FA8750-12-2-0295

Using Moving Target Defense for Secure Hardware Design

Ruby B. Lee

Princeton University

February 2016

Final Report

AIR FORCE RESEARCH LABORATORY

AFRL/RIT

525 Brooks Road

Rome, NY 13441-4505

NOTICE AND SIGNATURE PAGE

This report was cleared for public release by the Wright-Patterson Air Force Base (WPAFB) Public Affairs Office and is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

FA8750-12-2-0295 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH THE ASSIGNED DISTRIBUTION STATEMENT.

ROBERT DIMEO, Program Manager
AFRL, Program Manager

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
03/01/2016	Final Performance Report	09/28/2012-08/30/2015

4. TITLE AND SUBTITLE
Using Moving Target Defense for Secure Hardware Design

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-12-2-0295

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Ruby B. Lee, PI

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Princeton University
E-Quad, Rm B218
Princeton, NJ 08544

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER

12. DISTRIBUTION AVAILABILITY STATEMENT

13. SUPPLEMENTARY NOTES

14. ABSTRACT
A secure cache, called Newcache, is designed that can thwart cache side-channel attacks that can leak secret and critical information. All caches today are susceptible to these cache side-channel attacks, which are software attacks on shared hardware caches, despite software isolation of memory pages in virtual address spaces or Virtual Machines. These cache attacks can leak secret encryption keys or private identity keys, thus nullifying any protections provided by strong cryptography. Newcache uses a novel dynamic, randomized memory-to-cache mapping to thwart contention-based side-channel attacks, rather than the static, deterministic mapping used by conventional set-associative caches. It implements a Moving Target Defense in hardware caches. This project implements a cycle-accurate simulation of Newcache to verify its security against a comprehensive cache side-channel attack suite we develop. We also create a representative cloud computing benchmark suite and evaluate the system performance of Newcache for both cloud computing benchmarks and smartphone benchmarks, and find that Newcache performs as well as conventional set-associative caches. We also design a VLSI testchip with a 32Kbyte Newcache and a 32 Kbyte set-associative cache, and verify that the access latency and power of the two caches are comparable. Thus, feasibility and deployability of the secure cache is established. We also find a solution for cache reuse attacks to further enhance Newccche security.

15. SUBJECT TERMS					
Secure cache, Newcache, cache side-channel attacks, information leakage, randomized memory mapping, hardware security, secure hardware design, moving target defense, VLSI test chip, side-channel attack suite, cloud computing benchmark suite, security testing, behavioral modeling, performance evaluation using gem5, test chip design and fabrication.					
16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
				102	Ruby Lee
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPONE NUMBER (*Include area code*)**
					609.258.1426

Table of Contents

List of Tables

List of Figures

1. Summary

With the number of cyber attacks escalating, it is crucial that we protect the confidentiality and integrity of data and programs in our networked computers, like smartphones, notebook and desktop computers and cloud computing servers. Although strong cryptography can be used to encrypt and authenticate data, this protection is rendered useless if the secret crypto keys can be leaked out. It turns out that this can be done rather easily through cache side-channel attacks, which are software attacks on hardware caches. Today, all processors with caches are susceptible to these cache side-channel attacks – this essentially comprise *all* computers from embedded systems to smartphones to cloud computers.

Software isolation technology, like virtual address spaces for different processes, or partitioned machine memory for virtual machines, only isolate memory pages – the underlying hardware cache hierarchy is still shared at some cache level. These hardware caches are essential for the performance of modern computers to bridge the speed gap between fast processors and slow memories.

It is important to understand that such leakage of critical and secret information through cache side-channel attacks happen with correctly functioning caches. Unlike software security vulnerabilities that are often due to software flaws, cache side-channel attacks are not due to hardware flaws. They use the intrinsic behavior of caches, where cache hits are fast (when requested data is found in the cache) and cache misses are slow (when requested data is not in the cache and has to be brought from the next level of slower caches or from main memory).

While software solutions have been proposed, they incur significant performance degradation, reported at 3X to 10x slowdown. Also, their security is not assured, since software cannot directly control the hardware-managed caches, which can change with different implementations of the processor-cache subsystem. Furthermore, while it may be possible to change the code for well-known crypto libraries, it is not possible to change all legacy code with embedded crypto routines and other software using secret or sensitive data or code.

Hence, since this is a hardware problem, a hardware solution is desired. Secure caches have been proposed using various forms of cache partitioning, but these impact system performance of either the crypto software or other software on the system. A novel secure cache based on randomized memory-to-cache mapping, called Newcache, has the promise of defeating cache side-channel attacks without degrading performance [1]. However, no security testing, limited performance evaluation and no testchip was designed for the original Newcache, to prove its feasibility. This project does a thorough evaluation of the security and performance of Newcache at the system level, using a detailed behavioral model of Newcache on a full system simulator. We also design, fabricate and test a VLSI testchip implementing a 32 KByte Newcache and a conventional 8-way set-associative cache of the same size, to prove its feasibility, and measure its physical characteristics like its latency for a cache access, the power consumed and its area overhead.

1.1 Summary of Security Evaluation

We identified different classes of cache side-channel attacks based on contention versus reuse in the cache, access-based versus timing-based attacks, and whether the attacks are on instructions or on data. We developed the first attack suite of cache side-channel attacks as a contribution of this project. This included known published attacks, as well as new attacks that we constructed. Although our initial project goals were for contention based cache side-channel attacks, we expanded the scope to also consider reuse based side-channel attacks, like the cache collision attack.

The results of security testing with our attack suite show that Newcache defeats all known cache side-channel attacks on the L1 data and instruction caches, with one exception.

The only exception is the cache collision attack which Newcache mitigates by increasing the attacker's work factor by at least an order of magnitude. A cache collision attack depends only on the reuse of cache lines by the victim program itself (cache hits), thus exploiting the fundamental purpose of a cache. We are happy to report that, beyond the scope of our original project goals, we have also found an elegant solution to the cache collision attack by a modification to the cache controller, which we call a Random-Fill cache [31]). Thus, a Newcache with a Random Fill cache controller would defeat all cache attacks on the L1 D-cache. The cache controller of the I-cache does not need to change, and can be a conventional cache controller, while Newcache can be used as the L1 I-cache itself.

We also constructed a targeted attack on Newcache, i.e., an attack by an expert who understood Newcache to the last detail (Liu and Lee[4]). This resulted in an enhanced Newcache that defeated even such targeted attacks. The Newcache described and evaluated in this report is our enhanced Newcache.

1.2 Summary of Performance Evaluation results

PI Lee's previous work on Newcache [1] did performance evaluation only on the SPEC benchmark suite using the Simplescalar simulator. SPEC benchmarks are compute benchmarks that may not be representative of current and future computing environments. Hence, we perform thorough performance evaluation on smartphone benchmarks and cloud computing benchmarks, using the more advanced gem5 cycle-accurate simulator. For the smartphone benchmarks, we chose 4 benchmark suites (Bbench, 0xBench, coremark and Mibench). For cloud computing, we developed our own cloud benchmark suite with 6 representative cloud computing workloads: Web Server, Database Server, Mail Server, File Server, Streaming Server and Application Server (see section 5.3.1 and Table 24). To evaluate the server benchmark performance, we had to enhance the gem5 dual-system mode for client-server operation running on Intel x86 processors (see sections 3.3 and 5.3.2).

We summarize the detailed performance results presented in section 5 Performance Evaluation:

While there may be some impact on cache miss rates, the overall performance impact (in Instructions per Cycle, IPC) of substituting Newcache for an L1 data cache, L1 instruction cache, or L2 cache, is negligible.

For smartphone benchmarks, the parameter of Newcache known as “nebits” (number of extra index bits) need not be larger than 4 for best performance results, while for cloud computing server benchmarks, performance can be improved with nebit going up to 6 bits. This nebit parameter essentially increases the size of the ephemeral Logical Direct Mapped cache, but not the size of the physical cache of the Newcache architecture. Newcache actually increased the overall performance of the server benchmarks by more than 2%, when nebits=6 for a 32 Kbyte L1 Data cache. For both smartphone and server benchmarks, we find that Newcache with a larger cache size (e.g., 64 KB) incurs less L1 data cache miss rate increase relative to the 4-way or 8-way SA cache, respectively.

When Newcache is used as the L2 cache in server benchmarks, there is a slight impact on overall performance (IPC), of about 2-3%. Although Newcache as L2 cache incurs a relatively large increase of both local and global L2 cache miss rates, this has negligible impact on overall performance, because the global L2 cache miss rate is very small – about 0.1%.

When Newcache is used as both the L1 D-cache and the L1 I-cache, while leaving the L2 cache as a set-associative cache, we observed negligible impact on overall performance of less than 1%.

To summarize, there is no performance overhead using Newcache as a L1 D-cache and/or I-cache. The L2 cache can remain as a set-associative cache, since overall performance is slightly better than also using Newcache for the L2 cache.

1.3 Summary of Testchip Results

The feasibility of the Newcache physical design was established with the testchip we designed, fabricated, tested and measured. There is a small area overhead of 10% for the cache (which would mean a negligible overall processor core overhead of about 1%). The power overhead was measured at around 20% for our cache designs, but this can be optimized further in future, low power designs. The overhead in access time latency of Newcache versus the 8-way set-associative cache we implemented was inconclusive in our testchip, since the latency was limited by the data memory portion which was common to both caches and which we did not spend time optimizing. The novel part of Newcache is the dynamic indexing of the Data and Tag memory lines using a CAM (Content Addressable Memory) rather than a static address decoder. Our testchip showed that the CAM was not the performance limiter in the cache access latency.

In conclusion, this project has showed the feasibility of Newcache as a substitute for conventional set-associative caches that will defeat cache side channel attacks without degrading system performance. Physical characteristics of the Newcache design are within acceptable ranges that can be improved in future work.

2. Introduction

Security is an increasingly important consideration in cache design, in addition to performance and power efficiency. The fundamental cache behavior (cache hits and misses) can be exploited to reveal critical information through cache side channel attacks. When the victim program is performing an encryption or decryption these cache side-channel attacks can leak the secret key, thus nullifying any confidentiality or integrity protection provided by strong cryptography.

Although software isolation methods like disjoint Virtual Address Spaces are used, so that memory pages are not shared by two processes, the underlying hardware caches are still shared. This is also true for different Virtual Machines, with isolation enforced by the Virtual Machine Monitor (VMM) so that memory pages are not shared between Virtual Machines – but the underlying hardware caches may still be shared.

The attacker is just a user-level process that does not break any rules – it merely accesses memory in its own assigned address space. The victim process may be performing any computation; in this report, we focus on the victim performing an encryption with a secret key in symmetric-key ciphers, or with a private key in public-key ciphers, to show how easily the attacker can learn the victim's secret or private key. Since almost all processors, from client-side computers to server computers, contain caches, and all current caches are vulnerable to these cache side-channel attacks, these attacks are a very serious threat to the information leakage of critical, secret or sensitive information.

Software solutions involve re-writing the algorithm and the software cipher program, but this incurs significant performance degradation. Furthermore, while it may be possible to replace cryptography libraries with new software, it is essentially impossible to replace all the embedded cryptography found in legacy software. Also, side-channel attacks can leak non-cryptographic information, such as credit card numbers, passwords, and other secret or sensitive information. Hence, since these cache side-channel attacks are attacks on hardware caches, ideally, a hardware solution that intrinsically mitigates cache side-channel attacks is desired, that works for legacy programs and does not degrade the performance of the system. **This project details such a secure cache design and thoroughly evaluates its security, system performance, access time and power consumption.**

2.1 Background on Conventional Caches

A key characteristic of caches, that is a major cause of cache side channel attacks, is the *fixed memory-cache mapping*. Figure 1 shows a typical Set-Associative (SA) cache -- shown as 8-way SA in Figure 1 -- where a fixed number of bits from the memory address (labeled ADDR) is used to select (or index into) a set of cache lines (here, 8 cache lines in a set, for an 8-way SA cache). All cache lines in a set are compared with the "tag" bits of the memory address to see if there is a match (called a cache hit) or no match (cache miss). On a cache miss, one of the lines in the set is chosen to be replaced by the new cache line containing the data or instructions being requested by the executing program. The replacement algorithm is typically

Least Recently Used (LRU), meaning the line in the set whose last use by the program is furthest in the past. A Direct-Mapped (DM) Cache is a 1-way SA cache, while a Fully-Associative (FA) cache has only 1 set. Almost all Level-1 caches (caches closest to the processor) in microprocessors and multi-core processors today are 4-way or 8-way SA caches.

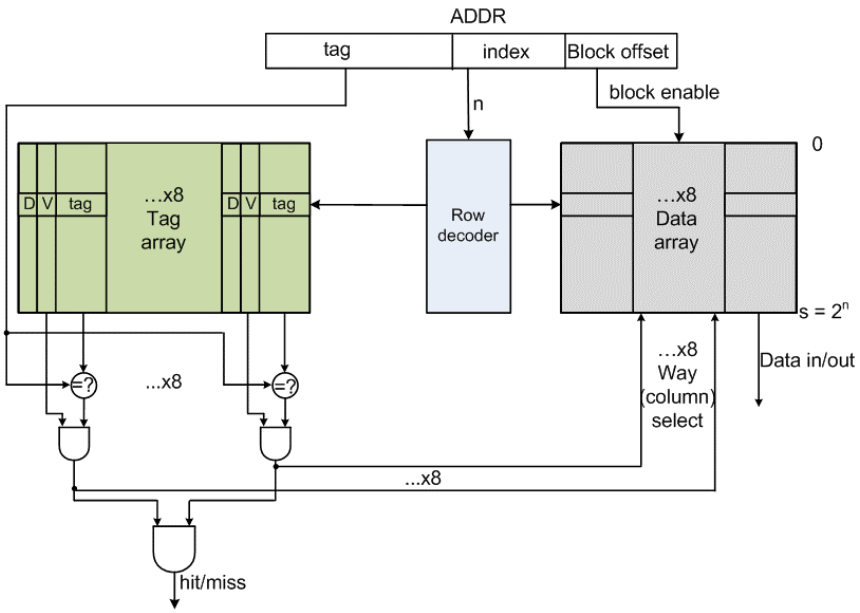

Figure 1 Block diagram of 8-way set associative cache

Different techniques have been proposed for mitigating cache-based side-channel attacks, from draconian measures like turning off the cache if possible (performance drops like a rock), and rewriting software (not completely secure and performance also drops very significantly up to 10X). Hardware techniques include cache isolation [2][3] and cache randomization [1][2].

2.2 Background on Newcache

Cache randomization is a novel mitigation strategy for defeating cache-based side-channel attacks. It makes the memory-to-cache mapping dynamic (rather than statically defined) and randomized (rather than deterministic like LRU). Hence, the memory-cache mapping is different even if the same program is executed on an identical machine, and even from one execution of the program to another on the same machine. This thwarts the attacker by using a moving target defense (MTD) in the design of hardware caches. This is the topic of our research program on "Using Moving Target Defense for Secure Hardware Design." We focus on a secure cache, using dynamic, randomized memory-cache mapping, which we call Newcache, that was originally designed in our PALMS lab [1]. This technical report is a thorough description of work done on evaluating Newcache; its behavioral modeling, its security, its system

performance evaluation, and the design and measurement (latency, power and area) of the Newcache test chip compared to a conventional 8-way set-associative cache of the same size.

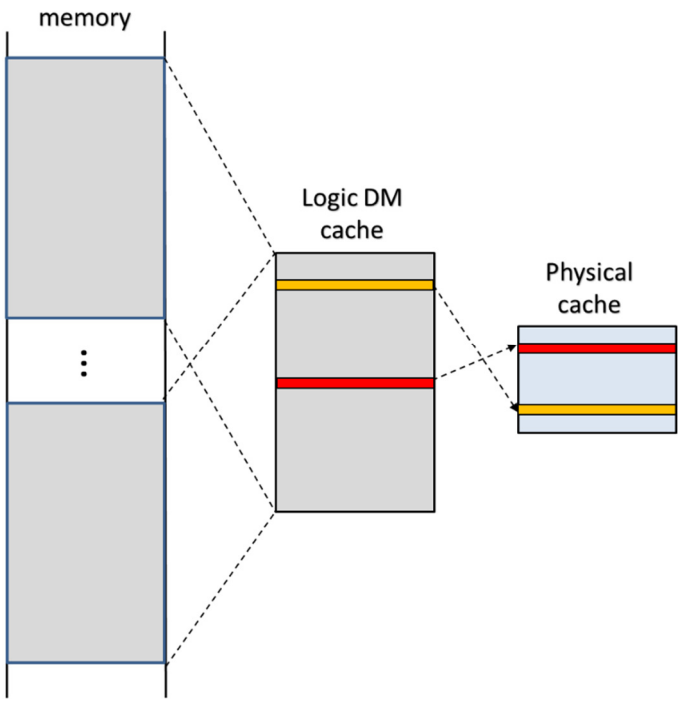

Figure 2 Conceptual two-step logical mapping in Newcache

Figure 2 is a conceptual diagram of Newcache, the secure cache of this project. Newcache memory-to-cache mapping can be described logically as follows: it first maps the memory address space to a logical direct mapped (LDM) cache that is typically larger than the physical cache, and then does a fully-associative mapping of cache lines from the LDM cache to the physical cache. The LDM cache does not actually exist, and the physical (circuit) implementation of Newcache is very similar to a Direct-Mapped cache except for the LNregs described next. As shown in Figure 3, a cache access starts by comparing part of the memory address (called index bits) with the mappings stored in the contents of the line number registers (LNregs); on a match with the contents of a LNreg, the single corresponding physical cache line is accessed, similar to a direct mapped cache (i.e., the tag corresponding to this cache line is compared with the tag bits of the memory address to check for a cache hit). Thus Newcache must have both an index hit and a tag hit to have a cache hit. It has a cache miss if there is either an index miss or a tag miss.

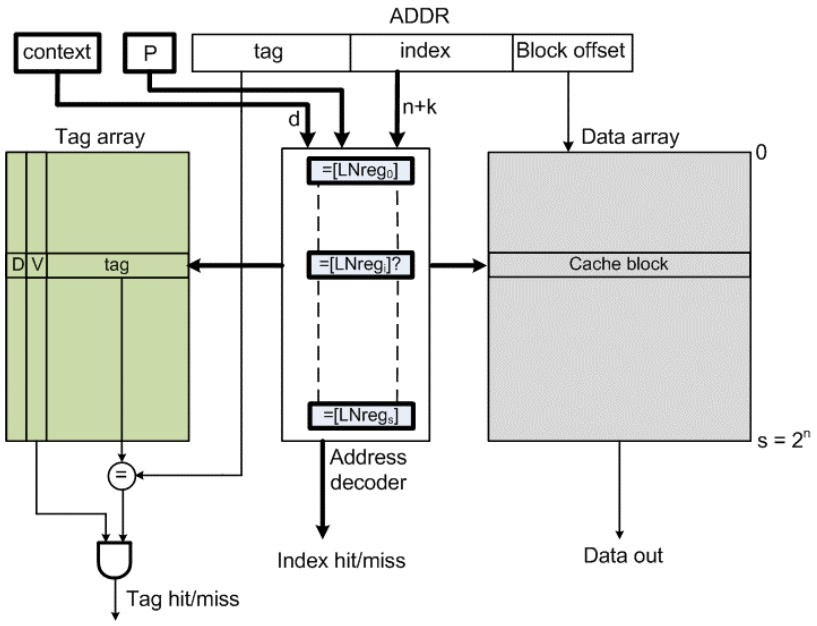

Figure 3 Block diagram of Newcache architecture

The Logical Direct Mapped cache (LDM) is implemented merely by making the width (n+k bits) of the LNregs wider (the "+k" bits) than that needed to select one of 2^n cache line slots (the "n" bits). These k extra bits are sometimes called "nebits" (number of extra bits) in this report. For example, with nebits = k = 2, we have a Logical Direct Mapped cache that is at least 4 times as large as the physical cache. (Note that another benefit of Newcache is that it does not require the number of sets in the physical cache to be a power of two.)

Although a fully-associative cache with a fully random replacement algorithm may also randomize the memory-cache mapping, it causes unacceptable power overhead and longer access times. In contrast, Newcache tries to achieve the best of both worlds – the low power and fast access time of Direct-Mapped cache with the better cache miss-rates and system performance of a Fully Associative cache. The latter system performance goal is also that of Set-Associative (SA) caches, however these SA caches are not secure, as every such conventional cache is susceptible to cache side-channel attacks which can leak critical information like secret cryptographic keys.

To eliminate potential cache conflicts (i.e., sharing of cache line slots) between a victim process and an attacker process, Newcache also provides each trust domain with a disjoint mapping (identified by RMT_ID) so that no conflicts would ever occur between two processes from different trust domains. Newcache can also eliminate the cache conflicts between protected memory regions and non-protected memory regions within a process, by using a "P" bit to indicate a protected cache line and using a security-aware replacement algorithm. By comparing the cache block diagrams in Figure 1 and Figure 3, we see that the most significant difference of Newcache (Figure 3) compared with a conventional Set-Associative cache (Figure 1) is that Newcache replaces the row decoder (commonly called the "address decoder") with an array of LNregs that store the dynamic mapping of memory addresses to cache line slots.

2.3 Multiple Levels of Caches and Non-blocking caches

Because of the great difference in speed between the fast processor and the main memory, there are typically two levels of caches (L1 and L2) in small computers like smartphones, and three levels of caches (L1, L2 and L3) in high-performance servers. Figure 4 shows two levels of caches, with separate Level-1 Instruction cache (L1I) and Level-1 Data cache (L1D) and a unified Level-2 cache (L2) containing both instructions and data.

A non-blocking cache is one where the processor is not blocked on a cache miss but is allowed to continue execution. Multiple outstanding cache misses are allowed, and these cache miss requests to the next level of cache or main memory are saved in a queue, called the MSHR (Miss Status Handling Registers) in Figure 4.

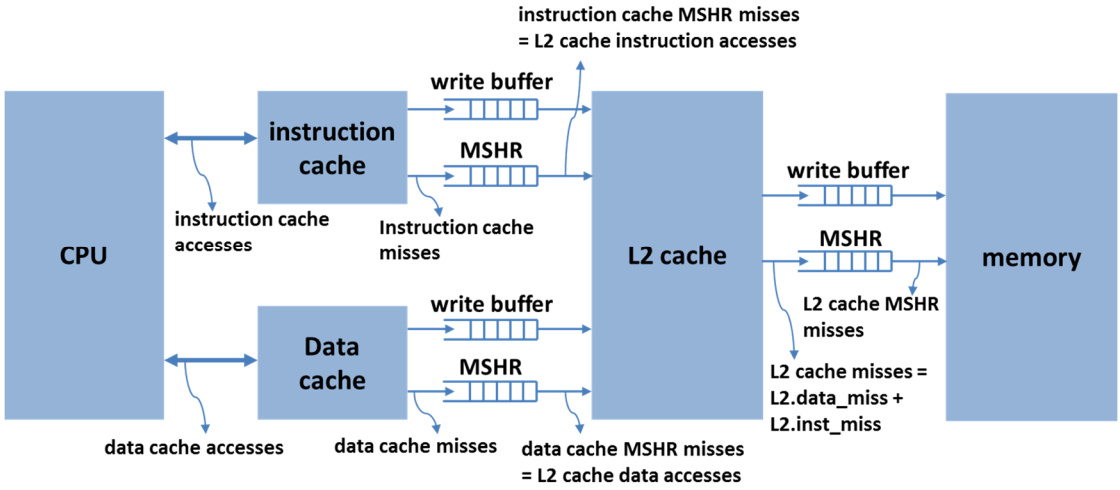

Figure 4 Hierarchy of two levels of non-blocking caches

In this report, we implemented and evaluated the enhanced Newcache, described in [4] and shown in Figure 3. In the enhanced Newcache [4], the P-bit is compared with the LNreg contents rather than with the tag, as in the original Newcache [1].

3. Methods, Assumptions and Procedures

In this section, we describe how we achieve a detailed, cycle-accurate simulation of the secure cache architecture represented by Newcache. We call this a behavioral model of Newcache, which can enable us to run Newcache as a Level-1 Data cache (L1 D-cache), and/or a Level-1 Instruction cache (L1 I-cache) and/or a Level-2 unified instruction and data cache (L2 cache).

First, we looked at various simulators, and selected the gem5 open-source simulator [5], because this is a very mature and well-honed simulator that gives cycle-accurate simulation with advanced cache and memory models, and also full simulations of both the Intel x86 processor architecture as well as the ARM processor architecture. We want to use the Intel x86 architecture for cloud server workloads, since it is the dominant processor used in cloud computing, and we want to use the ARM architecture for smartphones, since it is the dominant processor architecture in smartphones and tablet computers. The gem5 simulator also allows us to run a full system simulation, which is necessary both for security testing and for system performance evaluation.

3.1 Newcache Behavioral Model

We use the open source gem5 simulator [5] for Newcache behavioral modeling. The gem5 simulator has cycle-accurate simulation capability for different processors (e.g., ARM, x86, SPARC, MIPS, Alpha) and different cache and memory subsystems.

The gem5 simulator runs in Full System (FS) mode and System Emulation (SE) mode. In FS mode, the OS runs on the simulator as on a real machine, and the OS handles system calls. In SE mode, the OS is not supported and system calls are emulated – handled by the host machine. Also, FS mode handles multi-tasking while SE mode does not. Clearly, FS mode is more accurate but takes much longer to simulate.

The source code of our gem5 modeling of Newcache is available at https://github.com/eepalms/gem5-newcache.

3.1.1 Memory System in Gem5

Before going into the detail of modeling Newcache in gem5, we first take a look at how gem5 models the memory system. As shown in Figure 5, gem5 follows a modular design, in which all the hardware components are instances of C++ objects. There are only a small number of memory objects, such as cache, bus, DRAM etc. Different caches in the memory hierarchy are only different instances of the same cache class. Ports are used to interface memory objects (see Figure 6) and always come in pairs. Each port implements some standard interfaces to send and receive messages. Gem5 models the computer system at two different levels of complexity. The simplest model is the atomic model which only models the functionality of a component, while the more complicated model is the detailed model which models the detailed timing of a component. The message is represented by a packet to encapsulate the transfer between two memory objects.

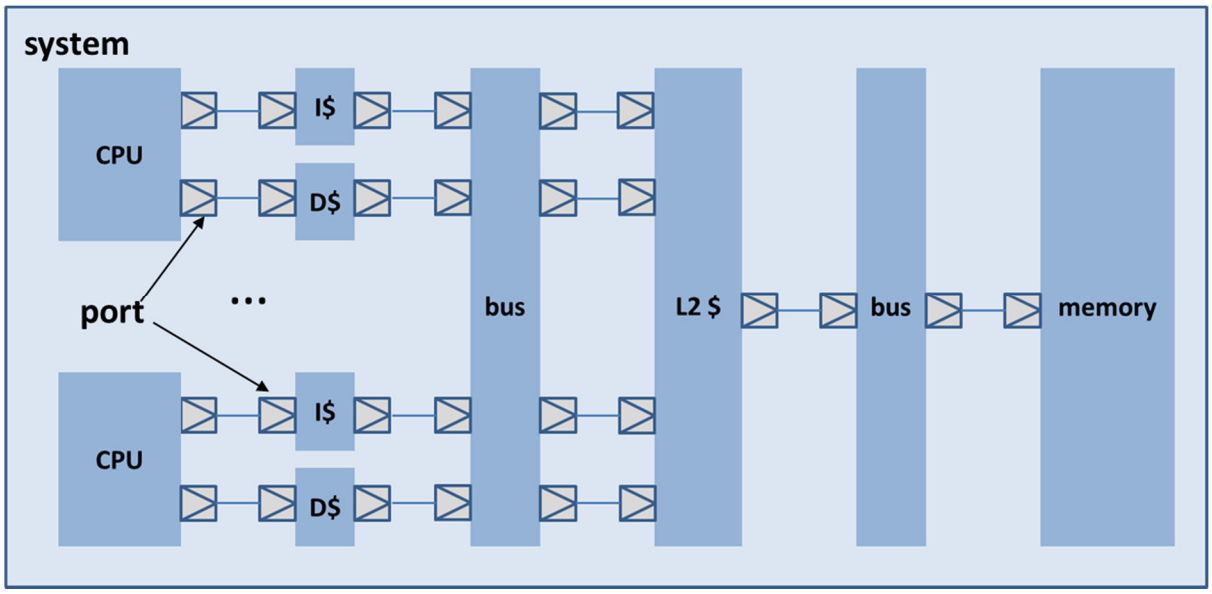

Figure 5 Memory system in gem5 simulator

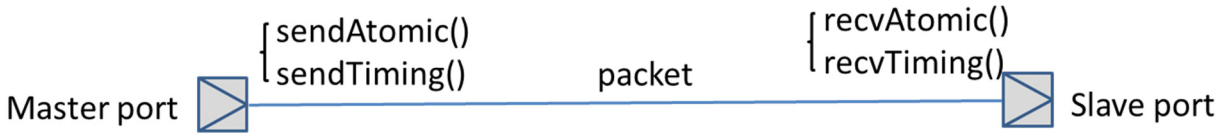

Figure 6 A pair of ports and message transfer in gem5

3.1.2 Cache object

Figure 7 shows the internal structure of a cache object. It consists of a cache and a tag store. The cache essentially implements a non-blocking cache controller, which interfaces with the processor and other parts of the memory system via a CPU side port and a memory side port, respectively. It also implements the write back policy and a simple MESI based cache coherence protocol. The Tag Store is the core component of the cache object, which models the data array, tag array, address decoder as well as the replacement algorithm. The Tag Store exposes several interfaces for the cache controller to access a cache block, find a victim cache block to replace, and insert a cache block into the Tag Store. In order to model the behavior of Newcache, we only need to change the internal implementation of the Tag Store, but can keep the cache controller and interfaces unchanged, making it as simple as adding a new replacement algorithm.

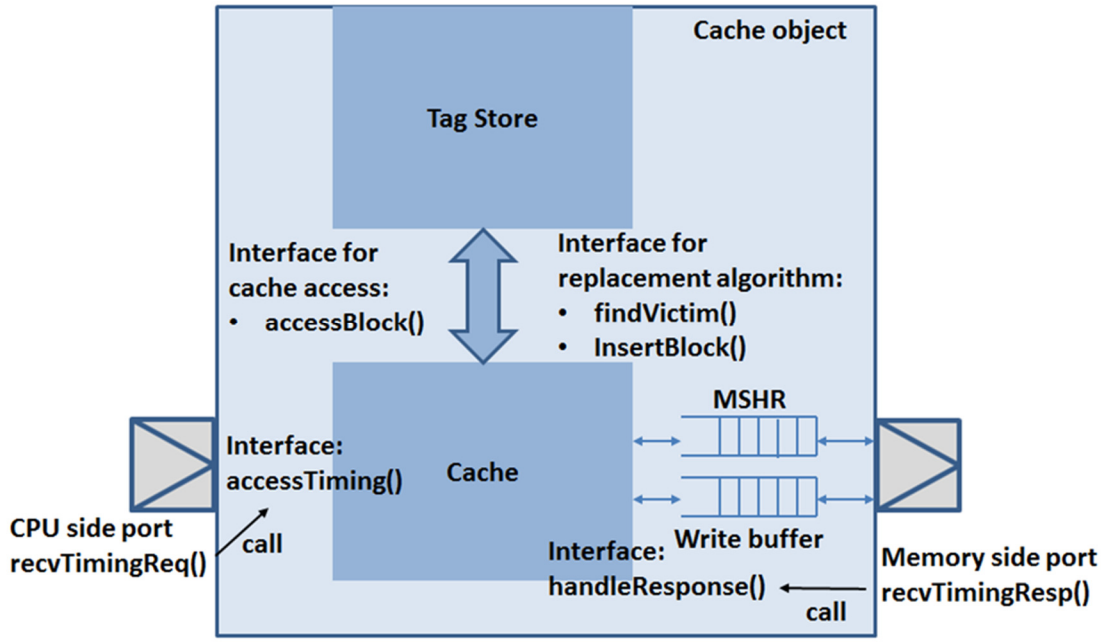

Figure 7 Internal structure of cache object

3.2 Implementation of Newcache in gem5

We implement the behavioral model of Newcache in gem5 as a new replacement algorithm, which allows us to reuse the common data structure for the Cache block and Cache Controller currently implemented in gem5 for the LRU cache. We also keep most of the interfaces unchanged. Although Newcache requires more information to index the cache (RMT_ID bits and "P" bit), we can encapsulate these information inside the packet and ensure the interfaces remain unchanged. In this section, we describe the implementation of the Newcache logic and system interfaces in detail.

3.2.1 Configurable cache parameters

The instance of a gem5 object can be specified using a Python script and a Python object can be automatically converted to a C++ object. Users can instantiate different cache objects by configuring the cache parameters. For example, Figure 8 shows that the typical configurable parameters for a conventional cache include cache size, cache block size, associativity, and replacement algorithm (with string type), as well as parameters related with the MSHR. Since Newcache is implemented as a new replacement algorithm, we can instantiate Newcache by configuring the replacement algorithm as "NEWCACHE". In addition, Newcache does not have the associativity parameter. Instead, it has its own parameter, known as the Number of extra index bits (nebits, or k).

	Tag Store related parameters	MSHR related parameters
Conventional cache	• Cache size ***C*** • Cache block size ***B*** • Associativity (for set-associative cache) ***A*** • Replacement algorithm ***R***	• Maximum number of MSHRs in MSHR queue • Maximum fetch requests allowed per MSHR
Newcache	• Cache size ***C*** • Cache block size ***B*** • Number of extra index bits ***k*** • Replacement algorithm ***R***	

Figure 8 Configurable cache parameters

3.2.2 Key data structure for Newcache

Gem5 implements the conventional set-associative cache with LRU replacement algorithm using several levels of abstractions. The first level of abstraction is a flat array of cache blocks, which contains as many as the number of cache blocks in a cache. Each cache block contains a tag, a data field, as well as associated status bits (e.g., dirty, valid etc.). A set-associative cache is built on the flat array of cache blocks. In particular, it contains an array of cache sets object. One important field of the cache set object is a pointer to an array of cache block pointers (a.k.a. the way pointers in Figure 9), which points to the cache block contained in the cache set. The array of way pointers facilitates the implementation of the LRU replacement algorithm. In particular, the first entry in the array of way pointers always points to the most recently used (MRU) cache block in a set, while the last entry in the array always points to the least recently used (LRU) cache block in a set.

As shown in Figure 10, to implement Newcache, we still reuse the same flat array of cache blocks. In particular, we add a field called LNreg which is the index field of the address. This allows us to reconstruct the block address for a cache block. We still keep the array of cache sets, which only consists of a single cache set object for Newcache. We replace the array of way pointers data structure with a Map data structure, to speed up the associative search. The Map data structure maps the search key LNreg to a pointer to a cache block. Unlike the way pointer for the set-associative cache, the pointer in the Map data structure is fixed. The search key for the LNregs is a concatenation of the *d* bits for the context RMT_ID, the *1* bit "P" bit and the *n+k* bits index bits. We use the Map template class in C++ STL (standard template library) to implement the map data structure. Since the search key is organized as a balanced binary search tree, the time complexity for associative search is O(logN). The number of entries in the map data structure is equal to the number of cache blocks in the cache. The operation for the insertion of a cache block into the cache ensures that one cache block can only be mapped to a single search key, therefore, all the mapping in the Map data structure is 1-to-1 mapping.

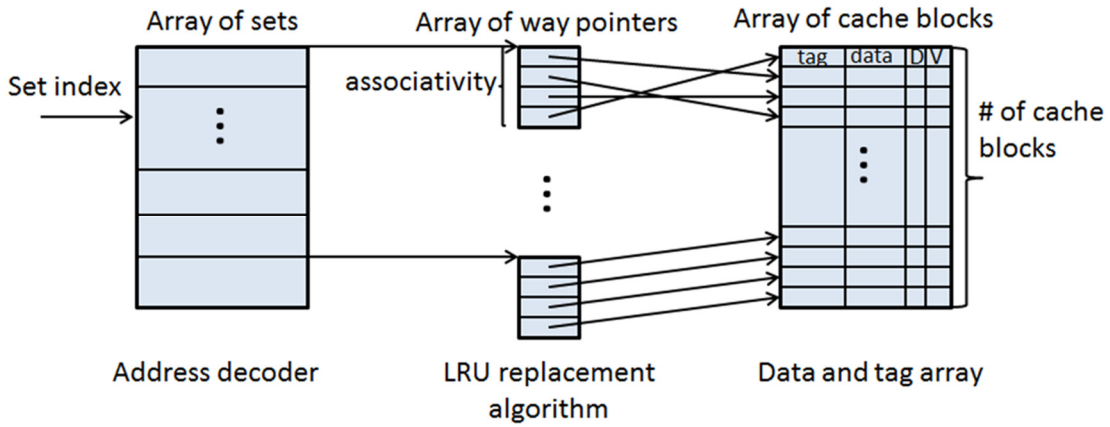

Figure 9 Gem5 data structure for a set-associative cache with LRU replacement algorithm

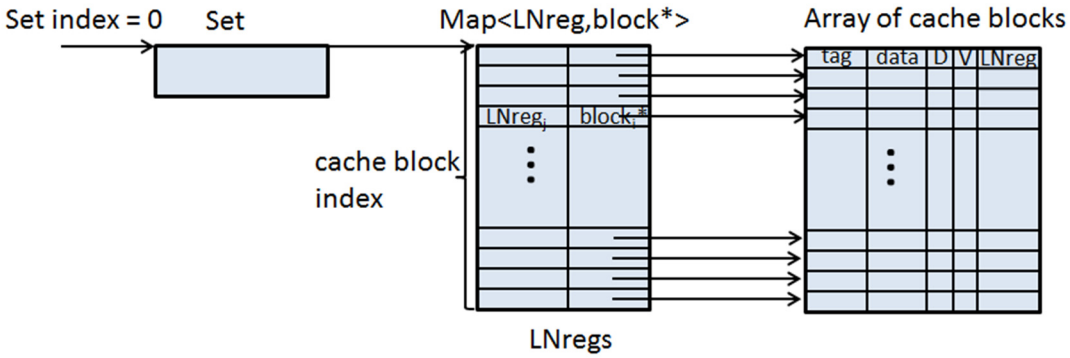

Figure 10 Data structure for Newcache

3.2.3 Implementation of system interfaces

Newcache requires that some trusted software sets the "P" bit for the security-sensitive code or data within a process, as well as the RMT_ID for a trust domain. If the Operation System (O.S.) is trusted, it can do this. If the O.S. is not trusted, but the hypervisor is trusted, then the hypervisor can do this. In the following, we assume the O.S. is trusted.

System interface for "P" bit

We extend page table entry and TLB entry with an extra bit to indicate whether a page is protected or not. The implementation is slightly different for the SE mode and the FS mode. In SE mode, the setup of the page table is done by gem5. We therefore provide several simulator options for a user to specify which address range needs to be protected. These simulator options include: *need_protect, protect_start, protect_end*, which specifies whether the program needs to be protected, and if so, the start and end address that needs to be protected. Gem5 reads in these options and sets the "P" bit in the page table entry correspondingly. In FS mode, the setup of the page table is done by the operating system. In particular, we use the 11th bit (available for user) in the page table entry as the "P" bit. The operating system can implement a new syscall, or extend the existing syscall (e.g., mprotect in Linux) to set the "P"

bit. We also modified the hardware page table walker so that the "P" bit in the page table entry can be copied to the "P" bit in the TLB entry. Since the "P" bit needs to be passed all the way down to the cache, we add a new flag in the request class to indicate whether the corresponding memory address in the request is protected or not. The flag in the request is set when accessing the TLB and the same request can be passed all the way to the cache.

System interface for RMT_ID

Ideally, RMT_ID can be implemented by adding a new model-specific register (MSR), which can be read and written by *rdmsr* and *wrmsr* instructions. For simplicity, we choose to use an unused (reserved) control register *CR1* to store RMT_ID, which can be read and written by the *mov* instruction. In SE mode, since one core can only run one task, *CR1* can be simply set to zero during the initialization phase. In FS mode, the operating system can associate each process with a different RMT_ID and set *CR1* during a context switch. Similar to the implementation of "P" bit, we add a RMT_ID field in the request. The CR1 register is read out and copied to the request in parallel with the TLB access.

3.3 Dual system configuration

In order to characterize Newcache performance for a cloud server, we need to set up a dual system on the gem5 simulator. We model the dual system on the x86 architecture since Intel x86 microprocessors are widely used in the server market. In the dual system (see Figure 11), the server is the test system we are interested in and runs in detailed CPU mode (with cycle-accurate timing), whereas the client is the drive system that is run in atomic CPU mode to save simulation time. The test system and the drive system are connected directly through an Ethernet link. Setting up x86 dual system is not immediately available in gem5 since gem5 does not provide a working x86 dual system. We had to modify the Python configuration file to explicitly add an Ethernet device (Figure 12) to the PCI bus and create a corresponding entry in the MPtable (a data structure in BIOS for configuring interrupts in a multiprocessor system) to properly connect the interrupt line of the Ethernet device to the I/O APIC (advanced programmable interrupt controller).

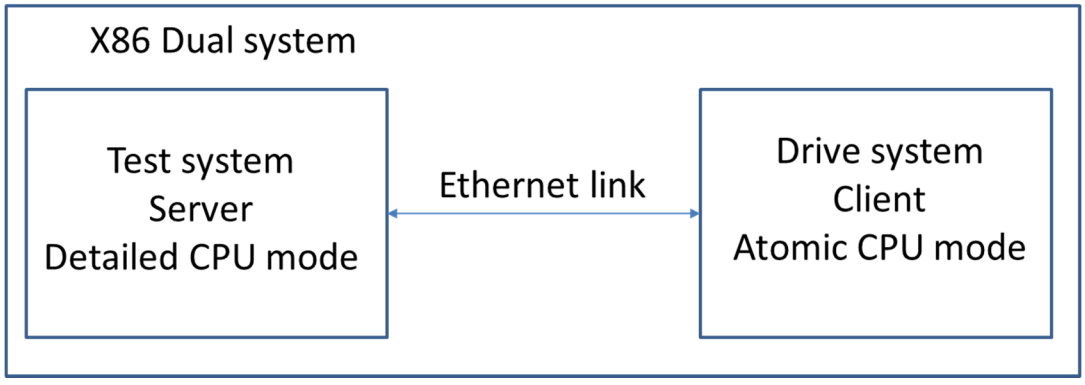

Figure 11 x86 dual system

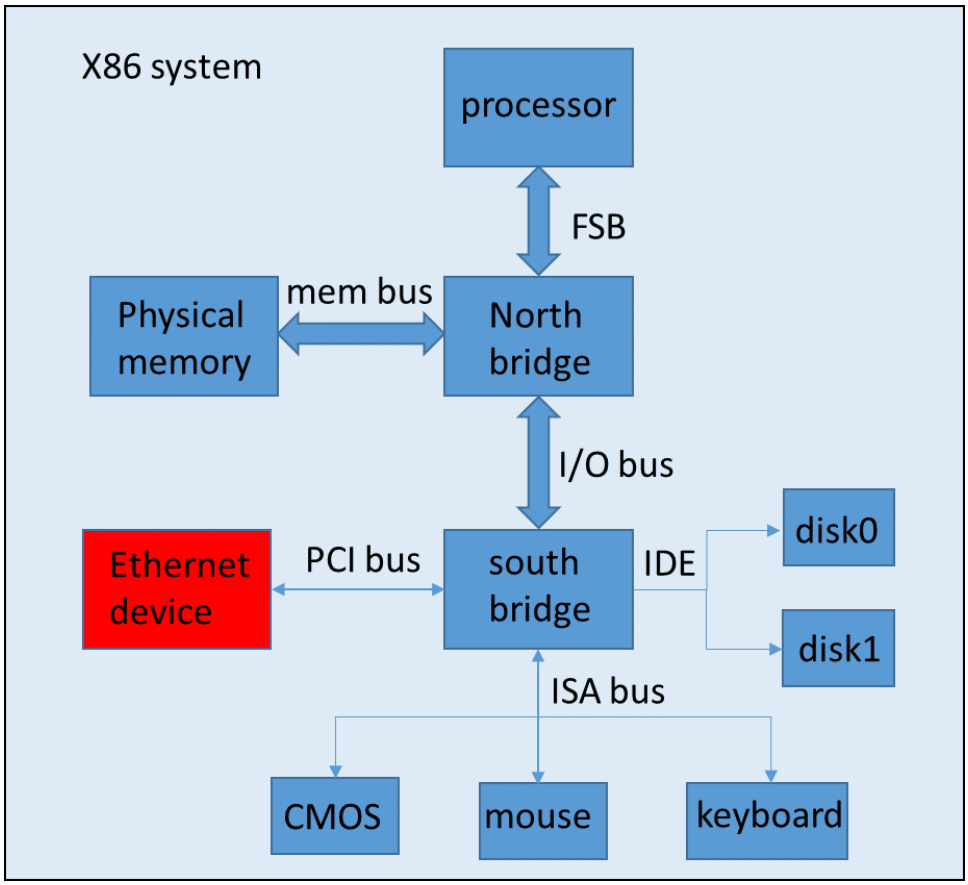

Figure 12 Internal connection of x86 system

The Ethernet device is a PCI device. Gem5 already models a generic PCI device, hence we only need to create an instance of a PCI device with a unique bus id (8-bits), device id (5-bits) and function (3-bits). The bus id should be the same as the PCI bus id. In order to make it visible to the north bridge, we need to manually specify the address range for the memory-mapped I/O. Furthermore, we need to connect the interrupt line of the Ethernet device to the I/O APIC.

The modeling of the interrupt controller is shown in Figure 13. Gem5 supports both PIC (programmable interrupt controller) and APIC. Interrupts connected to PIC are also routed to the APIC. In particular, PIN1~PIN15 of the I/O APIC are assigned to PIC interrupts. I/O APIC PIN16 is assigned to the IDE controller. Therefore, we can assign PIN17 of the I/O APIC for the Ethernet device. Lastly, we need to add the new I/O interrupt assignment in the MP configuration table (Gem5 already has corresponding model). The MP table is a data structure in BIOS, which is used to provide information about the multiprocessor configuration for the operating system. In particular, the MP table includes configuration information for the processor (one entry per processor), bus (one entry per bus), I/O APIC (one entry per I/O APIC), I/O interrupt assignment (one entry per bus interrupt source) and local interrupt assignment (one entry per system interrupt).

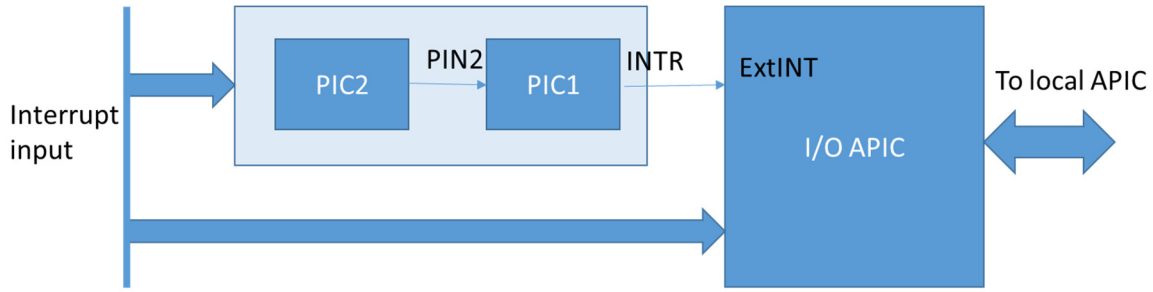

Figure 13 Connection of interrupt controller

After the Ethernet device is properly configured and connected to the system, both server and client need to do some extra work and explicit synchronization between the server and the client. This is achieved by executing a different shell script in the server and another script in the client.

In general, the work that the server must do include:

1) Boot the operating system
2) Configure its network interface with a static IP address (e.g., 10.0.0.1)
3) Start the service. Note that some Server-side benchmarks require some time to start up before their services are available to the client under gem5. We need to use a daemon program to test periodically if the service ports are opened up by the benchmark.
4) Configure the server if necessary.
5) Send 'ready' signal to the client for synchronization.

The work that the client must do include:

1) Boot the operating system
2) Configure its network interface with a static IP address (e.g., 10.0.0.2)
3) Listen for the 'ready' signal.

When the client receives the 'ready' signal, it starts sending requests and this begins the region of interest for which we need to collect statistics.

3.4 Develop Suite of Cache Side-Channel Attacks for Security Evaluation

To facilitate deployment, it is necessary to do security testing on both our secure Newcache design as well as on a conventional set-associative cache, to see if the cache side-channel attacks are defeated in the former, and not in the later. Since no suite of such cache side-channel attacks exist, we had to construct our own attack suite, which covers all known attacks. This is described in detail in Section 3 and Table 2.

While the original goal of the project was to consider only the contention-based cache attacks that Newcache was designed for, we found that to be comprehensive, we also have to consider the reuse-based attacks. In both cases, there are the conventional access-based attacks and timing-based attacks.

In addition, we considered also side-channel attacks on the Instruction cache as well as on the Data cache. (See Table 1 in Section 3).

In some caches, e.g., for the I-cache, we had to create a new attack for a reuse-based, access-based cache side-channel attack on the I-cache (see Table 2 and section 3.) We also created targeted attacks for Newcache, based on the attacker modifying the Prime and Probe attack or the Evict and Time attack on the Data cache (see section 3.3.1).

Section 3 will describe the attacks in detail, and the results we get from the security testing of Newcache and conventional set-associative caches with each attack.

3.5 Smartphone and Cloud Computing Benchmark Suites for Performance Evaluation

The early work on Newcache used only the SPEC benchmark suite. These are compute benchmarks and not very representative of today's computing landscape which comprises mobile smartphones and cloud computing. Hence, we need to re-evaluate the system performance of Newcache with respect to a representative suite of smartphone benchmarks and cloud computing benchmarks.

We were able to find such a suite of smartphone benchmarks from various sourcec. These are described in section 4.2, as well as the performance results we obtained.

For cloud computing, we had to assemble our own representative cloud computing benchmarks, since available "scale-out" benchmarks were not suitable for general-purpose cloud computing. We describe the cloud computing benchmarks we selected as representative in section 4.3, as well as the performance results we obtained.

3.6 Testchip to evaluate physical cache performance and power characteristics

To evaluate the security and performance of a secure cache within a system, we need to simulate the whole system using gem5. Simulation is necessary because currently there is no processor with our secure caches implemented in it. Hence the hardware has to be simulated, and gem5 allows us to run the operating system (O.S.) and the victim and attacker software on top of the O.S. , which runs on top of the simulated hardware system (x86 or ARM processor, caches, memory, etc.). However, we would also like to evaluate the performance characteristics of the hardware cache itself, like how long it takes to read or write data in the cache (called the cache access time or latency), and how much power it consumes to do a read or a write. The chip area taken by each cache can also be measured. For the most accurate measurements, we implemented a testchip of a 32 Kbyte Newcache and a conventional 8-way set-associative cache on the same chip. The same design team did both caches, with equivalent optimization

for both latency and power for both caches. We also designed Built-In Self Test (BIST) in the test-chip to test the cache access and power at speed.

The testchip design, fabrication, testing and results are described in section 5.

Hence, instead of a single section for “Results and Discussion”, we have three sections:

- section 3 for Security Evaluation (including the description of the attack suite we developed),
- section 4 for Performance Evaluation (including the Smartphone benchmark suites and the cloud Computing benchmark suite we developed), and
- section 5 for Testchip design and evaluation.

4. Security Evaluation

4.1 Threat Model and overview of cache side channel attacks

The attacker is an unprivileged user process running on the same machine as the victim. The victim computes on some secret information and the attacker tries to retrieve the secret information through cache side channel attacks. The attacker can manipulate the cache state and take timing measurements. Due to the timing difference between cache hits and misses, the timing measurements enable the attacker to infer the memory addresses accessed by the victim process. If the memory addresses depend on the secret information, the attacker can further deduce the secret.

We focus on the attacks against the L1 caches -- since they are closest to the processor, attacks against them are the fastest and most dangerous attacks. We assume the attacker can share the L1 cache with the victim through simultaneous multi-threading (SMT) or preemptive scheduling.

Cache attacks can target both the L1 D-cache and the L1 I-cache. The majority of attacks are D-cache attacks. They rely on *secret-dependent memory indexing*, which are used in many software cryptographic implementations for performance optimization. For example, Figure 14 shows a code snippet in AES (Advanced Encryption Standard), where each data item is 32 bits. For each round of AES encryption, AES uses the XOR of the input **in** and the round key **rk** to index the AES tables (**Te0,...,Te3**). If the attacker can get the index to the lookup table, he can derive the secret key.

```
s0 = GETU32(in )      ^ rk[0];
s1 = GETU32(in +4)    ^ rk[1];
s2 = GETU32(in + 8)   ^ rk[2];
s3 = GETU32(in + 12) ^ rk[3];
t0 = Te0[s0>>24] ^ Te1[(s1>>16) & 0xff] ^Te2[(s2>>8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[4];
```

Figure 14 Secret-dependent table lookup

```
if (secret==1) then
    code block 1;
else
    code block 2;
end
```

Figure 15 Secret-dependent instruction paths

I-cache attacks are fundamentally different from D-cache attacks. They exploit the *secret-dependent instruction paths* of program execution. Figure 15 illustrates an example code segment with secret-dependent instruction paths. If the attacker can distinguish which code path is executed, he can learn the secret. Secret-dependent instruction paths are not specific to cryptography, but can also be found in non-cryptographic applications that process secrets like passwords and credit card information.

New classification of cache attack types

In order to manipulate the cache state, the attacker exploits two fundamental cache properties: *where* and *when* to place data in the cache. Modern processors use set-associative (SA) caches where data is always placed in one of the cache slots in the cache set selected by the index bits in the memory address. If the attacker contends for the same cache set as the victim, this results in a conflict miss, which enables the attacker to infer the memory address used by the victim. We denote this class of attacks as *contention-based attacks*. Hence, "where" refers to which cache set had contention between victim and attacker.

In addition, a processor always fetches data from the next cache level (or memory) into the L1 cache when the data is needed and a cache miss occurs, known as a *demand fetch*. If a previously accessed data is cached, the reuse of the same data by a later memory access will result in a cache hit, which enables the attacker to know which address was previously accessed. We denote this class of attacks as *reuse-based attacks*. Hence, "when" refers to whether data not found in the cache is fetched on demand (i.e., when requested by the processor and not in the cache), or not (e.g., randomly prefetched or bypassed to the processor without placing it in the cache.)

Table 1 Classification of cache side channel attacks

		D-cache	***I-cache***
Contention based attacks	***Access-driven***	***Prime-Probe***	***Prime-Probe***
	Timing-driven	***Evict-Time***	***N.A.***
Reuse based attacks	***Access-driven***	***Flush-Reload***	***Flush-Reload***
	Timing-driven	***Cache-Collision***	***N.A.***

N.A. = not applicable

4.2 Attack benchmark suite

In order to evaluate the security of Newcache, we use an empirical analysis framework which consists of a suite of attack benchmarks. Our attack benchmarks consider all side-channel attacks on L1 caches, listed in 8 classes in Table 1, with an example attack in each class, where applicable.

Both contention based attacks and reuse based attacks can be further classified as access-driven attacks and timing-driven attacks, which is the conventional classification based on *what* to measure by the attacker. In access-driven attacks, the attacker measures how the victim process' memory accesses impact the cache state and hence his own memory accesses. In timing-driven attacks, the attacker measures how his memory accesses impact the total execution time of the victim's operation. (Trace-driven attacks are a special case of access-driven attacks.) To the best of our knowledge, Table 1 summarize all the possible ways to perform cache attacks.

Since the purpose of the attack benchmarks is to evaluate the security of different cache designs, we target well-known cryptographic algorithms with vulnerabilities.

Table 2 summarizes the attack benchmarks, and we describe each attack benchmark in detail in the following sections 4.3 and 4.4.

Table 2 Summary of attack benchmarks and attack results on Newcache

Type of attack	Type of cache	Attack benchmark	Result for Newcache
Contention based attacks	D-cache	Prime-Probe attack against first round AES	attack defeated
		Evict-Time attack against first round AES	attack defeated
	I-cache	Prime-Probe attack against modular exponentiation	attack defeated
Reuse based attacks	I-cache	Flush-Reload attack against modular exponentiation	attack defeated
	D-cache	Cache collision attack against final round AES	attack mitigated*

*One order of magnitude harder to attack than set-associative cache

A typical cache side channel attack consists of an online phase and an offline phase. In the online phase, the attacker collects measurement samples during the victim's encryption. In the offline phase, the attacker processes the measurement samples to recover secret keys. Only the online phase depends on the cache organization and the offline phase is uniquely determined by the attack strategy and the crypto algorithm. Therefore, we focus on the online phase of attacks. We use statistical analysis to quantify the measurement, without having to fully recover the key. Our attack benchmarks can be run on gem5, using the same behavioral model as for the performance evaluation.

Summary of results: Our takeaway is that (see **Error! Reference source not found.**) with proper allocation of RMT_ID, Newcache can defeat all the L1 cache attacks except the cache collision attack, since that attack only exploits reuse of data within a process. Nevertheless, Newcache can still make cache collision attacks one order of magnitude harder to succeed than on conventional set-associative caches.

4.3 Security evaluation for contention based attacks

4.3.1 Newcache used as the L1 D-cache

4.3.1.1 Prime-Probe attacks

The Prime-Probe attack is the most widely used attack technique. It works as follows:

Prime: An attacker *A* fills the cache (or just one or more cache sets) with its own data.

Idle: *A* waits for a pre-specified Prime-Probe interval while the victim process *V* gains control of the processor and utilizes the cache.

Probe: *A* gains control of the processor again, and measures the time to access the same cache sets to learn *V*'s cache activity.

If *V* uses some cache sets during the Prime-Probe interval, some of *A*'s cache lines in these cache sets will be evicted, which causes a longer load time for those cache sets during *A*'s Probe phase. It is

straightforward to do "prime" and "probe" for the entire L1 D-cache – the attacker only needs to load data from an array of the same size as the data cache.

Prime-Probe attacks against AES: Our attack benchmark is a concrete Prime-Probe attack against the first round encryption of AES-128. AES-128 uses a 16-byte key and the message is also chopped into 16-byte blocks. Each encryption contains 10 rounds of operations, each round indexes the AES tables using results obtained from the previous round. Each AES table has 256 entries with each entry being 4 bytes. In the first round, the table lookup indices $x_i^{(0)}$ are related with the plaintext byte p_i and key byte k_i as $x_i^{(0)} = p_i \oplus k_i$ for all $i = 0, ..., 15$.

To have a clean and synchronized environment for the attacker (favoring the attacker), the attack code uses the OpenSSL library calls as black-box functions. The attack code primes the whole D-cache by reading data from its own array before it calls the AES encryption library call to encrypt one block of random plaintext. Then the attack code probes the array and measures the time to access each cache set. For each block encryption, we record the plaintext and a vector of timing measurements, with each element of the vector being the time to access that cache set. After collecting enough measurements (2^{20}), we perform statistical analysis for the measurements. For each plaintext byte p_i, we can divide the samples into 256 packs according to the value of p_i and calculate the average probe time for each cache set in each pack. For a given cache set which contains AES table entry x^*, the probe time is expected to be higher than others when $p_i^* = x^* \oplus k_i^*$. Figure 16(a) shows the heatmap for the 8-way SA cache, where each point represents the probe time; the longer the probe time, the lighter the color. Since we use an all-zero key (for simplicity without losing generality), we can see a clear bright straight line.

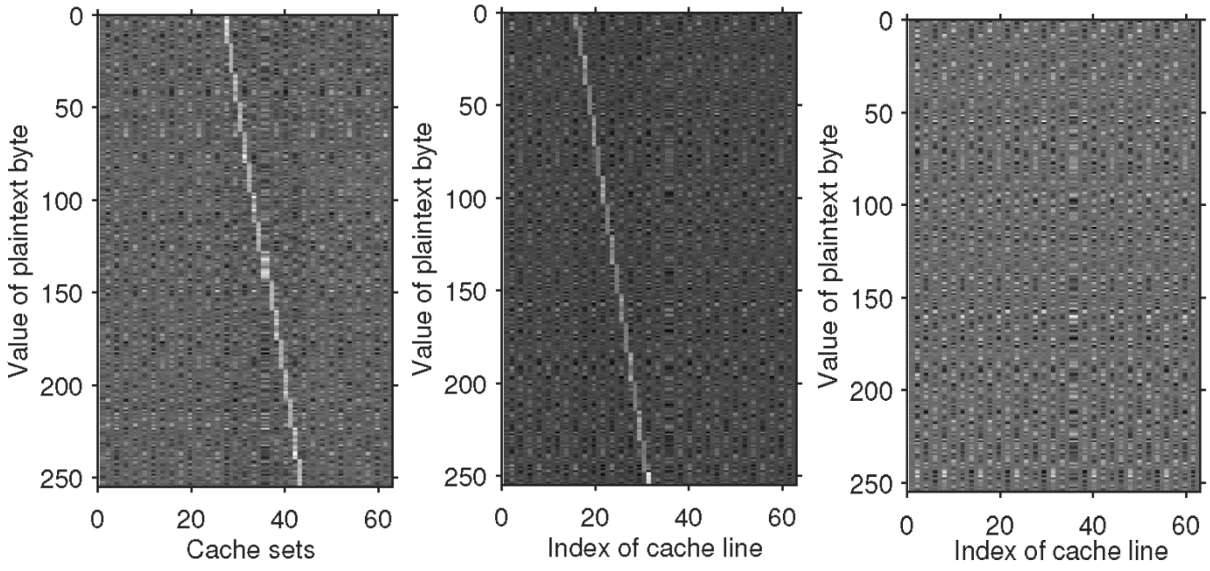

Figure 16 D-cache heatmap for (a) 8-way SA cache, (b) Newcache without RMT_ID and P-bit, (c) Newcache with RMT_ID and P-bit

Table 3 Pearson correlation coefficient for Prime-Probe attack

8-way SA	Newcache (no *RMT_ID and P-bit*)	Newcache (*with RMT_ID and P-bit*)
0.4209	0.4226	0.0032

Targeted attack against Newcache: We made two changes to adapt the attack to target Newcache. First, instead of measuring the time to probe a cache set, we measure the time to probe a single cache line. Second, a smarter attacker may exploit the fixed part of the memory-to-cache mapping. The attacker can allocate an array of the same size as the ephemeral LDM cache and prime the cache using a sub-array (same size as the AES tables) that conflicts in the LDM cache with the AES tables. Since Newcache is a PIPT (physically-indexed and physically-tagged) cache, the need to locate the conflicting sub-array increases the difficulty of the attack. However, the attacker can leverage "huge page" support in modern microprocessors to solve the address uncertainty problem.

Figure 16(b) shows the case for Newcache without RMT_ID and P-bit. It indicates that without RMT_ID and P-bit, the potential fixed conflicts can still be exploited by the targeted attack. However, if the AES tables are given a different P-bit value (P=1) from the rest of the program which is given P=0 (see Figure 16(c)), Newcache can defeat the targeted attack.

To quantitatively measure the vulnerability of a design to the Prime-Probe attacks, we calculate the Pearson correlation coefficient between the attacker's measurements (the D-cache heatmap) with the ground truth (the expected D-cache pattern, i.e., noiseless bright straight line with points corresponding to the straight line in white, with other points in black). The results are summarized in

Table 3. The closer the coefficient is to zero, the less accurately the attacker is observing patterns. When the coefficient is very near zero, the attacker is essentially observing noise. The quantitative result is in accordance with the visual heatmap.

4.3.1.2 Evict-Time attacks

The Evict-Time attack is a timing-driven attack and is a generalization of the well-known Bernstein's attack [30]. The attacker *A* works as follows:

Evict: *A* fills one specific cache set with his own data.
Time: *A* triggers the victim process to perform the security-critical operation and measures the victim's total execution time.
If the victim accesses the evicted cache set, his execution time tends to be statistically higher than when he does not access the evicted cache set (but may access other data in the lookup table), due to the victim having a cache miss. For a SA cache, it is straightforward to evict one specific cache line.

Evict-Time attack against AES: Similar to the Prime-Probe attack, our attack benchmark is a concrete Evict-Time attack against the first round encryption of AES-128. Before each library call for the AES encryption, the attack code evicts one cache line that contains AES table entries out of the D-cache. Each AES encryption encrypts one block of random plaintext. For each block encryption, we record the plaintext and corresponding encryption time. After collecting enough measurements (2^{20}), we perform statistical analysis for the measurements. For each plaintext byte p_i, we divide the samples into 256 packs according to the value of p_i and calculate the average encryption time for each pack. If for every encryption, the attacker constantly evicts the same AES table entry x^*, we would expect the average encryption time to be significantly higher when $p_i = x^* \oplus k_i^*$. Figure 17(a) shows the results for an 8-way SA cache.

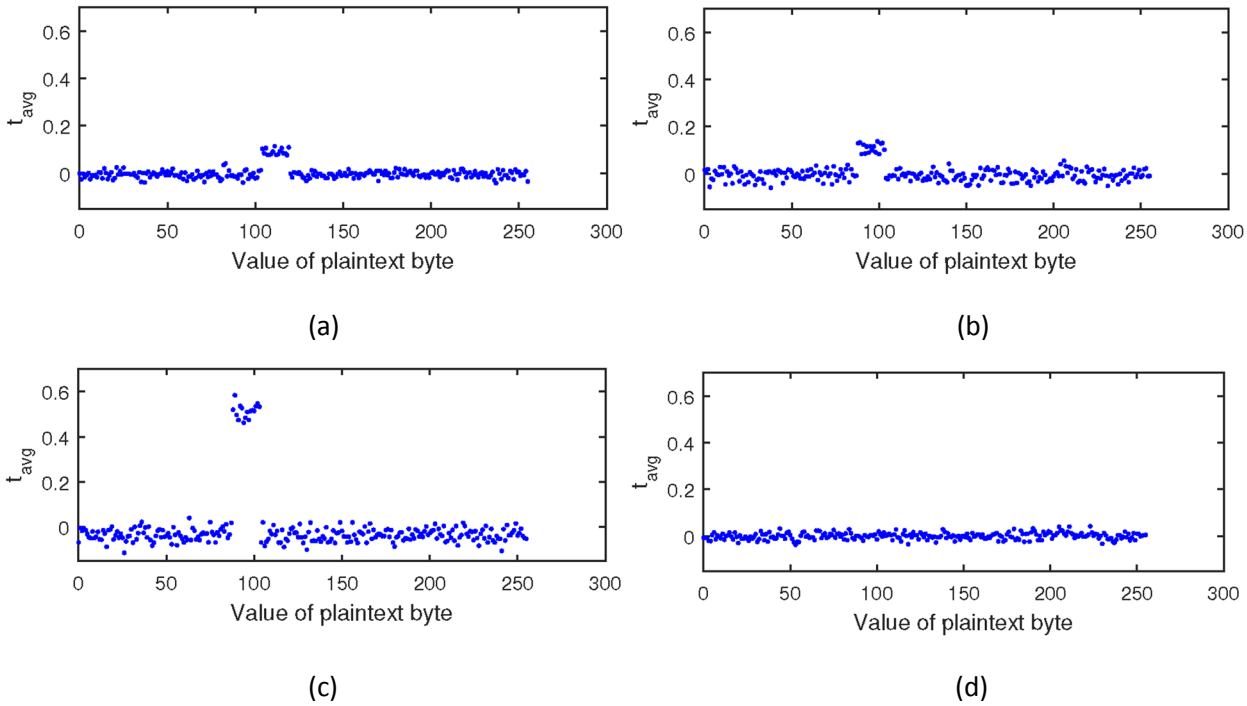

Figure 17 Evict-Time attack results for (a) 8-way SA cache, (b) Newcache without RMT_ID and P-bit, (c) original Newcache, (d) Newcache with RMT_ID and P-bit

Targeted attack against Newcache: An appropriate measurement for the Evict-Time attack requires the attacker to evict one specific cache line. Similar to the Prime-Probe attack, the attacker can allocate an array of the same size as the ephemeral LDM cache and find the memory line from the array that conflicts in the LDM cache with a specific cache line containing AES table entries. Similar to our findings for the Prime-Probe attack, the potential fixed conflicts in the LDM cache can still be exploited to perform the Evict-Time attack (see Figure 17(b)), but the attack is defeated if security-critical data is given a different trust domain and therefore, also a different mapping table (see Figure 17(d)). As for the Prime-Probe attacks, we can also quantitatively measure the vulnerability of a design to the Evict-Time attacks by correlating the attacker's measurement with the expected ground truth values ("1" for points with significantly higher average encryption time, "0" for others), as shown in

Table 4.

Table 4 Pearson correlation coefficient for Evict-Time attack

8-way SA	Newcache (no *RMT_ID nor P-bit*)	Original Newcache	Newcache (*with RMT_ID and P-bit*)
0.8068	0.7472	0.9187	0.0217

It is worth noting that with this attack benchmark, we discovered a very subtle design issue in the original Newcache, which also resulted in simplifying its replacement algorithm. In the original Newcache design, Newcache associates a protection bit (P-bit) with each cache line in the tag array, to avoid the eviction of a security-critical cache line by the wrapper code in the same trust domain (as in Bernstein's attack). For a tag miss, a memory line can replace another cache line only if neither of them have the P-bit set. Otherwise, the memory line will not be brought into the cache and a randomly selected cache line is evicted to give the attacker an illusion of cache eviction. It turns out that this original design even makes the Evict-Time attack easier (see Figure 17(c) and

Table 4). This is because once the attacker occupies the conflicting cache set, access to the protected conflicting AES table entries will always result in a cache miss. And that cache set will remain occupied by the attacker (except if randomly evicted) since the protected AES lines cannot replace the attacker's cache line.

4.3.2 Newcache used as I-cache

4.3.2.1 Prime-Probe attacks

A Prime-Probe attack for the I-cache is not as easy to achieve as that for the D-cache. First, priming or probing a specific set in the I-cache is trickier than just loading data from an attacker's array into the D-cache. Second, derivation of victim cache usage requires distinguishing cache footprints left in the I-cache by different secret-dependent code paths, rather than just timing how long it takes to probe a D-cache set. An I-cache footprint consists of all the cache sets that the memory lines containing the code block (including all the functions called by it) maps to. The Prime-Probe technique enables the attacker to determine which cache sets have been accessed by the victim, and hence its cache footprint.

Figure 18 show how to prime or probe exactly one set in the I-cache. This shows a cache with a size of S sets by W ways, with each cache block being B bytes. Memory addresses that are $B{\cdot}S$ bytes apart will be mapped to the same cache set. To prime one cache set, the attacker needs to allocate W cache blocks (the shaded blocks in Figure 18), where each cache block is $B{\cdot}S$ bytes away from the previous block. Inside each cache block is the instruction **jmp** $B{\cdot}S$, a relative jump instruction that jumps to an instruction that is $B{\cdot}S$ bytes away, which is the next block, except for the last block, which has a return (**ret**) instruction. When priming the cache set, the attacker's main program jumps 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 S-1 sets. The attacker can allocate a contiguous memory chunk that has the same size as the instruction cache to prime the whole I-cache. Probing is just like Priming, except that the **rdtsc** instruction, placed at the beginning and end of accessing each I-cache set, is used to measure the time to probe each I-cache set.

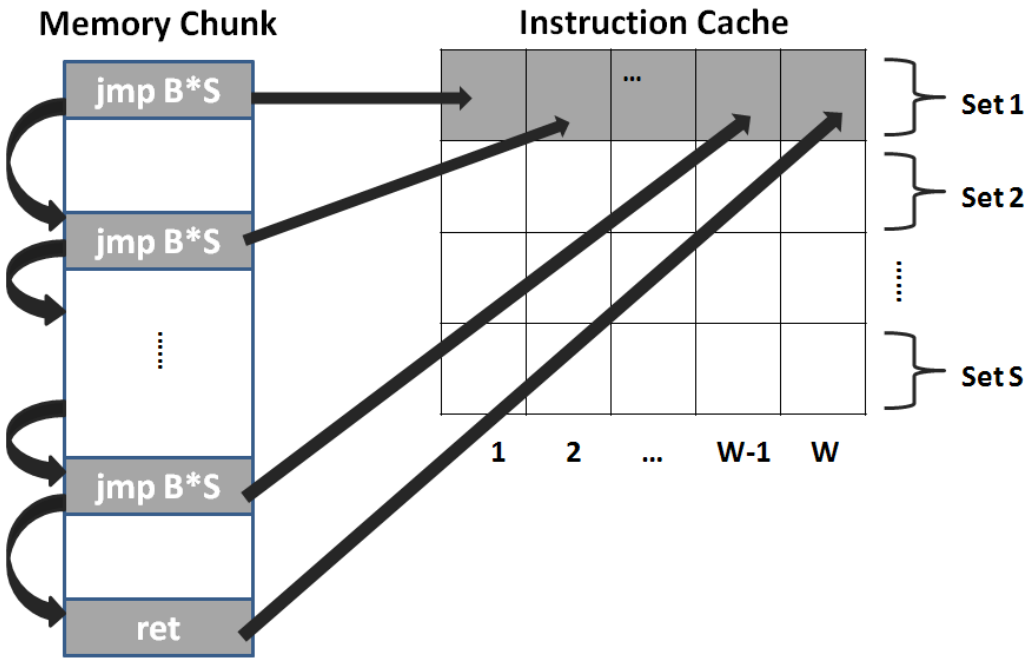

Figure 18 Instruction cache set Prime and Probe

Prime-Probe attack against modular exponentiation: Our attack benchmark consists of a concrete Prime-Probe attack against the modular exponentiation algorithm in libgcrypt v1.5.3 [16]. Modular exponentiation is the main computation in many public-key algorithms, like RSA, ElGamal, etc. libgcrypt

is a widely used cryptographic library, and also used in the Cloud Computing I-cache attack in [33]. It implements a square-and-multiply algorithm as shown in Figure 19. We label S, R, M to stand for calls to the functions, Square, ModReduce and Mult, respectively, inside the algorithm. The sequence of function calls in one execution of modular exponentiation with square-and-multiply can leak information about the exponent expo, which is the private key of many public-key ciphers (ElGamal, RSA etc.). Note that the most significant exponent bit en is always 1, so the information leaked is from e_{n-1} to e_1. For example, the sequence (SRMR)(SR) corresponds to expo = $110_2 = 6_{10}$. SRMR leaks information $e_2 = 1$, and SR gives $e_1 = 0$.

We first show how the attack can be performed on a real machine. The attacker and victim processes are run alternately via normal preemptive scheduling. Since the processor we use has a 32kB, 8-way SA I-cache with 64B line size, we have 64 sets for the I-cache. We use a method similar to [35] to trick the Linux complete fair scheduler (CFS) to get an attacker Prime-Probe interval of about 2.5µs, so that basically one M or S operation from the victim process can execute during this interval. The victim just performs modular exponentiation depicted in Figure 19 with an exponent unknown to the attacker. Each time the attacker performs a Prime-Probe, he gets a footprint, which is a vector of timings, one timing per cache set. For our test setup, this is a 64-element vector. After getting the trace of the footprints online, the attacker can do off-line analysis to classify each footprint as an S, M or R operation, to infer the exponent, which are the private key bits. For our purpose of evaluating the vulnerability of an I-cache to this attack, we label each S, R or M operation in Figure 19.

```
procedure SQUAREMUL(base, expo, mod)
    Let e_n,...,e_1 be the bits of expo, and e_n = 1
    for i ← n-1, 1 do
        y ← SQUARE(y, base)                    (S)
        y ← MODREDUCE(y, mod, base)            (R)
        if e_i = 1 then
          y ← MULT(y, mod, base)               (M)
          y ← MODREDUCE(y, mod, base)          (R)
        end
    end
```

Figure 19 Square-and Multiply algorithm in libgcrypt

Figure 20 shows 6000 of the attacker's Prime-Probe trials. To give a figure with visually clear patterns, we choose 2000 footprints in the trace with an S label, 2000 footprints with an M label and 2000 footprints with an R label, and pile them together based on their labels. So trials 1-2000 are all S operations; 2001-4000 are all M operations and 4001-6000 are all R operations, and we can see different patterns for these different operations. To quantitatively describe how well an operation can be correctly classified based on its cache footprint, we use libsvm [34] to train a multi-label SVM (Support Vector Machine) classifier. We choose its default SVM type C-SVC [36][37] and a linear kernel function. An SVM is a supervised machine learning tool that, when fed with enough training samples with different labels, builds a model that can categorize new testing samples into one of the labels. For this case, a label is one of S, R and M; a sample is a cache footprint with feature dimension of 64, and the i^{th} feature ($1 \le i \le 64$) is the probe time for cache set i. Thus, before an SVM can classify new instances, we need to train it with a set of sample-

label pairs. Similar to [33], we set the victim's exponent to be all 1's, which gives a sequence of SRMRSRMR... operations. We collect a trace of the attacker's Prime-Probe footprints, and we use the hooking function trick from [33] to label those footprints. In the experiment, we use 40,000 of these footprints to train an SVM classifier, and another 12,000 samples to do the testing.

Table 5 shows the classification matrix for the attack. The **diagonal** entries are the correct classifications, and the classification accuracy is 90.2%. The high classification accuracy indicates a high key-bits recovery by subsequent off-line stages of the attack [33]. The value in parenthesis is the percentage of trials having the value in that cell of the classification matrix.

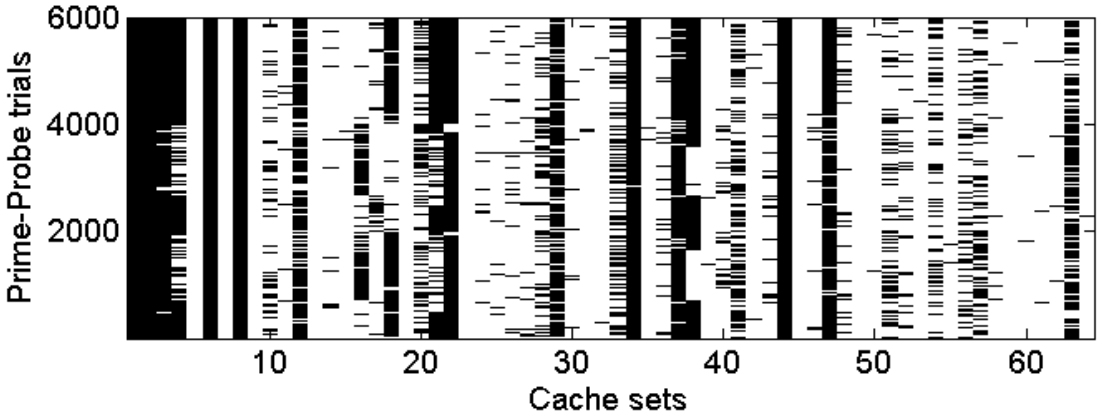

Figure 20 Cache footprint for S,M, and R operations in a real machine

Table 5 SVM classification matrix for real attacks on real machines

	Classification			Classification Accuracy
	Square	Multiply	Reduce	
Op: Square	3479 (0.87)	189 (0.05)	332 (0.08)	90.2%
Op: Multiply	375 (0.09)	3587 (0.90)	38(0.01)	
Op: Reduce	92(0.02)	148 (0.04)	3760 (0.94)	

Attack results for Newcache: Since gem5 does not support a high precision timer, it is very hard to achieve fine-grained preemption as in a real machine. Hence we emulate the Prime-Probe attacks by hacking the simulator to execute dummy memory accesses for the probe operations at some fixed time interval. The time interval is chosen so that the victim can only run for a very short time interval, similar to the real attack. As for the real attacks we did on real machines, we set the victim process to repeatedly perform modular exponentiations.

We first evaluate the case for Newcache without using the RMT_ID and P-bit. We varied the number of extra index bits k. We collect the Prime-Probe footprints, and input them as training and testing samples

to an SVM classifier. Figure 21(a)-(c) gives the visual patterns of Prime-Probe trials for these three cache configurations with k = 0, 2 and 4 bits, and Table 6 gives the classification matrix.

When $k = 0$, it is essentially a direct-mapped (DM) cache. This is why a clear pattern following the S, R,M operations can be seen in Figure 21(a). At the steady state, the size of the LDM cache is the same as the physical cache size and hence every entry in the LDM cache will have a mapping to a physical cache line and the mapping cannot be updated any more. Therefore, all the following memory accesses will follow the fixed, linear mapping. The amount of fixed, linear mapping decreases as k increases. When k is increased to 4, no clear pattern for the cache footprints can be seen (Figure 21(c)). In Table 6, the SVM classification accuracy decreases from 98.7% to 47.2%, as k increases from 0 to 4. According to [33], it would be extremely difficult for further offline analysis to extract key bits correctly with an SVM accuracy of 47.2%. Comparing Figure 7(b) and Table 6, we find that when $k = 2$, a lot of noise is introduced into the cache patterns. But if the attacker's prime-probe code happens to get even a few index conflicts with the victim operations' codes, the fixed linear mapping will give him a very high classification accuracy. By increasing the LDM cache size, we cannot eliminate the fixed mapping entirely, but we can reduce the probability of the index conflicts. This indicates that although it is not obvious as for the D-cache, the fixed mapping part can still be exploited in the I-cache attack. However, if the victim and the attacker are given different disjoint trust domains and mapping tables, no fixed pattern can be found even when $k = 0$ (see Figure 21(d)).

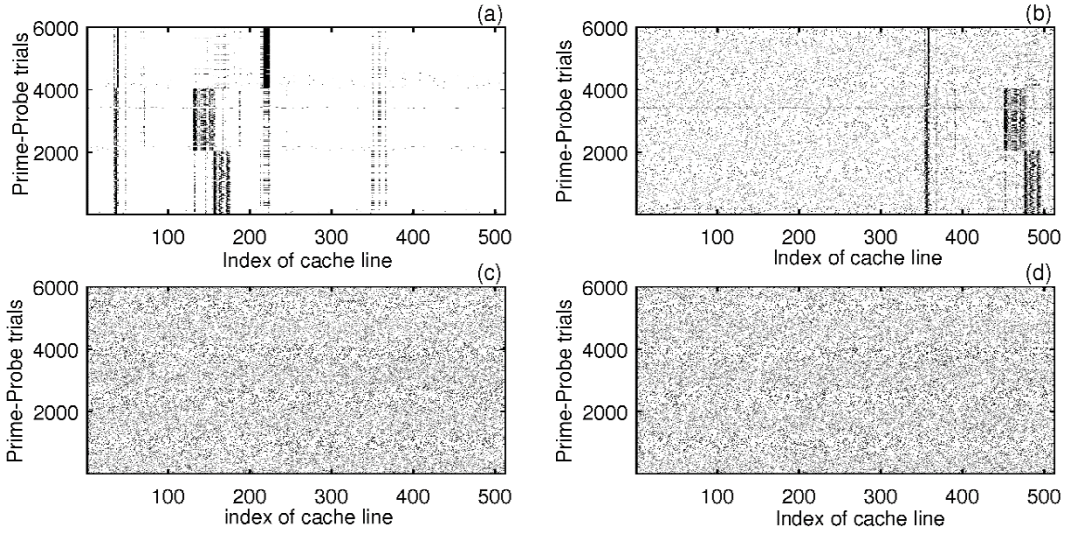

Figure 21 Cache footprint patterns of modular exponentiation on gem5 simulator, for Newcache without RMT_ID and P-bit and with different number of extra index bits for (a) k=0, (b) k=2, (c) k=4. In (d), Newcache with RMT_ID and P-bit, and k=0.

Table 6 Classification matrix for Newcache with different number of extra index bits

	Classification (k=0, without RMT_ID and P-bit)			**Classification (k=2, without RMT_ID and P-bit)**		
	Square	Multiply	Reduce	Square	Multiply	Reduce
Square	3978 (0.99)	0 (0.00)	22 (0.01)	3840	5 (0.00)	155 (0.04)

Multiply	1 (0.00)	3983 (1.00)	17 (0.00)	5 (0.00)	3850 (0.96)	145 (0.04)
Reduce	10 (0.00)	107 (0.03)	3883 (0.97)	62 (0.02)	61 (0.02)	3877 (0.97)
Accuracy	98.7%			96.4%		
	Classification (k=4, without RMT_ID and P-bit)			**Classification (k=0, with RMT_ID and P-bit)**		
	Square	Multiply	Reduce	Square	Multiply	Reduce
Square	1567 (0.39)	1535 (0.38)	898 (0.22)	1570	1545 (0.39)	885 (0.22)
Multiply	1450 (0.36)	1838 (0.46)	712 (0.18)	1484	1869 (0.47)	647 (0.16)
Reduce	828 (0.21)	908 (0.23)	2264 (0.57)	799	874 (0.21)	2327 (0.58)
Accuracy	47.2%			48.1%		

4.3.2.2 Evict-Time attacks

Although the Evict-Time attack has been demonstrated on the D-cache, it has never been done on the I-cache. Our analysis below shows that this attack technique is, in fact, not useful for the I-cache. Consider the code segment in Figure 15. Ideally, to apply the Evict-Time technique to the I-cache, the attacker can evict one specific cache set in one of the code paths. However, (1) the two code paths may have different lengths, and hence different execution times, so the attacker can already infer the secret from the total execution time, without needing the Evict-Time attack. Furthermore, (2) Evict-Time attacks generally leak less information than Prime-Probe attacks for the I-cache: if the secret-dependent branch is within a loop (e.g. Square-and-Multiply exponentiation as shown in Figure 19), a Prime-Probe attack can capture each execution of the branch. But for an Evict-Time attack, the evicted line can only increase the timing of the first execution of that code path. After both code paths are cached, there will be no further timing differences due to cache misses. In summary, Evict-Time attacks are generally not a threat for the I-cache, so we do not include this attack in our attack benchmark suite.

4.4 Security evaluation for reuse based attacks

Unlike contention based attacks, reuse based attacks only exploit the fact that the reuse of previously accessed data will result in a cache hit.

4.4.1 Flush-Reload attacks

The attacker and the victim process may share some address space. In particular, many cryptographic libraries are distributed as shared libraries. If an attacker *A* wants to learn which address *addr* in the shared library is accessed by the victim *V*, he can perform the following operations:

Flush: *A* flushes the cache line(s) containing *addr* out of the cache.

Idle: *A* blocks itself for a pre-specified interval.

Reload: *A* measures the time to reload *addr*.

Then if *V* accesses *addr* during the interval between *A*'s Flush and Reload, *A* will get a significantly lower reload time, since it will hit in the cache.

Flush-Reload attacks can be more serious threats for the I-cache since cryptographic libraries are usually distributed as shared libraries, while sharing of security-sensitive data is rare and can be easily disabled by not declaring the security-sensitive data as read-only. Therefore, we only include Flush-Reload attacks against the I-cache in our attack benchmark suite.

We first show how the Flush-Reload attacks on the I-cache can be constructed, which is non-trivial. Consider the code segment in Figure 15. To determine which code path is executed by the victim process *V*, the attacker *A* can Flush, Idle, then execute (Reload) a code segment C1 in code block 1. Then he can repeat this, but execute (Reload) a code segment C2 in code block 2 instead. However, choosing a code segment in each code path is tricky. First, unlike reloading the data, arbitrarily jumping into a library function would likely crash the process. Second, the execution must be able to return back to the attacker's own code immediately after executing the code segment in the shared library. Our solution is to call into the return (**ret**) instruction of a function called in the code segment, as shown in Figure 22. We assume that the address to probe is stored in the register rcx. We use two **rdtsc** instructions before and after the probe to measure the reload time. The **rdtsc** instruction will load the time in the Time Stamp Counter to registers eax (lower 32 bits) and edx (higher 32 bits), respectively. We only use the lower 32 bits time and save it to register esi before executing the next **rtdsc** instruction. The **lfence** instruction is a memory fence for serialization. This ensures minimum side effects for the sampling, and transfers control back to the attacker immediately after executing the **ret** instruction.

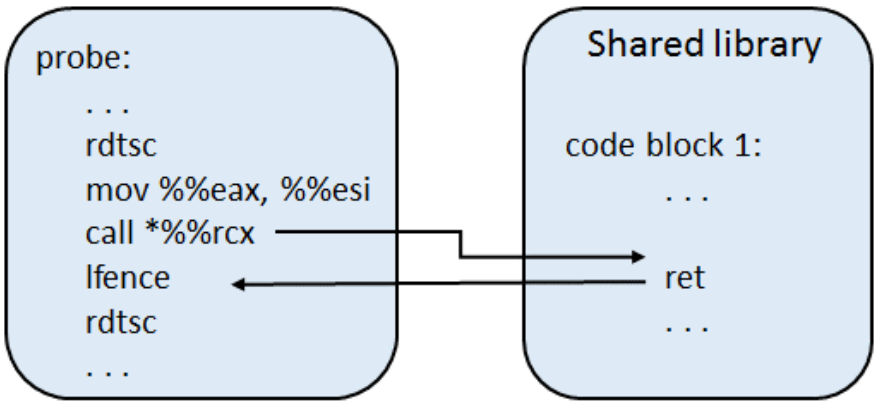

Figure 22 Code segment for sampling the execution of a code block in the shared library.

One complication is that use of address space layout randomization (ASLR) in the memory makes it hard to obtain the exact virtual address of the **ret** instruction in the shared library at runtime. One solution is to use some call back function (e.g., Linux *dl_iterate_phdr*) to obtain the base address of the shared library at run time and then add the file offset of the **ret** instruction in the executable file to the base address.

Flush-Reload attacks against modular exponentiation: Our Flush-Reload attack benchmark also targets the modular exponentiation in libgcrypt v1.5.3. We notice that the functions SQUARE() and MULT() are implemented by calling some sub-routines recursively, so the ret instruction of the sub-routines will be executed several times for one SQUARE or MULT operation. Similarly, the ret instruction of the MODREDUCE() function is in a loop and is also executed several times within one operation. So, the ret instruction in such sub-routines is a good choice for sampling the code path, since it will be executed frequently. We first show how the attack can be performed on a real machine. The attacker and victim processes are run alternately, via preemptive scheduling, and the Flush-Reload interval is 1µs. The flush of a cache line is done using the clflush instruction which can flush a certain linear address out of all caches in the cache hierarchy. Flush can also be done by running some dummy instructions by the attacker to evict the victim's cache lines. The key bits are set to be a repeated pattern (one '1' followed by seven '0's) The result is shown in Figure 23.We use a threshold of 200 cycles to distinguish lower or higher reload times, due mainly to I-cache hits or misses, respectively. Since the probe interval is much smaller than the time taken for each operation, we may get multiple measurements for each operation. The lower reload times of the three operations (below the threshold line) clearly shows the key bits, with "SRMR" indicating a "1" and "SR" a "0" bit. Compared with the Prime-Probe attack, the Flush-Reload attack is almost noiseless. Also, we note that the ret instruction is not the only instruction that can be used to distinguish a code path - any instruction can be used. However, the execution has to return to the main attack code, and **ret** also effectively serves this purpose.

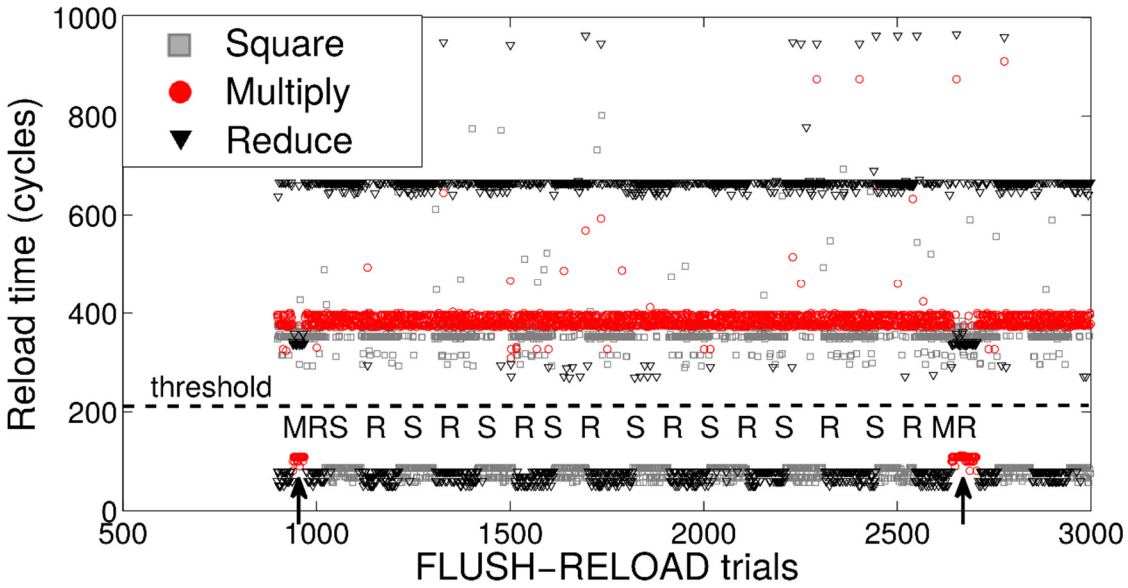

Figure 23 Measurement result for probing instruction cache using Flush-Reload.
The arrows at the bottom point to the multiply operations, which identify a "1" in the key.

Attack results for Newcache: Since gem5 does not support a high precision timer, we can only achieve a Flush-Reload interval of 100µs, which is roughly one S, R or M operation between two measurements for the modular exponentiation example. Therefore we only consider the distribution of the reload time (for S and M operations). We use the same key bits pattern as for the attacks on the real machine

("10000000..." repeated). If the Flush-Reload attack can succeed, two peaks are expected in the distribution of the reload time. One peak represents the case when the victim process does not perform the probed operation during the Flush-Reload interval (causing a cache miss and a relatively longer reload time). The other represents the case where the victim process performs the probed operation during the interval (causing a cache hit and a relatively shorter reload time). This is clearly shown in Figure 24(a) for the 8-way SA cache. Without loss of generality, we consider Newcache with $k = 2$. We find that without the protection of RMT_ID and P-bit, Newcache cannot defeat the Flush-Reload attacks (see Figure 24(b)). This is because it cannot prevent the shared instruction cache lines from being reused across different processes. However, if we give the attacker and victim process a different trust domain and mapping table, the Flush-Reload attack fails on Newcache. This again indicates the importance of properly assigning different RMT_IDs to mutually untrusting trust domains: it essentially does cache line duplication and provides a private copy of the shared memory line for each trust domain (and mapping table) at run time – thus defeating reuse based attacks.

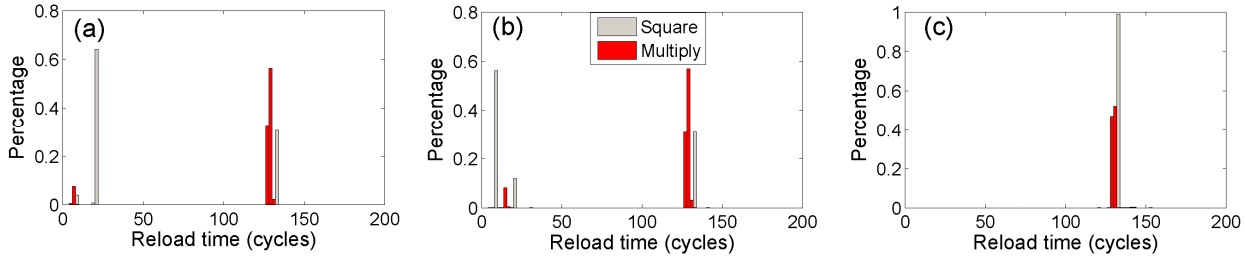

Figure 24 Reload time distribution for (a) 8-way SA cache, (b) Newcache without RMT_ID and P-bit, (c) Newcache with RMT_ID and P-bit.

4.4.2 Cache collision attacks

Flush-Reload attacks exploit the reuse of the shared code or data between the victim and the attacker. Cache collision attacks, however, exploit the reuse of the cached security-critical code/data within the victim process. In particular, if there is a cache collision (cache hit) between two security-critical memory accesses, the total execution time of the victim's operation tends to be statistically lower [32].

4.4.2.1 Newcache as D-cache

Cache collision attacks against AES: we use cache collision attacks against the final round encryption of AES as the attack benchmark. The final-round AES encryption uses a separate AES table T_4. The table lookup index x_u^{10} to T_4 is related with the final round key byte k_i^{10} and ciphertext byte ci as $T_4[x_u^{10}] \oplus k_i^{10} = c_i$. Hence, by learning two table lookups x_u^{10} and x_w^{10} collide, i.e., $x_u^{10} = x_w^{10}$, the attacker can infer that $k_i^{10} \oplus k_j^{10} = c_i \oplus c_j$. To perform the attack, the attacker first clears the cache so that each AES block encryption always starts with a clean cache. Then the attacker triggers one AES block encryption with random plaintext and measures the total encryption time. He repeats clearing the cache and getting a measurement until he gets enough samples, each sample containing the ciphertext and its block encryption time. Clearing the cache can be done by accessing an array of the same size as the cache. Since

a cache collision of two table lookups means a lower average execution time, the attacker can aggregate the time measurements according to the value of the XORed ciphertext bytes and find the minimum average encryption time. The attacker can infer 15 XORed key bytes ($k_0^{10} \oplus k_i^{10}$, i = 1,…,15), and recover all the 16 bytes of the secret key, by guessing just one key byte k_0^{10}.

Attack results for Newcache: We first consider Newcache without RMT_ID and P-bit, and varying the number of extra index bits, k. We record the number of measurements required to recover the full key, with the results shown in Table 7. When $k = 0$, Newcache requires similar number of measurements as the SA cache to recover the full key, since it is essentially a DM cache. With larger k, Newcache still succumbs to cache collision attacks, but can make the attack harder by about one-order of magnitude. This is because the dynamic randomized memory-to-cache mapping only randomizes the location that a memory line can be placed in the cache, but as long as the cache line is demand-fetched into the cache, the premise behind cache collision attacks still holds. Even if the security-critical data is protected by the P-bit, Newcache still succumbs to the cache collision attack, since the cache collisions occur between the security-critical data themselves.

Fortunately, cache collision attacks can be defeated by designing a secure cache controller [31], which can be built with Newcache. The reason why the dynamic randomized mapping makes the attack harder is because cache collision attacks require cleaning the cache before each AES block encryption, and the randomized mapping makes it harder to ensure that the cache is completely cleaned.

Table 7 Number of measurements required for a successful cache collision attack

	8-way SA	Newcache			
		k=0	k=2	k=4	k=0, with RMT_ID and P-bit
# measurements	78,000	117,000	1,203,000	1,100,000	1,077,000

4.4.2.2 Newcache as I-cache

It is not clear how a cache collision attack can be performed on the I-cache. Figure 25 Shows a possible attack construct, where a code fragment C1 belonging to one of the code blocks is repeated after the if-then-else statement. However, cache collision attacks against the I-cache have the same problem as for Evict-Time attacks on the I-cache. Even if code block 1 is executed in Figure 25, the total execution time may not be lower. Therefore, we do not consider cache collision attacks against the I-cache.

```
if (secret==1) then
        code block 1;
else
        code block 2;
end
code segment C1 in code block 1;
```

Figure 25 Cache collision of secret-dependent instruction accesses

5. Performance Evaluation

This section first describes the performance metrics used (section 5.1). Then we evaluate the system performance of Newcache used as the L1 Data cache for smartphone benchmarks (section 5.2), and then for server benchmarks (section 0). We then study how Newcache performs as an L2 cache (section 6)) and as an L1 instruction cache (section 0).

We study smartphone benchmarks as these are the dominant client platforms moving forward. We study cloud server benchmarks as cloud computing is becoming the pre-dominant computing paradigm today and moving forward.

5.1 Performance Metrics

Consider the two level non-blocking cache system as shown in Figure 4, where the miss status handler register (MSHR) is used to store information for outstanding cache misses. We use three performance metrics to characterize the performance of a SA cache or a Newcache used at different levels.

For Newcache used as L1 data cache for both smartphone and server benchmarks:

1) L1 data cache miss rate = L1 data cache misses / L1 data cache accesses.
   This metric directly indicates how Newcache impacts the L1 data cache performance.
2) L2 cache misses per kilo instructions (MPKI) = L2 cache MSHR misses / number of instructions * 1000.
   This indicates the overall misses to the main memory (or L3 cache if present).
3) Instructions per cycle (IPC) = Number of instructions committed / total time taken in cycles
   This indicates the overall performance.

For Newcache used as L2 cache, we further use two additional metrics:

4) Local L2 miss rate = L2 cache misses / data cache MSHR misses
5) Global L2 miss rate = L2 cache misses / L1 data cache accesses

The local L2 miss rate is dependent on the non-blocking features like the structure and number of MSHR entries, and the optimization features implemented. The smaller the number of MSHR misses (e.g., when more optimized), the larger the local L2 miss rate.

The global L2 miss rate is similar to MPKI, and is another representation of the overall misses to the main memory (or L3 cache if present). It depends also on the percent of instructions that are load and store instructions. It also depends indirectly on the non-blocking features and optimizations.

For Newcache used as L1 instruction cache, we further use two additional metrics:

6) Instruction cache miss rate = instruction cache misses / instruction cache accesses
7) Global L2 miss rate for instructions = L2.inst_misses / instruction cache accesses.

Note that for all these cache miss rate or MPKI metrics, smaller values are better. However, for the overall performance, in Instructions executed Per Second (IPC), larger values are better.

5.2 Smartphone Benchmark Performance

In this subsection, we first describe the smartphone benchmarks we selected and used (section 5.2.1) and the simulator configurations and evaluation method (section 5.2.2). We selected two full system benchmarks, Bbench and 0xbench, and two embedded system benchmark suites, Coremark and Mibench. We give detailed smartphone performance results (section 5.2.3) and then summarize these results (section 5.2.4).

5.2.1 Smartphone benchmark suites

Since ARM dominates the smartphone market, we focus on typical applications based on the ARM architecture. The version of the ARM processor implemented in the gem5 simulator is the ARMv7 architecture. For a thorough evaluation, we not only consider benchmark suites that are widely used for embedded processor performance benchmarking, but also include real-world Android full-system benchmarks.

5.2.1.1 Full system benchmarks

Bbench: Bbench [26] is a web-page rendering benchmark (written in JavaScript) specifically designed for gem5, which can terminate the simulator automatically when it finishes execution. It iteratively browses 9 of the most popular web-sites on the Internet, including Amazon, BBC, CNN, Craiglist, eBay, Google, MSN, Slashdot and Twitter. The web-pages consist of various types such as HTML, CSS, Javascript and images. The content for these sites are gathered using a tool (HTTrack) without the need to have network access.

0xbench benchmark suite: 0xbench [27] is an open source benchmark suite that can be found in Google Play for Android system benchmarking. It provides comprehensive benchmarks and contains 17 applications in 6 different categories, as shown in Table 8. We have successfully run 14 of the 17 applications.

Table 8 0xbench benchmark categories and application descriptions

category	application	Description	Status
Arithmetic	Linpack	Numerical linear algebra operation	✓
	Scimark2	Scientific calculation (FFT, Monte Carlo)	✓
web	Sunspider	A Javascript benchmark	Runtime error

2D rendering	CanvasRedraw	Use random color to redraw canvas repeatedly	✓
	Draw circle	A simple 2D animation program, calculate the refresh rate	✓
	DrawRect	Repeatedly add random colored, size rectangles on canvas	✓
	Draw Circle2	Repeatedly render random colored, size circles on canvas	✓
	DrawArc	Simple 2D animation	✓
	DrawText	Calculate text rendering speed	✓
	DrawImage	Calculate picture rendering speed	✓
3D rendering	GL cube	Sample program uses OpenGL ES to render a rotating Rubik's Cube	✓
	GL Teapot	Use OpenGL ES to render a rotating Utah Teapot	✓
	Lesson08	A rotating 3D cube with textured applied and alpha blending enabled	✓
	Lesson16	A rotating 3D cube with textured applied and GLFog enabled	✓
Dalvik	Garbage collection	Test the performance of the garbage collection mechanism on the Dalvik VM	✓
Native	LibMicro	Microbenchmarks that execute abundant system calls	Depends on other executables
	UnixByte	Provide indication on the performance of Unix-like system	Runtime error

5.2.1.2 Embedded processor benchmarks

CoreMark: CoreMark [28] is the most widely used processor core benchmark designed by EEMBC. It targets the evaluation of just the processor core with little I/O, and primarily on integer operations (only 16K lines of code). It contains four workloads: matrix manipulation, linked lists, state machines and CRCs (cyclic redundancy check). The four workloads are chained so that the output of each workload is the input of the next workload.

Mibench benchmark suite: Mibench [29] is the open-source version of EEMBC embedded system benchmark suite, designed by the University of Michigan. It contains 35 applications in 6 categories, all written in standard C language. Table 9 lists the 6 categories and 35 applications in Mibench with a simple description of each application. We have successfully run 16 of the applications on gem5. Most of the failures are due to incompatibility of the old source code with the available ARM cross compiler.

Table 9 Mibench description and current status

category	**application**	**Description**	**status**
automotive	Basicmath	simple mathematical calculations	✓
	bitcount	Bit count algorithm	✓
	qsort	sorts a large array of strings with quick sort alrotihm	✓
	Susan corner	recognize corners and edges in Magnetic Resonance Images	✓

	Susan edge		✓
	Susan smooth		Runtime error
telecom	Adpcm encoder	Adaptive differential code modulation for speed sample (encoding)	✓
	Adpcm decoder	Adaptive differential code modulation for speed sample (decoding)	✓
	FFT	Perform Fast Fourier Transform on an array of data	✓
	IFFT	Perform Inverse Fast Fourier Transform	Runtime error
	CRC32	32-bit cyclic redundancy check on a file	✓
	GSM encoder	Encode speech sample with TDMA/FDMA	✓
	GSM decoder	Decode speech sample with TDMA/FDMA	✓
network	dijkstra	Calculate shortest path using dijkstra algorithm	✓
	patricia	Route IP traffic from web server with route table represented by trie	✓
security	Blowfish encryption	Encrypt large text file using blowfish block cipher	Runtime error
	Blowfish description	decrypt large text file using blowfish block cipher	
	sha	Produce 160-bit message digest for a large text file	✓
	Rijndael encryption	Encrypt large text file using rijndael (AES) block cipher	✓
	Rijndael decryption	Decrypt large text file using rijndael (AES) block cipher	Runtime error
	Pgp sign	Public key encryption algorithm (digital signing)	Compile error
	PGP verify	Public key encryption algorithm (verify digital signature)	
office	String search	Search given words in phrases using a case sensitive comparison algorithm	✓
	ghostscript	Postscript language interpreter without graphical interface	Compile error
	ispell	Fast spelling checker similar to the Unix shell, but faster	
	rsynth	Text to speech synthesis program	
	sphinx	Speech decoder operating on finite-length segments of speech or utterances, one utterance at a time	
consumer	jpeg	Compress color image into jpeg format	
	tiff2bw	Converts a color TIFF image to black and white image	
	tiff2rgba	Formatted TIFF image	
	tiffdither	Dithers a black and white TIFF bitmap to reduce the resolution and size of the image at the expense of clarity	
	tiffmedian	Converts an image to a reduced color palette by taking several medians of the current color palette	
	lame	GPL'ed MP3 encoder that supports constant, average and variable bit-rate encoding	
	mad	High-quality MPEG audio decoder	
	typeset	A general typesetting tool, that has a front-end processor for HTML	

To summarize, Mibench has 35 applications in 6 categories:

- Auto/Industrial(6):Basic math, bitcount, qsort, susan (edges, corner, smooth)
- Telecom (7): CRC32, FFT, IFFT, ADPCM (enc, dec), GSM (enc, dec)
- Network (2): Dijkstra, Patricia
- Security (7): Blowfish (enc, dec), PGP (sign, verify), Rijndael (enc, dec), sha
- Office (5): Ghostscript, ispell, rsynth, sphinx, stringsearch
- Customer (8): Jpeg, lame, mad, tiff2-(bw, rgba), tiff-(dither, median), typeset.

These are very old programs, and some would not compile with the ARM cross compiler, or had runtime errors. We used Mibench since it is a free (academic) benchmark suite, modeled after EEMBC, which is a commercially maintained benchmark for embedded systems that is quite expensive.

5.2.2 Experiments Performed and Simulator Configurations

We take the single-core ARM cortex-a15 as the reference processor and the baseline configurations are summarized in Table 11. The ARM cortex-a15 processor is an out-of-order superscalar processor with two levels of cache hierarchy. We model Newcache based on the gem5 classic memory system non-blocking cache, as described in section 3.1. We evaluate the performance of Newcache as a L1 data cache, compared with a conventional set-associative L1 data cache.

Table 10 shows the 6 experiments we performed for each of the 4 smartphone benchmarks or benchmark suites. We evaluate the L1 Data cache miss rate, the L2 MPKI (Miss Per Kilo Instruction) and the overall performance in IPC (Instructions Per Second), while varying either Newcache and set-associativity parameters with a fixed cache size, or varying the cache size with fixed Newcache and set-associative cache parameters.

Table 10 Experiments for each benchmark for Smartphone Performance Evaluation

Performance Metric	Parameters varied	Parameters fixed
(i) L1 D-cache miss rate	SA cache: 2-, 4-, 8-way SA Newcache, nebit k = 3, 4, 5, 6	Cache size = 32 Kbytes
(ii) L1 D-cache miss rate	Cache size = 16KB, 32 KB, 64 KB	SA cache: 4-way SA Newcache, k=4
(iii) L2 MPKI (Misses Per Kilo Instructions)	SA cache: 2-, 4-, 8-way SA Newcache, nebit k = 3, 4, 5, 6	Cache size = 32 Kbytes
(iv) L2 MPKI	Cache size = 16KB, 32 KB, 64 KB	SA cache: 4-way SA Newcache, k=4
(v) Overall Performance, ILP (Instructions Per Cycle)	SA cache: 2-, 4-, 8-way SA Newcache, nebit k = 3, 4, 5, 6	Cache size = 32 Kbytes
(vi) Overall Performance, ILP	Cache size = 16KB, 32 KB, 64 KB	SA cache: 4-way SA Newcache, k=4

For Newcache, the parameter varied is the number of extra index bits (nebit), also called k. This is the number of extra bits in the Line Number registers (LNregs) which essentially increases the size of the ephemeral Logical Direct Mapped cache (LDM cache) without increasing the size of the actual physical cache.

The L2 MPKI shows the traffic that is generated to the next (slower) level of the cache-memory hierarchy, to see if the randomized memory-to-cache mapping of Newcache generates more traffic at the backend than set-associative caches.

Table 11 shows the baseline configurations for the Set-associative cache and for Newcache.

Table 11 Baseline Configurations for Smartphone Simulations

Single-core out-of-order ARM	**L1-I cache (private)**	**L1-D cache (private)**	**L2 cache (unified)**	**Memory**
	4-way 32 kB 1-cycle latency	4-way 32 kB 1-cycle latency	8-way 512 kB 10-cycle latency	2GB 100-cycle latency
Cache block size	64 B			
Clock frequency	1 GHz			
Evaluate the performance of Newcache as a L1 data cache, (baseline Newcache configuration has nebit=k=4 extra bits, with cache size of 32 KB), 1 cycle latency.				

For the embedded processor benchmark suites (CoreMark and Mibench), we run the benchmark in gem5 system-call emulation mode from start to completion.

For the full-system benchmarks (Bbench and 0xbench), we run the benchmark in gem5 full-system mode with Android Gingerbread distribution as the Operating System (OS). Instead of running from start to completion in timing mode, we first take a checkpoint right after booting the operating system (OS) in atomic CPU mode (without detailed timing), then we can restore from the checkpoint and collect statistics for the region of interest (ROI). Check-pointing not only avoids collecting statistics for an uninteresting region (booting OS), but also saves simulation time. Since Bbench uses gem5 pseudo instruction to terminate the simulator immediately after the execution of the benchmark is finished, it can be run to completion. However, for 0xbench, since there is no synchronization to terminate the simulator once the benchmark completes, we collect statistics for the execution of 2 billion or 4 billion instructions from the restored checkpoint.

5.2.3 Smartphone performance evaluation results

We first discuss the performance results for the benchmarks we ran in gem5 full system mode: Bbench (section 5.2.3.1) and the Oxbench benchmark suite (section 5.2.3.2).

We then discuss the Embedded processor benchmarks:

Coremark (section 0) and Mibench (section 5.2.3.4) benchmark suites.

Although we started out with the smaller embedded processor benchmarks, we found that they are less representative of smartphone performance than the full system benchmarks of Bbench and Oxbench, hence we describe those first.

For ease of reading, we start each benchmark on a new page. For each of the 4 benchmarks, the text summarizing the performance figures comes first, then the 6 figures. There are 2 figures each for the 3 performance metrics: L1 D-cache miss rate, L2 MPKI for backend traffic, and IPC for overall performance, as described in Table 10.

5.2.3.1 Bbench

L1 data cache miss rate

Figure 26 shows the L1 data cache miss rate for various associativity and nebit (number of extra bits in Newcache) values. We find that the data cache miss rate of Bbench is relatively low (on the order of 10^{-2}). There is a noticeable drop in miss rate when nebit is increased from 3 to 4.

(b) Figure 27 shows that for both 4-way SA cache and Newcache, the cache miss rate decreases as the cache size increases (as expected). The percentages show the relative increase in L1 D cache miss rate for Newcache at the same cache size. While this typically also decreases with increasing cache sizes, Bbench shows a little abnormality since a cache size of 32KB gives the smallest increase in cache miss rate between Newcache and 4-way SA cache, for L1 data cache performance.

L2 MPKI

Figure 28 shows L2 Misses per Kilo Instruction (MPKI), which is a measure of the number of requests to main memory, for every thousand instructions. As shown in Figure 28, the overall misses to the L2 cache is only slightly increased (up to 2% more for k=3 or 4) or essentially unchanged (for k=5 or 6) for Newcache with various nebit (k) values.

We also note from Figure 29 that a larger cache size may reduce Newcache's relative increase of overall cache misses to the memory; specifically, Newcache with size of 64KB incurs less L2 MPKI than the SA cache.

Overall Performance, IPC

Figure 30 shows that the overall performance, in Instructions executed Per Cycle (IPC), increases for Newcache as the nebit increases for the L1 D-cache. However, for SA caches, there is an abnormality as the IPC decreases from 2-way to 4-way L1 D-cache.

From Figure 31, we find that Newcache works best for Bbench when the L1 Data cache size is 32KB, where IPC is improved by about 0.6%. The results for IPC are slightly contrary to the results for L1 data cache miss rate and L2 MPKI: IPC is slightly improved compared with the 32KB SA cache even though the L1 data cache miss rate and L2 miss rate are increased. This may be due to the use of non-blocking caches and an out-of-order processor – both of which can hide the cache miss latencies even when the cache miss rate increases.

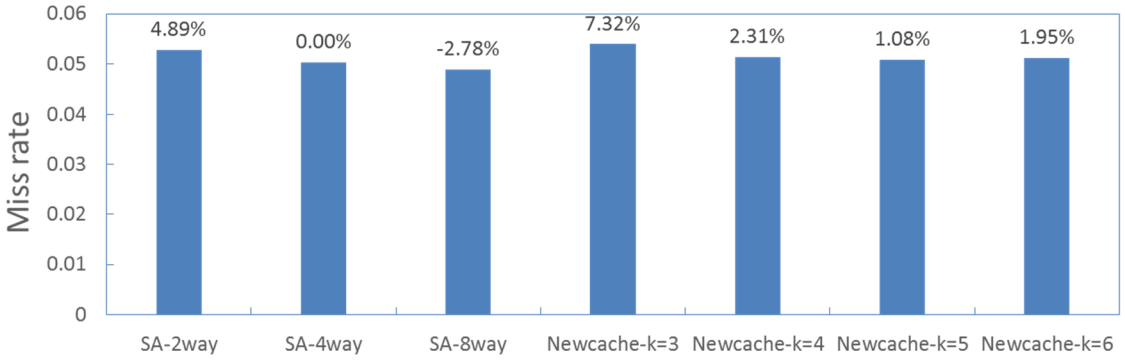

Figure 26 Bbench: L1 data cache miss rate for various associativity and nebit
(Percents shown relative to 4-way SA cache. Smaller miss rates are better.)

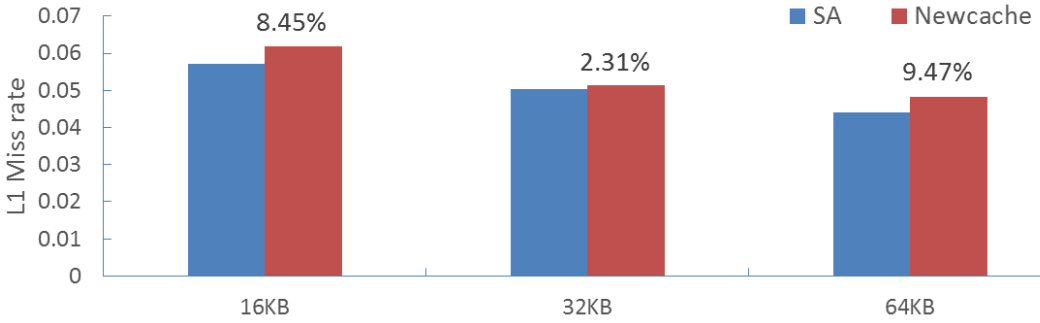

Figure 27 Bbench: L1 data cache miss rate vs. cache size
(Percents shown relative to 4-way SA cache. Smaller miss rates are better)

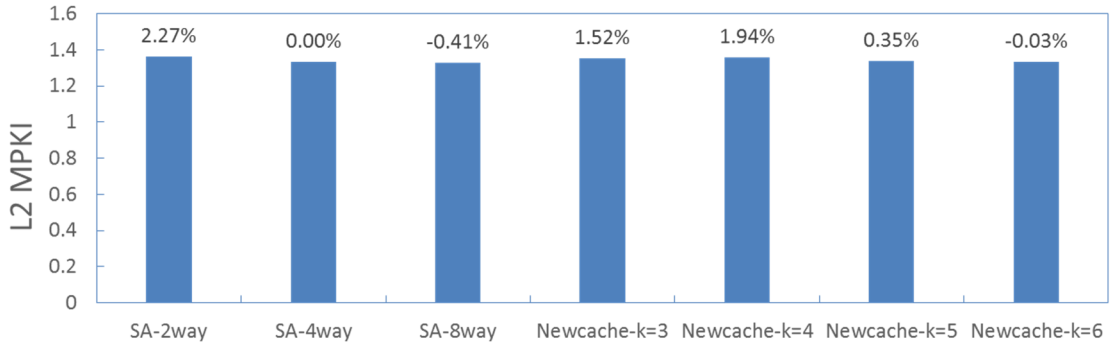

Figure 28 Bbench: L2 MPKI for various associativity and nebit (k) values
(Percents shown relative to 4-way SA cache. Smaller Misses Per Kilo Instruction, MPKI, are better.)

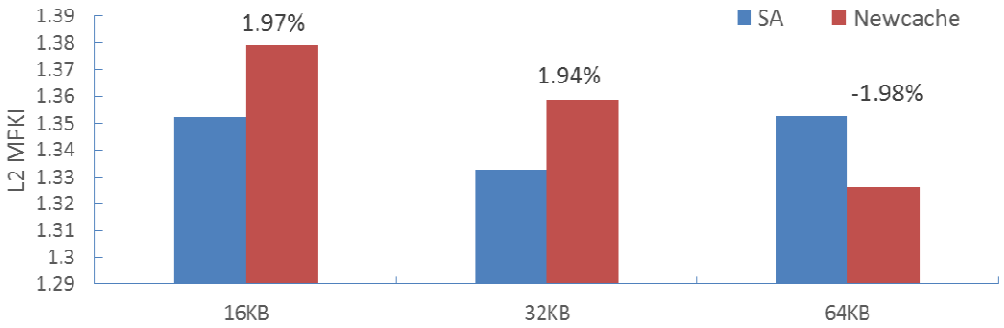

Figure 29 Bbench: L2 MPKI vs. cache size
(Percents shown are Newcache k=4 relative to 4-way SA cache, for a given cache size. Smaller Misses Per Kilo Instruction, MPKI, are better.)

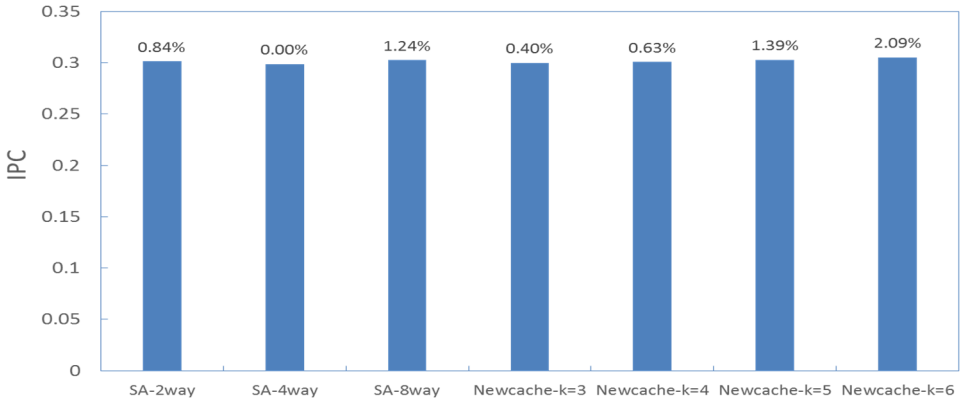

Figure 30 Bbench: IPC for various associativity and nebit
(Percents shown relative to 4-way SA cache. Larger IPC (Instructions Per Cycle) values are better.)

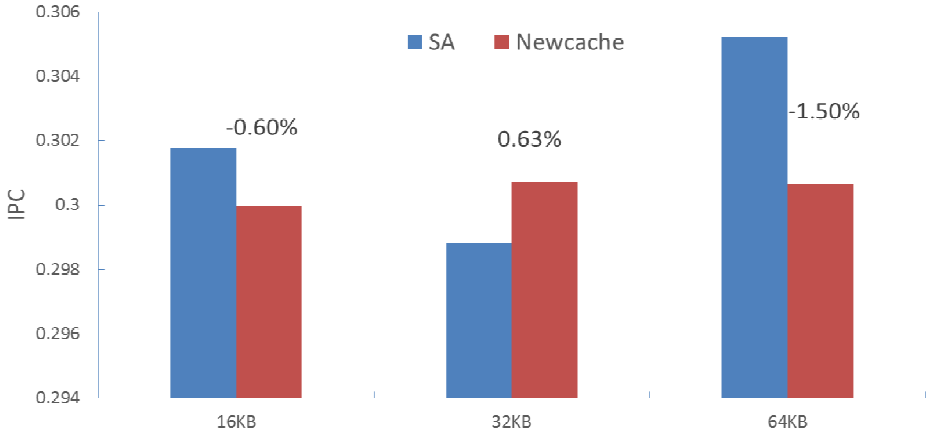

Figure 31 Bbench: IPC vs. cache size
(Percents shown are Newcache k=4 relative to 4-way SA cache, for a given cache size. Larger IPC values are better.)

5.2.3.2 0xbench benchmark suite

L1 data cache miss rate

The data cache miss rate results for the 14 0xbenchmarks are shown in Figure 32 and Table 12 for various associativity and nebit. As an Android full system benchmark, we find that 0xbench benchmarks have very similar data cache miss rates as Bbench (on the order of 10^{-2}). On average, Newcache incurs about 5% - 12% more data cache miss rate. Similar to Bbench, data cache miss rate decreases with the increasing of nebit, but is less sensitive than Bbench. We also find a relatively large drop of miss rate when nebit is increased from 3 to 4 (see for example, gc, lesson16 and teapot). The last set of bars is the average, "avg", overall all the benchmarks.

Figure 33 and Table 5 show the dependence of data cache miss rate on the cache size. For each benchmark, the first 2 bars are for 16KB caches for SA cache, and Newcache, the next 2 for 32KB caches, and the last 2 for 64KB caches. Unlike Bbench, we find that the relative overhead of miss rate compared with SA cache decreases for larger cache size, which is consistent with our findings for embedded processor benchmarks.

L2 MPKI

Again, we find the increase in overall misses to the memory is negligible (as shown in Figure 34 and Table 14) although data cache miss rate is increased by 5% - 12%. Also L2 MPKI shows almost no dependence on both set-associative and nebit. Cache size also has negligible impact (as shown in Figure 35 and Table 15).

Overall Performance, IPC

From Figure 30 and Table 16, we find that the overall performance, IPC, of Newcache is about the same as a baseline 4-way SA cache, when nebit is equal to or larger than 4. The improvement in IPC when Newcache nebit goes from k=3 to k=4 can be seen in drawarc, drawimage, drawtext, lesson08, teapot.

From Figure 37 and Table 17 we find that when the cache size is increased from 16KB to 64KB, the relative performance is changed from about 2% degradation to 1% improvement.

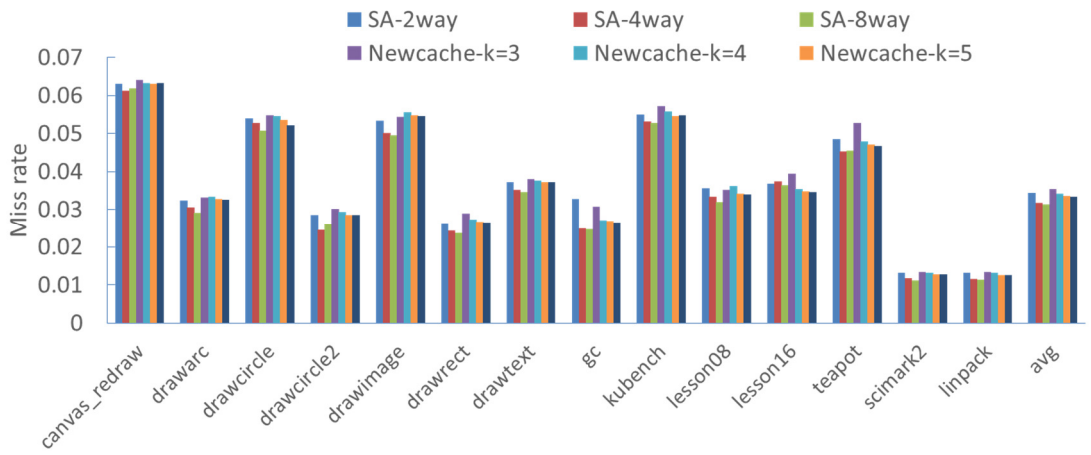

Figure 32 0xbench: L1 data cache miss rate for various associativity and nebit
(Smaller miss rates are better)

Table 12 0xbench: L1 data cache miss rate for various associativity and nebit

	SA-2way	SA-4way	SA-8way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
average	0.0344	0.0317	0.0313	0.0354	0.0342	0.0335	0.0333
increase relative to SA-4way	8.37%	0.00%	-1.53%	11.50%	7.89%	5.65%	4.95%

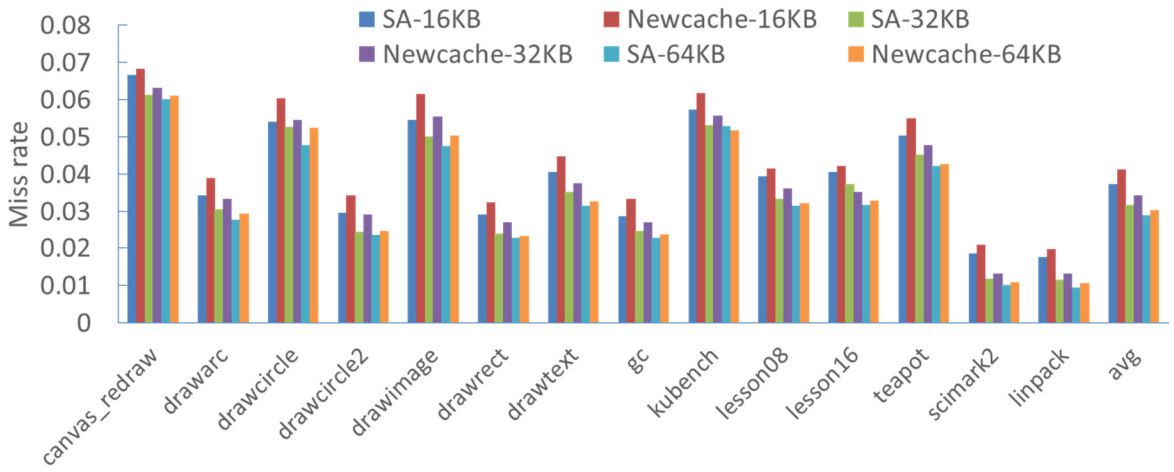

Figure 33 0xbench: L1 data cache miss rate vs. cache size
(Smaller miss rates are better)

Table 13 0xbench: L1 data cache miss rate increase relative to SA vs. cache size

	16KB		32KB		64KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	0.037	0.041	0.032	0.034	0.029	0.030
Increase relative to SA-4way		10.32%		7.89%		4.51%

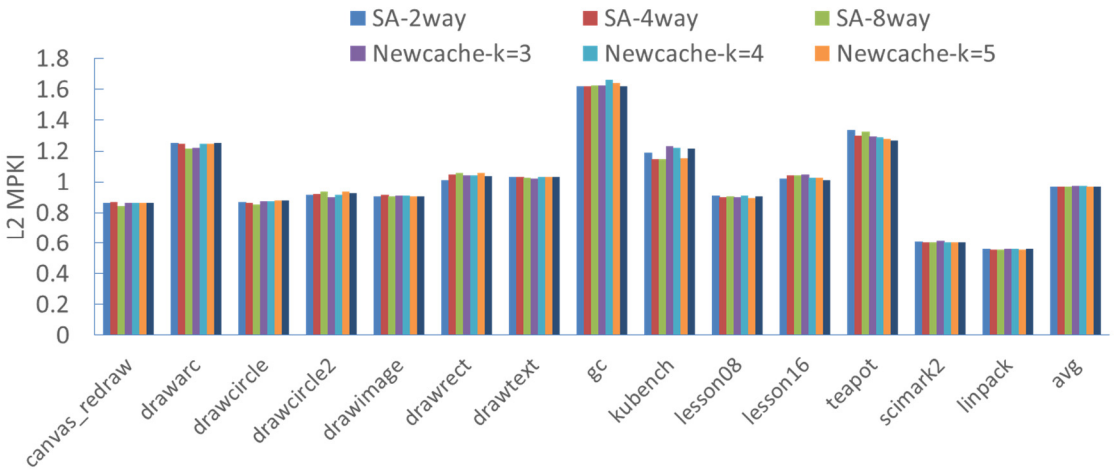

Figure 34 0xbench: L2 MPKI for various associativity and nebit
(Smaller MPKIs are better)

Table 14 0xbench: L2 MPKI increase relative to SA cache for various associativity and nebit

	SA-2way	SA-4way	SA-8way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
average	0.972	0.970	0.968	0.972	0.974	0.971	0.972
increase relative to 4-way SA	0.12%	0.00%	-0.23%	0.19%	0.40%	0.07%	0.13%

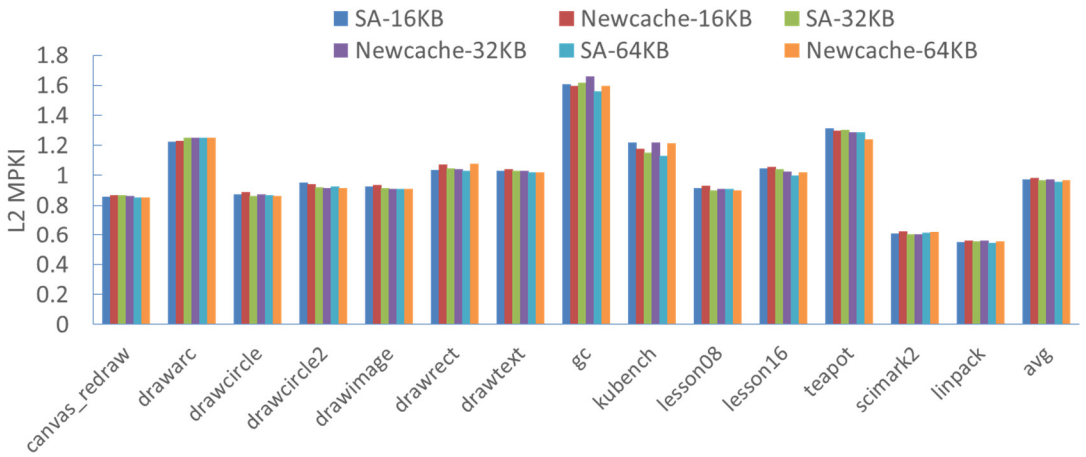

Figure 35 0xbench: L2 MPKI vs. cache size
(Smaller MPKIs are better)

Table 15 0xbench: L2 MPKI increase relative to SA cache vs. cache size

	16KB		32KB		64KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	0.976	0.983	0.970	0.974	0.960	0.970
Increase relative to 4-way SA		0.80%		0.40%		1.04%

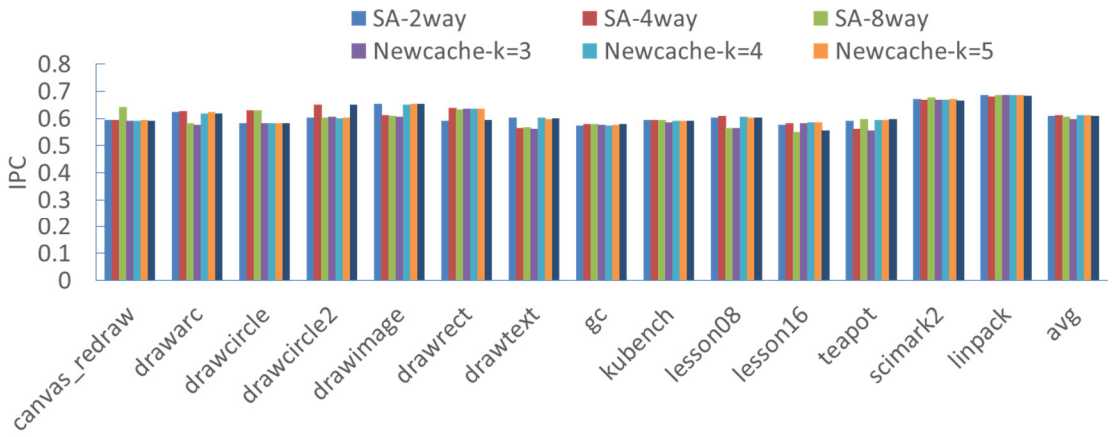

Figure 36 0xbench: IPC for various associativity and nebit
(Larger IPCs are better)

Table 16 0xbench: IPC increase relative to SA cache for various associativity and nebit

	SA-2way	SA-4way	SA-8way	Newcache-k=3	Newcache-k=4	Newcache-k=5	Newcache-k=6
average	0.611	0.613	0.607	0.598	0.613	0.614	0.611
increase relative to 4-way SA	-0.36%	0.00%	-0.87%	-2.40%	0.07%	0.15%	-0.30%

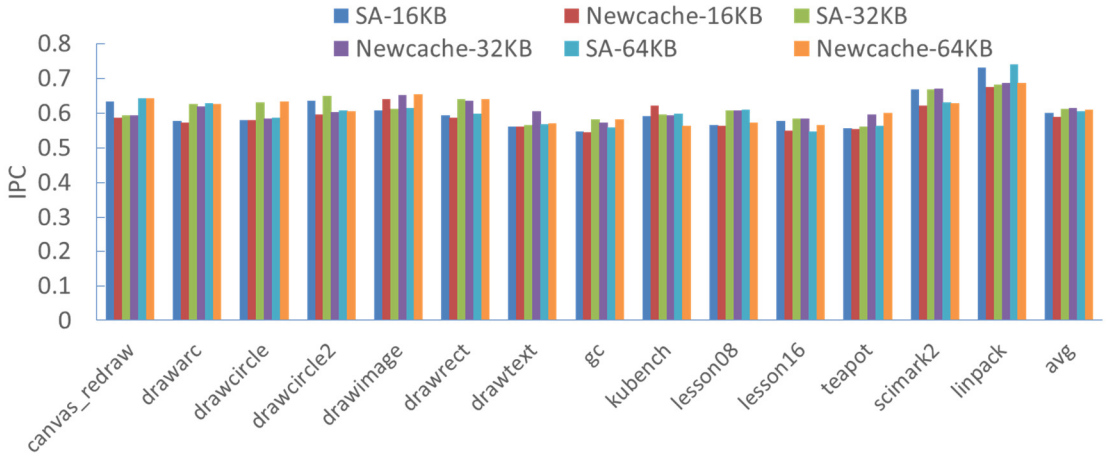

Figure 37 0xbench: IPC vs. cache size
(Larger IPCs are better)

Table 17 0xbench: IPC increase relative to SA vs. cache size

	16KB		32KB		64KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	0.599	0.588	0.613	0.613	0.605	0.611
Increase relative to 4-way SA		-1.90%		0.07%		0.99%

5.2.3.3 CoreMark

Coremark is a widely used processor benchmark designed by EEMBC. It is a simple benchmark for embedded systems, targeting at evaluating just the processor core. It consists of only 16K lines of code, with little I/O. It focuses primarily on integer operations. It has 4 workloads: matrix multiplication, linked lists, state machines and CRCs. The output of each workload is the input of the next workload. Coremark has very small memory footprint and extremely low cache miss rates. However, it is widely used in the industry.

L1 data cache miss rate

Figure 38 shows the L1 data cache miss rate for set associative (SA) cache with different associativity and Newcache with different number of extra index bits (nebit) *k*. The label gives the increase of miss rate relative to the 4-way SA cache. We note that Coremark has a very small memory footprint and a 32 KB L1 data cache is large enough to hold the most frequently used data which gives a miss rate as low as 10^{-6}. Newcache with nebit 3 and 4 does not work as well as the LRU replacement algorithm due to extra tag misses and the miss rate is increased by about 2 orders of magnitude (nevertheless, it is still as low as 10^{-4}).

Figure 39 shows the L1 data cache miss rate for different cache sizes. Newcache uses the baseline configuration with nebit of 4. We find that when the cache size is increased to 64KB, Newcache performs exactly the same as the SA cache.

L2 MPKI

Figure 40 and Figure 41 show the L2 MPKI for various associativity, nebit and cache sizes. From the results we find that even though Newcache with nebit of 3 and 4 leads to significant increase of L1 data cache miss rate, the L2 cache MPKI (overall misses to the memory) is almost unchanged due to the small memory footprint compared with the size of L2 cache.

IPC

The memory footprint of CoreMark is so small that it has no impact (degradation is within 0.1%) on the overall system performance, as shown in Figure 42 and Figure 43. Note that CoreMark has a relatively high IPC of over 1 instruction per cycle.

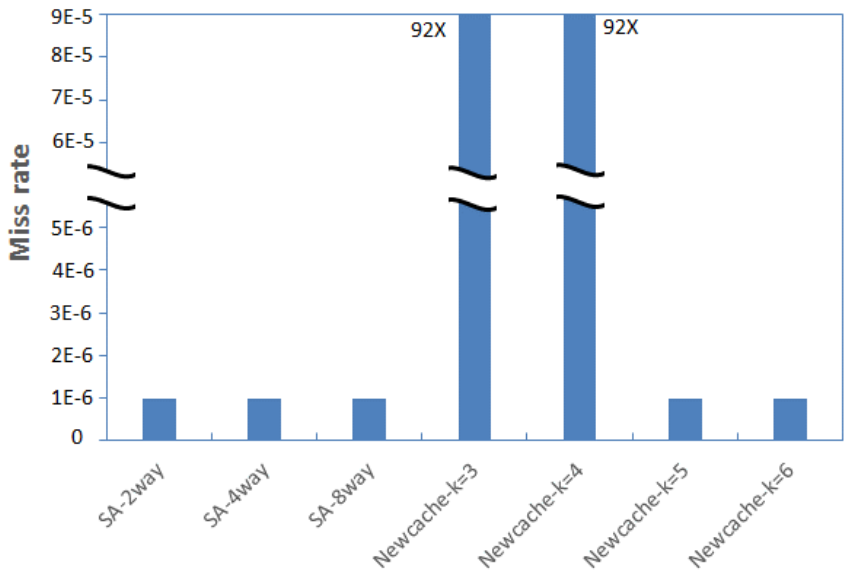

Figure 38 CoreMark: L1 data cache miss rate for various associativity and nebit
(Percents relative to 4-way SA. Smaller miss rates are better)

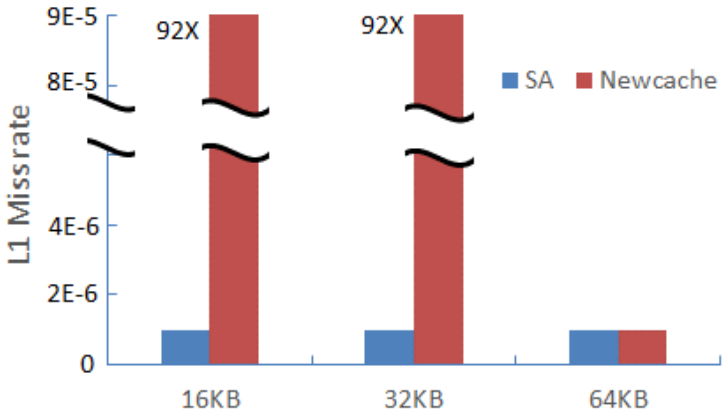

Figure 39 CoreMark: L1 data cache miss vs. cache size

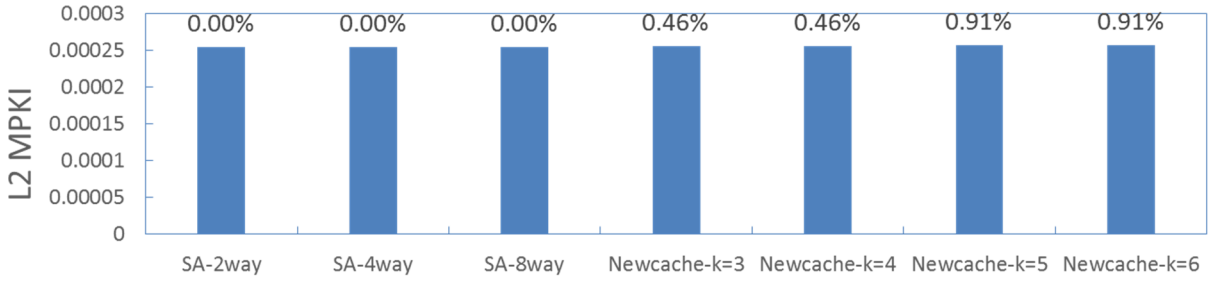

Figure 40 CoreMark: L2 MPKI for various associativity and nebit
(Percents relative to 4-way SA. Smaller MPKIs are better)

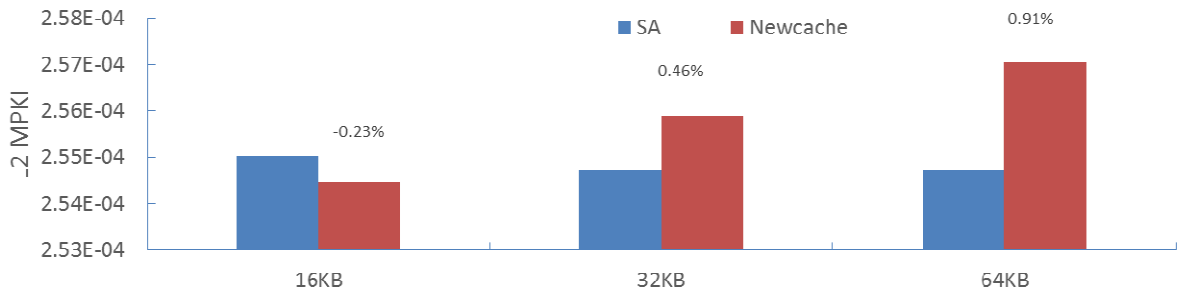

Figure 41 CoreMark: L2 MPKI vs. cache size
(Percents relative to 4-way SA. Smaller MPKIs are better)

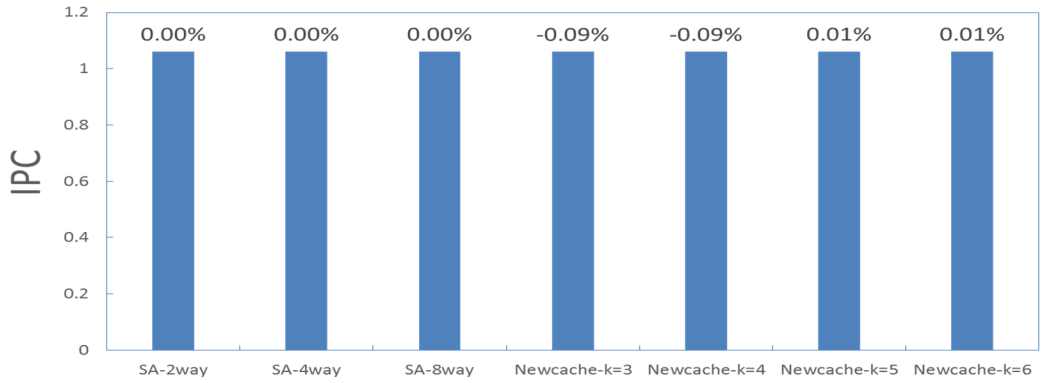

Figure 42 CoreMark: IPC for various associativity and nebit
(Percents relative to 4-way SA. Larger IPCs are better)

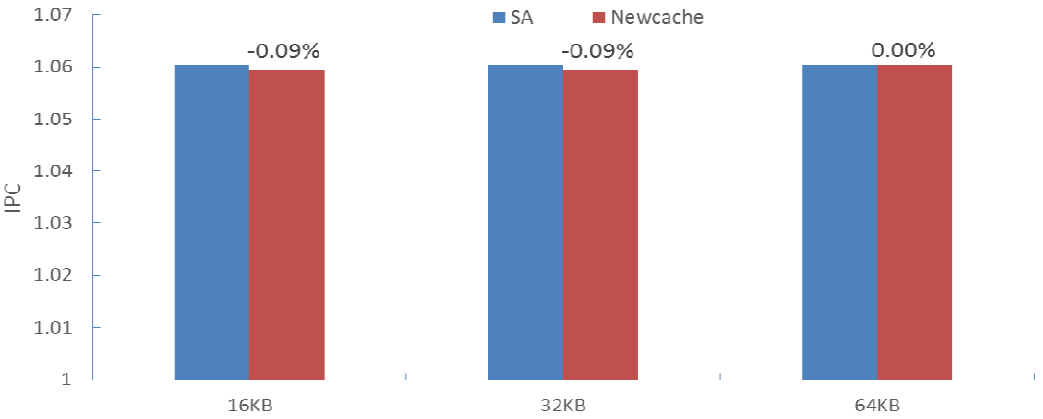

Figure 43 CoreMark: IPC vs. cache size
(Percents relative to 4-way SA. Larger IPCs are better)

5.2.3.4 Mibench benchmark suite

Mibench has 35 applications in 6 categories:

- Auto/Industrial(6):Basic math, bitcount, qsort, susan (edges, corner, smooth)
- Telecom (7): CRC32, FFT, IFFT, ADPCM (enc, dec), GSM (enc, dec)
- Network (2): Dijkstra, Patricia
- Security (7): Blowfish (enc, dec), PGP (sign, verify), Rijndael (enc, dec), sha
- Office (5): Ghostscript, ispell, rsynth, sphinx, stringsearch
- Customer (8): Jpeg, lame, mad, tiff2-(bw, rgba), tiff-(dither, median), typeset.

These are very old programs, and some would not compile with the ARM cross compiler, or had runtime errors. We used Mibench since it is a free (academic) benchmark suite, modeled after EEMBC, which is a commercially maintained benchmark for embedded systems that is quite expensive.

L1 data cache miss rate

Figure 44 shows the L1 data cache miss rate for various associativity and nebit and Table 18 summarizes the increase of miss rate relative to the 4-way SA cache. Similar to CoreMark, the L1 data cache miss rate is also very low (on the order of 10^{-4}). In general, the baseline Newcache (k=4) incurs more data cache miss rate than the SA cache. We find that nebit has a significant impact on the data cache miss rate and the miss rate is dropped very significantly when nebit is increased from 3 to 4. Some of the miss rates are so small they cannot even been seen in the figure.

Figure 45 and Table 19 show the impact of cache size on L1 data cache miss rate. The increase of L1 data cache misses relative to SA decreases for larger caches. This is because a larger cache size corresponds to a larger logical direct mapped cache (LDM) in Newcache, and hence fewer conflict misses.

L2 MPKI

Since L2 cache size is relatively large for Mibench, which only has a small memory footprint, the increase of L2 MPKI is negligible, as shown in Figure 46 and Table 20. From Figure 47 and Table 21 we find that similar to the L1 data cache miss rate, the relative increase of L2 MPKI decreases with the increase of cache size.

Overall Performance, IPC

Similar to CoreMark, Mibench does not stress the memory system enough, hence the overall performance is almost unchanged compared with the SA cache. Even with small nebit (k=3) (see Figure 48 and Table 14) and small cache size (16KB) (see Figure 49 and Table 23), the IPC degradation is less than 0.5%.

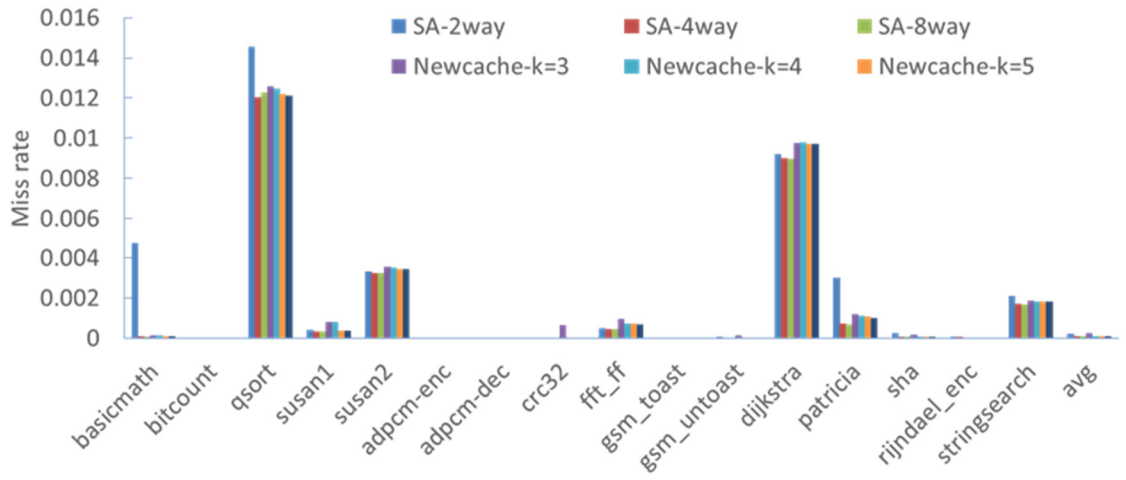

Figure 44 Mibench: L1 data cache miss rate for various associativity and nebit
(Smaller miss rates are better)

Table 18 Mibench: L1 data cache miss rate for various associativity and nebit

	SA-2way	SA-4way	SA-8way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
average	2.09E-04	1.06E-04	8.78E-05	2.53E-04	1.16E-04	1.08E-04	1.07E-04
increase relative to SA-4way	97.44%	0.00%	-17.03%	139.40%	9.37%	2.06%	1.33%

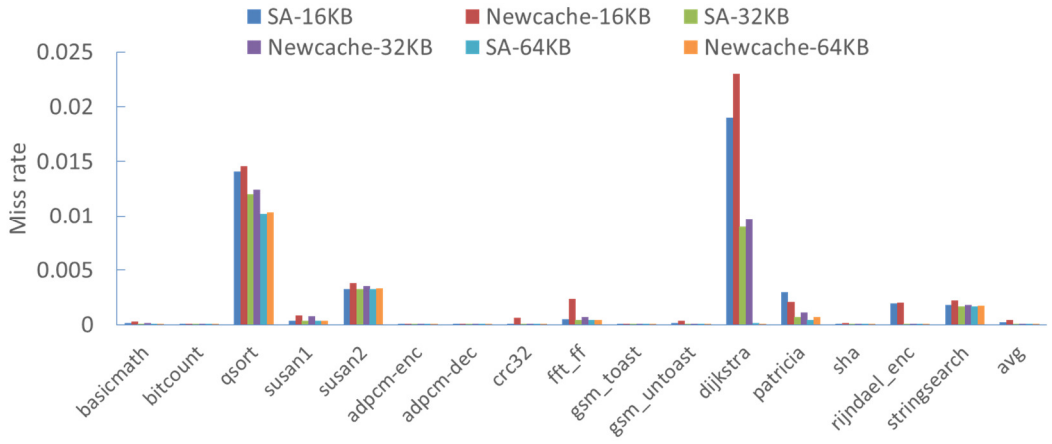

Figure 45 Mibench: L1 data cache miss rate vs. cache size
(Smaller miss rates are better)

Table 19 Mibench: L1 data cache miss rate relative to SA vs. cache size

	16KB		32KB		64KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	2.16E-04	4.42E-04	1.06E-04	1.16E-04	5.67E-05	5.79E-05
Increase relative to SA		104.30%		9.37%		2.04%

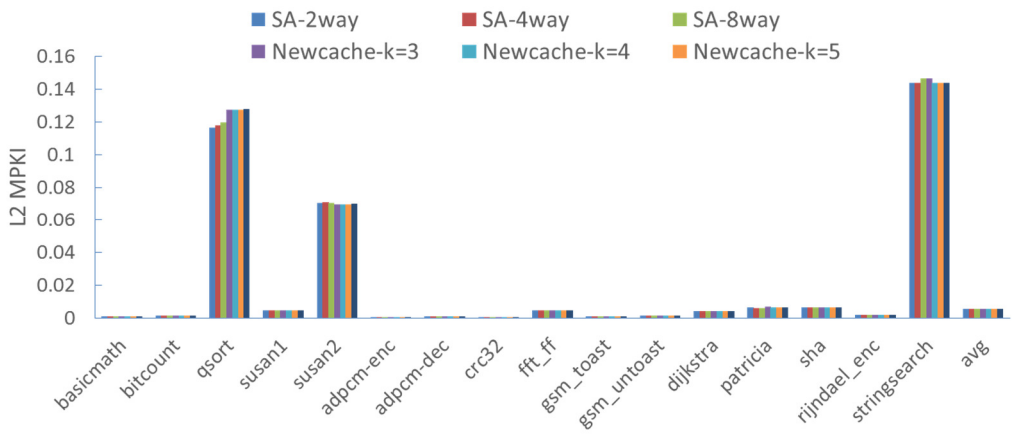

Figure 46 Mibench: L2 MPKI for various associativity and nebit
(Smaller miss rates are better)

Table 20 Mibench: L2 MPKI relative to SA for various associativity and nebit

	SA-2way	SA-4way	SA-8way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
average	5.70E-03	5.72E-03	5.72E-03	5.77E-03	5.78E-03	5.76E-03	5.78E-03
increase relative to SA-4way	-0.30%	0.00%	0.05%	0.97%	1.03%	0.77%	0.99%

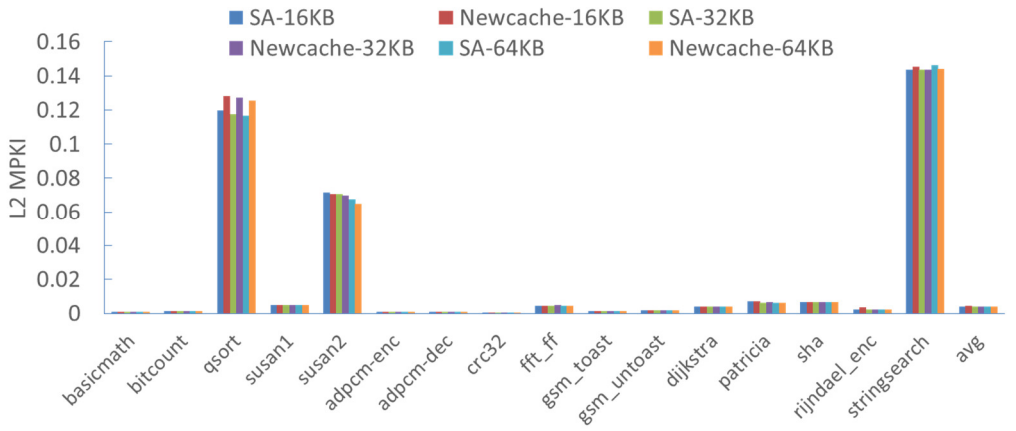

Figure 47 Mibench: L2 MPKI vs. cache size
(Smaller MPKIs are better)

Table 21 Mibench: L2 MPKI relative to SA vs. cache size

	16KB		32KB		64KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	4.07E-03	4.22E-03	4.00E-03	4.03E-03	4.00E-03	4.00E-03
Increase relative to SA		3.85%		0.86%		0.11%

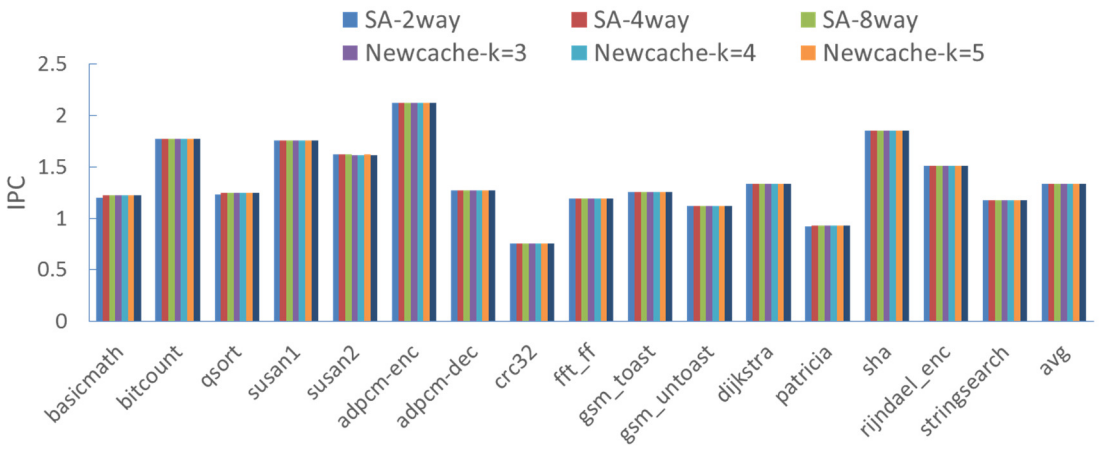

Figure 48 Mibench: IPC for various associativity and nebit
(Larger IPCs are better)

Table 22 Mibench: IPC for various associativity and nebit

	SA-2way	SA-4way	SA-8way	Newcache-k=3	Newcache-k=4	Newcache-k=5	Newcache-k=6
average	1.338	1.342	1.342	1.341	1.341	1.341	1.341
increase relative to SA-4way	-0.28%	0.00%	0.00%	-0.08%	-0.07%	-0.08%	-0.08%

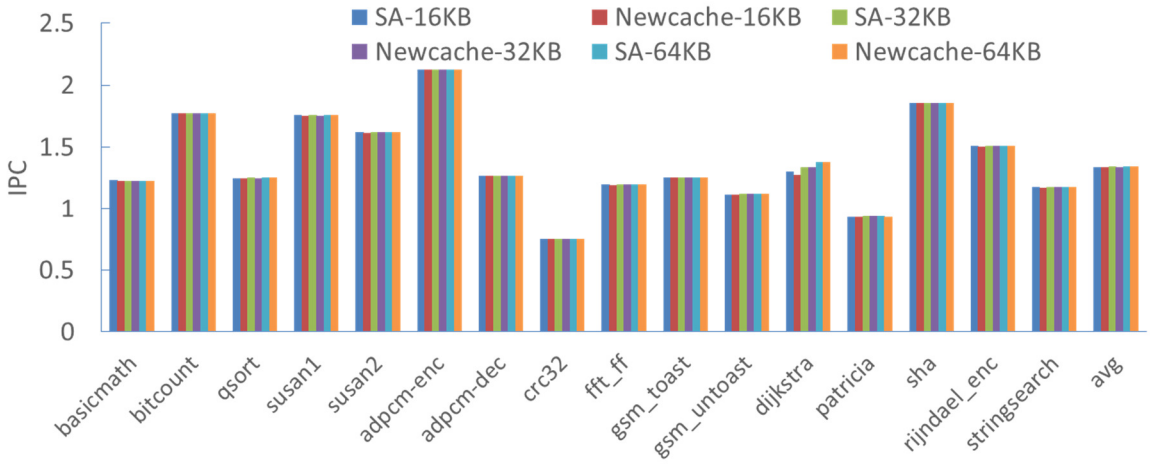

Figure 49 Mibench: IPC vs. cache size
(Larger IPCs are better)

Table 23 Mibench: IPC vs. cache size

	16KB		32KB		64KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	1.338	1.335	1.342	1.341	1.344	1.344
Increase relative to SA-4way		-0.23%		-0.07%		-0.03%

5.2.4 Summary of Smartphone Performance evaluation results

1) Benchmark Characteristics

The full-system benchmarks (Bbench and 0xbench) are closer to the real-world smartphone applications, while the embedded processor benchmarks (Coremark and Mibench) do not stress the memory system sufficiently. The L1 data cache miss rate is on the order of 10^{-6} – 10^{-4} for embedded processor benchmarks and on the order of 10^{-2} for full system benchmarks. The L2 MPKI for the full system benchmarks are also one order of magnitude larger than the embedded processor benchmarks. The full system workloads are also more I/O intensive with less instruction level parallelism. Therefore, the IPC of the full system workloads are much lower (~0.3 for Bbench and ~0.6 for 0xbench) than embedded processor benchmarks (> 1).

2) Performance comparison of Newcache and conventional set-associative caches

The baseline Newcache (nebit k=4 and 32KB) incurs moderately more data cache miss rate (mostly less than 7%) and slightly increased backend traffic to memory in overall misses of the L2 cache (L2 MPKI), compared with the baseline 4-way SA cache.

However, due to the use of non-blocking cache technology that can hide cache miss latency, as well as the non-memory-intensive property of the smartphone benchmarks, there is almost no impact on the overall performance in Instructions executed Per Cycle (IPC).

3) Impact of extra index bits *k* for Newcache

Increasing extra index bits (nebits or k) reduces L1 data cache conflict misses for all the benchmarks. The impact is more obvious for the embedded processor benchmarks than for the full system benchmarks. For all benchmarks except CoreMark, L1 data cache miss rate has a significant drop when k is increased from 3 to 4. There is no dependence for the L2 miss rate on the extra index bits in the L1 data cache.

4) Sensitivity to cache size

For all benchmarks except Bbench, we find Newcache works better with larger cache size, due to the reduced conflict misses caused by the larger Logical Direct Mapped (LDM) cache aspect of Newcache.

5.3 Server performance for Newcache used as L1 D-cache

We now study the server benchmark performance when Newcache is used as a L1 data cache in a cloud computing server. For this, we surveyed many cloud server applications and workloads, and assembled our cloud computing benchmark suite as described below, since we could not find a representative cloud benchmark suite. This cloud computing benchmark suite, ready to simulate on the gem5 cycle accurate simulator, is another contribution of this project.

Table 24 summarizes the testing benchmarks and the driving tools we use for the 6 representative server workloads we selected.

Table 24 Summary of Cloud Server Benchmarks

	Server-side Benchmarks	**Client-side Driving Tools**
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

It is important to note that some Server-side benchmarks require some time to start up before their services are available to the client under gem5. We 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.

Since this is a new cloud benchmark suite, and it is considerably more difficult to simulate cloud server workloads than smartphone benchmarks, we describe each cloud server workload (benchmark) below and how we use it in section 5.3.1. We describe the enhancements to the gem5 simulator we had to make to do dual-system simulation for the client-driven server workloads in section 5.3.2.

5.3.1 Cloud Server Benchmarks: Description and Selection

We divide cloud server benchmarks into several categories: Web Server, Database Server, Mail Server, File Server, Streaming Server, and Application Server. We describe the potential possible benchmarks for each category and the method and parameters we use for the performance measurements.

5.3.1.1 Web Server and Client

In order 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, etc.) to store data and a server-side scripting engine (PHP, etc.) to generate dynamic web pages if necessary.

Server:

- Apache HTTP server (httpd) [7]: 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) [6]: 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/*.

5.3.1.2 Database Server and Client

Server:

- **MySQL [18]:** 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 [13] : 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 [24]: 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.

5.3.1.3 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 [22]: 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 [22]: 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 the command line when using postal.

Testing:

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

5.3.1.4 File Server and Client

A client can interact with the remote server file system via several 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 [23]: 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 [10]: 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 userpassword 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 performance 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 64 KB files. By replacing loadfile with smb-readfiles.txt, we generate Dbench_read, which launches 3 clients and each client opens and reads five 64 KB files.

5.3.1.5 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 the 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 [11]: 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 [17]: 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 [25]: 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 [19]: 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.

Testing:

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.

5.3.1.6 Application Server and Client

An application server [9] 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. However, many application servers also provide services like clustering, fail-over, and load-balancing. The currently most used Java application server is actually an extension of Java virtual machine for running applications. The server handles connection to the database on one side and connections to the web client on the other side. 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 [8]: We use Tomcat on the server side. Tomcat, Jetty [15], Jboss [14] and Glassfish [12]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:

For the client, we use Apache ab to send HTTP requests to Tomcat (Apache ab is described previously as a web server client).

Testing:

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.

5.3.2 Client-Server gem5 Simulation Methodology

In order to characterize how the server performs, we use the dual system setup as described in section 3.3. It is not easy to install server benchmarks under the gem5 simulator. 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 disk image. Similar to the full-system smartphone benchmarks, we can take a checkpoint immediately before the client starts sending requests and then run the dual system in detailed CPU mode, after checkpoint restore, to completion.

The reference processor for the cloud server simulated is the Intel core-i7 with three levels of cache hierarchy, with parameters shown in Table 25. The baseline caches are 8-way set-associative L1 data cache, 4-way set-associative L1 instruction cache, a unified 8-way set-associative L2 cache and a 16-way set-associative L3 cache.

We replace the L1-D cache with Newcache in our evaluation. The baseline Newcache configuration has k=4 extra index bits.

Table 25 Baseline cloud server configuration

Single-core out-of-order X86 processor	L1-I cache (private)	L1-D cache (private)	L2 cache (unified)	L3 cache	memory
	4-way 32 kB	8-way 32 kB	8-way 256 kB	16-way 2MB	2GB

	4-cycle latency	4-cycle latency	10-cycle latency	35-cycle latency	100-cycle latency
Cache block size	64 B				
Clock frequency	3GHz				

5.3.3 Server Performance Evaluation Results for Newcache used as L1 D-cache

Based on the baseline cache configurations in Table 25, we replace the L1 data cache with Newcache of the same size (32 KB). We run the cloud benchmark services on the server side and driving tools on the client side. The reference L1 data cache is an 8-way set-associative cache.

Note that for these server benchmarks, the reference is an 8-way set-associative cache since server processors tend to be Intel processors with 8-way SA for their L1 Data caches. For the smartphone benchmarks, the reference used was 4-way set-associative caches since smartphone processors tend to be ARM processors with 4-way SA for their L1 Data caches.

5.3.3.1 L1 data cache miss rate for server benchmarks

Figure 50 shows the L1 data cache miss rate for different associativity for SA caches and number of extra bits (nebit) *k* for Newcache, Table 26 shows the average L1 data cache miss rate increase relative to the baseline 8-way SA cache. From the results, we can see that Newcache (with nebit k=4 or 5) has higher miss rate than 8-way SA cache but lower miss rate than 4-way SA cache. **We can also see that when nebit is increased to 6, Newcache has almost the same miss rate as that of the 8-way SA cache.**

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 L1 data cache miss rate increases a lot. This is because as the server handles more user requests and more threads, the data from different handled threads tend to contend for the limited slots in the cache, thus evicting each other's data and causing larger L1 data cache miss rates.

Figure 51 shows the impact of cache size on the data cache miss rate. The average increase compared with the 8-way SA cache is summarized in Table 27. Newcache tends to benefit more from larger cache sizes since a larger cache size gives a larger logical direct mapped (LDM) cache with less conflict misses.

The results show that Newcache decreases the data cache miss rate by 3.9% (improvement) when the cache size is increased to 64KB.

We do not have the result for mysql with 16 KB SA cache as the data cache. Under a small cache size, some benchmark service cannot start normally on gem5, which might be caused by gem5's internal implementation.

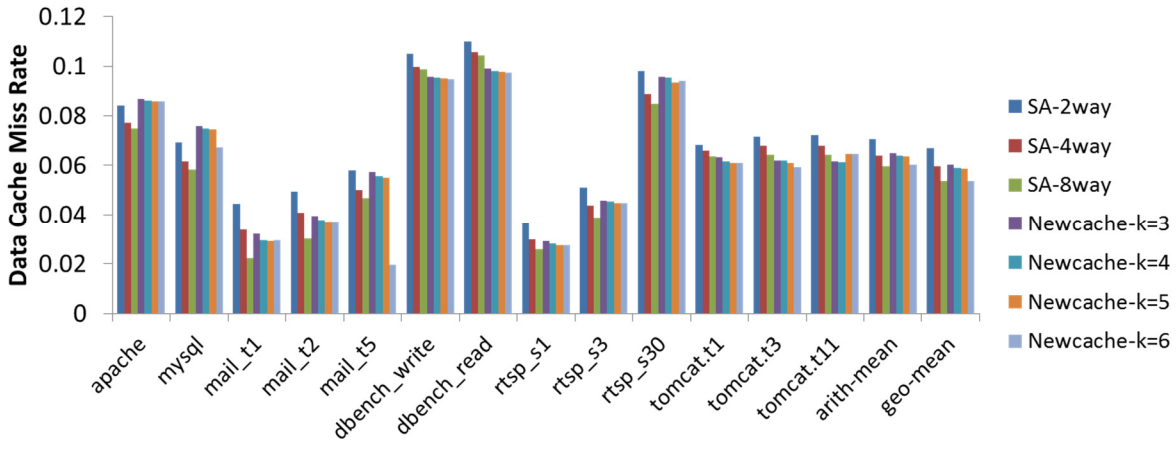

Figure 50 Newcache as L1 data cache: L1 data cache miss rate for various associativity and nebit

Table 26 Newcache as L1 data cache: data cache miss rate compared to 8-way SA cache

	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
ncrease relative to SA-8way	18.12%	7.22%	0.00%	8.68%	6.97%	6.45%	0.70%

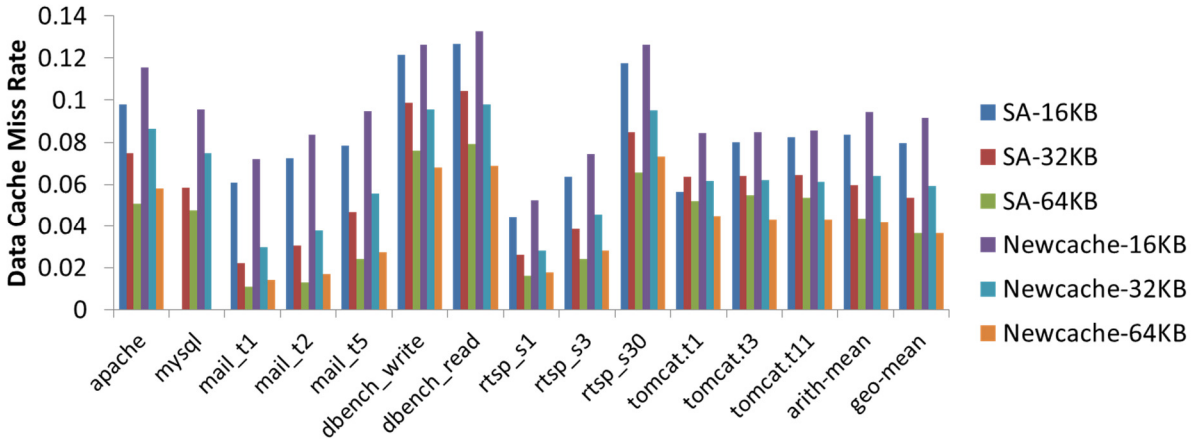

Figure 51 Newcache as L1 data cache: L1 data cache miss rate vs. cache size

Table 27 Newcache as L1 data cache: Increase of data cache miss rate relative to SA vs. cache size

	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 8-way SA		13.05%		6.97%		-3.91%

5.3.3.2 L2 MPKI for server benchmarks

Here the L2 Cache is the baseline conventional 256kB 8-way SA cache. Figure 52 shows the L2 MPKI for different associativity and nebit, and Table 28 shows the different configurations' L2 MPKI increase relative to 8-way SA cache.

When Newcache is used as the L1 Data cache, the L2 MPKI are even better than those of the baseline 8-way SA-cache. This is because the access stream to the L2 cache may have better locality when the L1 data cache is Newcache due to the random replacement algorithm. In the case of least recently used (LRU) replacement algorithm that is used in conventional SA caches, data that are not present in the L1 cache is seldom used recently, and has a higher chance of not being in the L2 cache too. Whereas in the case of Newcache, a randomly evicted data that may be reused in the near future may still be present in the L2 cache. (L2 cache is a conventional SA cache with LRU replacement algorithm).

Figure 53 shows the L2 MPKI under different cache sizes for all the benchmarks. This is similar to the data cache miss rate results. As analyzed above, for both SA-cache and Newcache, as the cache size increases, the data cache Miss Rate decreases for each benchmark. This is because a larger cache tends to have more data available for immediate use, which leads to lower data cache miss rate. So the number of data requests from the data cache to the L2 cache also shrinks, which is why L2 MPKI also decreases. **The L2 MPKI actually is 2.5% better (smaller) than the 8-way SA cache for 64KB L1 data cache.**

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 L2 MPKI. Table 29 summarizes the different cache sizes' L2 MPKI Increase relative to the 8-way SA cache. Still, Newcache basically performs as well as the SA-cache. **It's interesting to find that after changing data cache to Newcache, L2 MPKI benefits a lot for 64kB cache sizes.**

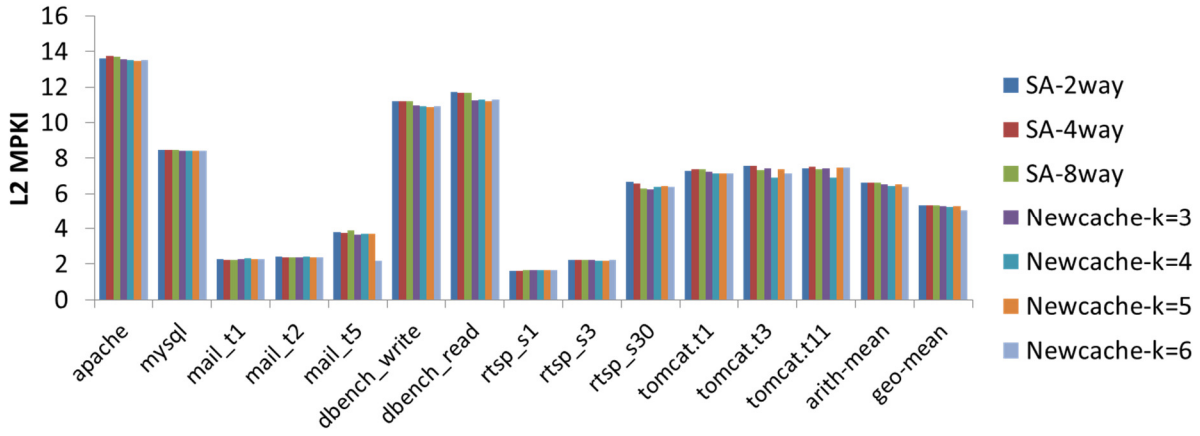

Figure 52 Newcache as L1 data cache: L2 MPKI for various associativity and nebit

Table 28 Newcache as L1 data cache: L2 MPKI increase relative to SA

	SA 2-way	SA 4-way	SA 8-way	Newcache k=3	Newcache k=4	Newcache k=5	Newcache k=6
Average	6.631	6.642	6.602	6.514	6.441	6.513	6.382
Increase relative to SA 8-way	0.44%	0.61%	0.00%	-1.33%	-2.43%	-1.37%	-3.33%

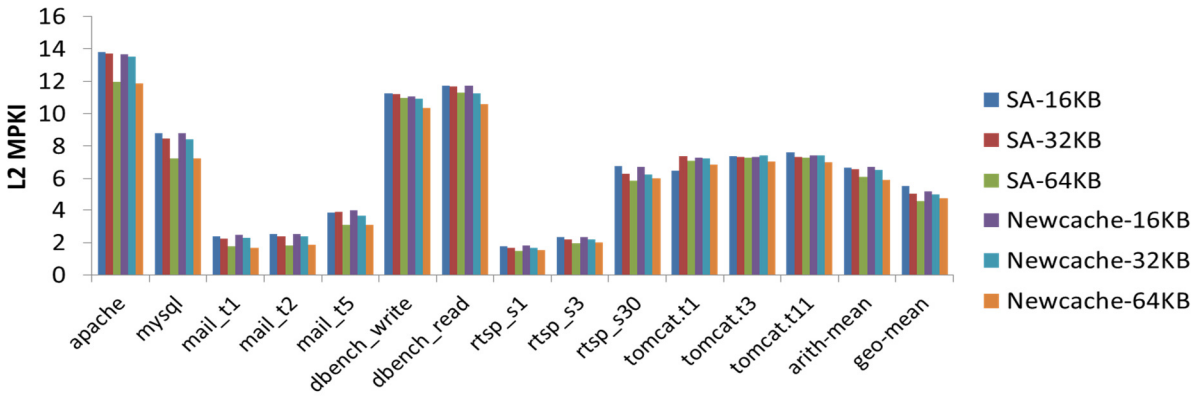

Figure 53 Newcache as L1 data cache: L2 MPKI vs. cache size

Table 29 Newcache as L1 data cache: L2 MPKI increase relative to SA vs. cache size

	16KB		32KB		64KB	
	SA	Newcache	SA	Newcache	SA	Newcache
Average	6.674	6.705	6.602	6.514	6.088	5.936
Increase relative to SA cache of same size		0.46%		-1.33%		-2.50%

5.3.3.3 Overall Server Performance in IPC

Figure 54 shows the IPCs for different associativity and nebit. We see that:

1) 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, because they spend a lot of time in the kernel: accessing the network stack, the disk subsystem, handling the user connections and requests, and syncing large amounts of threads all require the program trapping into the kernel, which also induce a lot of context switches. Kernel codes have a lot of dependencies, thus causing low IPCs.

2) Similar to the results for the data cache miss rate and global L2 cache miss rate (similar to L2 MPKI), as the client requests increase, the IPC decreases. 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 cause lower IPCs.

Table 30 summarizes all these configurations' IPC Increase relative to an 8-way set-associative cache, as an average of all the benchmarks. We can see that overall Newcache performs as well as the 8-way SA-cache. **The IPCs tend to be better than that of 8-way SA cache when nebit is increased to 6 (2.8% better on average).** A 32 KB Newcache with nebit k=3 has a 256 KB LDM cache, while a 32 KB Newcache with nebit k=6 has a 2048 KB LDM cache. Unlike desktop applications, server applications tend to fetch data from a wider range of memory space (larger memory footprint), 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 longer nebit than smartphone applications.

Figure 55 shows the IPC for different cache sizes for all the benchmarks.

Table 31 summarizes the Newcache's IPC increase relative to SA for different cache sizes on average. We can see that basically **Newcache performs as well as SA cache for overall performance in Instructions executed Per Cycle. As the size increases, Newcache tends to perform a little better than SA cache. For 64 KB L1 data cache, Newcache has 1.36% higher IPC than the 8-way SA cache.**

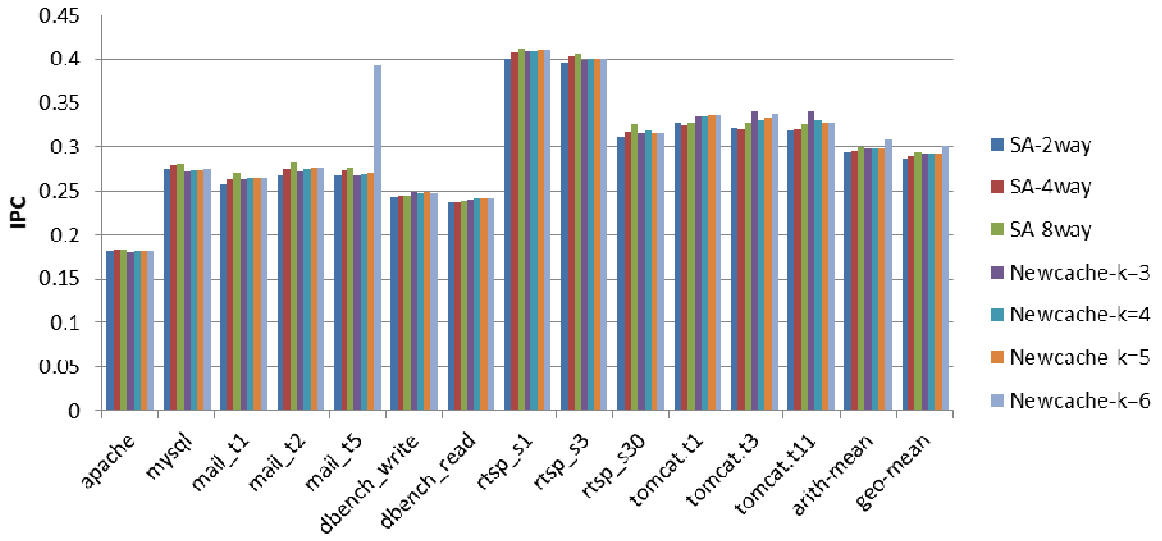

Figure 54 Newcache as L1 data cache: IPC for various associativity and nebit

Table 30 Newcache as L1 data cache: IPC increase relative to SA cache

	SA 2-way	SA 4-way	SA 8-way	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 8-way	-2.35%	-1.26%	0.00%	-0.28%	-0.52%	-0.53%	2.77%

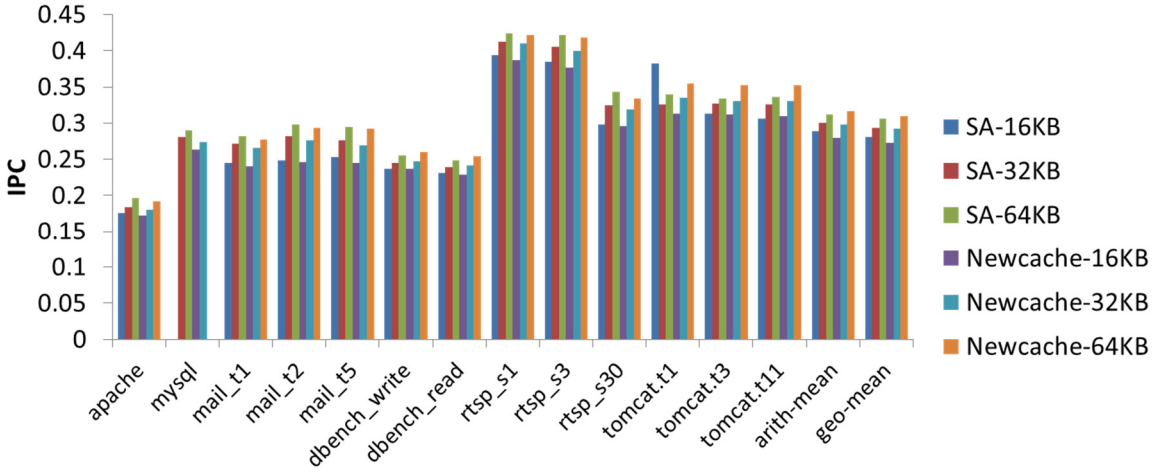

Figure 55 Newcache as L1 data cache: IPC vs. cache size

Table 31 Newcache as L1 data cache: IPC increase relative to SA vs. cache size

	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 8-way SA		-3.45%		-0.52%		1.36%

5.3.4 Summary of Server Performance Results

1) Server benchmarks tend to be very I/O intensive with very low instruction level parallelism and hence low IPC. Compared with smartphone benchmarks, server benchmarks are more memory-intensive and the memory system has greater impact on the overall performance.

2) We find that the baseline Newcache (k=4, 32KB) has moderately larger data cache miss rate than the 8-way SA cache, but smaller miss rate than the 4-way SA cache.

3) However, the overall misses for the next level cache (L2 MPKI) may be reduced for Newcache. This may be because for Newcache, randomly evicted data that may be reused in the near future may still be present in the L2 cache, whereas for conventional set-associative caches with LRU replacement, data that had been evicted from the L1 cache is seldom used recently, and has a higher chance of not being in the L2 cache too.

4) The overall performance in IPC of baseline Newcache (k=4, 32 KB) is slightly lower than the 8-way SA cache but better than the 4-way SA cache.

5) Increasing Newcache nebit to k=6 bits significantly reduces both L1 and L2 cache miss rates and improves overall performance compared with the SA cache. **A 32 KB Newcache with k=6 bits has better overall performance than an 8-way SA cache of the same size.**

6) Similar to smartphone benchmarks, Newcache benefits more from larger L1 data cache size: **when the cache size is increased to 64 KB, Newcache can reduce both L1 and L2 cache miss rates and improve overall performance.**

5.4 Server Performance for Newcache used as L2 Cache

Using the base CPU and cache configuration in Table 25 for servers, we replace the L2 cache with Newcache in this subsection. We run the benchmark services on the server side and driving tools on the client side. We then collect both the local and global L2 miss rate and IPC for all the benchmarks. Definitions of the metrics were given in Section 5.1, and are repeated below:

Local L2 miss rate = L2 cache misses / data cache MSHR misses

Global L2 miss rate = L2 cache misses / L1 data cache accesses

Note that the global L2 miss rate is similar to the L2 MPKI metric used earlier – they have the same numerator and slightly different denominators. The global L2 miss rate would tend to be larger than L2 MPKI, since it depends on the fraction of total instructions executed that are memory access instructions (typically about 30-40%).

5.4.1 Local L2 cache miss rate

Figure 56 shows the local L2 miss rate for different associativity and nebit, and Table 32 shows the increase of local L2 miss rate relative to the 8-way SA cache. We find that even when nebit is increased to 6, Newcache still incurs relatively large local L2 miss rate.

Figure 57 shows the local L2 miss rate for different cache sizes of 128 Kbytes, 256 Kbytes and 512 Kbytes, for both the baseline 8-way set-associative L2 cache and for Newcache k=4, (used as the L2 cache). Table 33 shows the average increase of L2 local miss rate for Newcache relative to the SA cache. From the result, we notice that Newcache as L2 cache does not perform as well as for the L1 data cache. Basically, Newcache tends to incur more local L2 cache misses with a larger L2 cache size.

The main cause is due to the less-optimum random replacement algorithm for L2 caches compared with the LRU replacement algorithm. Server applications tend to do a lot of stuff, like accessing network stack, the disk subsystem, handling the user connections and requests, syncing large amount of threads etc., which induce a lot of context switches and kernel trapping and the kernel may frequently access these data. For a large SA cache with LRU replacement algorithm, those data tend to remain in the L2 cache for a long time. However, Newcache's random eviction may cause those constantly-needed data and instructions to be replaced, which can cause a lot of penalties on local L2 miss rate. This is contrary to the case when Newcache is used as the L1 data cache, in which the random replacement may increase the locality of the reference stream to the L2 cache and potentially decrease the L2 cache miss rate.

Also note that with optimizations of the MSHR mechanism for the L1 non-blocking caches, the denominator in the local L2 miss rate is decreased, causing an increase in the local L2 miss rate.

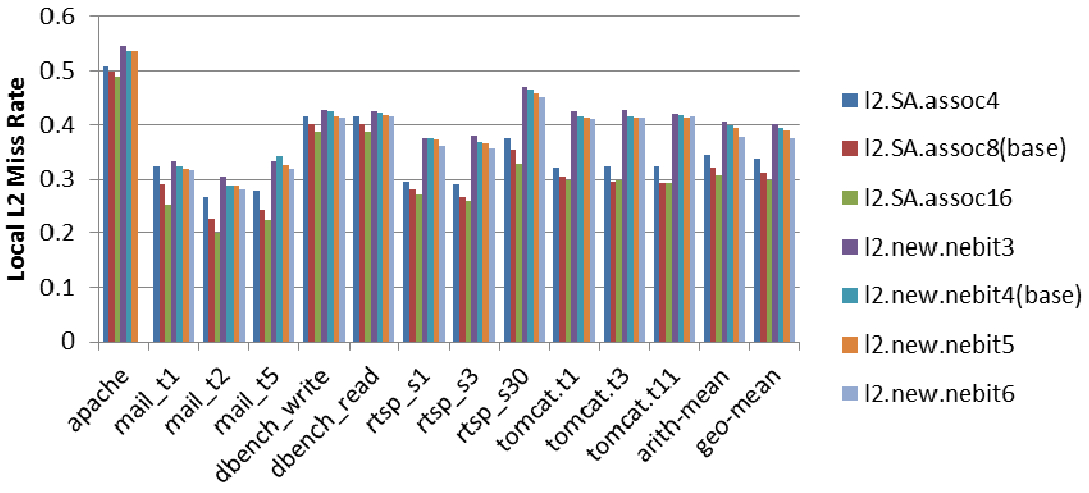

Figure 56 Newcache as L2 cache: local L2 miss rate vs. associativity and nebit

Table 32 Newcache as L2 cache: increase of local L2 miss rate relative to SA vs. associativity and nebit

	SA 4-way	SA 8-way	SA 16-way	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 8-way	7.25%	0.00%	-4.40%	26.19%	24.26%	22.78%	17.40%

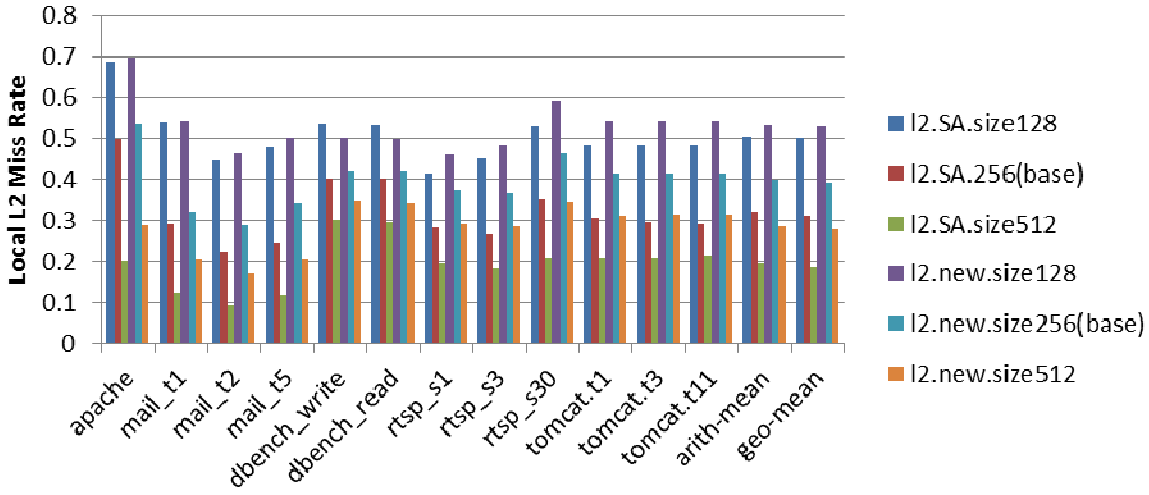

Figure 57 Newcache as L2 cache: local L2 miss rate vs. cache size

Table 33 Newcache as L2 cache: increase of local L2 miss rate relative to SA vs. cache size

	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%

5.4.2 Global L2 cache miss rate

Figure 58 shows the global L2 miss rate with different associativity and nebit, and Table 34 summarizes the increase of global L2 miss rate relative to 8-way SA. Newcache incurs 12% more global L2 misses even when nebit is increased to 6.

Figure 59 shows the global L2 miss rate for different cache sizes and the increase of global L2 miss rate relative to SA is summarized in Table 35. The results are similar to what we have got from the previous local L2 miss rate. 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 the SA cache with the same size, Newcache used as L2 cache incurs much more misses, especially for larger L2 cache sizes.

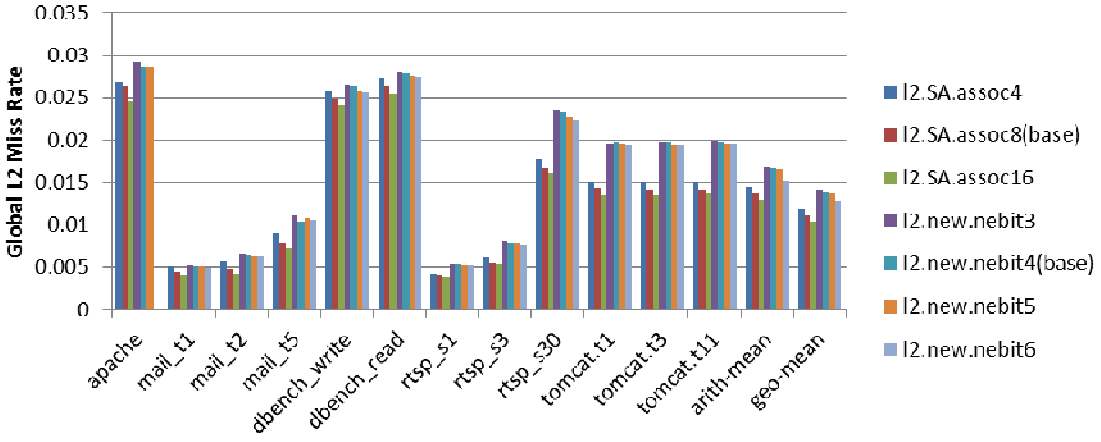

Figure 58 Newcache as L2 cache: global L2 miss rate vs. associativity and nebit

Table 34 Newcache as L2 cache: increase of global L2 miss rate relative to SA vs. associativity and nebit

	SA 4-way	SA 8-way	SA 16-way	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 8-way	5.68%	0.00%	-5.08%	23.83%	22.15%	21.14%	12.04%

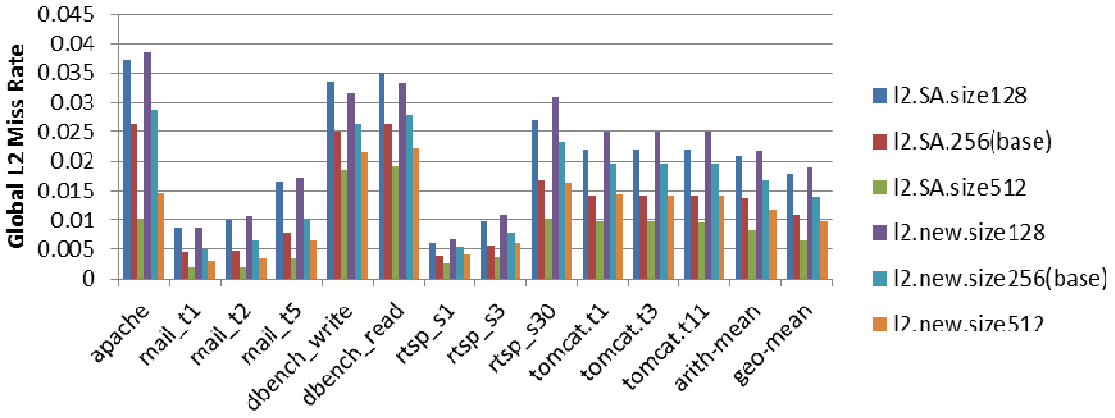

Figure 59 Newcache as L2 cache: global L2 miss rate vs. cache size

Table 35 Newcache as L2 cache: increase of global L2 miss rate relative to SA vs. cache size

	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%

5.4.3 Overal Performance in IPC

The impact of associativity and nebit is shown in Figure 60 and Table 36. It is worth noting when nebit is increased to 6, the overall performance (in IPC) will be slightly increased instead of being degraded, compared to an 8-way SA cache, when Newcache is used as the L2 cache.

The overall performance for various cache sizes is shown in Figure 61 and Table 37. **On average, the IPC for Newcache is only slightly worse than the SA cache by about 2% despite the significant increase of both local and global L2 miss rate.** This is because the L1 data cache filters out most cache misses and the global L2 miss rate is already extremely low and thus a degradation of this very low L2 miss rate will hardly impact the overall system performance. Contrary to Newcache as L1 data cache, the overall performance does not benefit from larger L2 cache size.

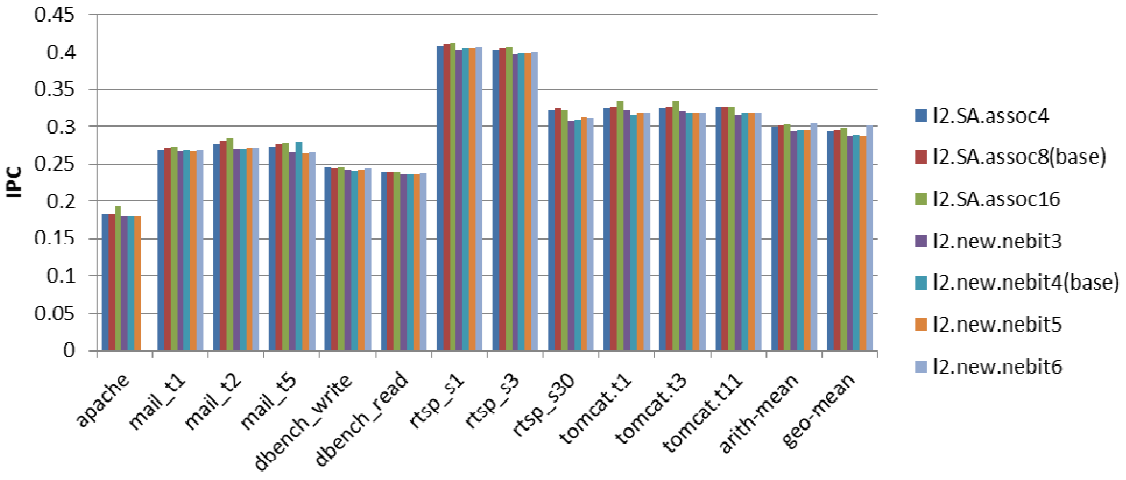

Figure 60 Newcache as L2 cache: overall performance in IPC vs. associativity and nebit

Table 36 Newcache as L2 cache: increase of IPC relative to SA vs. associativity and nebit

	SA 4-way	SA 8-way	SA 16-way	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%

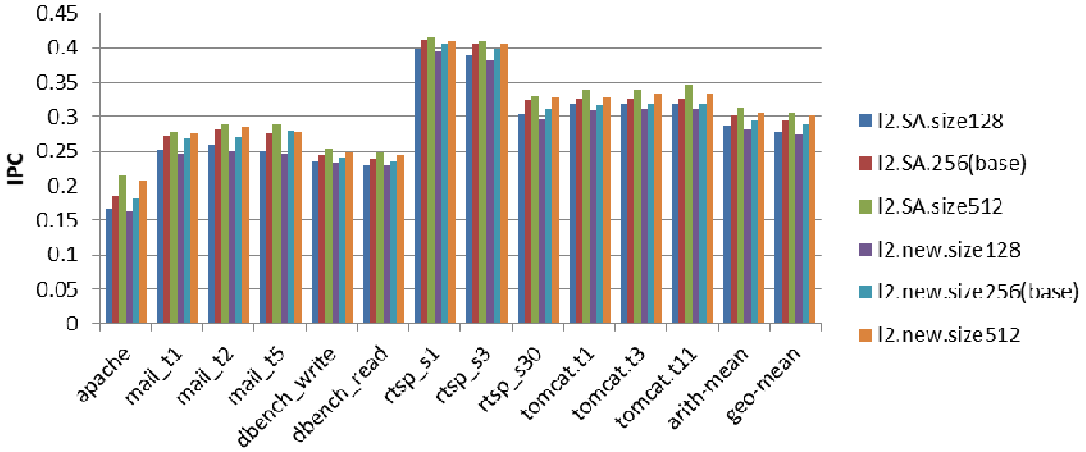

Figure 61 Newcache as L2 cache: overall performance in IPC vs. cache size

Table 37 Newcache as L2 cache: increase of IPC relative to SA vs. cache size

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

5.4.4 Summary of performance results for Newcache used as L2 cache

Basically, Newcache as L2 cache incurs significant increase of both L2 local miss rate and L2 global miss rate. Unlike for Newcache used as the L1 data cache, increasing L2 cache size, when Newcache is used as the L2 cache, does not reduce the relative increase of miss rate compared with the SA cache. Nevertheless, since the global miss rate of L2 cache is very low, the impact to the whole system is very small, and the overall performance degradation is less than 2-3%. It is worth noting that when k is increased to 6, the overall performance is even increased (by over 1%) despite the increase of global miss rate.

If side-channel attacks on the L2 cache are achievable by an attacker (which has yet to be shown), it is possible that variations of Newcache used as an L2 cache that can further improve its performance. However, considering just performance, there appears to be no advantage to using Newcache for the L2 cache.

5.5 Server Performance for Newcache used as Both L1 Data Cache and L2 Cache

In this section, we study how the system performs with Newcache used as both L1 data cache and L2 cache. We measure the L1 data cache miss rate, local L2 miss rate and global L2 miss rate for all the benchmarks. We use l1d.new and l1d.SA to represent Newcache as L1 data cache and SA cache as L1 data cache; we use l2.new and l2.SA to represent Newcache as L2 cache and SA cache as L2 cache, respectively. 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 baseline configuration (base size, associativity for SA caches and nebit for Newcache).

5.5.1 Data cache miss rate

Figure 62 shows the data cache miss rate of the four configurations for all the benchmarks, and Table 38 summarizes on average the data cache miss rate increase of the four configurations relative to l1d.SA-l2.SA, i.e., conventional set-associative L1 D-cache and L2 caches. l1d.new-l2.new induce only slightly more data cache miss rate than that of l1d.SA-l2.SA. Basically, even when both L1 data cache and L2 cache are Newcache, L1 data cache miss rate does not increase a lot (around 6.6%) compared to all SA caches.

5.5.2 Local L2 cache miss rate

Figure 63 shows the local L2 miss rate of the four configurations for all the benchmarks, and Table 39 summarizes on average the local L2 miss rate increase of the four configurations relative to l1d.SA-l2.SA. Similar to the results we get in the previous section, using Newcache as L2 cache incurs large L2 miss rate compared with SA cache, no matter whether the L1 data cache is a SA cache or Newcache. Note, however, that **using Newcache as only L1 Data cache but not as L2 cache (2nd column in** Table 39**) has a better (lower) miss rate than when the SA cache is used as both L1 cache and L2 cache (a decrease of 6.8%)**.

5.5.3 Global L2 cache miss rate

Figure 64 shows the global L2 miss rate of the four configurations for all the benchmarks, and Table 40 summarizes on average the global L2 miss rate increase of the four configurations relative to l1d.SA-l2.SA. Using Newcache as L2 cache incurs large global L2 miss rate compared with SA cache, no matter whether the L1 data cache is a SA cache or Newcache. **Again, using Newcache as only the L1 D-cache has a slightly better global L2 miss rate (by 1.9%).**

5.5.4 Overall Performance in IPC

From Figure 65 and Table 41, on average, the IPC for l1d.new-l2.new is degraded by about only 2.7% despite the significantly increased L2 miss rate. **Again, using Newcache as only the L1 D-cache results in almost the same overall performance in IPC, compared to the using set-associative caches for both L1 D-cache and L2 cache.**

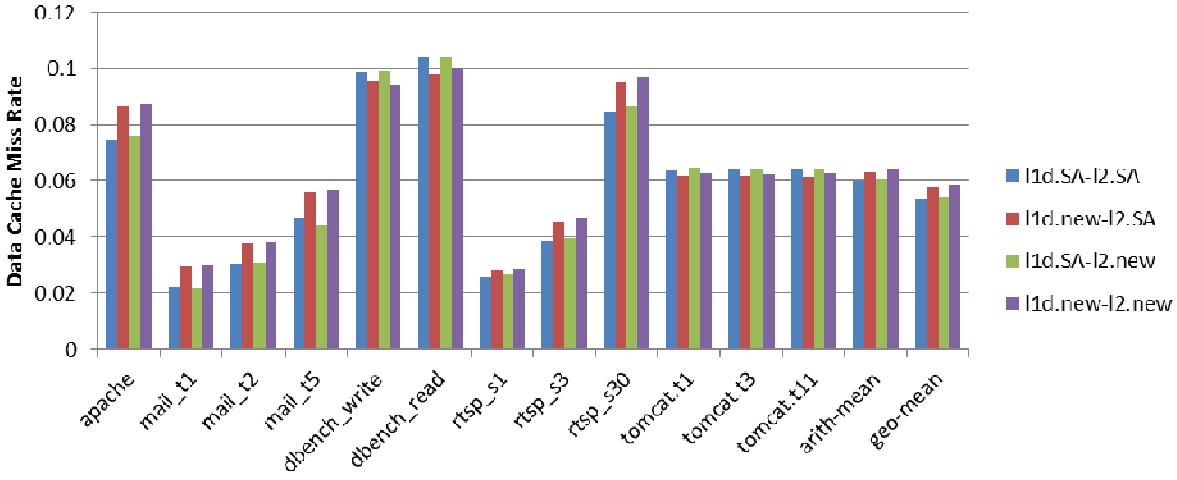

Figure 62 Newcache as both L1 data cache and L2 cache: data cache miss rate

Table 38 Newcache as both L1 data cache and L2 cache: increase of data cache miss rate

	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%

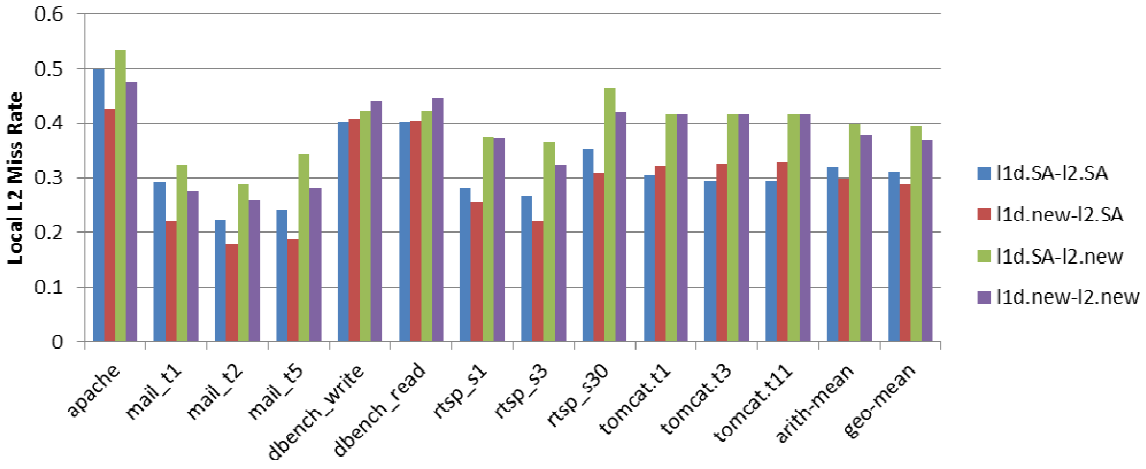

Figure 63 Newcache as both L1 data cache and L2 cache: local L2 cache miss rate

Table 39 Newcache as both L1 data cache and L2 cache: increase of local L2 miss rate

	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%

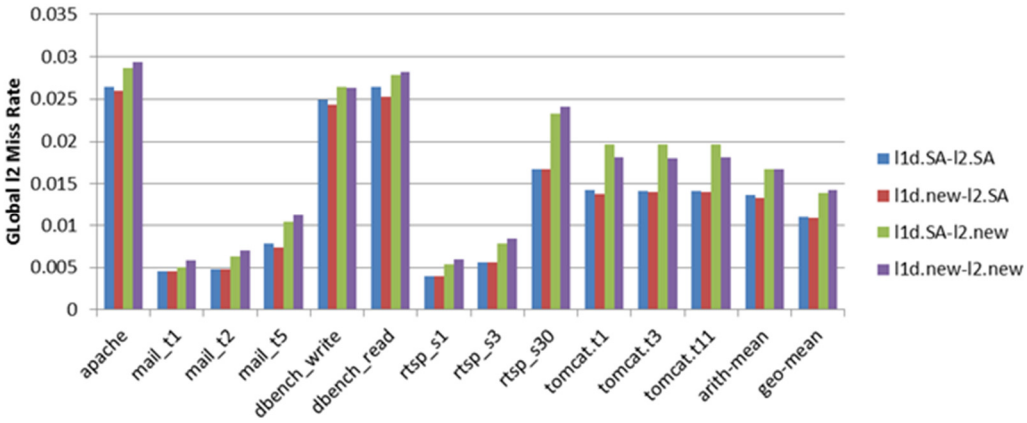

Figure 64 Newcache as both L1 data cache and L2 cache: global L2 cache miss rate

Table 40 Newcache as both L1 data cache and L2 cache: increase of global L2 cache miss rate

	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%

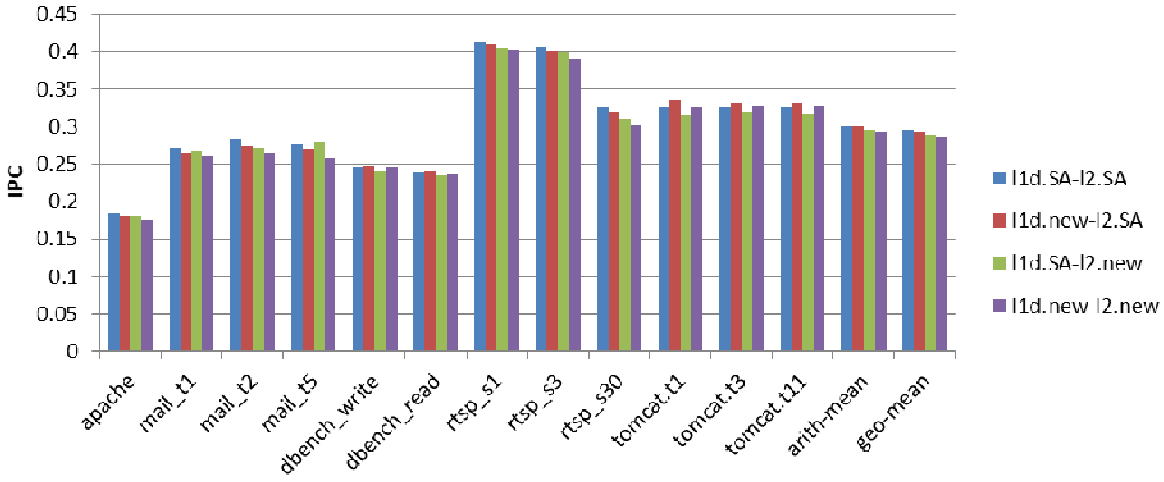

Figure 65 Newcache as both L1 data cache and L2 cache: IPC

Table 41 Newcache as both L1 data cache and L2 cache: increase of IPC

	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%

5.5.5 Summary of Performance results for Newcache used as both L1 D-cache and L2 cache

Similar to the results obtained in the last section, when Newcache is used as the L2 cache, there is significant increase of L2 cache miss rate, relative to conventional SA caches used for both the L1-D-cache and the L2 cache. However, because of the very low L2 miss rates to start with, this degradation has very little impact on the overall performance in IPC. The overall performance is still within 3% degradation.

When Newcache is used only as the L1 D-cache, there is a 5.25% performance degradation for L1 D-cache miss rate, but there is an improvement in both the local L2 miss rate and the global l2 miss rate, and no impact on overall performance in IPC. This jives with the results in section 5.3.3 .

5.6 Performance for Newcache used as L1 Instruction Cache

In this subsection, we study the system performance for Newcache used as the L1 instruction cache by measuring the server side's IPC, instruction cache miss rate, and local L2 instruction miss rate for all the benchmarks, using the following 6 different cache configurations:

1) All.SA.base: all L1 instruction cache, data cache and L2 cache are SA caches;
2) L1i.SA.base: SA cache for L1 instruction cache, Newcache for L1 data cache and L2 cache (same size as SA cache and nebit k=4);
3) L1i.new.nebit3: Newcache as instruction cache (k=3), Newcache for data cache and L2 cache (k=4);
4) L1i.new.nebit4: Newcache as instruction cache (k=4), Newcache for data cache and L2 cache (k=4);
5) L1i.new.nebit5: Newcache as instruction cache (k=5), Newcache for data cache and L2 cache (k=4);
6) L1i.new.nebit6: Newcache as instruction cache (k=6), Newcache for data cache and L2 cache (k=4)

The L1 instruction cache fetches Intel x86 instructions in cache lines, rather than data in cache lines as in the L1 D-cache. 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 and instruction references tend to have smaller memory footprint. Also, instruction references have better locality than the data references since programs tend to execute instructions in sequence, if the current executing instruction is not a taken branch.

5.6.1 Instruction cache miss rate

Figure 66 and Table 42 show the instruction cache miss rate results. When Newcache is used for the L1 I-cache, L2 D-cache and L2 cache, a moderate increase in I-cache miss rate (similar to that for Newcache used as a data cache) is seen (between 4-6.6% degradation compared to the All.SA base configuration). The instruction cache miss rate decreases (improves) as the Newcache nebit increases from k=3 to k=6.

When the instruction cache is a SA cache, but Newcache is used for the L1 D-cache and L2 cache, then only a slight 1.3% degradation is observed for the L1 instruction cache miss rate.

5.6.2 Global L2 cache miss rate for instructions

Figure 67 and Table 43 show the global L2 instruction miss rate results. Similar to what we have analyzed in previous sections, because the test uses Newcache as L2 cache, it will bring big negative impact on L2 cache's performance. On average, L2 cache as Newcache increases the global L2 instruction miss rate by about 40% compared with L2 cache with normal SA cache. However, this is a large percentage degradation of a very small number (0.02 global L2 miss rate for All.SA base configuration).

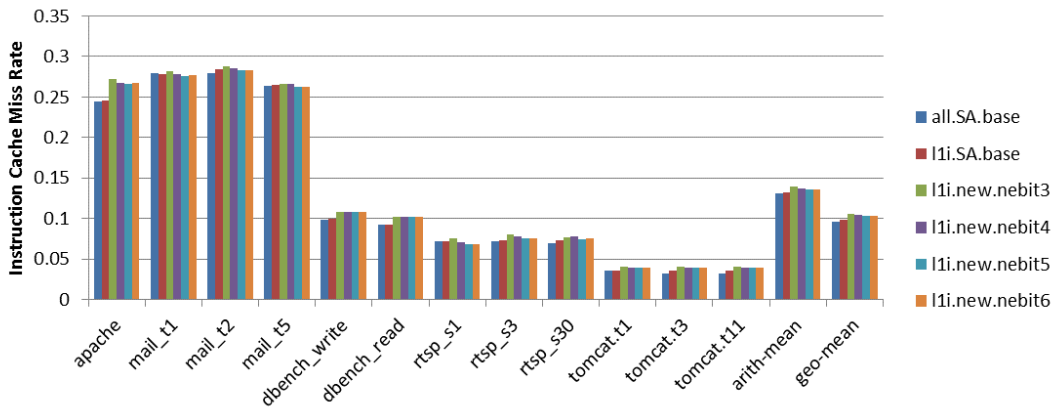

Figure 66 Newcache as instruction cache: instruction cache miss rate

Table 42 Newcache as instruction cache: increase of instruction cache miss rate

	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%

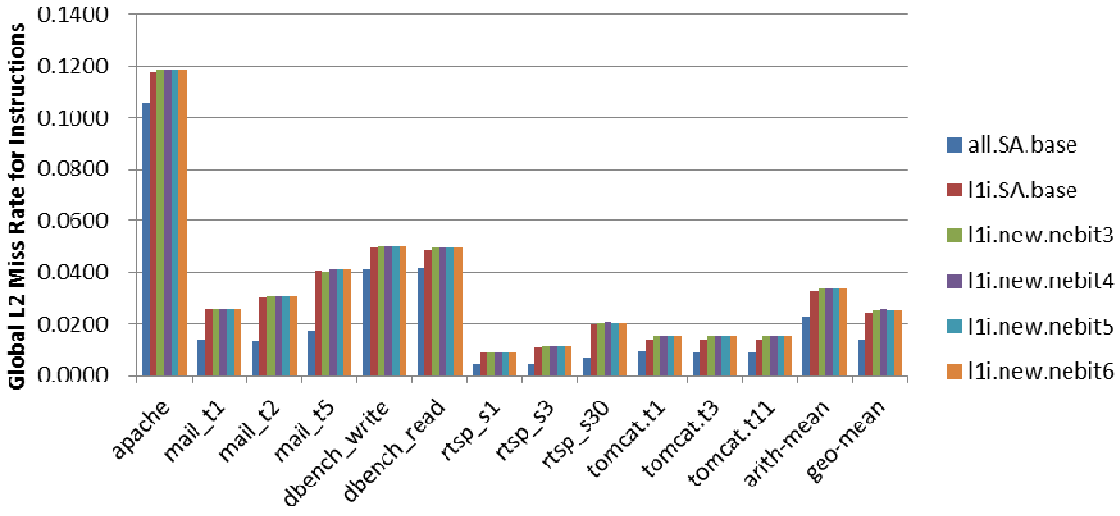

Figure 67 Newcache as instruction cache: global L2 miss rate for instructions

Table 43 Newcache as instruction cache: increase of global L2 miss rate

	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%

5.6.3 Overall Performance in IPC

In spite of the increased L2 cache miss rate, the overall performance in IPC is only slightly degraded by about 3%., as shown in Figure 68 and Table 44.

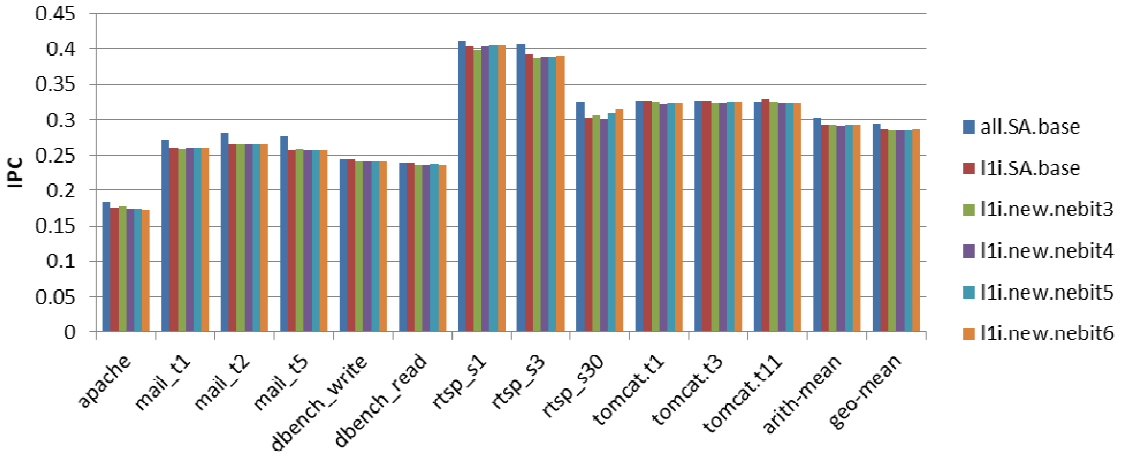

Figure 68 Newcache as instruction cache: IPC

Table 44 Newcache as instruction cache: increase of IPC

	all.SA .base	l1i.SA .base	l1i.new .nebit3	l1i.new .nebit4	l1i.new .nebit5	l1i.new .nebit6
Average	0.3016	0.2933	0.292052	0.2914	0.2927	0.293008
Increase relative to all.SA.base	0.00%	-2.73%	-3.15%	-3.39%	-2.95%	-2.84%

5.6.4 Summary of Performance results for Newcache used as the L1 I-cache

The overall performance in IPC is only slightly impacted by about 3% when Newcache is used for all the L1 I-cache, L1 D-cache and L2 unified cache, despite large relative increases in the L2 cache miss rate and the moderate increase in I-cache miss rate. It is possible that Newcache used as L2 cache causes the larger cache miss rates. Hence, we next study the performance when the L2 cache is a conventional SA cache.

5.7 Performance for Newcache used as both L1 Instruction and L1 Data Cache

In this subsection, we let L2 Cache remain a SA cache with the parameters as in Table 25, but use Newcache for both the L1 I-cache and L1 D-cache. We test the server performance for all the server benchmarks under the following 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)

We want to see the performance of Newcache as I-Cache, with D-Cache set to be Newcache, but L2 Cache still fixed as the conventional SA cache.

5.7.1 ICache Miss Rate

Figure 69 and Table Table 45 show the I-Cache Miss Rate results. On average Newcache as both I-Cache and D-cache induces about 3% more I-Cache Miss Rate than that of the all.SA.base case. Still, the random eviction policy does pose some negative effects on the I-Cache 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 I-Cache almost induces no penalty on I-Cache Miss Rate.

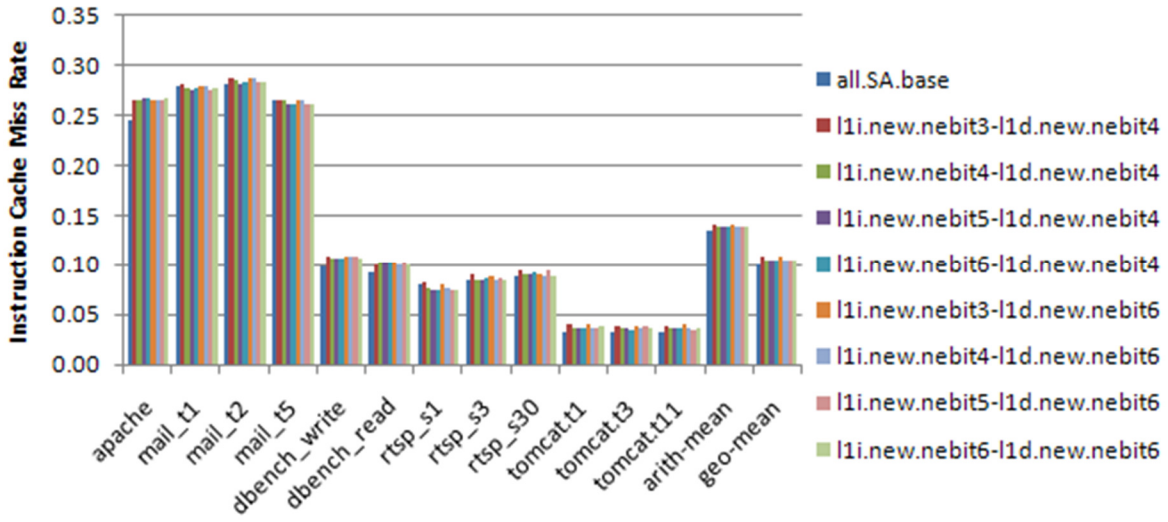

Figure 69 Newcache as both instruction and data cache: instruction Cache Miss Rate

Table 45 Newcache as both instruction and data cache: increase of instruction cache 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
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%

5.7.2 Local L2 Miss Rate for Instructions

Figure 70 and Table Table 46 show that the Local L2 Miss Rates for Instructions for most benchmarks are much lower with Newcache used as I-cache (by about 7%). Actually, the higher L1 I-Cache 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. This may explain why we have lower L2 Miss Rate for l1i.new.nebit3-6 cases.

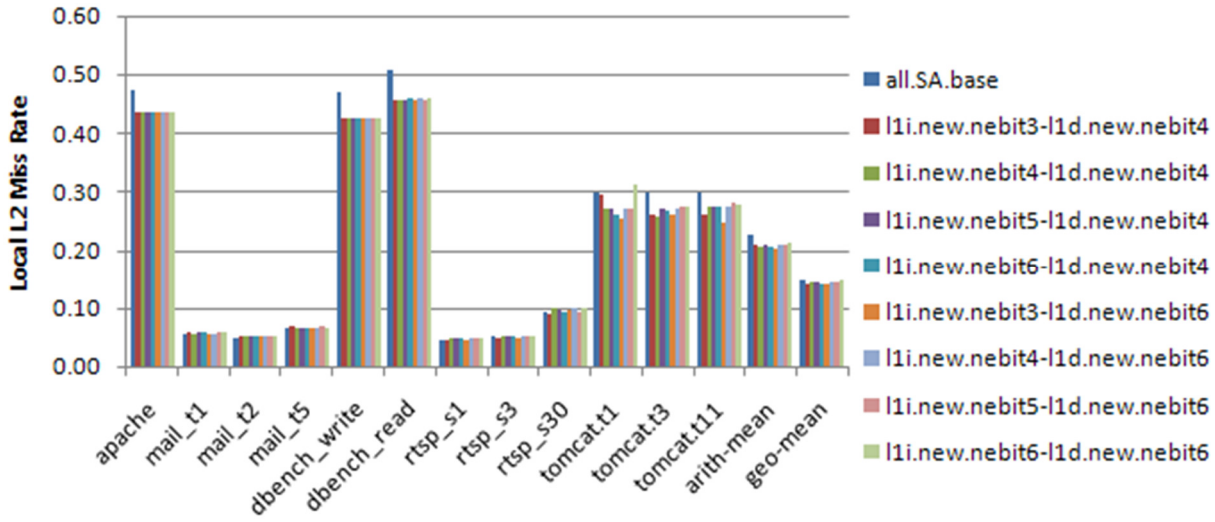

Figure 70 Newcache as both instruction and data cache: Local L2 Instruction Miss Rate

Table 46 Newcache as both instruction and data cache: increase of 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
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%

5.7.3 Global L2 Miss Rate for Instructions

Figure 71and Table 47 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.

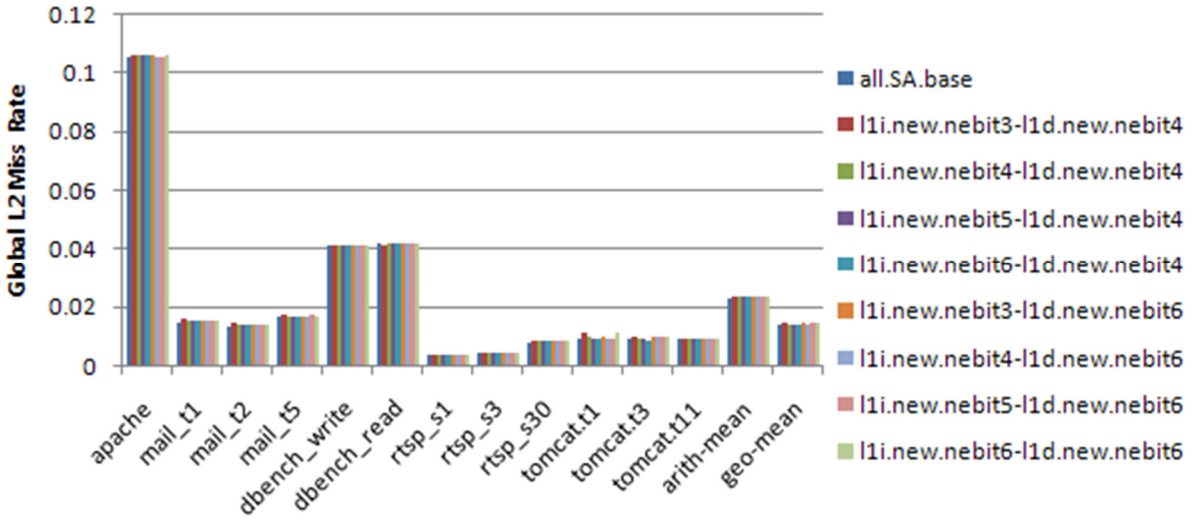

Figure 71 Newcache as both instruction and data cache: Global L2 Instruction Miss Rate

Table 47 Newcache as both instruction and data cache: increase of 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
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%

5.7.4 Overall Performance in IPC

According Figure 72 to and Table 48, 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 does not benefit from increasing the nebit value of I-Cache. Since instructions tend to be fetched from the same page (virtual and physical), increasing the LDM cache size does not influence IPC.

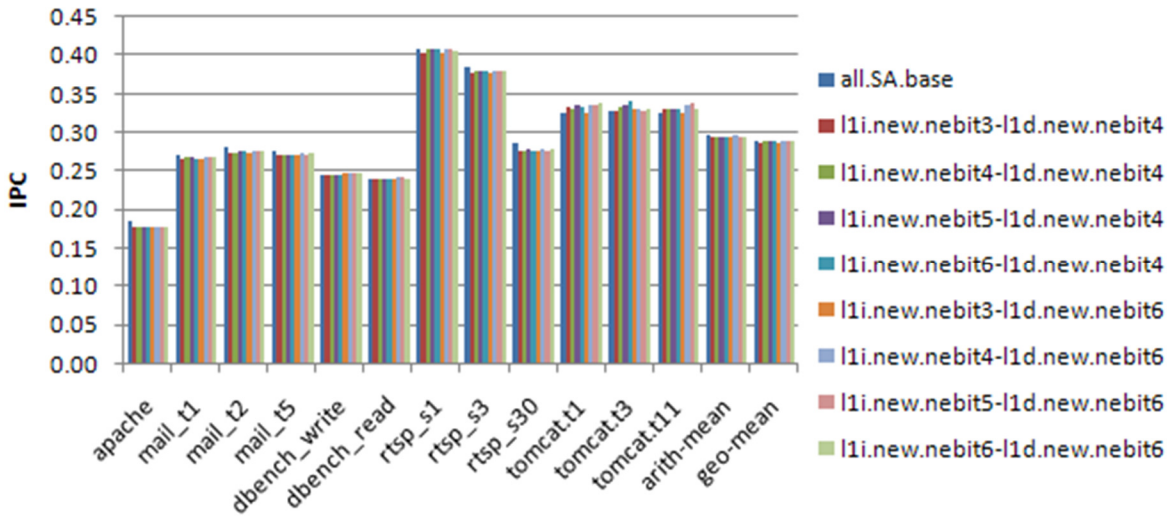

Figure 72 Newcache as both instruction and data cache: IPC

Table 48 Newcache as both instruction and data cache: increase of 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
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%

5.7.5 Summary of Performance results for Newcache used as L1 I-cache and D-cache

The overall performance in IPC is essentially not impacted (less than 1%) when Newcache is used for both the L1 I-cache and L1 D-cache, but not for the L2 unified cache, when compared to the case when all caches are SA caches. This is better overall performance than when the L2 cache was also Newcache, where the IPC degradation was 3% There is very slight impact on the global L2 miss rate (1-2% degradation) while the local L2 miss rate improved by about 7-8%, compared to the all SA configuration. When the nebit is greater than 3, the I-cache miss rate degraded by only 2-3.5%. These instruction cache and L2 cache miss rates are better than when the Newcache is also used as the L2 cache.

Newcache used as the L1 instruction cache has overall performance comparable to SA cache. When the L2 cache remains as a set-associative cache, the L1 and L2 cache miss rates are also not significantly degraded. Therefore, Newcache is a better candidate for L1 caches (either L1 instruction cache or data cache) than replacing the L2 cache, in terms of system performance for cloud benchmarks.

6. Testchip Design and Evaluation

6.1 Circuits

As mentioned above, the Newcache requires a fully associative structure for the LNreg portion of the cache. In hardware, such a structure is typically built using a contents addressable memory (CAM) structure using custom cells with an embedded comparison structure. However, designers often eschew CAM designs due to the power and latency overhead of the search operation and the larger cell size required for the embedded comparison circuits. However, as the CAM functionality significantly enhanced the security of the cache design, we endeavored to design the LNreg CAM to mitigate the conventional CAM deficiencies and overheads, the block diagram is shown in Figure 73.

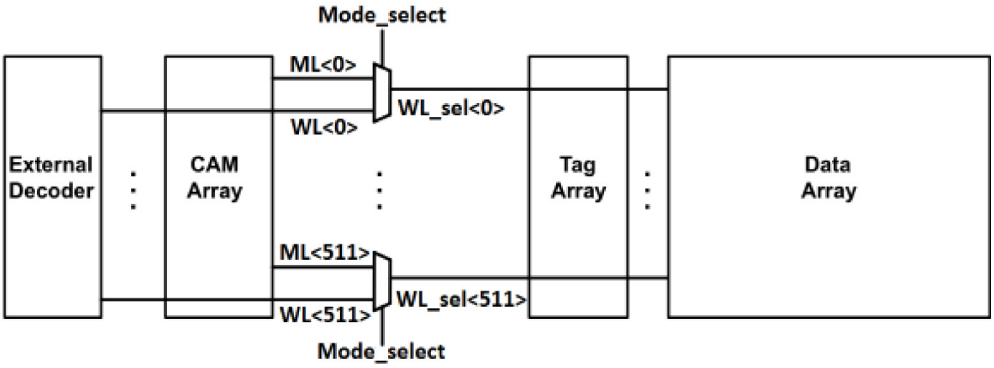

Figure 73 Memory access mode block diagram.
Data and Tag arrays can be accessed by either external decoder or CAM match operation. Mode select signal is used to select between decoded WLs from the external decoder and MLs from CAM.

CAMs are classified as either NOR or NAND style depending on the arrangement and logical function of the embedded comparision circuits. Typically NOR style CAMs have a lower search latency, but increased power consumption due to most (or all on a miss) matchlines charging and discharging during each search operation. As our LNreg CAM would have a large number of entries, a NOR style CAM would have too high of a power overhead. Additionally, with velocity saturation occuring in modern deep sub-micron transistors, the speed advantage of the NOR style over the NAND style is relatively small. Thus, we chose to use a NAND style topology/cell for the LNreg.

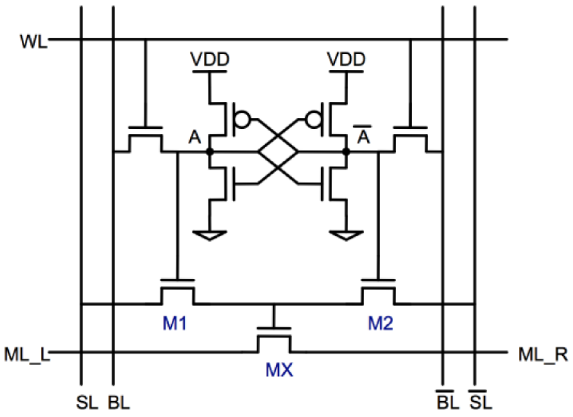

Figure 74 Schematic of 9T CAM cell using NAND-style matchline (ML).

The 9-transistor CAM cell we chose to use is shown in Figure 74**Error! Reference source not found.**, where the cell VDD is boosted to avoid a degraded level at the gate of MX. The cell uses a modern uni-directional poly layout for enhanced manufacturability and compatibility with deep sub- micron process design rules. Additionally, the modern "short and fat" layout aspect ratio allows enough room to run the search lines and bitlines separately. This reduces the loading on each, and thus improves the power and latency of search, read, and write operations without increasing the cell area. The NAND string device (MX) would ordinarily receive a degraded level at its gate, reducing its performance, but we boost the Vdd of the CAM cells to reduce the search latency. Boosting of the CAM cell Vdd requires a slightly higher Vdd for the cells, but this is typically available due to the higher I/O voltage and some SRAM designs already use a higher Vdd for increased cell stability.

Even with the boosted Vdd and velocity saturation effects, the 17b LNreg match could not be built as a single NAND string and still meet performance requirements. Thus, we use a hierarchical matchline, splitting the 17b into one string of 9b and another of 8b. These two matchlines are combined using a simple static CMOS NAND gate as shown in Figure 75. Additionally, the hierarchical matchline reduces the power, since in some mismatch cases, only one side of the matchline will discharge.

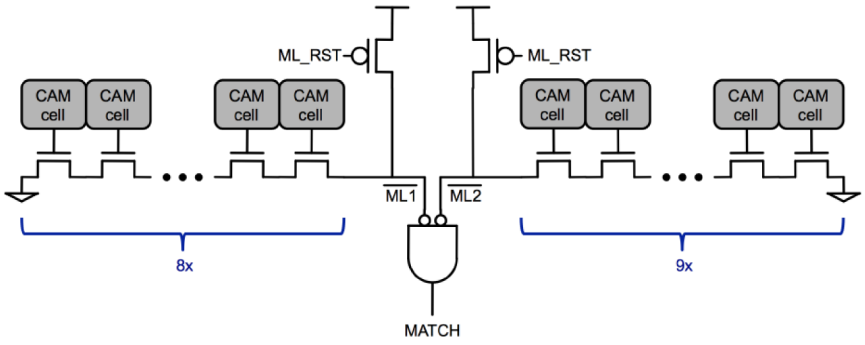

Figure 75 Hierarchical matchline with combining static NAND gate.

Besides the matchlines, another significant source of power dissipation in CAMs is the vertical search lines that drive the key value being searched for to all the cells. These are heavily loaded lines that span the entire height of the CAM. We reduce the search line loading as much as possible by splitting the search lines from the bitlines, as discussed above. Further, we use statically driven search lines for all but the topmost (closest to NAND gate) lines to reduce search line toggling. The topmost two search line pairs are dynamically driven to simplify the clocking of the CAM and reduce the search latency.

If all SLs were driven statically, then we would require a control signal to activate the dynamic NAND strings, similar to the evaluate signal sent to the footer device of conventional dynamic logic gates. This would need to be timed (with additional timing margin) against the delay of driving in the SL value, which would increase the search latency due to the needed timing margin to be safe against variability and skew. However, with the topmost SLs being dynamically driven, we enable flow through timing, and do not require any additional control signals or timing margining.

6.2 Testchip

We implemented a 32kB direct mapped Newcache (NC) and a 32kB conventional 8-way set associative (SA) cache on a prototype testchip in a 65nm Bulk CMOS 7-metal copper process. The 8-way SA cache provides a baseline conventional design to compare the Newcache version against. The die microphotograph is shown in Figure 80. The prototype Newcache consists of three memory arrays: a 512 x 17b LNreg CAM, a 512 x 15b Tag array, and a 4k x 64b Data array. The Data array is split into two 512 x 256b arrays with the CAM and Tag in the middle to mitigate the wordline RC. SRAM and 9T CAM cell areas are 1.1 µm2 and 2.7 µm2, respectively. The corresponding area utilizations for Data, Tag, and CAM arrays are 40.2%, 31.3%, 19%, respectively.

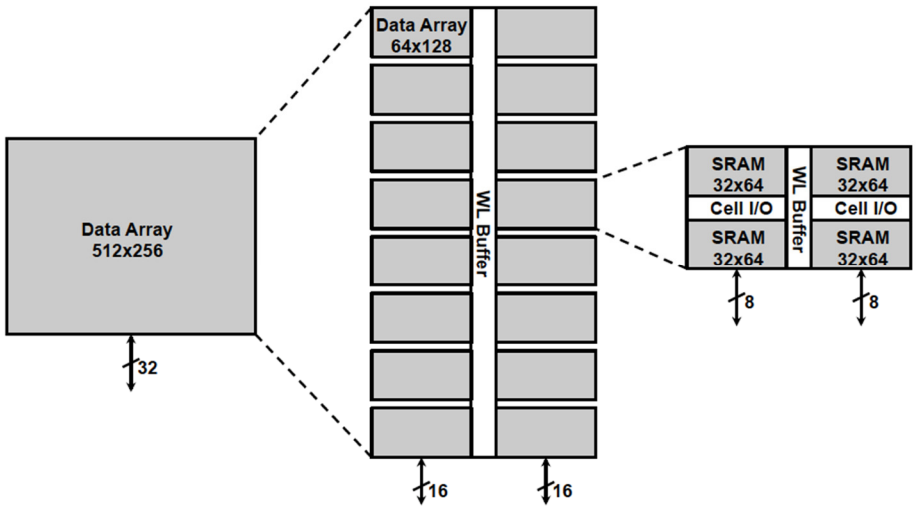

Figure 76 Array sub-blocking for Newcache and conventional cache data arrays.
Diagram shows one of two data array blocks in each cache.

Array partitioning is used to achieve high-speed and low- power memory operation (Figure 76). There are eight 64 x 256b partitions and each partition row consists of two 64 x 128b sub-blocks. Each sub-block is further divided into four 32 x 64b cell arrays. We use 8:1 column multiplexing to improve the speed and aspect ratio of the memory. Hence, each sub-block provides 8b data. Similar array partitioning is used for CAM and Tag arrays except for the 8:1 column multiplexing. Both CAM and Tag arrays are divided into eight 64-row sub-blocks with 17b and 15b -wide columns, respectively.

The short bitlines (BLs) allow us to use full swing signaling for reads, thus eschewing the need for large sense amplifiers and complex sense timing. Each sub-block output is connected to a global bitline (GBL) to get the selected data out. Array partitioning significantly decreased the BL and WL length for each sub-block, increasing the overall memory performance and decreasing power consumption. Apart from the performance and power gains, such aggressive array partitioning enabled us to re-use most of the peripheral circuitry (i.e., cell I/O, wordline drivers, etc.) across all the memory blocks, which significantly decreased the design complexity of the system.

The central row decoder is hierarchical using two 3:8 static pre-decoders and 64 final row-decoders. The first 6 bits of the address are sent to row decoder to activate one of the 64 WLs. The remaining 3b are sent to a 3:8 static decoder to generate eight block-select (BS) signals, which are used to activate one of the eight 64-row sub-blocks.

	Area (mm^2)	
	32kB Newcache	**32kB 8-way SA conventional cache**
Data array	0.76 (80%)	0.80 (93%)
Tag array	0.03 (3%)	0.06 (7%)
LnReg CAM	0.14 (15%)	NA
Cache total	0.95	0.86
Built-in self-test	0.15	0.13

Figure 77 Newcache and conventional cache area breakdown.

A custom test board is designed to test the chip. A cavity up ceramic pin grid array package is used for the test chip. National Instruments Data Acquisition (NI-DAQ) system is used for both sending the inputs to the chip and sampling the outputs from the chip. The test board also has level shifters to convert 5V signals that are required by the NI-DAQ interface to 2.5V that is I/O pad voltage of the chip.

Built-in self-test (BIST) circuits are used for testing cache design at speed. Each BIST circuit has scan-enabled shift registers that provide input data and capture output data. The input registers can hold 16 commands (e.g., memory read, memory write, and match operation) and are configured to operate in a

circular manner to continuously provide input data, and the output registers store the most recently processed 16 commands.

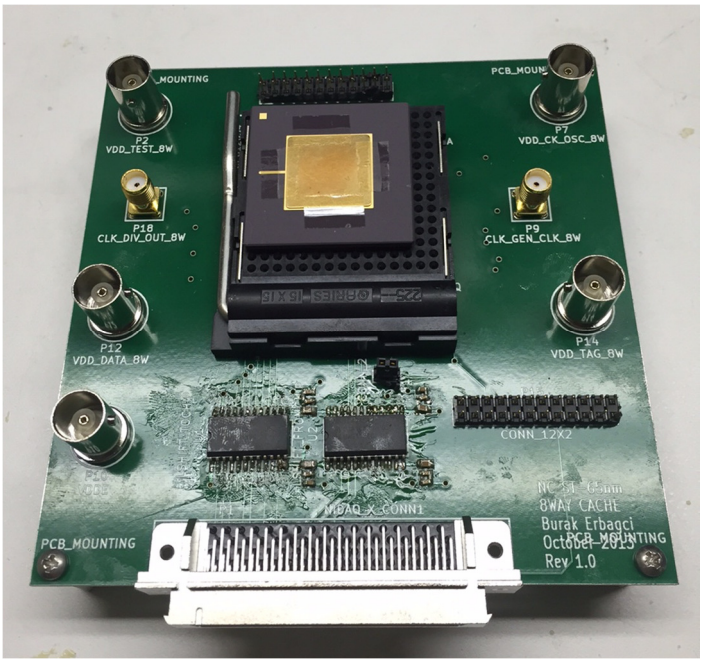

Figure 78 PCB used for testing of Newcache testchip.

There are two test PCBs (as shown in Figure 78), one for testing the Newcache and one for testing the conventional 8-way set associative cache. The 8-way board is shown. The testchip is packaged in a PGA package and socketed into the board. Also shown are the level converters (two chips on the bottom) to interface the workstation to the PCB via the right-angle connector on the bottom of the board.

The following test procedure is used for cache sub blocks (i.e., Data, Tag, and CAM). Multiple tests are run to verify functionality at each memory address, since BIST can hold 16 commands. First, BIST input registers are filled with 8 write and 8 read commands. Supply voltage and operating frequency are swept; and for each voltage-frequency point the functionality is verified. Then, the write and read power are measured at the functional voltage-frequency points. For write operation, the BIST input registers are filled with 16 write commands. Supply voltage and operating frequency are swept; and the average write power for each voltage-frequency point is measured. For read operation, the BIST input registers are filled with 8 write and 8 read commands. Supply voltage and operating frequency are swept; and the average power for each voltage-frequency point is measured. Then, the average read power is back calculated by these measured power values and average write power values. For write and read operations, various input test patterns are used. These input test vectors include alternating patterns (i.e., A's, 5's), all 1's and 0's, and some random patterns.

Process	65nm bulk CMOS 7 Metal Cu
Nominal supply	1.0V
NC CAM area	0.14 mm^2
NC Data memory area	0.76 mm^2
NC Tag memory area	0.03 mm^2
Total NC area	0.95 mm^2
Die area	4 mm^2
Operating supply range	0.8 - 1.2V

Figure 79 Newcache implementation details.

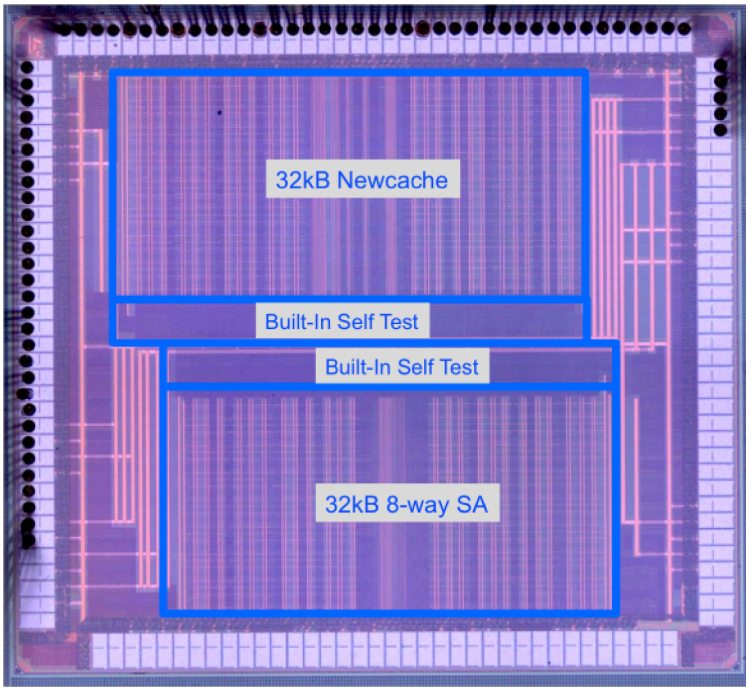

Figure 80 Die microphotograph of the 2mm x 2mm Newcache testchip in 65nm bulk CMOS. The chip has 144 I/O pads.

6.3 Experimental results

We began by testing each memory component (CAM, Tag, Data) of Newcache individually. At the nominal 1.0V VDD, Data, Tag and CAM memories operate at 525, 650 and 570 MHz respectively. Schmoo plots for these memories are shown in Figure 81 (a), Figure 82 (a) and Figure 83(a), respectively. In these figures, the Green area represents the voltage-frequency points in which the memory is functional.

In Figure 81 (a), we see that Data memory operates at 420-600 MHz across a supply voltage of 0.8-1.2V. In Figure 82 (a), we see that Tag memory operates at 500-800 MHz across a supply voltage of 0.8-1.2V. In Figure 84 (a), we see that CAM operates at 420-700 MHz across a supply voltage of 0.8-1.2V.

Read and write power measurements for Newcache memory components are shown in Figure 81 (b), Figure 82 (b) and Figure 83(b).

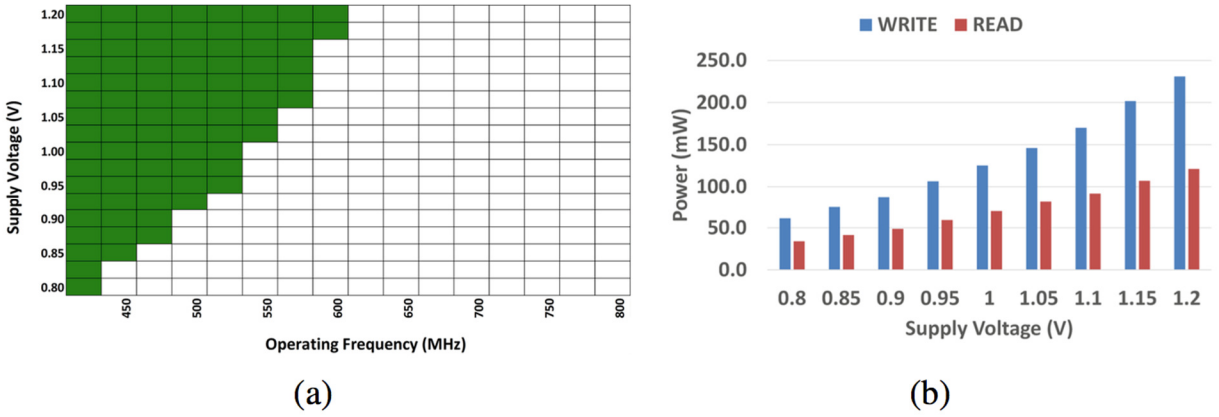

Figure 81 (a) Schmoo plot of the Newcache Data memory. (b) Read and write power measurements for Data memory.

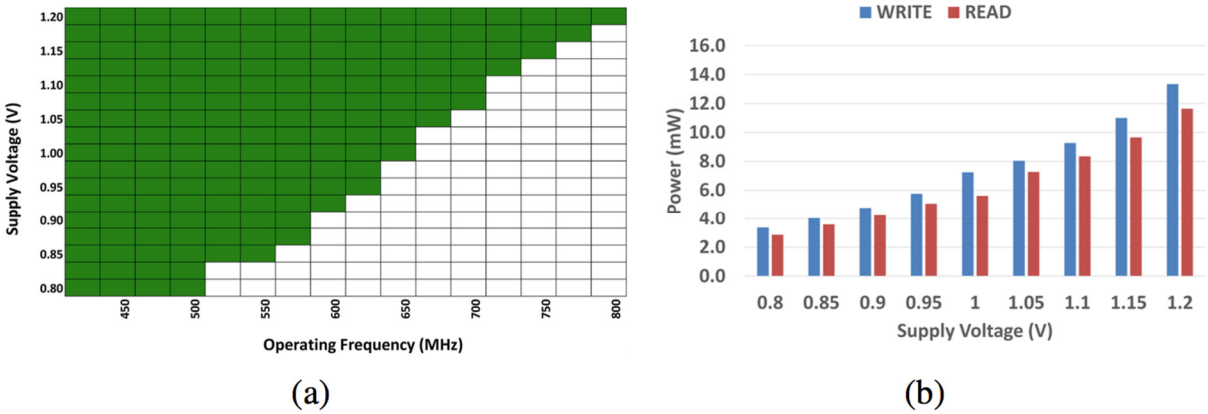

Figure 82 (a) Schmoo plot of the Newcache Tag memory. (b) Read and write power measurements for Tag memory.

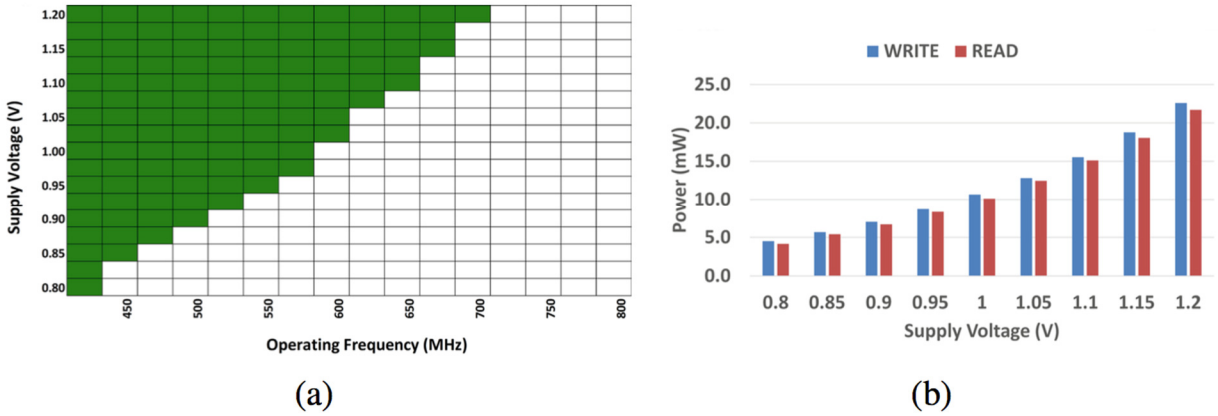

Figure 83 (a) Schmoo plot of the Newcache CAM. (b) Read and write power measurements for CAM.

The cycle time of Newcache is limited by the Data array. Hence, the Newcache operating frequency is set by the Data and all blocks are operated at the same frequency. Read and write power measurements for all Newcache memory components when accessed by CAM match operation are shown in Figure 84 (a) and (b). The power overhead for Newcache over the 8-way SA cache is small (20%) since the majority of power is consumed by Data.

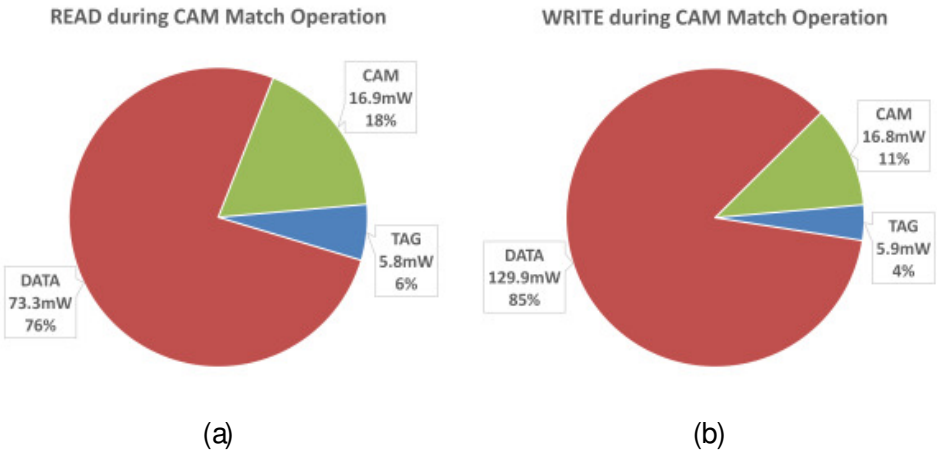

Figure 84 (a) Read and (b) write power breakdown for Newcache at 1.0 VDD, 500MHz, and room temperature.

	Area (mm^2)	Frequency @ 1V	Cache read power @ 500 MHz (mW)	Cache write power @ 500 MHz (mW)
Newcache 32kB Direct Mapped	0.95	500 MHz	96	152.6
Conventional 32kB 8-way SA	0.86	500 MHz	82.5	123
Newcache overhead	10%	-	16.3%	24%

Table 85 Newcache versus conventional 8-way set associative cache overhead summary.

The critical path of the design was in the data array, and thus the Newcache CAM did not limit the performance of the design. However, the data array access time was longer than the simulated value likely due to a speedpath in the data array decoder. Future designs would seek to eliminate the speedpath using device sizing or logic alterations. The overall overhead of the Newcache over the conventional design was still quite modest, especially when taking into account the fact that the L1 cache would only account for a small percentage of the overall microprocessor area and power.

With the testchip, we have demonstrated a secure Newcache design with randomized replacement that was implemented using novel circuits to mitigate the disadvantages of using a small CAM-based decoding scheme. The prototype testchip contained both a conventional 8-way SA cache and a Newcache. As shown in Table 85, measured results show only small overheads in area, power, and delay for using this technique for securing the cache against software-based cache timing side-channel attacks.

7. Conclusions

We have done a thorough evaluation of Newcache, a secure cache that is resilient to cache-based side-channel attacks. Our evaluation was along three dimensions: security, system performance and physical characteristics (cache access time latency, power for reads and writes, and area).

Security:

For security, we developed a new, comprehensive classification of cache side-channel attacks targeting the fastest information leakage through Level 1 instruction and data caches. We developed the first cache side-channel attack suite, including constructing some new attacks and attacks specifically targeting Newcache's architecture. We performed security testing of Newcache versus conventional set-associative caches with this attack suite.

We have shown, using concrete real attacks specifically targeting Newcache, that Newcache can completely defeat all the contention based attacks and Flush-Reload attacks, for both D-cache and I-cache. This assumes that the RMT_ID and P-bit are correctly assigned to represent different trust domains, both across processes and within a process. The only attack Newcache eventually succumbs to is the cache collision attack. However, Newcache makes it one order of magnitude harder for the cache collision attack to work compared to set-associative caches.

We have also found a new defense for cache collision attacks. These reuse-based attacks can be defeated by modifying the cache controller using a Random Fill strategy [31]. Hence, Newcache combined with a Random Fill cache controller [31] can defeat all cache side-channel attacks.

Performance:

We performed extensive performance evaluation using both smartphone benchmark suites and a cloud computing benchmark suite that we developed. We tested Newcache used as a L1 D-cache, a L1-I-cache, as a L2 cache, and various combinations of these.

While there may be some impact on cache miss rates, the overall performance impact (in Instructions per Cycle, IPC) of substituting Newcache for an L1 or L2 cache is negligible.

When Newcache is used as the L1 data cache (with L1 instruction cache and L2 cache as conventional set-associative caches):

- For smartphone benchmarks, there is negligible impact on overall performance (IPC). Newcache incurs a moderate increase in L1 data cache miss rate compared to a conventional 4-way SA cache. It has negligible impact on the L2 cache miss rate. But since the smartphone benchmarks are not memory-intensive, the moderate increase in L1 data cache miss rate has negligible impact on the overall system performance. There is a noticeable improvement in cache miss performance when the extra index bits in Newcache is increased from k = 3 to 4, but not much improvement beyond k = 4 bits.

- For server benchmarks, there is negligible impact (or slight improvement) in overall performance. Newcache incurs moderate increase in L1 data cache miss rate compared to an 8-way SA cache, but it may potentially ***reduce*** L2 cache miss rate. The overall performance is almost unchanged. When the number of extra index bits in Newcache is increased to k = 6, we find significant reduction of L1 cache miss rate and Newcache may increase the overall performance by more than 2%.
- For both smartphone and server benchmarks, we find that Newcache with a larger cache size (e.g., 64 KB) incurs less L1 data cache miss rate increase relative to the 4-way or 8-way SA cache of the same size (64 KB).

When Newcache is used as the L2 cache, using the server benchmarks, there is negligible impact on overall performance in Instructions executed Per Cycle (IPC). Although Newcache as L2 cache incurs a relatively large increase of both local and global L2 cache miss rate, this has negligible impact on overall performance, because the global L2 cache miss rate is very small – about 0.1%.

When Newcache is used as an instruction cache, we get similar performance results as when it is used as an L1 data cache.

Since there are cache side-channel attacks against both the L1 instruction and data caches, we are interested in the system performance when Newcache is used for both these caches. We find that the overall performance in IPC is essentially not impacted (less than 1%) when Newcache is used for both the L1 I-cache and L1 D-cache, but not for the L2 unified cache, when compared to the case when all caches are SA caches. This is better overall performance than when the L2 cache is also Newcache, where the IPC degradation was 3%. There is very slight impact on the global L2 miss rate, while the local L2 miss rate improved by about 7-8%, compared to the all SA configuration. These L2 cache miss rates are better than when Newcache is also used as the L2 cache.

Physical Characteristics

Our testchip design showed that Newcache has small overheads compared to an 8-way set-associative cache of the same size. When both caches are 32 Kbytes, Newcache is about 10% larger in our design, which is less than 1% for a processor core. The cache access time latency was limited by the data memory common to both our Newcache and our set-associative cache in our testchip, and not by the new CAM memory of Newcache. Future designs can improve the access time latency of the data memory for both caches. The power measured for cache reads (16%) and writes (24%) at 1V and 500 MHz was larger than we obtained from the layout extraction (0% reads, 3% writes). Future work can further optimize the power.

In summary, we have verified that Newcache can perform as well as set-associative caches, on average, for both smartphone and cloud computing benchmarks. The testchip shows the design is feasible, although currently, there remains an overhead for power, which can be further optimized in future work. Most importantly, it is very resilient against all known cache side-channel attacks, while no existing cache can claim that.

Hence, we recommend Newcache for use as both a L1 I-cache and a L1 D-cache. The L2 cache can remain the conventional SA cache. First, no L2 cache attacks have been found, and L1 cache attacks are much faster than any hypothetical L2 cache attacks. Second, the L2 cache misses can be significantly increased (40%) when Newcache is used as the L2 cache (with Newcache used as the L1 instruction and/or data cache), relative to a conventional set-associative L2 cache. Third, because the L2 cache is much larger than the L1 cache, using the same circuit design for a Newcache L2 as for a Newcache L1 cache would incur more power, or otherwise would require additional lookup table storage.

8. Project Publications

Except for the first publication below, upon which this project started, the rest of the publications were produced for this project's goals for a secure cache that is resilient to software cache-based side-channel attacks.

[1] Wang, Z., Lee, R.B., "A Novel Cache Architecture with Enhanced Performance and Security", Proceedings of the *41st. Annual IEEE/ACM International Symposium on Microarchitecture (Micro-41)*, pp. 88-93, December 2008.

[2] Liu, F., Lee, R.B., "Security Testing of a Secure Cache Design", in Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP), June 24, 2013.

[3] Liu, F., Lee, R.B., "Random Fill Cache Architecture", Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-47), Cambridge, pp. 203-215, December 2014.

[4] Zhang, T., Lee, R.B., "New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels", Proceedings of Annual Computer Security Applications Conference (ACSAC), December 2014, pp. 96-105, 2014.

[5] Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B., "Last-Level Cache Side-Channel Attacks are Practical", Proceedings of IEEE Symposium on Security and Privacy, San Jose, pp. 605-622, May 2015.

[6] Liu, F., Wu, H., Lee, R.B., "Can randomized mapping secure instruction caches from side-channel attacks?", in Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP), Portland, June 13, 2015.

[7] Fuchs, A., Lee, R.B., "Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs", Proceedings of the 8th ACM International Systems and Storage Conference, Haifa, Israel, May 2015.

[8] Yarom, Y., Ge, Q., Liu, F., Lee, R.B., Heiser, G., "Mapping the Intel Last-Level Cache", IACR Cryptology ePrint Archive, Report 2015/905, 2015.

[9] B. Erbagci, Fangfei Liu, C. Cakir, N. E. C. Akkaya, R. B. Lee, and K. Mai, "A 32kB Secure Cache Memory with Dynamic Replacement Mapping in 65nm bulk CMOS," Proceedings of IEEE Asian Solid-State Circuits Conference (A-SSCC 2015).

[10] Wu, H., "Performance Measurement and Security Testing of a Secure Cache Design", MSE Thesis, Electrical Engineering Department, Princeton, Princeton University, 2015.

[11] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser and Ruby B. Lee, "CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing", accepted to be published in the IEEE International Symposium on High Performance Computer Architecture (HPCA), March 2016, Barcelona, Spain.

[12] Liu, F., "Hardware based solutions for thwarting cache side channel attacks", PhD. Thesis, Electrical Engineering Department, Princeton, Princeton University, 2016. (In preparation),

9. Recommendations

Our recommendation is to urge all processor vendors to implement secure caches to prevent critical information leakage through cache side channel attacks. Where performance is important and running legacy code is important, we recommend Newcache to be used as both the L1 Instruction cache and the L1 Data cache. Our extensive performance studies show that this will not degrade system performance. In addition, this will defeat all known L1 cache side channel attacks, except for the cache collision attack which will be significantly mitigated – requiring at least one order of magnitude more effort on the attacker's part. The cache collision attack affects only the L1 Data cache (not the I-cache), and it can also be defeated if we use the Random Fill technique in the cache controller of the L1 D-cache. (The L1 I-cache does not need these changes to the cache controller.) **Hence, for the most secure solution, we recommend using Newcache for the L1 I-cache, and using Newcache with a Random Fill cache controller for the L1 D-cache.** The L2 and L3 caches can remain as a conventional SA caches.

In a cloud computing environment, cloud providers can ensure that Virtual Machines belonging to different customers are allocated and scheduled so that they do not share processor cores. Then, since both L1 and L2 caches are core-private, they will not be shared by a potential attacker and victim. However, the L3 cache (also called the Last Level Cache, LLC) is still shared by all cores, and LLC side-channel attacks have been demonstrated to be practical [5]. We have also found a solution to these LLC cache attacks that only require small changes to the OS and hypervisor [11]. Hence, we recommend these system-level LLC defenses be used for cloud servers, leveraging already introduced Intel hardware CAT technology for LLC QoS (Quality of Service).

The processor vendors are all interested in secure caches, such as Newcache, but they need to hear from customers that they want certain features, before they will implement these features, especially if it involves replacing such a well-honed component as a cache, which is so important for performance. Currently, they may also have their hands full implementing more basic security mechanisms than those for side-channel attack nullification. We have provided the hardware solution, Newcache, for the most performance-critical L1 caches. In time, it will be deployed in commodity processors. Greater urgency is required in high security usage scenarios like military uses, and we recommend that secure cache technology be implemented as soon as possible in all high security processors.

We recommend that the next step is to take a processor core, replace the I-cache and D-cache with Newcache, and subject the computer to extensive cache attacks, to test the resilience of these secure caches in the field.

10. Acknowledgements

We thank DHS for supporting this secure Newcache project, especially the DHS Program Manager Ed Rhyne, and AFRL for managing the contract, especially Robert DiMeo. The PI thanks Fangfei Liu of Princeton University for her many invaluable contributions to this project, which included detailed, comprehensive and insightful security and performance studies of Newcache and very innovative work on the defense for cache collision attacks and LLC attacks and defenses. We thank Prof. Kenneth Mai of Carnegie Mellon University for leading the testchip design team, consisting of both CMU and Princeton students, especially Burak Erbagci, Fangfei Liu, C. Cakir, N. E. C. Akkaya Burak. We also thank Princeton students Hao Wu and Tianwei Zhang for their useful contributions to the project.

11. References

[1] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in IEEE/ACM International Symposium on Microarchitecture (MICRO-41 2008), pp. 83-93, November 2008.

[2] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in IEEE/ACM International Symposium on Computer Architecture (ISCA 2007), p.494 – 505, June 2007.

[3] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in 22nd Annual Computer Security Applications Conference (ACSAC'06), p.473-482, December 2006

[4] F. Liu and R. B. Lee, "Security testing of a secure cache design," in Hardware and Architectural Support for Security and Privacy (HASP), June 24, 2013.

[5] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K., Sewell, M. Shoaib, N. Vaish, M. Hill, D. Wood, "The gem5 simulator," in ACM SIGARCH Computer Architecture News, vol. 39, issue 2, pp. 1-7, May 2011.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26]Gutierrez A, Dreslinski R G, Wenisch T F, et al. Full-system analysis and characterization of interactive smartphone applications[C]//Workload Characterization (IISWC), 2011 IEEE International Symposium on. IEEE, 2011: 81-90.

[27]0xbench benchmark suite. https://code.google.com/p/0xbench/

[28] Coremark benchmark: http://www.eembc.org/coremark/index.php

[29] Guthaus M R, Ringenberg J S, Ernst D, et al. MiBench: A free, commercially representative embedded benchmark suite[C]//Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on. IEEE, 2001: 3-14.

[30]Daniel J. Bernstein. 2005. Cache-timing Attacks on AES. Technical Report.

[31]Liu, F., Lee, R.B., "Random Fill Cache Architecture", Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-47), Cambridge, pp. 203-215, December 2014.

[32]Joseph Bonneau and Ilya Mironov, “Cache-Collision Timing Attacks against AES”, In Proceedings of Cryptographic Hardware and Embedded Systems (CHES’06). 201–215.

[33]Zhang Y, Juels A, Reiter M K, et al. Cross-VM side channels and their use to extract private keys[C]//Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012: 305-316.

[34]libsvm. http://www.csie.ntu.edu.tw/ cjlin/libsvm/.

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

[36]Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. 1992. A Training Algorithm for Optimal Margin Classifiers. In Proceedings of the Fifth Annual Workshop on Computational Learning Theory (COLT ’92). ACM, New York, NY, USA, 144–152. DOI:http://dx.doi.org/10.1145/130385.130401

[37]Corinna Cortes and Vladimir Vapnik. 1995. Support-Vector Networks. Mach. Learn. 20, 3 (Sept. 1995), 273–297. DOI:http://dx.doi.org/10.1023/A:1022627411411