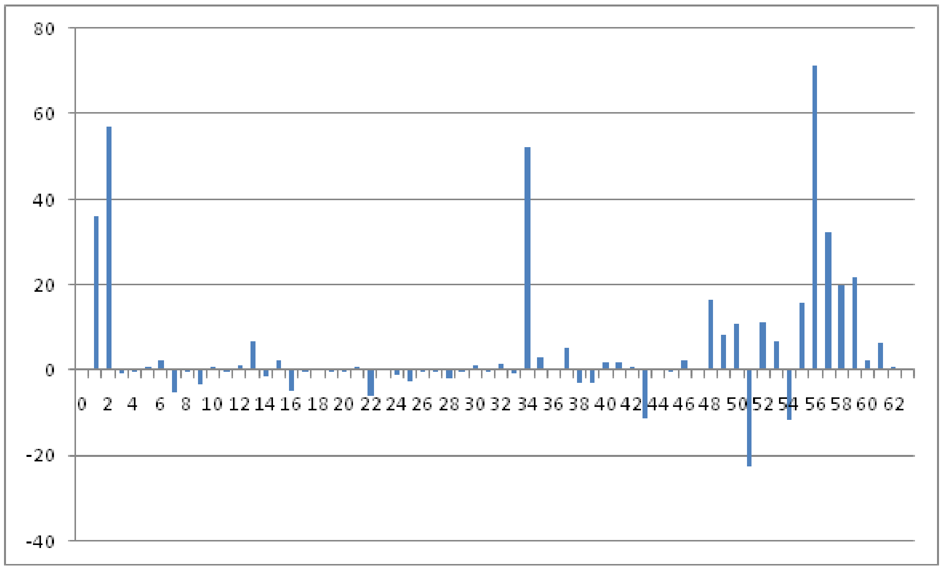


Figure 4.10: Reduce's Footprint after Subtracting the Clean Prime-Probe

The Reduce operation: _gcry_mpih_divrem() has virtual address ranging from 0x53c10 to 0x5440f, which is about 2047 bytes. These instructions occupy set48 to set63, and set0 to set8. From the figure above, we can see that probing these sets takes longer time. Also, not all the instructions are executed during each run of _gcry_mpih_divrem(). Moreover, _gcry_mpih_divrem() also calls other functions: _gcry_mpih_submul_1 at address 0x57080 (set2), _gcry_mpih_add_n at address 0x56f70 (set61), and _gcry_mpih_sub_n at 0x56fb0 (set62).

From Figure 4.6, Figure 4.8 and Figure 4.10, we can distinguish the ICache footprints left by Square, Mult and Reduce clearly, because different operations have different PROBE-ing time patterns for different cache sets.

4.1.4 Modular Exponentiation Experiments

For the actual attack, we need an attacker process and a victim process. However my experiments are done in an ideal setting. I assume the attacker has the ability to PROBE at the exact time that one of Square, Mult or Reduce is finished. The clear experimental results would give us an intuition about how the attack works.

Message (base) has 1000 bits, key (exponent) and modular both have 1024 bits. Message is chosen to be 10101010...1010b, where there is a 1 in every other bit. Modular is chosen to be 101010...1010b, where also there is a 1 in every other bit. It does not matter what values we choose for message and modular, because different message and different modular values do not influence the execution of Mult, Square and Reduce. Mult, Square and Reduce functions have fixed virtual addresses; No matter what message and modular we choose, the Mult, Square, and Reduce footprints left in the SA cache will be the same.

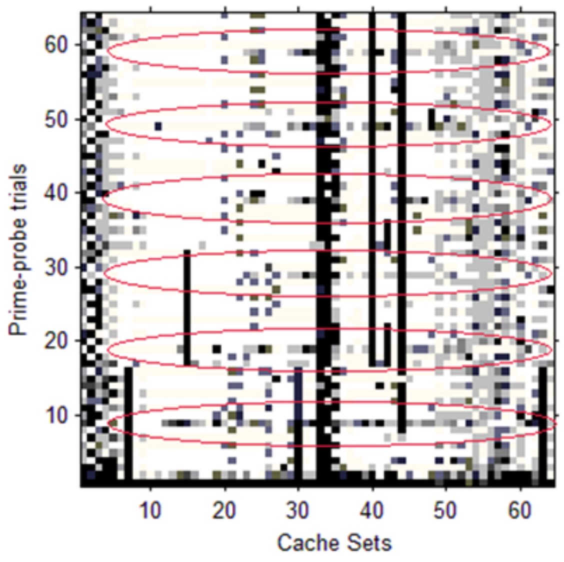


Figure 4.11: Gray Scale Matrix I

First, I set key to be 100010001000...1000b, where there is a 1 in every four bits. Prime-probe is done at the same time that the program is executing modular exponentiation. I get 2556 prime-probe results for the Square, Mult and Reduce operations. The first 64 prime-probe trials are extracted from the results and are printed as gray-scale matrix shown in Figure 4.11. The white-black scale is set from 100 to 200 (cycles) (white is 100 cycles and black is 200 cycles). Comparing with the unique footprints we got above, especially Figure 4.8, we can clearly find that red-circled trial 9,19,29,39,49, 59 ... are Mult operations. The whole execution path of the experiment is: SRSRSR(SRMR)SRSRSR(SRMR)..., which clearly shows that the 2nd bit to 9th bit are 00010001 (we should note that the first bit is always 1). From the whole 2556 prime-probe trials, the attacker can easily know each bit of the key.

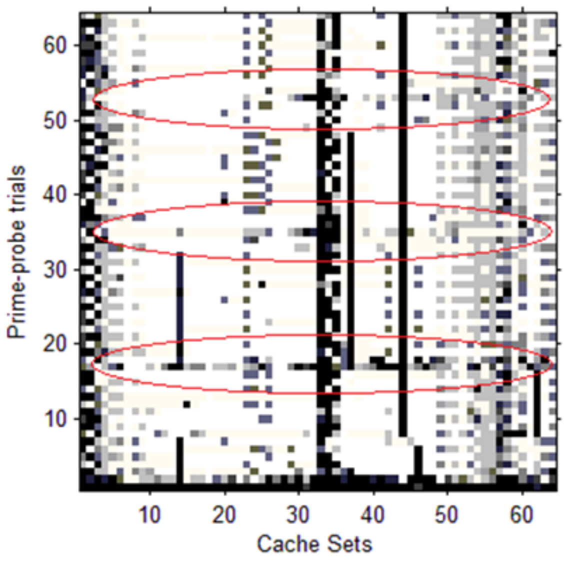


Figure 4.12: Gray Scale Matrix II

Similarly, I now set key to be 100000010000000...10000000b, where there is a 1 in every eight bits. I get 2300 prime-probe results for the Square, Mult and Reduce operations. Again, the first 64 prime-probe trials are extracted from the results and are printed as gray-scale matrix shown in Figure 4.12. We can clearly find that trial 17, 35, 53 ... are Mult operations. The whole execution path of the experiment is: SRSRSRSRSRSRSR(SRMR)SRSRSRSRSRSRSR(SRMR)..., which clearly shows that the 2nd bit to 17th bit are 0000000100000001.

Finally, I set the key bits randomly to be 1 or 0. In this case, the 1st bit, 6th bit, the 21st bit ... from the MSB side are randomly choosen to be bit 1. Again the first 64 prime-probe trials are extracted from the results and are printed as gray-scale matrix shown in Figure 4.13. The Mult operations clearly manifest themselves in the red-circles (trial 11 and trial 43), indicating the 6th bit and 21st bit's operations are (SRMR), which correspond to bit 1. It is also easy to distinguish between a Square operation and a Reduce operation by looking into the PROBE time of set19, 20 and 21.

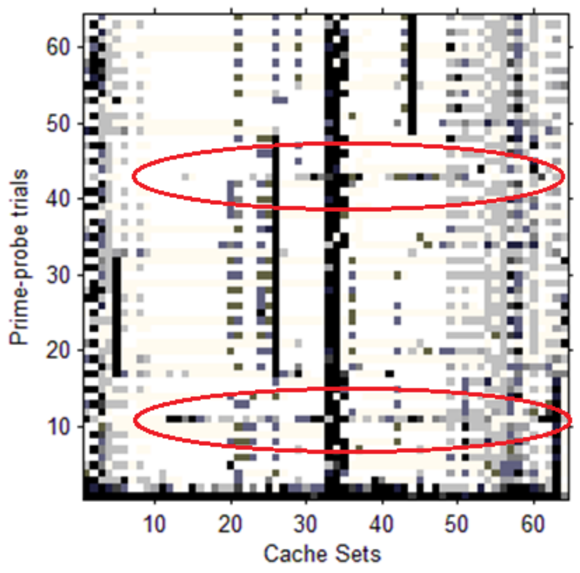


Figure 4.13: Gray Scale Matrix III

The above experiment showed that ICache footprints left behind by different operations are different, and thus can be used to infer the execution path of the program, from which we can get the cryptographic key. However, I just used a simplified method to get the clear footprint left by each operation. In a real scenario, attackers cannot modify the source code of the crypto-library, neither can they prime-probe at the exact time after one typical operation is finished. In Zhang's work [44], they use a support vector machine (SVM) to classify the instruction cache PROBE-ed footprints into the corresponding function calls (S, M or R) during the training phase. To make the attack actually work, they have to deal with issues like observation granularity, observation noise etc. Actually they use an artificial intelligence method called Hidden Markov Model to reduce noise and increase prime-probe accuracy when they cannot control the exact PROBE time. Furthermore, when their work elevates the side-channel attack from process level to virtual machine level, they will have to consider more issues, e.g. core migration, error correction, operation sequence reassembly etc.

The original Libgcrypt 1.5.0 does not provide secure memory execution, so attackers can detect each bit of the exponent from the executing sequence of these instructions through an ICache side channel. Libgcrypt 1.5.3 provides some secure execution: if the exponent is inside secure memory, then no matter what value e_i is, Mult will always be executed. However, libgcrypt 1.5.3 gives programmers the option to use secure memory or not (actually not using secure memory is the default option). So, programmers still tend to sacrifice security for execution efficiency.

4.2 Newcache in Defending against Instruction Cache Side-Channel Attack

Newcache is implemented in gem5 [20] by F.Liu [35]. The testing environment is shown in Table 4.1. The whole architecture simulates a single core of the latest Intel i7 processor. In the configuration, ICache is simulated as a 32kB Newcache with $k = 4$ (The details of Newcache are described in Chapter 2.3). I did two experiments and got the results. One is the clean prime-probe result, another is the probe patterns after executing the Square, Mult and Reduce operations in the modular exponentiation.

Table 4.1: Simulator Configurations

Single-core out-of-order X86 processor	L1-ICache (private)	L1-DCache (private)	L2 Cache (unified)	L3 Cache	Memory
	32 kB Newcache 4-cycle latency	32 kB 8-way 4-cycle latency	256 kB 8-way 10-cycle latency	2MB 16-way 35-cycle latency	2GB 100-cycle latency
Cache line size	64B				
Clock freq	3 GHz				
Evaluate Newcache as a L1 instruction cache, k=4 for Newcache configuration					

4.2.1 Clean PROBE-ing Result

A clean primeprobe means probing happens directly after priming. For this experiment, the 32kB Newcache has 32kB/64B=512 LNregs, each represents one cache-line. Because there is no notion of cache-set in Newcache, I slightly modify the PRIME method. I still allocate a contiguous 32kB memory for the attacker, and access the memory to evict out the entries in the ICache. But instead of PRIME-ing each of the sets for SA cache, I prime each of the 512 LNregs for ICache as Newcache. However, for this case, not like the traditional SA cache, we cannot prime all the cache-lines due to Newcache's random eviction mechanism. In the PRIME phase, imagine that the Newcache is gradually filled with more and more attacker's cache-lines, then it's also more and more likely that the next access to another attacker's cache-line may randomly evict a cache-line belonging to the attacker. Also, in the PROBE phase, access to some cache-line not already in the cache may still possibly evict a cache-line belonging to the attacker.

Algorithm 2 shows the PROBE algorithm. $virtual\_code\_address$ is the start of the 32kB memory. We do $num\_set = 512$ iterations. Each iteration calls $temp\_code\_address$, which just contains a ret instruction. At the end of each iteration, $temp\_code\_address$ is increased by the cache $block\_offset$. PRIME is just like PROBE without $rdtsc$ instructions. Doing this clean PROBE-ing experiment will also allow us to see how many of the 512 LNregs can be primed after one PRIME phase.

Algorithm 2 PROBE-ing for 32kB Newcache

```
procedure ProbeNewcache()
    $temp\_code\_address \leftarrow virtual\_code\_address$
    $num\_set \leftarrow 512$
    $block\_offset \leftarrow 64$
    for $i = 0 \rightarrow num\_set$ do
        $cycles\_begin \leftarrow$ RDTSC()
        Call $*temp\_code\_address$
        $cycles\_end \leftarrow$ RDTSC()
        $elapsed\_time[i] \leftarrow cycles\_end - cycles\_begin$
        $temp\_code\_address = temp\_code\_address + block\_offset$
    end for
end procedure
```

Figure 4.14 and Figure 4.15 are two clean prime-probe results. The LNreg numbers with lower probe time (about 30 cycles) are the cache-lines already in the cache, while those with higher probe-time are the cache-lines not successfully occupying one slot in the cache. From the two figures below, we may find that those evicted and those kept in the cache are totally randomized, which shows the successful implementation of random eviction policy for Newcache due to index miss.

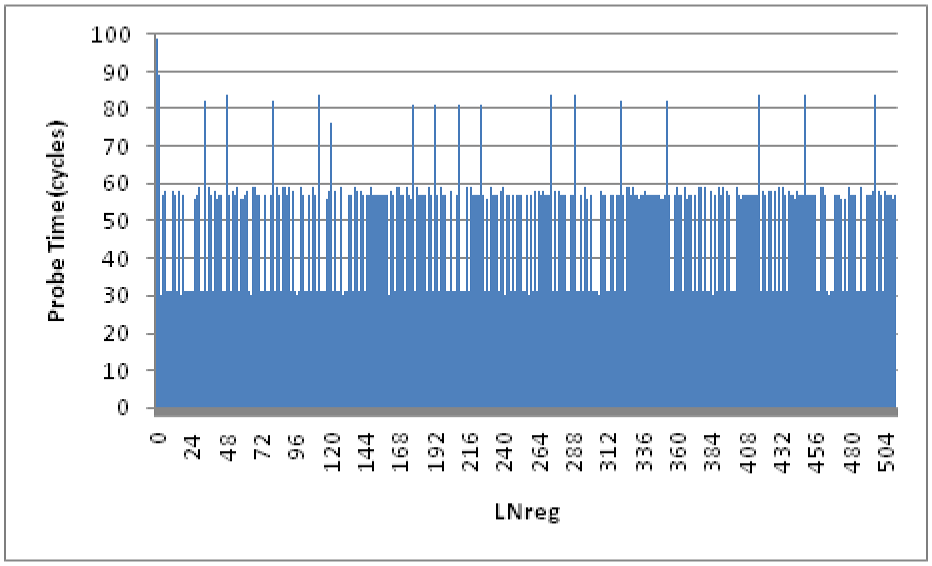


Figure 4.14: Clean PROBE-ing Result I for 32kB Newcache

Also, I collect about 800 clean prime-probe timings, average the probe-time for each of LNregs and get Figure 4.16

Figure 4.16 shows that the cache-eviction is truly random, the average probe time for all these LNregs are all about 40 cycles. However, there are small peaks in the average results for some LNregs (e.g. LNreg 17,33,49...). Actually, from Figure 4.14 and Figure 4.15, we may find that if LNreg 17,33,49... encountered a cache miss, their PROBE-time will be around 80 cycles, much

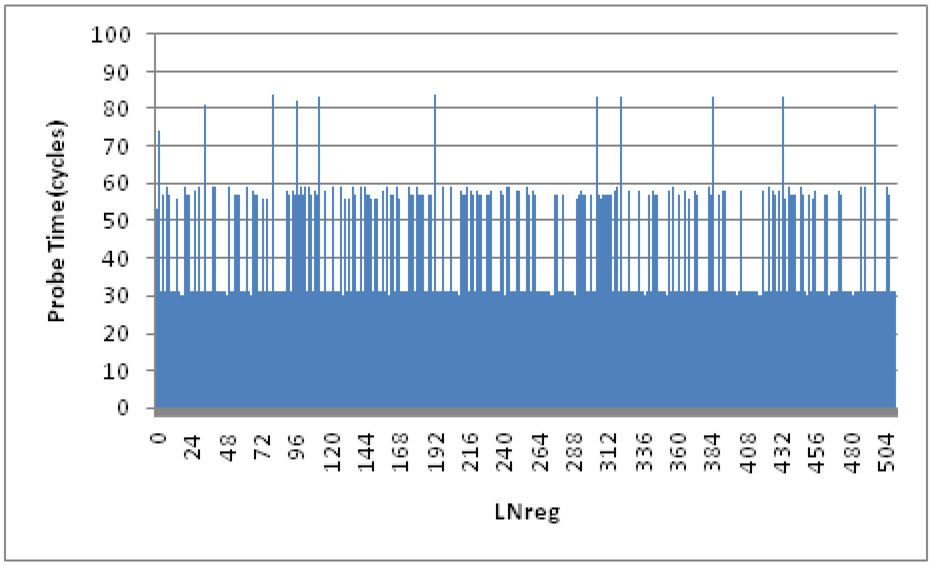


Figure 4.15: Clean PROBE-ing Result II for 32kB Newcache

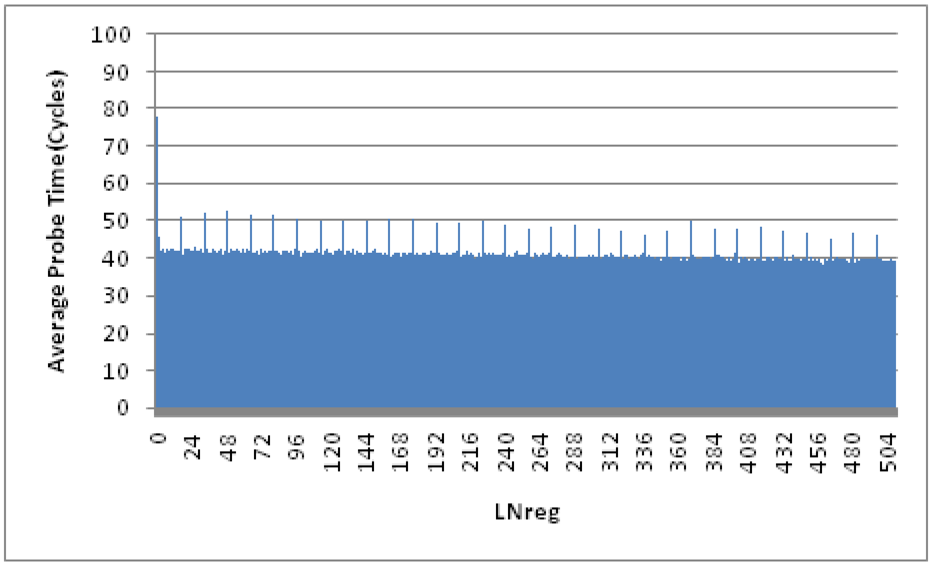


Figure 4.16: Averge PROBE Time for 800 Clean Prime-Probe

higher than other cache-miss lines' (about 60 cycles). These glitches may be caused by the memory architecture implementation in gem5, but they do not influence the functionality of Newcache.

Also, from these more than 800 clean prime-probe results, we get the number of successfully-primed cache lines (Figure 4.17). We can see that on average about 330 out of 512 cache-lines can successfully occupy Newcache after the PRIME-phase.

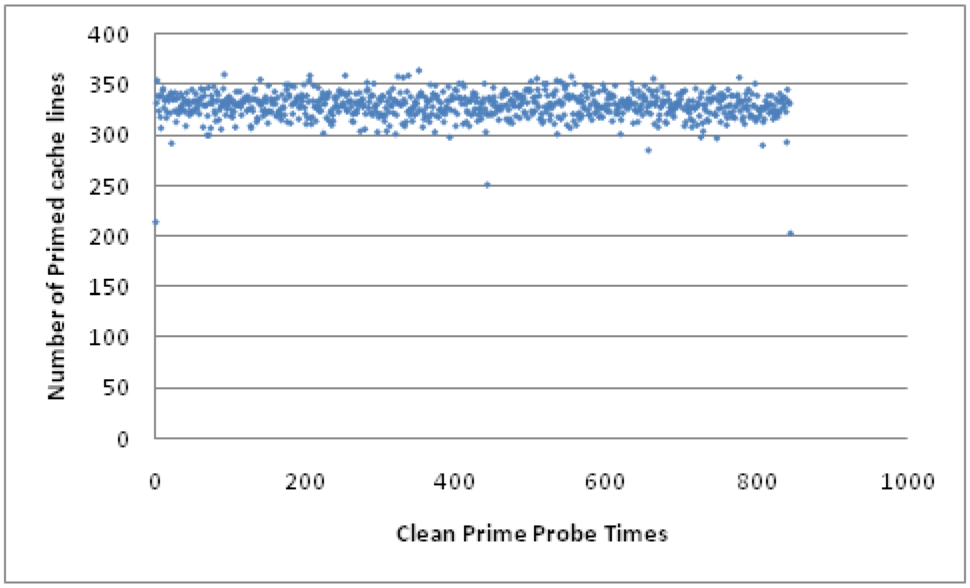


Figure 4.17: Number of PRIME-ed Cache-lines

4.2.2 ICache footprint left by Square, Mult and Reduce

- Square:

For this case a PRIME is done just before calling the Square function, and a PROBE is done just after the function returns. Figure 4.18 and Figure 4.19 are the two PROBE results for the Square function. We find that Newcache's random eviction policy helped prevent the ICache side-channel attack a lot; the two footprints left by Square function are not even alike, which successfully conceal the information that a Square operation has just executed.

Averaging about 800 probe-time for each of LNregs (Figure 4.20) shows similar result as that of clean prime-probe. The cache-eviction is truly random, each LNreg has equal probability to be evicted, and the average probe time for each LNreg is about 45 cycles. The attacker cannot get any useful information about what instructions are executed under the random eviction scheme of Newcache.

- Mult:

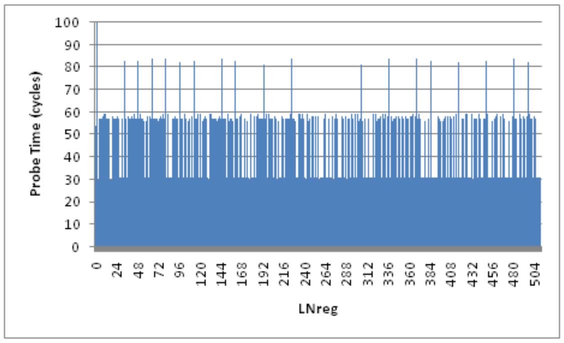


Figure 4.18: PROBE Result I for Square

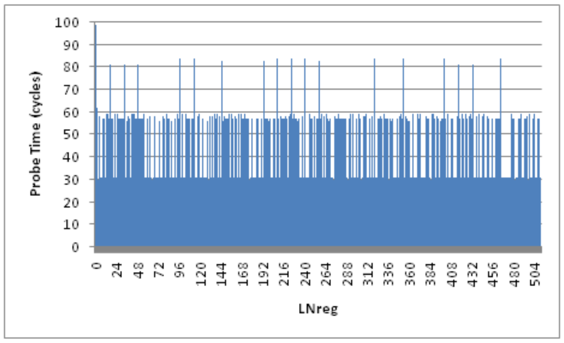


Figure 4.19: PROBE Result II for Square

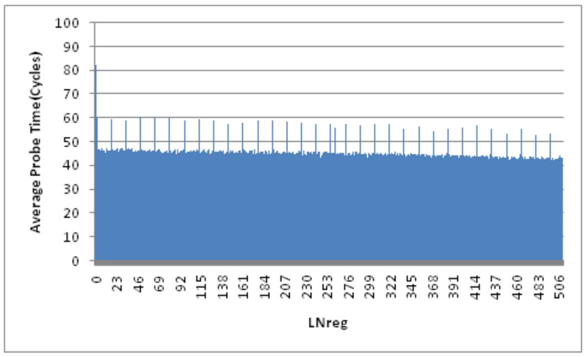


Figure 4.20: Average PROBE Result for Square

Figure 4.21 and Figure 4.22 are the two PROBE results for Mult function. Figure 4.20 shows the average PROBE result for Mult.

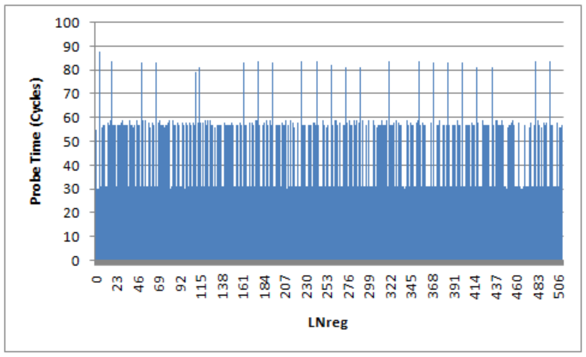


Figure 4.21: PROBE Result I for Mult

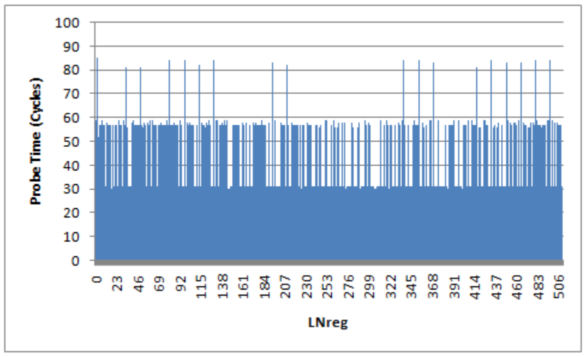


Figure 4.22: PROBE Result II for Mult

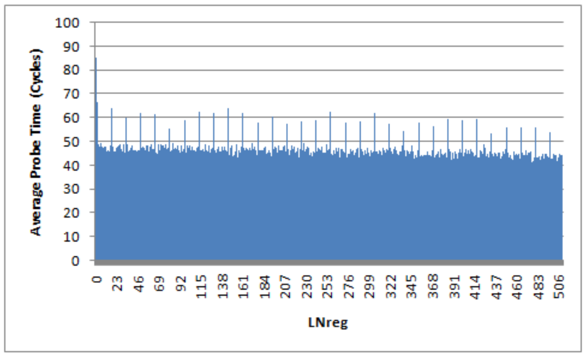


Figure 4.23: Average PROBE Result for Mult

- Reduce:

Figure 4.24 and Figure 4.25 are the two PROBE results for Reduce function. Figure 4.26 shows the average PROBE result for Reduce.

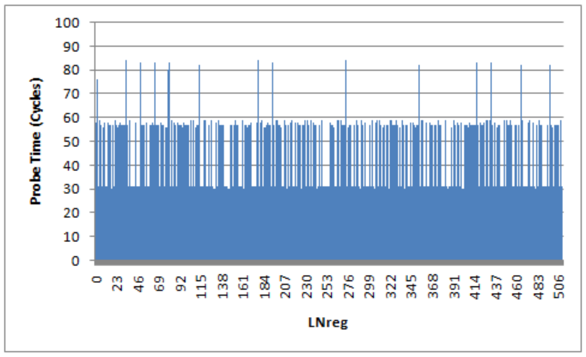


Figure 4.24: PROBE Result I for Reduce

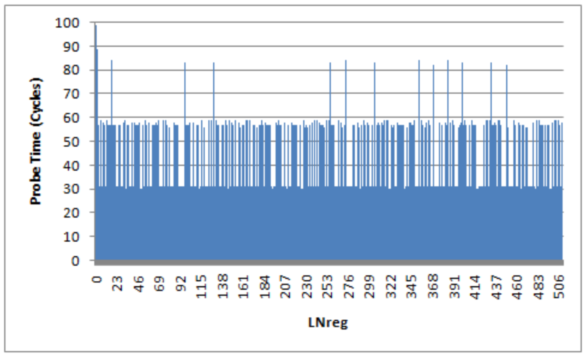


Figure 4.25: PROBE Result II for Reduce

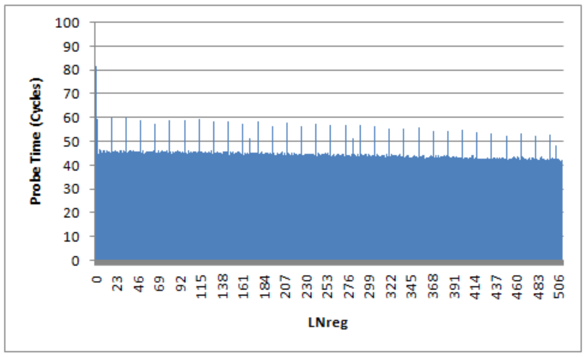


Figure 4.26: Average PROBE Result for Reduce

The PROBE results and the average PROBE results for Mult and Reduce are similar to those for Square operation, which shows the successful implementation of Newcache's random eviction policy. Also, from Figure 4.18, Figure 4.21 and Figure 4.24, an attacker cannot tell which footprint corresponds to which operation.

4.2.3 More Analysis

1. Carefully comparing Figure 4.20 and Figure 4.16, people could find that the average probing time for each LNreg of Square() function seems longer than that of clean prime-probe result. Is it possible that by exploiting this kind of timing difference, we can figure out which of the Square, Mult and Reduce function is executing? The answer is no. The reason that Figure 4.20 has longer average probing time is due to the length of the executed operation. As described in Section 4.1.3, Square function has about 973 bytes instructions, Mult has about 863 bytes, and Reduce has about 2047 bytes. The experiments are done in an ideal case (i.e. we do a prime directly before the function and a probe directly after the function). The more instructions the function has, the more likely the attackers' cache-lines will be evicted by the function, which is then reflected in the longer average probing time in the PROBE phase. However, in the real case, an attacker cannot prime and probe at any time he wants. Usually, the cpu is interrupted constantly to context switch to different processes, but the attacker does not know how far the victim process has gone. So the probing time of each LNreg cannot reveal any useful information about which of Square, Mult or Reduce has been executed.

2. There does exist a possible redesigned attack targeting Newcache. The redesigned attack is similar to the attack described by F.Liu [35]. Her attack is specifically designed for Newcache used as a data cache, while my attack is for Newcache used as an instruction cache.

If the attacker knows exactly where the Square(), Mult() and Reduce() instructions reside in the memory, he can manually create memory locations that will contend for the same locations in LDM Cache by Square, Mult and Reduce (Figure 4.27). Figure 4.27 shows (a) the cache contents directly after attacker's PRIME phase; we can see that none of the cache-lines of Square, Mult and Reduce is in the LDM Cache or physical cache. After some PRIME-PROBE interval, the attacker probes again. Let's assume at this point, (b) part of the Square() function has been executed, and we get the cache contents in Figure 4.28. Then the probing time for the memory location 1 will be longer. The attacker can easily get the information that the Square() function has executed.

However, people may think that the random eviction policy of Newcache cannot get the attacker a clean prime, which means the attacker is not able to prime every single cache-line for Newcache. From Figure 4.17, for more than 800 clean prime-probe results, we get the number of successfully-primed cache lines for each prime-probe. We can see that on average about 330 out of 512 cache-lines can

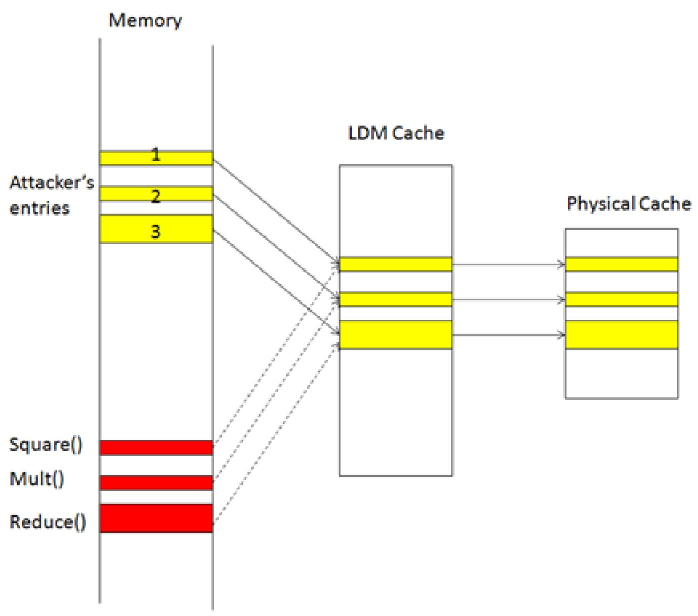


Figure 4.27: Illustration of the Attack (a)

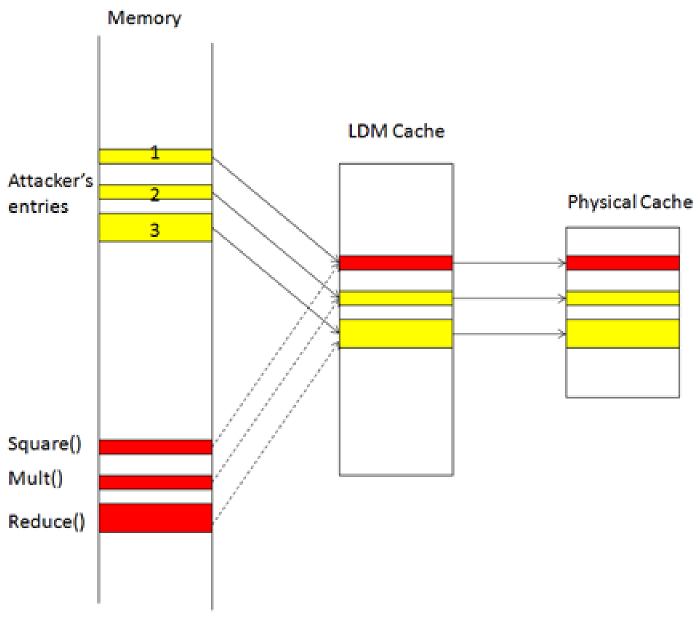


Figure 4.28: Illustration of the Attack (b)

successful occupy the Newcache after the PRIME phase. It's quite intuitive: if the attacker primes fewer cache-lines, these cache-lines tend to stay in the cache without being evicted by further primes.

From the previous description in Chapter 4.1.3, Square() has virtual address ranging from 0x54e50 to 0x5521d (973 bytes), Mult() has virtual address ranging from 0x555a0 to 0x558ff (863 bytes), Reduce has virtual address ranging from 0x53c10 to 0x5440f (2047 bytes); these three function totally occupy instructions no more than 6kB, which are about 96 cache-lines, far less than 330 cache-lines. Thus just priming 96 cache-lines is very likely to result in a successfully priming.

Moreover, how could the attacker know where the Square, Mult and Reduce functions reside in the physical memory? For a traditional 32kB 8-way set-associative cache, the last 12 bits decide which cache-set a memory-line should go into. Since the OS always assigns memory by 4KB memory pages, the last 12 bits of any virtual address is always fixed. So Square, Mult and Reduce instructions will always be fetched into the same sets in 32kb 8-way set-associative cache. However, for Newcache, we have n+k= 9+4 =13 bits, and the size of the LDM cache is 512kB. Everytime, the Square, Mult and Reduce instructions from the dynamic libraries may have different n+k index bits. The attacker thus needs to perform some independent trials [35] before he can locate the memory locations of Square, Mult and Reduce.

3. One more thing to notice, the attack described here and the attack described by F. Liu [35] both assume that the attacker's process has the same RMT_ID as the victim's process, otherwise the attacker's cache-line access during prime-probe will be considered as an index-miss in the first place, which will trigger random eviction instead of tag-miss eviction. So our attack model is mostly unrealistic: to have the same RMT_ID as the victim process, the attacker has to modify the victim's binary and inject his own codes. More detailed analysis is in Chapter 2.3.3.

4. The experiment I did is under an ideal model. I used this ideal model to get the clear footprint left by each operation for the SA cache. In a real scenario, attackers cannot modify the source code of the crypto-library, neither can they prime-probe at the exact time after one typical operation is finished. Under the ideal case (where the attacker is given enough power), our experiment shows that for SA cache, the attacker could get distinguished footprints left by different operations, while for Newcache, the attacker failed. If an attacker cannot even get any useful information when given enough power, he is doomed to fail in a non-ideal, real scenario.

In Zhang's work [44], they use a support vector machine (SVM) to classify the instruction cache PROBE-ed footprints into the corresponding function calls (S, M or R) during the training phase. To make the attack actually work for the SA cache, they have to deal with issues like observation granularity, observation noise etc. Actually they use an artificial intelligence method called Hidden

Markov Model to reduce noise and increase prime-probe accuracy when they cannot control the exact PROBE time.

5. In the previous analysis 4, I have described that under the ideal attack scenario in Newcache, no visual difference can be made from cache footprints of Square, Mult and Reduce. Afterwards, we refer to Zhang's work [44], and train a SVM model based on the labeled training samples of Square, Mult and Reduce. However, when we use the SVM model to do testings, we get a very high classification accuracy (71.2%, Table 4.3), which means that the ideal attack case (prime directly before the function and probe directly after the function) will actually add information of each operation's length into the final footprint. Such information is not reflected in the visual pattern, but can be detected by a trained SVM model.

We then simulate the real attack under gem5, so that the victim can only execute for a constant period of time before being preempted. This simulation is actually based on the real attack described by [29]. We train a SVM model again after this simulation, and we get a much lower classification accuracy (41.0%, Table 4.4), which indicates the Newcache's capability in preventing the successful information extractions from cache footprints.

Table 4.2 is the classification matrix for SA Cache used as ICache with LRU replacement algorithm, and the victim executes in a fixed interval between prime-probe phases (simulating the real attack [29]), while Table 4.3 and Table 4.4 are the classification matrices for Newcache with Nebit k=4 used as ICache. In table 4.3, we have data of the ideal prime-probe attack, so complete S, M and R operations are performed between prime-probe phases, while Table 4.4 is simulating the real attack [29] in the simulator, so the victim also executes in a fixed interval between prime-probe phases. Note that to simulate the real attack, we actually modify the simulator code directly, and let the simulator do attacker's prime-probe work. Modifying simulator directly allows us to quickly collect prime-probe data, thus we can easily collect 40,000 training samples and 12,000 testing samples, and use these testing samples to get the classification accuracy in Table 4.2 and Table 4.4. However, the ideal attack is done with two processes (one attacker process and one victim process) running under the simulator; collecting attack footprints is slow, and I only collected 3000 training samples and 1000 testing samples in 2 days. That's why we see much fewer testing samples in Table 4.3.

Table 4.2: Classification Matrix for 8-way SA Cache with LRU Replacement Algorithm used as ICache , with Fixed Interval for Victim to Execute, between Prime-Probe Phases

	Classification			Accuracy
	Square	Multiply	Reduce	
S	3985 (1.00)	0 (0.00)	15 (0.00)	99.7%
M	1 (0.00)	3991 (1.00)	8 (0.00)	
R	8 (0.00)	6 (0.00)	3986 (1.00)	

Table 4.3: Classification Matrix for Newcache with Nebit k=4 used as ICache, with Complete S, M or R Operation Performed, between Prime-Probe Phases

	Classification			Accuracy
	Square	Multiply	Reduce	
S	319 (0.96)	6 (0.02)	9 (0.03)	71.2%
M	1 (0.00)	189 (0.57)	143 (0.43)	
R	4 (0.01)	125 (0.38)	204 (0.61)	

Table 4.4: Classification Matrix for Newcache with Nebit k=4 used as ICache, with Fixed Interval for Victim to Execute, between Prime-Probe Phases

	Classification			Accuracy
	Square	Multiply	Reduce	
S	1143 (0.29)	940 (0.24)	1917 (0.47)	41.0%
M	1154 (0.29)	1098 (0.27)	1748 (0.44)	
R	696 (0.17)	620 (0.16)	2684 (0.67)	

Chapter 5

Closing Words

In this thesis, we did a thorough measurement of the performance (e.g. IPC, Cache Miss Rate etc.) of Newcache as data cache, L2 cache and instruction cache, and a detailed performance analysis and comparison with a conventional set-associative cache for carefully selected cloud server benchmarks under gem5. We find that for the L1 data cache and L1 instruction cache, Newcache has performance comparable with a conventional set-associative cache, with Newcache sometimes having even better performance. Newcache as L2 cache has higher local and global L2 miss rate compared with SA-cache as L2 cache, which is possibly due to Newcache randomly evicting lines out of the L2 cache that may have still been useful. However, the performance of program execution, which is directly reflected by the IPC, is hardly impacted. We also found that Newcache with $k = 6$ extra index bits often gave better performance for the server benchmarks studied in this work, whereas previous studies on the SPEC benchmarks [42] showed that $k = 4$ was sufficient.

We also did experiments and security analysis on Newcache as an instruction cache. The results showed that Newcache can thwart representative attacks targeting instruction cache side channels. Under an attack model which may not be realistic, the attacker can still use a redesigned *Prime-and-Probe* technique targeting Newcache to extract some secret key information.

Future work might include making a mathematical probability model about the Newcache random replacement policy, as well as realizing possible instruction side channel attacks targeting Newcache.

Appendix A

Data for Newcache Performance Measurement

A.1 Data for Newcache as L1 Data Cache

Table A.1: Newcache as L1-DCache: IPC for Different Cache Sizes

	SA-size16	SA-size32	SA-size64	new-size16	new-size32	new-size64
apache	0.1760	0.1837	0.1962	0.1722	0.1806	0.1924
mysql		0.2803	0.2901	0.2631	0.2734	
mail_t1	0.2447	0.2710	0.2815	0.2407	0.2655	0.2773
mail_t2	0.2485	0.2822	0.2977	0.2461	0.2758	0.2931
mail_t5	0.2535	0.2766	0.2952	0.2451	0.2691	0.2921
dbench_write	0.2364	0.2447	0.2550	0.2368	0.2470	0.2599
dbench_read	0.2308	0.2390	0.2479	0.2294	0.2414	0.2546
rtsp_s1	0.3935	0.4118	0.4241	0.3872	0.4100	0.4212
rtsp_s3	0.3843	0.4058	0.4215	0.3767	0.3999	0.4178
rtsp_s30	0.2986	0.3251	0.3434	0.2952	0.3193	0.3341
tomcat.t1	0.3820	0.3263	0.3395	0.3135	0.3351	0.3542
tomcat.t3	0.3135	0.3265	0.3343	0.3114	0.3308	0.3522
tomcat.t11	0.3065	0.3261	0.3360	0.3101	0.3309	0.3522
arith-mean	0.2890	0.2999	0.3125	0.2791	0.2984	0.3167
geo-mean	0.2813	0.2936	0.3062	0.2729	0.2918	0.3098

Table A.2: Newcache as L1-DCache: IPC for Different Associativity and Nebit

	SA-assoc2	SA-assoc4	SA-assoc8	new-nebit3	new-nebit4	new-nebit5	new-nebit6
apache	0.1816	0.1834	0.1837	0.1805	0.1806	0.1809	0.1808
mysql	0.2757	0.2786	0.2803	0.2731	0.2734	0.2734	0.2758
mail_t1	0.2568	0.2636	0.2710	0.2639	0.2655	0.2655	0.2656
mail_t2	0.2679	0.2748	0.2822	0.2730	0.2758	0.2760	0.2761
mail_t5	0.2676	0.2738	0.2766	0.2679	0.2691	0.2699	0.3928
dbench_write	0.2425	0.2441	0.2447	0.2483	0.2470	0.2484	0.2473
dbench_read	0.2375	0.2384	0.2390	0.2410	0.2414	0.2415	0.2418
rtsp_s1	0.4008	0.4085	0.4118	0.4095	0.4100	0.4101	0.4101
rtsp_s3	0.3965	0.4025	0.4058	0.3997	0.3999	0.4004	0.4007
rtsp_s30	0.3120	0.3176	0.3251	0.3155	0.3193	0.3150	0.3148
tomcat.t1	0.3271	0.3237	0.3263	0.3349	0.3351	0.3364	0.3363
tomcat.t3	0.3216	0.3206	0.3265	0.3406	0.3308	0.3330	0.3375
tomcat.t11	0.3198	0.3205	0.3261	0.3404	0.3309	0.3279	0.3275
arith-mean	0.2929	0.2962	0.2999	0.2991	0.2984	0.2983	0.3082
geo-mean	0.2867	0.2900	0.2936	0.2924	0.2918	0.2919	0.3008

Table A.3: Newcache as L1-DCache: DCache Miss Rate for Different Cache Sizes

	SA-size16	SA-size32	SA-size64	new-size16	new-size32	new-size64
apache	0.0979	0.0748	0.0507	0.1154	0.0863	0.0581
mysql		0.0584	0.0478	0.0954	0.0749	
mail_t1	0.0609	0.0222	0.0111	0.0721	0.0296	0.0143
mail_t2	0.0726	0.0303	0.0130	0.0837	0.0377	0.0171
mail_t5	0.0785	0.0467	0.0241	0.0947	0.0557	0.0272
dbench_write	0.1217	0.0987	0.0762	0.1262	0.0955	0.0681
dbench_read	0.1265	0.1044	0.0793	0.1326	0.0980	0.0690
rtsp_s1	0.0446	0.0260	0.0159	0.0523	0.0281	0.0177
rtsp_s3	0.0635	0.0387	0.0243	0.0745	0.0456	0.0280
rtsp_s30	0.1175	0.0848	0.0655	0.1262	0.0952	0.0734
tomcat.t1	0.0566	0.0637	0.0521	0.0844	0.0618	0.0448
tomcat.t3	0.0802	0.0642	0.0547	0.0850	0.0619	0.0433
tomcat.t11	0.0822	0.0644	0.0537	0.0856	0.0614	0.0433
arith-mean	0.0836	0.0598	0.0437	0.0945	0.0640	0.0420
geo-mean	0.0796	0.0538	0.0364	0.0916	0.0591	0.0365

Table A.4: Newcache as L1-DCache: DCache Miss Rate for Different Associativity and Nebit

	SA-assoc2	SA-assoc4	SA-assoc8	new-nebit3	new-nebit4	new-nebit5	new-nebit6
apache	0.0841	0.0773	0.0748	0.0870	0.0863	0.0859	0.0858
mysql	0.0692	0.0618	0.0584	0.0759	0.0749	0.0745	0.0674
mail_t1	0.0445	0.0339	0.0222	0.0323	0.0296	0.0294	0.0294
mail_t2	0.0494	0.0407	0.0303	0.0396	0.0377	0.0373	0.0373
mail_t5	0.0581	0.0500	0.0467	0.0575	0.0557	0.0551	0.0197
dbench_write	0.1050	0.0999	0.0987	0.0957	0.0955	0.0952	0.0948
dbench_read	0.1101	0.1057	0.1044	0.0991	0.0980	0.0979	0.0975
rtsp_s1	0.0366	0.0299	0.0260	0.0291	0.0281	0.0275	0.0276
rtsp_s3	0.0511	0.0437	0.0387	0.0458	0.0456	0.0449	0.0448
rtsp_s30	0.0982	0.0887	0.0848	0.0959	0.0952	0.0933	0.0940
tomcat.t1	0.0684	0.0660	0.0637	0.0634	0.0618	0.0610	0.0609
tomcat.t3	0.0715	0.0680	0.0642	0.0621	0.0619	0.0610	0.0592
tomcat.t11	0.0723	0.0681	0.0644	0.0616	0.0614	0.0648	0.0647
arith-mean	0.0706	0.0641	0.0598	0.0650	0.0640	0.0637	0.0602
geo-mean	0.0671	0.0597	0.0538	0.0604	0.0591	0.0588	0.0537

Table A.5: Newcache as L1-DCache: Global L2 Miss Rate for Different Cache Sizes

	SA-size16	SA-size32	SA-size64	new-size16	new-size32	new-size64
apache	0.0264	0.0264	0.0233	0.0261	0.0260	0.0230
mysql		0.0190	0.0162	0.0196	0.0188	
mail_t1	0.0048	0.0045	0.0035	0.0050	0.0046	0.0034
mail_t2	0.0050	0.0048	0.0037	0.0051	0.0048	0.0038
mail_t5	0.0078	0.0079	0.0062	0.0081	0.0074	0.0063
dbench_write	0.0250	0.0249	0.0245	0.0245	0.0244	0.0231
dbench_read	0.0264	0.0264	0.0254	0.0264	0.0252	0.0239
rtsp_s1	0.0043	0.0040	0.0036	0.0043	0.0040	0.0037
rtsp_s3	0.0059	0.0056	0.0050	0.0059	0.0056	0.0050
rtsp_s30	0.0180	0.0168	0.0157	0.0178	0.0167	0.0162
tomcat.t1	0.0139	0.0142	0.0138	0.0138	0.0138	0.0132
tomcat.t3	0.0142	0.0141	0.0141	0.0138	0.0141	0.0134
tomcat.t11	0.0145	0.0141	0.0142	0.0141	0.0140	0.0133
arith-mean	0.0139	0.0141	0.0130	0.0142	0.0138	0.0123
geo-mean	0.0113	0.0115	0.0103	0.0118	0.0113	0.0097

Table A.6: Newcache as L1-DCache: Global L2 Miss Rate for Different Associativity and Nebit

	SA-assoc2	SA-assoc4	SA-assoc8	new-nebit3	new-nebit4	new-nebit5	new-nebit6
apache	0.0262	0.0264	0.0264	0.0260	0.0260	0.0259	0.0260
mysql	0.0189	0.0189	0.0190	0.0188	0.0188	0.0188	0.0188
mail_t1	0.0045	0.0045	0.0045	0.0046	0.0046	0.0046	0.0046
mail_t2	0.0048	0.0048	0.0048	0.0048	0.0048	0.0048	0.0048
mail_t5	0.0077	0.0076	0.0079	0.0075	0.0074	0.0074	0.0052
dbench_write	0.0249	0.0249	0.0249	0.0241	0.0244	0.0242	0.0243
dbench_read	0.0264	0.0264	0.0264	0.0254	0.0252	0.0251	0.0253
rtsp_s1	0.0039	0.0039	0.0040	0.0040	0.0040	0.0040	0.0040
rtsp_s3	0.0056	0.0056	0.0056	0.0055	0.0056	0.0056	0.0056
rtsp_s30	0.0178	0.0176	0.0168	0.0170	0.0167	0.0172	0.0170
tomcat.t1	0.0140	0.0142	0.0142	0.0137	0.0138	0.0137	0.0137
tomcat.t3	0.0145	0.0145	0.0141	0.0131	0.0141	0.0140	0.0136
tomcat.t11	0.0142	0.0144	0.0141	0.0131	0.0140	0.0143	0.0142
arith-mean	0.0141	0.0141	0.0141	0.0137	0.0138	0.0138	0.0136
geo-mean	0.0115	0.0115	0.0115	0.0112	0.0113	0.0113	0.0110

A.2 Data for Newcache as L2 Cache

Table A.7: Newcache as L2 Cache: IPC for Different Cache Sizes

	l2.SA size128	l2.SA size256	l2.SA size512	l2.new size128	l2.new size256	l2.new size512
apache	0.1658	0.1837	0.2142	0.1633	0.1803	0.2072
mail_t1	0.2521	0.2710	0.2777	0.2474	0.2676	0.2758
mail_t2	0.2603	0.2822	0.2893	0.2505	0.2702	0.2843
mail_t5	0.2495	0.2766	0.2887	0.2468	0.2797	0.2787
dbench_write	0.2351	0.2447	0.2542	0.2332	0.2409	0.2488
dbench_read	0.2296	0.2390	0.2481	0.2288	0.2355	0.2438
rtsp_s1	0.3987	0.4118	0.4153	0.3957	0.4052	0.4106
rtsp_s3	0.3886	0.4058	0.4095	0.3819	0.3982	0.4042
rtsp_s30	0.3031	0.3251	0.3287	0.2954	0.3106	0.3276
tomcat.t1	0.3177	0.3263	0.3383	0.3102	0.3161	0.3277
tomcat.t3	0.3178	0.3265	0.3383	0.3103	0.3183	0.3345
tomcat.t11	0.3181	0.3261	0.3453	0.3105	0.3181	0.3340
arith-mean	0.2864	0.3016	0.3123	0.2812	0.2951	0.3064
geo-mean	0.2788	0.2947	0.3067	0.2738	0.2885	0.3007

Table A.8: Newcache as L2 Cache: IPC for Different Associativity and Nebit

	l2.SA assoc4	l2.SA. assoc8	l2.SA. assoc16	l2.new nebit3	l2.new nebit4	l2.new nebit5	l2.new nebit6
apache	0.1833	0.1837	0.1941	0.1803	0.1803	0.1805	
mail_t1	0.2684	0.2710	0.2725	0.2664	0.2676	0.2670	0.2682
mail_t2	0.2778	0.2822	0.2838	0.2699	0.2702	0.2712	0.2708
mail_t5	0.2733	0.2766	0.2787	0.2646	0.2797	0.2642	0.2653
dbench_write	0.2459	0.2447	0.2456	0.2429	0.2409	0.2430	0.2437
dbench_read	0.2381	0.2390	0.2394	0.2351	0.2355	0.2357	0.2362
rtsp_s1	0.4086	0.4118	0.4128	0.4042	0.4052	0.4054	0.4063
rtsp_s3	0.4030	0.4058	0.4064	0.3959	0.3982	0.3982	0.3991
rtsp_s30	0.3217	0.3251	0.3220	0.3084	0.3106	0.3128	0.3115
tomcat.t1	0.3244	0.3263	0.3331	0.3214	0.3161	0.3184	0.3191
tomcat.t3	0.3255	0.3265	0.3332	0.3208	0.3183	0.3186	0.3184
tomcat.t11	0.3267	0.3261	0.3266	0.3163	0.3181	0.3184	0.3190
arith-mean	0.2997	0.3016	0.3040	0.2938	0.2951	0.2945	0.3053
geo-mean	0.2930	0.2947	0.2976	0.2872	0.2885	0.2878	0.3007

Table A.9: Newcache as L2 Cache: DCache Miss Rate for Different Cache Sizes

	l2.SA size128	l2.SA size256	l2.SA size512	l2.new size128	l2.new size256	l2.new size512
apache	0.0773	0.0748	0.0692	0.0785	0.0758	0.0710
mail_t1	0.0233	0.0222	0.0214	0.0229	0.0219	0.0214
mail_t2	0.0316	0.0303	0.0294	0.0327	0.0312	0.0306
mail_t5	0.0496	0.0467	0.0446	0.0486	0.0438	0.0463
dbench_write	0.0993	0.0987	0.0975	0.0993	0.0991	0.0981
dbench_read	0.1053	0.1044	0.1015	0.1053	0.1045	0.1016
rtsp_s1	0.0268	0.0260	0.0255	0.0270	0.0266	0.0259
rtsp_s3	0.0403	0.0387	0.0377	0.0422	0.0397	0.0388
rtsp_s30	0.0879	0.0848	0.0816	0.0899	0.0866	0.0839
tomcat.t1	0.0631	0.0637	0.0625	0.0641	0.0648	0.0630
tomcat.t3	0.0631	0.0642	0.0625	0.0641	0.0645	0.0616
tomcat.t11	0.0631	0.0644	0.0602	0.0641	0.0645	0.0616
arith-mean	0.0609	0.0599	0.0578	0.0616	0.0602	0.0587
geo-mean	0.0546	0.0535	0.0516	0.0552	0.0537	0.0525

Table A.10: Newcache as L2 Cache: DCache Miss Rate for Different Associativity and Nebit

	l2.SA assoc4	l2.SA. assoc8	l2.SA. assoc16	l2.new nebit3	l2.new nebit4	l2.new nebit5	l2.new nebit6
apache	0.0751	0.0748	0.0709	0.0761	0.0758	0.0759	
mail_t1	0.0223	0.0222	0.0222	0.0223	0.0219	0.0224	0.0219
mail_t2	0.0304	0.0303	0.0301	0.0308	0.0312	0.0307	0.0314
mail_t5	0.0466	0.0467	0.0463	0.0476	0.0438	0.0476	0.0475
dbench_write	0.0969	0.0987	0.0988	0.0972	0.0991	0.0975	0.0975
dbench_read	0.1040	0.1044	0.1041	0.1046	0.1045	0.1049	0.1043
rtsp_s1	0.0260	0.0260	0.0260	0.0265	0.0266	0.0265	0.0265
rtsp_s3	0.0388	0.0387	0.0387	0.0397	0.0397	0.0397	0.0397
rtsp_s30	0.0842	0.0848	0.0854	0.0872	0.0866	0.0871	0.0860
tomcat.t1	0.0640	0.0637	0.0623	0.0630	0.0648	0.0645	0.0645
tomcat.t3	0.0638	0.0642	0.0623	0.0635	0.0645	0.0644	0.0645
tomcat.t11	0.0634	0.0644	0.0638	0.0648	0.0645	0.0645	0.0645
arith-mean	0.0596	0.0599	0.0592	0.0603	0.0602	0.0605	0.0589
geo-mean	0.0533	0.0535	0.0529	0.0539	0.0537	0.0541	0.0523

Table A.11: Newcache as L2 Cache: Local L2 Miss Rate for Different Cache Sizes

	l2.SA size128	l2.SA size256	l2.SA size512	l2.new size128	l2.new size256	l2.new size512
apache	0.6867	0.4995	0.2030	0.6985	0.5349	0.2884
mail_t1	0.5382	0.2913	0.1249	0.5401	0.3233	0.2059
mail_t2	0.4493	0.2247	0.0924	0.4632	0.2886	0.1688
mail_t5	0.4791	0.2427	0.1164	0.5034	0.3414	0.2044
dbench_write	0.5361	0.4018	0.3024	0.5034	0.4236	0.3485
dbench_read	0.5308	0.4018	0.2951	0.4995	0.4217	0.3411
rtsp_s1	0.4143	0.2817	0.1962	0.4601	0.3747	0.2938
rtsp_s3	0.4504	0.2668	0.1811	0.4822	0.3668	0.2854
rtsp_s30	0.5298	0.3517	0.2083	0.5949	0.4653	0.3452
tomcat.t1	0.4829	0.3041	0.2096	0.5411	0.4161	0.3131
tomcat.t3	0.4827	0.2943	0.2091	0.5413	0.4163	0.3138
tomcat.t11	0.4828	0.2939	0.2145	0.5411	0.4165	0.3142
arith-mean	0.5053	0.3212	0.1961	0.5307	0.3991	0.2852
geo-mean	0.5013	0.3132	0.1858	0.5274	0.3941	0.2785

Table A.12: Newcache as L2 Cache: Local L2 Miss Rate for Different Associativity and Nebit

	l2.SA assoc4	l2.SA. assoc8	l2.SA. assoc16	l2.new nebit3	l2.new nebit4	l2.new nebit5	l2.new nebit6
apache	0.5053	0.4995	0.4896	0.5441	0.5349	0.5349	
mail_t1	0.3250	0.2913	0.2517	0.3340	0.3233	0.3182	0.3159
mail_t2	0.2669	0.2247	0.1986	0.3039	0.2886	0.2875	0.2799
mail_t5	0.2775	0.2427	0.2239	0.3335	0.3414	0.3255	0.3187
dbench_write	0.4154	0.4018	0.3876	0.4275	0.4236	0.4141	0.4123
dbench_read	0.4163	0.4018	0.3870	0.4250	0.4217	0.4182	0.4161
rtsp_s1	0.2951	0.2817	0.2714	0.3754	0.3747	0.3713	0.3612
rtsp_s3	0.2899	0.2668	0.2598	0.3786	0.3668	0.3646	0.3544
rtsp_s30	0.3736	0.3517	0.3289	0.4699	0.4653	0.4576	0.4520
tomcat.t1	0.3212	0.3041	0.2971	0.4253	0.4161	0.4138	0.4100
tomcat.t3	0.3234	0.2943	0.2968	0.4262	0.4163	0.4126	0.4125
tomcat.t11	0.3240	0.2939	0.2923	0.4204	0.4165	0.4139	0.4148
arith-mean	0.3445	0.3212	0.3071	0.4053	0.3991	0.3944	0.3771
geo-mean	0.3385	0.3132	0.2981	0.4005	0.3941	0.3892	0.3733

Table A.13: Newcache as L2 Cache: Global L2 Miss Rate for Different Cache Sizes

	l2.SA size128	l2.SA size256	l2.SA size512	l2.new size128	l2.new size256	l2.new size512
apache	0.0372	0.0264	0.0100	0.0385	0.0286	0.0146
mail_t1	0.0088	0.0045	0.0019	0.0087	0.0050	0.0031
mail_t2	0.0100	0.0048	0.0019	0.0107	0.0063	0.0036
mail_t5	0.0166	0.0079	0.0036	0.0171	0.0104	0.0066
dbench_write	0.0335	0.0249	0.0186	0.0316	0.0264	0.0216
dbench_read	0.0352	0.0264	0.0192	0.0332	0.0278	0.0223
rtsp_s1	0.0060	0.0040	0.0028	0.0068	0.0054	0.0042
rtsp_s3	0.0098	0.0056	0.0038	0.0110	0.0078	0.0060
rtsp_s30	0.0270	0.0168	0.0098	0.0309	0.0233	0.0163
tomcat.t1	0.0220	0.0142	0.0097	0.0250	0.0197	0.0145
tomcat.t3	0.0220	0.0141	0.0097	0.0249	0.0196	0.0142
tomcat.t11	0.0220	0.0141	0.0095	0.0249	0.0196	0.0142
arith-mean	0.0208	0.0136	0.0084	0.0219	0.0167	0.0118
geo-mean	0.0179	0.0110	0.0063	0.0191	0.0139	0.0096

Table A.14: Newcache as L2 Cache: Global L2 Miss Rate for Different Associativity and Nebit

	l2.SA assoc4	l2.SA. assoc8	l2.SA. assoc16	l2.new nebit3	l2.new nebit4	l2.new nebit5	l2.new nebit6
apache	0.0267	0.0264	0.0244	0.0292	0.0286	0.0287	
mail_t1	0.0051	0.0045	0.0039	0.0053	0.0050	0.0050	0.0049
mail_t2	0.0057	0.0048	0.0042	0.0066	0.0063	0.0062	0.0062
mail_t5	0.0090	0.0079	0.0072	0.0111	0.0104	0.0108	0.0106
dbench_write	0.0257	0.0249	0.0241	0.0265	0.0264	0.0257	0.0256
dbench_read	0.0274	0.0264	0.0254	0.0280	0.0278	0.0276	0.0275
rtsp_s1	0.0042	0.0040	0.0039	0.0054	0.0054	0.0054	0.0052
rtsp_s3	0.0061	0.0056	0.0054	0.0081	0.0078	0.0078	0.0076
rtsp_s30	0.0178	0.0168	0.0162	0.0235	0.0233	0.0228	0.0224
tomcat.t1	0.0151	0.0142	0.0135	0.0195	0.0197	0.0195	0.0193
tomcat.t3	0.0151	0.0141	0.0135	0.0196	0.0196	0.0194	0.0194
tomcat.t11	0.0151	0.0141	0.0137	0.0199	0.0196	0.0195	0.0195
arith-mean	0.0144	0.0136	0.0130	0.0169	0.0167	0.0165	0.0153
geo-mean	0.0119	0.0110	0.0104	0.0142	0.0139	0.0138	0.0128

A.3 Data for Newcache as Both L1 Data Cache and L2 Cache

Table A.15: Newcache as Both L1-DCache and L2 Cache: IPC

	l1d.SA-l2.SA	l1d.new-l2.SA	l1d.SA-l2.new	l1d.new-l2.new
apache	0.1837	0.1806	0.1803	0.1768
mail_t1	0.2710	0.2655	0.2676	0.2597
mail_t2	0.2822	0.2758	0.2702	0.2645
mail_t5	0.2766	0.2691	0.2797	0.2573
dbench_write	0.2447	0.2470	0.2409	0.2448
dbench_read	0.2390	0.2414	0.2355	0.2380
rtsp_s1	0.4118	0.4100	0.4052	0.4024
rtsp_s3	0.4058	0.3999	0.3982	0.3917
rtsp_s30	0.3251	0.3193	0.3106	0.3022
tomcat.t1	0.3263	0.3351	0.3161	0.3273
tomcat.t3	0.3265	0.3308	0.3183	0.3276
tomcat.t11	0.3261	0.3309	0.3181	0.3277
arith-mean	0.3016	0.3004	0.2951	0.2933
geo-mean	0.2947	0.2934	0.2885	0.2864

Table A.16: Newcache as Both L1-DCache and L2 Cache: DCache Miss Rate

	l1d.SA-l2.SA	l1d.new-l2.SA	l1d.SA-l2.new	l1d.new-l2.new
apache	0.0748	0.0863	0.0758	0.0876
mail_t1	0.0222	0.0296	0.0219	0.0298
mail_t2	0.0303	0.0377	0.0312	0.0382
mail_t5	0.0467	0.0557	0.0438	0.0567
dbench_write	0.0987	0.0955	0.0991	0.0941
dbench_read	0.1044	0.0980	0.1045	0.1000
rtsp_s1	0.0260	0.0281	0.0266	0.0285
rtsp_s3	0.0387	0.0456	0.0397	0.0466
rtsp_s30	0.0848	0.0952	0.0866	0.0971
tomcat.t1	0.0637	0.0618	0.0648	0.0626
tomcat.t3	0.0642	0.0619	0.0645	0.0624
tomcat.t11	0.0644	0.0614	0.0645	0.0626
arith-mean	0.0599	0.0631	0.0602	0.0639
geo-mean	0.0535	0.0580	0.0537	0.0587

Table A.17: Newcache as Both L1-DCache and L2 Cache: Local L2 Miss Rate

	l1d.SA-l2.SA	l1d.new-l2.SA	l1d.SA-l2.new	l1d.new-l2.new
apache	0.4995	0.4268	0.5349	0.4749
mail_t1	0.2913	0.2204	0.3233	0.2767
mail_t2	0.2247	0.1803	0.2886	0.2571
mail_t5	0.2427	0.1886	0.3414	0.2816
dbench_write	0.4018	0.4083	0.4236	0.4400
dbench_read	0.4018	0.4046	0.4217	0.4475
rtsp_s1	0.2817	0.2564	0.3747	0.3715
rtsp_s3	0.2668	0.2199	0.3668	0.3236
rtsp_s30	0.3517	0.3088	0.4653	0.4210
tomcat.t1	0.3041	0.3215	0.4161	0.4168
tomcat.t3	0.2943	0.3265	0.4163	0.4172
tomcat.t11	0.2939	0.3279	0.4165	0.4173
arith-mean	0.3212	0.2992	0.3991	0.3788
geo-mean	0.3132	0.2873	0.3941	0.3714

Table A.18: Newcache as Both L1-DCache and L2 Cache: Global L2 Miss Rate

	l1d.SA-l2.SA	l1d.new-l2.SA	l1d.SA-l2.new	l1d.new-l2.new
apache	0.0264	0.0260	0.0286	0.0294
mail_t1	0.0045	0.0046	0.0050	0.0059
mail_t2	0.0048	0.0048	0.0063	0.0070
mail_t5	0.0079	0.0074	0.0104	0.0113
dbench_write	0.0249	0.0244	0.0264	0.0264
dbench_read	0.0264	0.0252	0.0278	0.0282
rtsp_s1	0.0040	0.0040	0.0054	0.0059
rtsp_s3	0.0056	0.0056	0.0078	0.0084
rtsp_s30	0.0168	0.0167	0.0233	0.0241
tomcat.t1	0.0142	0.0138	0.0197	0.0181
tomcat.t3	0.0141	0.0141	0.0196	0.0180
tomcat.t11	0.0141	0.0140	0.0196	0.0181
arith-mean	0.0136	0.0134	0.0167	0.0167
geo-mean	0.0110	0.0108	0.0139	0.0142

A.4 Data for Newcache as L1 Instruction Cache (Part A)

Table A.19: Newcache as L1-ICache: IPC

	all.SA base	l1i.SA base	l1i.new nebit3	l1i.new nebit4	l1i.new nebit5	l1i.new nebit6
apache	0.1837	0.1768	0.1780	0.1744	0.1742	0.1739
mail_t1	0.2710	0.2597	0.2595	0.2601	0.2605	0.2600
mail_t2	0.2822	0.2645	0.2652	0.2648	0.2653	0.2655
mail_t5	0.2766	0.2573	0.2581	0.2574	0.2575	0.2577
dbench_write	0.2447	0.2448	0.2414	0.2416	0.2416	0.2418
dbench_read	0.2390	0.2380	0.2360	0.2355	0.2367	0.2360
rtsp_s1	0.4118	0.4024	0.3990	0.4031	0.4041	0.4041
rtsp_s3	0.4058	0.3917	0.3871	0.3889	0.3893	0.3900
rtsp_s30	0.3251	0.3022	0.3066	0.3002	0.3094	0.3154
tomcat.t1	0.3263	0.3273	0.3259	0.3229	0.3245	0.3234
tomcat.t3	0.3265	0.3276	0.3231	0.3235	0.3251	0.3251
tomcat.t11	0.3261	0.3277	0.3248	0.3239	0.3238	0.3231
arith-mean	0.3016	0.2933	0.2921	0.2914	0.2927	0.2930
geo-mean	0.2947	0.2864	0.2854	0.2844	0.2856	0.2859

Table A.20: Newcache as L1-ICache: ICache Miss Rate

	all.SA base	l1i.SA base	l1i.new nebit3	l1i.new nebit4	l1i.new nebit5	l1i.new nebit6
apache	0.2447	0.2454	0.2721	0.2670	0.2666	0.2670
mail_t1	0.2790	0.2784	0.2820	0.2783	0.2761	0.2776
mail_t2	0.2792	0.2846	0.2873	0.2858	0.2829	0.2835
mail_t5	0.2636	0.2651	0.2668	0.2660	0.2628	0.2631
dbench_write	0.0984	0.1000	0.1082	0.1078	0.1081	0.1080
dbench_read	0.0922	0.0925	0.1023	0.1018	0.1017	0.1018
rtsp_s1	0.0722	0.0724	0.0756	0.0710	0.0686	0.0685
rtsp_s3	0.0720	0.0726	0.0804	0.0777	0.0753	0.0753
rtsp_s30	0.0699	0.0728	0.0766	0.0783	0.0743	0.0752
tomcat.t1	0.0352	0.0357	0.0409	0.0396	0.0392	0.0391
tomcat.t3	0.0320	0.0358	0.0409	0.0393	0.0392	0.0389
tomcat.t11	0.0320	0.0358	0.0408	0.0394	0.0393	0.0391
arith-mean	0.1309	0.1326	0.1395	0.1377	0.1362	0.1364
geo-mean	0.0958	0.0985	0.1063	0.1042	0.1028	0.1029

Table A.21: Newcache as L1-ICache: Local L2 Instruction Miss Rate

	all.SA base	l1i.SA base	l1i.new nebit3	l1i.new nebit4	l1i.new nebit5	l1i.new nebit6
apache	0.4743	0.5277	0.5000	0.4887	0.4894	0.4884
mail_t1	0.0535	0.0959	0.0955	0.0972	0.0970	0.0973
mail_t2	0.0487	0.1126	0.1124	0.1133	0.1146	0.1145
mail_t5	0.0696	0.1605	0.1594	0.1625	0.1643	0.1640
dbench_write	0.4690	0.5588	0.5197	0.5216	0.5210	0.5213
dbench_read	0.5077	0.5978	0.5466	0.5480	0.5483	0.5488
rtsp_s1	0.0603	0.1294	0.1279	0.1367	0.1401	0.1415
rtsp_s3	0.0614	0.1604	0.1509	0.1564	0.1618	0.1621
rtsp_s30	0.1087	0.2828	0.2794	0.2780	0.2845	0.2800
tomcat.t1	0.2834	0.4295	0.4101	0.4260	0.4305	0.4327
tomcat.t3	0.2964	0.4289	0.4094	0.4267	0.4304	0.4334
tomcat.t11	0.2978	0.4288	0.4095	0.4274	0.4307	0.4338
arith-mean	0.2276	0.3261	0.3101	0.3152	0.3177	0.3181
geo-mean	0.1559	0.2687	0.2588	0.2642	0.2670	0.2673

Table A.22: Newcache as L1-ICache: Global L2 Instruction Miss Rate

	all.SA base	l1i.SA base	l1i.new nebit3	l1i.new nebit4	l1i.new nebit5	l1i.new nebit6
apache	0.1056	0.1174	0.1190	0.1190	0.1189	0.1189
mail_t1	0.0144	0.0256	0.0258	0.0259	0.0257	0.0259
mail_t2	0.0131	0.0307	0.0309	0.0310	0.0310	0.0310
mail_t5	0.0176	0.0406	0.0405	0.0412	0.0411	0.0411
dbench_write	0.0411	0.0496	0.0502	0.0502	0.0502	0.0502
dbench_read	0.0415	0.0487	0.0496	0.0495	0.0495	0.0496
rtsp_s1	0.0041	0.0088	0.0092	0.0092	0.0091	0.0092
rtsp_s3	0.0042	0.0110	0.0115	0.0115	0.0115	0.0116
rtsp_s30	0.0073	0.0196	0.0203	0.0207	0.0201	0.0200
tomcat.t1	0.0093	0.0142	0.0155	0.0156	0.0156	0.0156
tomcat.t3	0.0088	0.0142	0.0155	0.0155	0.0156	0.0156
tomcat.t11	0.0088	0.0142	0.0155	0.0156	0.0157	0.0157
arith-mean	0.0230	0.0329	0.0336	0.0337	0.0337	0.0337
geo-mean	0.0140	0.0246	0.0255	0.0256	0.0256	0.0256

A.5 Data for Newcache as L1 Instruction Cache (Part B)

Table A.23: Newcache as L1-ICache: IPC

	all.SA base	l1i.new nebit3	l1i.new nebit4	l1i.new nebit5	l1i.new nebit6
apache	0.1837	0.1810	0.1808	0.1812	0.1810
mail_t1	0.2703	0.2708	0.2714	0.2713	0.2711
mail_t2	0.2798	0.2802	0.2806	0.2808	0.2807
mail_t5	0.2768	0.2770	0.2779	0.2791	0.2780
dbench_write	0.2447	0.2433	0.2435	0.2433	0.2434
dbench_read	0.2390	0.2378	0.2378	0.2374	0.2376
rtsp_s1	0.4071	0.4050	0.4099	0.4101	0.4097
rtsp_s3	0.3848	0.3816	0.3839	0.3841	0.3834
rtsp_s30	0.2857	0.2843	0.2864	0.2861	0.2862
tomcat.t1	0.3247	0.3088	0.3225	0.3269	0.3288
tomcat.t3	0.3267	0.3209	0.3334	0.3258	0.3254
tomcat.t11	0.3261	0.3228	0.3273	0.3264	0.3439
arith-mean	0.2958	0.2928	0.2963	0.2960	0.2974
geo-mean	0.2896	0.2868	0.2898	0.2896	0.2908

Table A.24: Newcache as L1-ICache: ICache Miss Rate

	all.SA base	l1i.new nebit3	l1i.new nebit4	l1i.new nebit5	l1i.new nebit6
apache	0.2447	0.2663	0.2668	0.2666	0.2668
mail_t1	0.2798	0.2803	0.2782	0.2766	0.2782
mail_t2	0.2812	0.2833	0.2826	0.2803	0.2803
mail_t5	0.2660	0.2654	0.2641	0.2619	0.2621
dbench_write	0.0984	0.1072	0.1071	0.1070	0.1071
dbench_read	0.0922	0.1017	0.1013	0.1013	0.1013
rtsp_s1	0.0810	0.0812	0.0762	0.0745	0.0748
rtsp_s3	0.0837	0.0883	0.0851	0.0843	0.0846
rtsp_s30	0.0883	0.0894	0.0875	0.0879	0.0877
tomcat.t1	0.0331	0.0411	0.0356	0.0363	0.0346
tomcat.t3	0.0327	0.0377	0.0373	0.0360	0.0350
tomcat.t11	0.0320	0.0358	0.0351	0.0338	0.0367
arith-mean	0.1344	0.1398	0.1381	0.1372	0.1374
geo-mean	0.0997	0.1067	0.1039	0.1030	0.1032

Table A.25: Newcache as L1-ICache: Local L2 Instruction Miss Rate

	all.SA base	l1i.new nebit3	l1i.new nebit4	l1i.new nebit5	l1i.new nebit6
apache	0.4743	0.4347	0.4345	0.4350	0.4348
mail_t1	0.0535	0.0538	0.0538	0.0543	0.0553
mail_t2	0.0487	0.0493	0.0496	0.0495	0.0495
mail_t5	0.0643	0.0674	0.0680	0.0663	0.0675
dbench_write	0.4690	0.4281	0.4286	0.4294	0.4289
dbench_read	0.5077	0.4591	0.4609	0.4616	0.4612
rtsp_s1	0.0444	0.0430	0.0462	0.0470	0.0469
rtsp_s3	0.0529	0.0486	0.0502	0.0504	0.0511
rtsp_s30	0.0919	0.0945	0.0958	0.0949	0.0952
tomcat.t1	0.2999	0.3030	0.2762	0.2688	0.2813
tomcat.t3	0.2972	0.2672	0.3126	0.2656	0.2812
tomcat.t11	0.2978	0.2681	0.2772	0.2822	0.3171
arith-mean	0.2251	0.2097	0.2128	0.2088	0.2142
geo-mean	0.1477	0.1417	0.1445	0.1424	0.1456

Table A.26: Newcache as L1-ICache: Global L2 Instruction Miss Rate

	all.SA base	l1i.new nebit3	l1i.new nebit4	l1i.new nebit5	l1i.new nebit6
apache	0.1056	0.1059	0.1061	0.1061	0.1062
mail_t1	0.0143	0.0145	0.0144	0.0144	0.0148
mail_t2	0.0132	0.0134	0.0135	0.0134	0.0133
mail_t5	0.0165	0.0172	0.0173	0.0167	0.0170
dbench_write	0.0411	0.0412	0.0412	0.0413	0.0412
dbench_read	0.0415	0.0417	0.0418	0.0418	0.0417
rtsp_s1	0.0034	0.0033	0.0034	0.0034	0.0034
rtsp_s3	0.0042	0.0041	0.0041	0.0041	0.0041
rtsp_s30	0.0078	0.0081	0.0080	0.0080	0.0080
tomcat.t1	0.0092	0.0115	0.0091	0.0091	0.0091
tomcat.t3	0.0090	0.0094	0.0108	0.0089	0.0091
tomcat.t11	0.0088	0.0089	0.0091	0.0089	0.0108
arith-mean	0.0229	0.0233	0.0232	0.0230	0.0232
geo-mean	0.0138	0.0142	0.0141	0.0138	0.0141

A.6 Data for Newcache as L1 Instruction Cache (Part C)

Table A.27: Newcache as L1-ICache: IPC

	all.SA base	l1i.new .nebit3 l1d.new .nebit4	l1i.new .nebit4 l1d.new .nebit4	l1i.new .nebit5 l1d.new .nebit4	l1i.new .nebit6 l1d.new .nebit4	l1i.new .nebit3 l1d.new .nebit6	l1i.new .nebit4 l1d.new .nebit6	l1i.new .nebit5 l1d.new .nebit6	l1i.new .nebit6 l1d.new .nebit6
apache	0.1837	0.1777	0.1775	0.1775	0.1774	0.1775	0.1778	0.1778	0.1776
mail_t1	0.2703	0.2656	0.2667	0.2667	0.2663	0.2666	0.2670	0.2673	0.2667
mail_t2	0.2798	0.2738	0.2741	0.2750	0.2749	0.2742	0.2745	0.2752	0.2753
mail_t5	0.2768	0.2696	0.2703	0.2708	0.2707	0.2715	0.2726	0.2716	0.2721
dbench_write	0.2447	0.2453	0.2455	0.2454	0.2451	0.2462	0.2462	0.2479	0.2461
dbench_read	0.2390	0.2402	0.2402	0.2397	0.2401	0.2403	0.2421	0.2408	0.2398
rtsp_s1	0.4071	0.4024	0.4074	0.4074	0.4075	0.4026	0.4073	0.4072	0.4070
rtsp_s3	0.3848	0.3779	0.3792	0.3793	0.3793	0.3768	0.3792	0.3797	0.3788
rtsp_s30	0.2857	0.2760	0.2763	0.2782	0.2749	0.2758	0.2786	0.2744	0.2789
tomcat.t1	0.3247	0.3342	0.3299	0.3343	0.3327	0.3257	0.3343	0.3353	0.3372
tomcat.t3	0.3267	0.3283	0.3317	0.3350	0.3419	0.3291	0.3299	0.3268	0.3298
tomcat.t11	0.3261	0.3315	0.3294	0.3316	0.3303	0.3256	0.3345	0.3369	0.3310
arith-mean	0.2958	0.2935	0.2940	0.2951	0.2951	0.2927	0.2953	0.2951	0.2950
geo-mean	0.2896	0.2871	0.2875	0.2884	0.2883	0.2864	0.2888	0.2885	0.2884

Table A.28: Newcache as L1-ICache: ICache Miss Rate

	all.SA base	l1i.new .nebit3 l1d.new .nebit4	l1i.new .nebit4 l1d.new .nebit4	l1i.new .nebit5 l1d.new .nebit4	l1i.new .nebit6 l1d.new .nebit4	l1i.new .nebit3 l1d.new .nebit6	l1i.new .nebit4 l1d.new .nebit6	l1i.new .nebit5 l1d.new .nebit6	l1i.new .nebit6 l1d.new .nebit6
apache	0.2447	0.2661	0.2665	0.2666	0.2673	0.2665	0.2662	0.2661	0.2667
mail_t1	0.2798	0.2810	0.2779	0.2763	0.2767	0.2800	0.2787	0.2761	0.2777
mail_t2	0.2812	0.2873	0.2866	0.2825	0.2833	0.2871	0.2868	0.2831	0.2829
mail_t5	0.2660	0.2659	0.2648	0.2621	0.2618	0.2655	0.2647	0.2617	0.2622
dbench_write	0.0984	0.1078	0.1075	0.1074	0.1076	0.1077	0.1077	0.1081	0.1074
dbench_read	0.0922	0.1014	0.1022	0.1020	0.1022	0.1025	0.1014	0.1021	0.1013
rtsp_s1	0.0810	0.0815	0.0762	0.0748	0.0746	0.0814	0.0763	0.0747	0.0748
rtsp_s3	0.0837	0.0902	0.0854	0.0845	0.0866	0.0887	0.0854	0.0865	0.0848
rtsp_s30	0.0883	0.0940	0.0900	0.0897	0.0929	0.0912	0.0883	0.0935	0.0891
tomcat.t1	0.0331	0.0396	0.0369	0.0361	0.0360	0.0398	0.0363	0.0358	0.0373
tomcat.t3	0.0327	0.0382	0.0366	0.0357	0.0335	0.0385	0.0370	0.0373	0.0365
tomcat.t11	0.0320	0.0375	0.0362	0.0357	0.0355	0.0398	0.0355	0.0349	0.0357
arith-mean	0.1344	0.1409	0.1389	0.1378	0.1382	0.1407	0.1387	0.1383	0.1380
geo-mean	0.0997	0.1077	0.1049	0.1038	0.1037	0.1080	0.1044	0.1045	0.1042

Table A.29: Newcache as L1-ICache: Local L2 Instruction Miss Rate

	all.SA base	l1i.new .nebit3 l1d.new .nebit4	l1i.new .nebit4 l1d.new .nebit4	l1i.new .nebit5 l1d.new .nebit4	l1i.new .nebit6 l1d.new .nebit4	l1i.new .nebit3 l1d.new .nebit6	l1i.new .nebit4 l1d.new .nebit6	l1i.new .nebit5 l1d.new .nebit6	l1i.new .nebit6 l1d.new .nebit6
apache	0.4743	0.4359	0.4352	0.4350	0.4355	0.4361	0.4347	0.4350	0.4345
mail_t1	0.0532	0.0574	0.0562	0.0567	0.0567	0.0562	0.0564	0.0569	0.0568
mail_t2	0.0487	0.0514	0.0504	0.0513	0.0510	0.0508	0.0505	0.0514	0.0513
mail_t5	0.0643	0.0677	0.0662	0.0668	0.0668	0.0657	0.0650	0.0685	0.0657
dbench_write	0.4690	0.4250	0.4258	0.4267	0.4255	0.4255	0.4261	0.4265	0.4267
dbench_read	0.5077	0.4574	0.4569	0.4572	0.4590	0.4554	0.4605	0.4553	0.4603
rtsp_s1	0.0444	0.0437	0.0467	0.0476	0.0475	0.0436	0.0466	0.0472	0.0474
rtsp_s3	0.0529	0.0487	0.0502	0.0504	0.0506	0.0479	0.0499	0.0504	0.0508
rtsp_s30	0.0919	0.0896	0.0978	0.0965	0.0911	0.0972	0.0997	0.0933	0.0992
tomcat.t1	0.2999	0.2951	0.2707	0.2716	0.2615	0.2548	0.2699	0.2716	0.3137
tomcat.t3	0.2972	0.2607	0.2578	0.2718	0.2661	0.2607	0.2707	0.2728	0.2748
tomcat.t11	0.2978	0.2619	0.2740	0.2757	0.2727	0.2453	0.2727	0.2808	0.2781
arith-mean	0.2251	0.2079	0.2073	0.2089	0.2070	0.2033	0.2086	0.2092	0.2133
geo-mean	0.1477	0.1416	0.1424	0.1437	0.1422	0.1391	0.1430	0.1438	0.1460

Table A.30: Newcache as L1-ICache: Global L2 Instruction Miss Rate

	all.SA base	l1i.new .nebit3 l1d.new .nebit4	l1i.new .nebit4 l1d.new .nebit4	l1i.new .nebit5 l1d.new .nebit4	l1i.new .nebit6 l1d.new .nebit4	l1i.new .nebit3 l1d.new .nebit6	l1i.new .nebit4 l1d.new .nebit6	l1i.new .nebit5 l1d.new .nebit6	l1i.new .nebit6 l1d.new .nebit6
apache	0.1056	0.1060	0.1060	0.1060	0.1064	0.1062	0.1058	0.1058	0.1059
mail_t1	0.0143	0.0155	0.0150	0.0151	0.0151	0.0151	0.0151	0.0151	0.0152
mail_t2	0.0132	0.0142	0.0139	0.0139	0.0139	0.0140	0.0139	0.0140	0.0140
mail_t5	0.0165	0.0172	0.0168	0.0168	0.0168	0.0167	0.0165	0.0172	0.0165
dbench_write	0.0411	0.0411	0.0411	0.0411	0.0411	0.0411	0.0412	0.0414	0.0411
dbench_read	0.0415	0.0414	0.0417	0.0417	0.0419	0.0417	0.0417	0.0415	0.0417
rtsp_s1	0.0034	0.0034	0.0034	0.0034	0.0034	0.0034	0.0034	0.0034	0.0034
rtsp_s3	0.0042	0.0042	0.0041	0.0041	0.0042	0.0041	0.0041	0.0042	0.0041
rtsp_s30	0.0078	0.0081	0.0084	0.0083	0.0081	0.0085	0.0084	0.0084	0.0085
tomcat.t1	0.0092	0.0108	0.0093	0.0091	0.0088	0.0095	0.0091	0.0090	0.0108
tomcat.t3	0.0090	0.0093	0.0088	0.0090	0.0083	0.0094	0.0093	0.0095	0.0093
tomcat.t11	0.0088	0.0091	0.0092	0.0091	0.0090	0.0091	0.0090	0.0091	0.0092
arith-mean	0.0229	0.0234	0.0231	0.0231	0.0231	0.0232	0.0231	0.0232	0.0233
geo-mean	0.0138	0.0143	0.0140	0.0140	0.0138	0.0141	0.0140	0.0141	0.0142

Appendix B

Performance Measuring Scripts and Data Collecting Scripts

B.1 Sample Performance Measuring Script for Dbench_Write

Listing B.1 is a sample bash script for running the benchmark dbench_write. We set the –serverbench option to be “smbd” and the –clientbench option to be “smbd.0’, which corresponds to the smbd-server.rcS and smbd.0-client.rcS scripts in Listing B.2 and Listing B.3. In Listing B.1, we set L1-DCache to be Newcache, L1-ICache and L2 cache as conventional SA caches. We also set the stats output directory and all the other configuration details (cache associativity, size, etc.) in the script. The server and client binaries are installed in a gem5 disk image ain.img, which can be found in `haow@palms.ee.princeton.edu:~/archive/` (it is too big to be uploaded to github). All the other testing scripts for different benchmarks can be found in `https://github.com/eepalms/Newcache-performance-testing/tree/master/scripts-dual`. Actually people can just follow the sample script to create their own testing scripts, by either writing another auto scripts generator, or by doing simple keywords replacement in the sample script.

Listing B.1: Sample Bash Script for dbench_write

```
cd /home/haow/newcache_v2/
build/X86/gem5.opt --outdir=../testing/outdir/smbd.0/smbd.0-new-base
configs/example/fs.py --kernel=vmlinux --disk-image=ain.img
--mem-size="2048MB" --dual --serverbench="smbd" --clientbench="smbd.0"
--checkpoint-dir=../testing/checkpoint/smbd.0 -r 1 --clock="3GHz"
--mem_lat="66.7ns" --cpu-type=detailed --caches --l1d_size="32kB"
 --cacheline_size=64 --l1d_assoc=512 --l1d_nebit=4 --l1d_alg="NEWCACHE"
```

```
--l1d_index_pos="" --l1i_size="32kB" --l1i_assoc=4 --l2cache
--l2_size="256kB"  --l2_assoc=8 --l3cache --l3_size="2MB" --l3_assoc=16
 --maxinsts=1000000000 &
```

B.2 Python Script to Generate Performance Measuring Scripts

Listing B.2 is the Python script for generating performance measuring scripts under different configurations for different benchmarks. The benchmarks[] array defines the benchmarks, the disk images, the server/client configuration scripts to use and the maximum execution instruction numbers. The configs[] array defines the configurations you want to use to measure the performance for each of these benchmarks. The whole version of the script can be found under the haow@palms.ee.princeton.edu:~/archive/testing/ directory.

Listing B.2: Python Script to Generate Performance Measuring Scripts

```
import os,sys,stat


gem5_base = 'cd /home/haow/newcache_v2/'
gem5_exe = 'build/X86/gem5.opt'
outdir_base = '/home/haow/testing/outdir/'
checkpoint_base = '/home/haow/testing/checkpoint/'

scripts_base = './'

# benchmark parameters
# bm[0]: benchmark suite name / client script name / outdir name
# bm[1]: disk image
# bm[2]: server script name
benchmarks = [  ['apache', 'big.img', 'apache', 'NA'] ,
                ['mysql', 'mysqltest.img', 'mysql', 'NA'],
                ['mail_t1', 'ain.img', 'mail', '1000000000'],
                ['mail_t2', 'ain.img', 'mail', '1000000000'],
                ['mail_t5', 'ain.img', 'mail', '1000000000'],
                ['smbd.0', 'ain.img', 'smbd', '1000000000'],
                ['smbd.1', 'ain.img', 'smbd', '1000000000'],
                ['ffserver.x1', 'ain.img', 'ffserver', 'NA'],
                ['ffserver.x3', 'ain.img', 'ffserver', '2000000000'],
                ['ffserver.x30', 'ain.img', 'ffserver', '3000000000'],
                ['tomcat.3', 'ain.img', 'tomcat', '2000000000'],
                ['tomcat.4', 'ain.img', 'tomcat', '2000000000'],
                ['tomcat.5', 'ain.img', 'tomcat', '3000000000'],
             ]

# gem5 configurations
#configuration[0]:                                  [1]line    [2]data-cache
[6]instruction-cache           [10]l2
configs = [  ['-l1i.new.base-l1d.SA.base-l2.SA.base',
             '64', 'LRU', '32kB', '8', '', 'NEWCACHE', '32kB', '512',
             '4', 'LRU', '256kB', '8', ''],
             ['-l1i.new.nebit3-l1d.SA.base-l2.SA.base',
              '64', 'LRU', '32kB', '8', '', 'NEWCACHE', '32kB',
               '512', '3', 'LRU', '256kB', '8', ''],
             ['-l1i.new.nebit5-l1d.SA.base-l2.SA.base',
             '64', 'LRU', '32kB', '8', '', 'NEWCACHE', '32kB',
             '512', '5', 'LRU', '256kB', '8', ''],
             ['-l1i.new.nebit6-l1d.SA.base-l2.SA.base',
             '64', 'LRU', '32kB', '8', '', 'NEWCACHE', '32kB',
             '512', '6', 'LRU', '256kB', '8', ''],
             ......
          ]



for bm in benchmarks:

    dir_name = scripts_base + bm[0]
    if not os.path.exists(dir_name):
      os.mkdir(dir_name)

    for configuration in configs:
      file_name = dir_name + '/' + bm[0] + configuration[0]
```

```
#if not os.path.exists(file_name):
fd = open(file_name, 'w')

fd.write(gem5_base)
fd.write('\n')
fd.write(gem5_exe)

#--outdir
temp_string = ' --outdir=' + outdir_base + bm[0] + '/' + bm[0]
 + configuration[0]
fd.write(temp_string)

#configs and --kernel
fd.write(' configs/example/fs.py --kernel=vmlinux')

#--disk-image
temp_string = ' --disk-image=' + bm[1]
fd.write(temp_string)

#--mem-size
fd.write(' --mem-size="2048MB"')

#--dual --serverbench --clientbench
temp_string = ' --dual --serverbench="' + bm[2]
+ '" --clientbench="' + bm[0] + '"'
fd.write(temp_string)

#--checkpoint-dir
temp_string = ' --checkpoint-dir='
+ checkpoint_base + bm[0] + ' -r 1'
fd.write(temp_string)

#
fd.write(' --clock="3GHz"
 --mem_lat="66.7ns" --cpu-type=detailed --caches')
```

```
#--cacheline_size
temp_string = ' --cacheline_size=' + configuration[1]
fd.write(temp_string)

#l1d
temp_string = ' --l1d_alg="'+ configuration[2]
+ '" --l1d_size="' + configuration[3]
+ '" --l1d_assoc=' + configuration[4]
fd.write(temp_string)

if configuration[2] == 'NEWCACHE':
  temp_string = ' --l1d_nebit=' + configuration[5]
  fd.write(temp_string)

fd.write(' --l1d_index_pos=""')

#l1i
temp_string = ' --l1i_alg="'+ configuration[6]
+ '" --l1i_size="' + configuration[7]
 + '" --l1i_assoc=' + configuration[8]
fd.write(temp_string)

if configuration[6] == 'NEWCACHE':
  temp_string = ' --l1i_nebit=' + configuration[9]
  fd.write(temp_string)

#l2
fd.write(' --l2cache')
temp_string = ' --l2_alg="'+ configuration[10]
+ '" --l2_size="' + configuration[11]
+ '" --l2_assoc=' + configuration[12]
fd.write(temp_string)
if configuration[10] == 'NEWCACHE':
  temp_string = ' --l2_nebit=' + configuration[13]
  fd.write(temp_string)
```

```
    #l3
    fd.write(' --l3cache --l3_size="2MB" --l3_assoc=16')

    #max-inst
    if bm[3] != 'NA':
      temp_string = ' --maxinsts=' + bm[3]
      fd.write(temp_string)

    fd.write(' &\n')
    fd.close()

    st = os.stat(file_name)
    os.chmod(file_name, st.st_mode | stat.S_IEXEC)
```

B.3 Sample Server-Side and Client-Side Configuration Scripts for Dbench_Write on Gem5

Listing B.3 and Listing B.4 are the file server's server-side and the client-side configuration scripts for the gem5 x86 dual system. In smbd-server.rcS, we set up the network, give server-side an IP address 10.0.0.1, and start up the smbd file server. In smbd.0-client.rcS, we set up the network, and after we confirm that the server-side is ready, we use dbench to drive the server (10.0.0.1) with some workload.

Listing B.3: smbd-server.rcS

```
#!/bin/sh
#
# /etc/init.d/rcS
#

echo -n "setting up network..."
mount proc /proc -t proc
/sbin/ifconfig eth0 10.0.0.1 netmask 255.255.255.0 txqueuelen 1000 up
/sbin/ifconfig lo 127.0.0.1 netmask 255.255.255.0 up

echo "1" > /proc/sys/net/ipv4/tcp_tw_recycle
echo "1" > /proc/sys/net/ipv4/tcp_tw_reuse
echo "1" > /proc/sys/net/ipv4/tcp_window_scaling
echo "0" > /proc/sys/net/ipv4/tcp_timestamps
echo "0" > /proc/sys/net/ipv4/tcp_sack
echo "15" > /proc/sys/net/ipv4/tcp_fin_timeout
echo "16384" > /proc/sys/net/ipv4/tcp_max_syn_backlog
echo "262144" > /proc/sys/net/ipv4/ip_conntrack_max
echo "1024 65535" > /proc/sys/net/ipv4/ip_local_port_range
echo "10000000 10000000 10000000" > /proc/sys/net/ipv4/tcp_rmem
echo "10000000 10000000 10000000" > /proc/sys/net/ipv4/tcp_wmem
echo "10000000 10000000 10000000" > /proc/sys/net/ipv4/tcp_mem
echo "524287" > /proc/sys/net/core/rmem_max
echo "524287" > /proc/sys/net/core/wmem_max
echo "524287" > /proc/sys/net/core/optmem_max
echo "300000" > /proc/sys/net/core/netdev_max_backlog
echo "131072" > /proc/sys/fs/file-max
echo "done."

echo -n "starting smbd..."
cd /home/haow/install/samba-4.0.9/sbin
./smbd start
sleep 10

echo "done."

echo "server ready" | nc 10.0.0.2 8000

echo -n "starting bash shell..."
/bin/bash
```

Listing B.4: smbd.0-client.rcS

```
#!/bin/sh
#
# /etc/init.d/rcS
#

echo -n "setting up network..."
```

```
mount proc /proc -t proc
/sbin/ifconfig eth0 10.0.0.2 netmask 255.255.255.0 txqueuelen 1000 up
/sbin/ifconfig lo 127.0.0.1 netmask 255.255.255.0 up

echo "1" > /proc/sys/net/ipv4/tcp_tw_recycle
echo "1" > /proc/sys/net/ipv4/tcp_tw_reuse
echo "1" > /proc/sys/net/ipv4/tcp_window_scaling
echo "0" > /proc/sys/net/ipv4/tcp_timestamps
echo "0" > /proc/sys/net/ipv4/tcp_sack
echo "15" > /proc/sys/net/ipv4/tcp_fin_timeout
echo "16384" > /proc/sys/net/ipv4/tcp_max_syn_backlog
echo "262144" > /proc/sys/net/ipv4/ip_conntrack_max
echo "1024 65535" > /proc/sys/net/ipv4/ip_local_port_range
echo "10000000 10000000 10000000" > /proc/sys/net/ipv4/tcp_rmem
echo "10000000 10000000 10000000" > /proc/sys/net/ipv4/tcp_wmem
echo "10000000 10000000 10000000" > /proc/sys/net/ipv4/tcp_mem
echo "524287" > /proc/sys/net/core/rmem_max
echo "524287" > /proc/sys/net/core/wmem_max
echo "524287" > /proc/sys/net/core/optmem_max
echo "300000" > /proc/sys/net/core/netdev_max_backlog
echo "131072" > /proc/sys/fs/file-max
echo "done."

echo "waiting for server..."
nc -l 8000
#sleep 1
echo -n "running dbench to test smbd file server..."
cd /home/haow/install/dbench/bin
/sbin/m5 checkpoint
/sbin/m5 dumpstats
/sbin/m5 resetstats
./dbench -B smb --smb-share=//10.0.0.1/share \
--smb-user=% --loadfile=smb-writefiles.txt --run-once --skip-cleanup 3
echo "done."

echo -n "halting machine"
```

```
m5 exit

echo -n "starting bash shell..."
/bin/bash
```

B.4 Sample Performance Stats Collecting Script

Listing B.5 is the Python script for collecting performance stats of different benchmarks. After running all the benchmark performance testing under gem5, we put all these results in a base_dir (/home/haow/testing/outdir/). In the Listing B.4, we basically need to adjust several parameters defined in the global variables area. *keywords* indicate the system stats that you want to collect (DCache miss rate, IPC, etc). *prog_names* are the benchmark names you want to collect stats from, and *configs* are the different configurations you want to collect stats from for each benchmark.

Listing B.5: Data Collecting Python Script

```
import sys
import re
# keyword is a list of keywords in the stats file
def process_raw_stats(file_name, keywords):
        fin = open(file_name, 'r')
        lines = fin.readlines()
        results = []
        for i in range(len(keywords)):
                results.append('')

        for line in lines:
                linefield = re.split('\s+', line)
                count = keywords.count(linefield[0])
                if count == 1:
                        index = keywords.index(linefield[0])
                        results[index] = linefield[1]
        fin.close()
        return results

############## global variables ###########
result_all = []
```

```
fout = open('/home/haow/stats_mse2.dat', 'w')
keywords = ['testsys.switch_cpus.ipc_total',
'testsys.cpu.dcache.overall_miss_rate::total',
'testsys.cpu.dcache.overall_mshr_misses::total',
'testsys.switch_cpus.commit.committedInsts',
'testsys.l2.overall_misses::switch_cpus.data',
'testsys.cpu.dcache.overall_accesses::total']

prog_names = ['apache-','mysql-','mail_t1-',
'mail_t2-','mail_t5-','smbd.0-','smbd.1-',
'ffserver.x1-', 'ffserver.x3-', 'ffserver.x30-',
'tomcat.3-', 'tomcat.4-', 'tomcat.5-']

base_dir = '/home/haow/testing/outdir/'

configs = ['LRU-assoc2', 'LRU-assoc4', 'LRU-assoc8',
'LRU-size16', 'LRU-size64', 'new-base', 'new-size16',
 'new-size64', 'new-nebit3', 'new-nebit5', 'new-nebit6']

for program in prog_names:
        for configuration in configs:
                file_name = base_dir + program + configuration + '/stats.txt'
                result = process_raw_stats(file_name, keywords)
                result_all.append(result)
                print result



#################### write formatted data to file

num_configs = len(configs)
num_columns = len(keywords)
num_programs = len(prog_names)

for col in range(num_columns):
        for prog in range(num_programs):
                for config in range(num_configs):
                        row = prog*num_configs + config
                        fout.write(result_all[row][col]+'\t')
                fout.write('\n')

fout.close()
```

Bibliography

[1] Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[2] Apache HTTP Server Project. http://httpd.apache.org/.

[3] Apache Tomcat. http://tomcat.apache.org/.

[4] Application Server Definition. http://en.wikipedia.org/wiki/Application_server.

[5] Dbench Workloads Generator. http://dbench.samba.org/.

[6] ffserver Streaming Server. http://www.ffmpeg.org/ffserver.html.

[7] Glassfish Application Server. https://glassfish.java.net/.

[8] IBM DB2 Database Software. http://www-01.ibm.com/software/data/db2/.

[9] JBoss Application Server. http://www.jboss.org/overview/.

[10] Jetty Application Server. http://www.eclipse.org/jetty/.

[11] Libgcrypt. http://www.gnu.org/software/libgcrypt/.

[12] LIVE555 Media Server. http://www.live555.com/mediaServer/.

[13] MySQL Database Management System. http://www.mysql.com/.

[14] openRTSP: a Command-line RTSP Client. http://www.live555.com/openRTSP/.

[15] OpenSSL Cryptography and SSL/TLS Toolkit. http://www.openssl.org/.

[16] PHP. http://php.net/.

[17] Postal: The Mad Postman. http://doc.coker.com.au/projects/postal/.

[18] SMBD File Server. http://www.samba.org/samba/docs/man/manpages/smbd.8.html.

[19] SysBench: A System Performance Benchmark. http://sysbench.sourceforge.net/.

[20] The gem5 Simulator System. http://gem5.org/Main_Page.

[21] VLC Media Server. http://www.videolan.org/vlc/.

[22] O. Aciiçmez. Yet Another Microarchitectural Attack: Exploiting I-cache. In *ACM Workshop on Computer Security Architecture*, pages 11–18, October 2007.

[23] Joseph Bonneau and Ilya Mironov. Cache-Collision Timing Attacks against AES. In *Proceedings of Cryptographic Hardware and Embedded Systems*, CHES'06, pages 201–215, 2006.

[24] E. Brickell1, G. Graunke, M. Neve1, and J.-P Seifert. Software Mitigations to Hedge AES against Cache-based Software Side Channel Vulnerabilities. Technical report, IACR ePrint Archive, Report 2006/052, Feb 2006.

[25] C. Percival. Cache Missing for Fun and Profit. In *Proc. of BSDCan*, 2005.

[26] D. J. Bernstein. Cache-timing Attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[27] J.-F. Dhem, F. Koeune, P.-A. Lerous, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A Practical Implementation of the Timing Attack. In *Proceedings of the International Conference on Smart Card Research and Applications*, pages 167–182, London, UK, 2000. Springer-Verlag.

[28] J. D. Gelas. Bulldozer for Servers: Testing AMD's "Interlagos" Opteron 6200 Series. http://www.anandtech.com/show/5058/amds-opteron-interlagos-6200/2, November 2011.

[29] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games — Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, 2011.

[30] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, USA, 2012.

[31] P. Kocher, R. B. Lee, G. McGraw, A. Raghunathan, and S. Ravi. Security as a New Dimension in Embedded System Design. In *Proceedings of the Design Automation Conference (DAC)*, pages 753–760, San Diego, CA, USA, June 2004.

[32] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, London, UK, 1999. Springer-Verlag.

[33] R. Könighofer. A Fast and Cache-timing Resistant Implementation of the AES. In *Proceedings of the 2008 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, pages 187–202, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] R. B. Lee and Y.-Y Chen. Processor Accelerator for AES. In *Proceedings of the IEEE 8th Symposium on Application Specific Processors*, Anaheim, CA, USA, 2010.

[35] F. Liu and R. B. Lee. Security Testing of a Secure Cache Design. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, New York, NY, USA, 2013. ACM.

[36] Fangfei Liu and R.B. Lee. Random fill cache architecture. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 203–215, Dec 2014.

[37] K. Mowery, S. Keelveedhi, and H. Shacham. Are AES x86 Cache Timing Attacks Still Feasible? In *Proceedings of the ACM Workshop on Cloud Computing Security Workshop*, pages 19–24, New York, NY, USA, 2012. ACM.

[38] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the Cryptographers' Track at the RSA conference on Topics in Cryptology*, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[39] D. Page. Partitioned Cache Architecture as a Side-Channel Defense Mechanism. Technical report, Cryptology ePrint Archive, Report 2005/280, 2005.

[40] D. Perez-Botero. Pwnetizer: Improving Availability in Cloud Computing Through Fast Cloning and I/O Randomization. In *MSE Thesis, Computer Science Department, Princeton University*, Princeton, NJ, USA, 2013.

[41] Z. Wang and R. B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 494–505, San Diego, CA, USA, June 2007.

[42] Z. Wang and R. B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *IEEE/ACM International Symposium on Microarchitecture*, pages 83–93, 2008.

[43] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. Cryptology ePrint Archive, Report 2013/448, 2013.

[44] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of 2012 ACM Conference on Computer and Communications Security*, pages 305–316, New York, NY, USA, 2012.

[45] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 990–1003, New York, NY, USA, 2014. ACM.