Evading Malicious Code with Concurrent Programming in Parallel Architectures and Their Protection Methods

Caglar SAYIN

Masteroppgave
Master i Teknologi - Medieteknikk
30 ECTS
Avdeling for informatikk og medieteknikk
Høgskolen i Gjøvik, 2014

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Evading Malicious Code with Concurrent Programming in Parallel Architectures and Their Protection Methods

Caglar SAYIN

2007/05/30

Revision history

Version #	Description of change (why, what where - a few sentences)
0.1	First version is made available via Fronter

Abstract

Have you ever realize how secure and safe environment they claim in security product companies' advertisement? Does Anti-Malware software really provide 100 percent security against Malware? As Everybody knows, It is not. Today, it is nothing more than cat and dog fight. Malware authors purpose an new architecture, an new approach and Anti Virus companies just try to fix vulnerabilities. Due to this fact, Parallel and concurrent architectures are elusive field for malware. They are new, trendy, popular, complex.

In this thesis, we will try to show vulnerabilities on concurrent and parallel cpu schedulers and non-uniform memory architecture. The weaknesses on hardware layer of the computer are hard to be observed by software solution. Therefore; It is time to pay attention for them, since it is not hard to predict that attackers will focus them.

This thesis purpose an offensive security approach, how malware can be evade autonomous malware detection systems, and also purpose and experimental method to detect and mitigate them.

Preface

I would like to thank Erik Hjelmås for encouraging me to write this small LaTeX class for GUC's master's theses, and in particular for making sure that I will not be responsible for maintaining and supporting it...

Ivar Farup, 2007/05/30

Contents

List of Figures

List of Tables

1 Introduction

The purpose of introduction chapter is giving the readers blueprint of the subject, the problems that we face, the change in the solutions, as well as motivation of its importance. In addition, It also purpose to form proper research question which will guide thesis.

1.0.1 Topic covered by the project

The thesis purposes an architecture of the malware which process parallel, access memory concurrently, conceal itself systematically, shortly that it is likely to be rocket science. However, everything actually started with a simple mathematical theory by John Von Neumann [1] and the first example of practical malware is writen by Bob Thomas at BBN, and it was called Creeper

The malware is abbreviation of malicious software. It could be any piece of code which is defined malicious. There is no formal definition of malicious, it could be some software advertise without any contest or it could be self-producing code piece which aim to distribute itself and steal your private information, and it turned an arm race between power holders today.

With development of the first malware, their counter software are created and anti malware software have evolved with them so far. In this race, malware authors are always one step further, because of security's nature. This race between black and white side raised the bar above. The motivation of the information amount and severity influence both today, and that information can be sometimes vital.

1.0.2 Keywords

Security, Concurrent Malware Design, Malware, Concurrent, Parallelism and Concurrency

1.0.3 Problem description

The one of the main and indecipherable problem in security discipline is formulating general threat definition and recognizing malicious activity and all this problems unsurprisingly reflect on information and computer security concept. Security is defined by system's identification, which involve with purpose, crowd, design structures, network model and so on, and today's information system which is designed with various architectural forms is protected against malware by general purpose protection tools. In the market, The anti malware tools producers focused on pragmatic solutions to survive, but it leads to that most of these tools are utterly reverse engineering process which works on result instead of reason.

With usual and pragmatic signature based methods, there are two mainstream techniques to detect malicious code which are called static and dynamic analysis.Static analysis identifies malwares mainly with code flow graph and data flow graph on stored file which is not processing. However, On the dynamic side it is a bit more tricky to analyze process, because you are working on the running pieces of codes without knowledge of structures and worse than this, it must concern race condition and memory coherency flaws.

The detection methods and techniques have been adequately worked so far because of the simplicity of architectures and usage of the massive generic computers, However, with increasing of the not standardized, parallel and popular devices like arm's SoC, it is not hard to estimate their new challenges. It is really likely to evade and obfuscate properly your on-the-fly processes with using uncertain charactership of parallel processing, complexity of concurrent programming, and structure of "Non Uniform Memory Architecture".

1.0.4 Justification, motivation and benefits

If malware designing is superficially considered, you could fall in usual fallacy that It is not beneficial and exactly opposite. However, if we can design it, there is always more skillful author who already abuse this vulnerabilities on the black side of the moon. The work we are obligated to actually proof this vulnerabilities and design counter measure against them. In this way, our blessed motivation is finding possible vulnerabilities, and mitigate or eliminate their risk. Otherwise; if we confront with unknown attack, it could be too late to fix and analyze it. For example, some of the most sensational and beneficial papers are criticize malware as same as the thesis ([2],[3],[4]), and their values are undoubted today.

1.0.5 Research questions

1. Can a malware model be designed with using parallel and concurrent architecture in order to conceal its presence from detection mechanism?

2. If we can design the mentioned malware, can we build a detection mechanism against these kind of malware's presence?

3. If we build the detection mechanism, What is detection complexity of the algorithms?

1.0.6 Planned contributions

This Master thesis is looking for better understanding on concurrent malware abilities and their counter-measure. Especially, It will try to show how possible to abuse concurrent memory accessing and how durable recent detection kits. It is quite unique work which we have to consider on the future. Ultimate goal is to eliminate any uncertainties which detection methods encounter with concurrent memory accessing.

1.1 Related work

This chapter will give an overview of researches about Malware‘s self-defense technique, the methods to analyze them, and their application on concurrent architecture.

1.1.1 Malware Self-Defense

This section will try to give the literature about malware evasion techniques. This techniques are generally antonym solution which are found by malware authors, however, there are enough surveys about known technique. We classified all these methods in six categories which are code obfuscation, code reuse, anti debugging, anti emulator and visualization and covert channel over network traffic. This taxonomy is well defined by Jonathan A.P Marpaung, et al [5], yet malware authors used them to protect their own properties.

Code obfuscation was originally found for protecting intellectual property[6], but

It aims to puzzle code‘s binary against merely static analysis and disassembling[7]. The first known obfuscation method used encryption in order to hide its content. It was called Cascade which is seen first 1986[8]. This simple architecture of the obfuscation is called packing[9]. It involve with two part of binary which are slub part, in order to decipher and encipher.[5]. Cascade was using simple XOR encryption and that was increasing performance.

Early of the 1990s , oligomorphism and polimorphism started to show up[8]. The main idea behind them is basicly transforming their slub part in each attempt of encryption process[7]. Today, there are two type of polymorphic approach to generate different variants of slub.[10]

- Rewriting the code each time from pseudo-code so it differs code synthetically which is actually transformation based obfuscation.
- Self-cipher itself different, order of these ciphers and using different keys.

One of the other important milestone of polymorphic malware is Mutation Engine(MtE) is writen by a Bulgarian virus Author, called The Dark Avanger. It was automated obfuscation tool which actually considered impossible in those times.[11]

There are also several methods to prevent unpacking process. These methods are collected carefully by Peter Ferrie [12]. These methods are especially obstacle for automated analysis.

Compare with polymorphic methods, metamorphic approach is more complicated. It is transformation based method instead of encryption approach.[13] Fundamentally, it produce different codes which doing same blue printed semantic. That just mitigate detection possibility because of lack of static code.

Network traffic, which malware produce are generally Achilles heel for malware, because they are generally adequately unique traffics to be identified[5]. They usually cover their overt malicious traffic with covert channel methods.[14]

Code reuse attacks are strong attacks because they do not inject any code in them as obfuscation methods did. They aim to use legitimate software to evade themselves. There are there well known applied version which are return-into-libc, return oriented programming and Frankenstein.

Return into libc attacks were demonstrated by solar designer in 1997 as a method of bypassing non executable stack to executable libc libraries[15]. It’s object is to change the "ret" infrastructure argument to the known address possibly libc library(stdio, system, etc). However, this attack is limited with libc libraries, which we improved with return oriented programming.

Return oriented programming is more flex version of retur-into-libc attack, which is introduced by Shacham in 2007[16]. Return oriented programming purpose a programming language with small gadgets(instruction bound) which involve all ability of Turing’s machine[17]. Frankenstein is one of the novel application of return oriented programming by Vishwath Mohan and Kevin W. Hamlen[18].

Anti debugging and anti emulator methods are really usual for today’s malware. The survey of Chen Xu et al. showed us in 2008, majority of 6900 on-the-air malware could evade their self with exhibiting benign behavior in sandboxes, debuggers, and virtual machines.[19]. VM and debuggers are most important element of dynamic analysis techniques in autonomous sector, because it must run the file just before it touch the working environment. Yet, it is not that knotty to determine whether working environment is virtual or not. Fuzzing cpu bechmarks and comparing results entropy is a good way to determine virtual machines.[20]

Rootkits are the piece of malicious code which aims to crack integrity of the system state. The idea of the remaining invisible to the system state is traced backed one of the oldest virus "Brain"[21]. It was changing the boot process and activate virus during booting. "Tequila" and "1689" viruses followed "Brain" in 1991 and 1993[22]. There are NTRootkit and HackerDeffender rootkits today. The proper classification of the rootkit are prepared by Adnan Abdakka[23].

1.1.2 Malware analysis methods

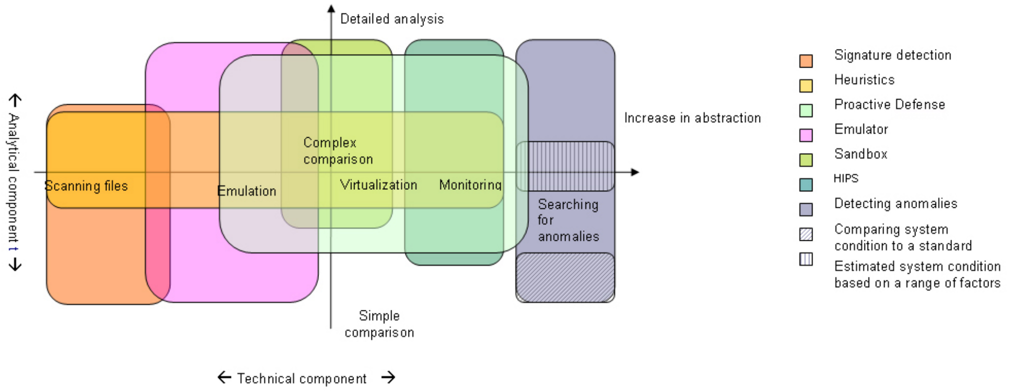

Figure 1: Detection models [?]

Malware analysis medhods could be considered two dimensional plane which are "anomaly, signature based detection technique" and "Statistic and dynamic analysis method".

There are also several applied techniques, which combine terminology above.

N-gram It is a anomaly based heuristic detection method algorithm. It is a bit costly process and not practical for client side analysis. It could be fit for honey pot analysis [24] [25] [26]. They are capable against to zero time malwares and that could makes it futures malware detection system.

Sequential approach on system and funtion call This approach is anomaly based dynamic analysis and observing and recording the flow graph of systems and function calls and try to analysis anomaly behaviors.[27]

Taint It is also called data flow analysis or data flow graph. It is basic tracking marked data values during execution.[28][28][29]

Control Flow Graph They are one of the most important arm of commercial autonomous malware detection tools[30] [31] [32]. After the invention of the polymorphic and metamorphic, syntactic analysis could not bear with them. Then we moved to upper layer of information, semantic layer. Semantic can be representation of code flow, and the routes of the code are adequate to produce signature to identify malware. This methods are a member of static analysis and disassembling and source code analysis job.

Network Monitoring Malware intention of the communication over network actually big clue to detect them. They generally use unique hostname, ip adress or specific protocol with particular way [5].

1.2 Choice of methods

This thesis will use a technical approach to the problem. It will use quantitative and model building approach. The methodology consist 8 circular step which are, asking question, building new hypothesis, planning methods, developing software, preparing genering testbed, testing, analyzing, reporting results. In addition to this, the large portion of the time will be given researching related topics, based architecture, and learning tools and technologies. Consequently, the accuracy of the thesis is lies on the proper scientific methodology.

circular diagram will be Drawn here

To address the first question, designing proper malware evasion technique with concurrent and parallel architecture haven‘t been researched well so for, therefore; it will lies on so much experiment and we might have to reflection of other evasion and obfuscation attacks analogy. We will chose several known malware, which is on air, to evade them during testing part. We will strong probably use kernel modules and android operating systems for testing bed, however we could linux OS without android layer to simplify and closure test period. In testing step, we will use several anti-virus system such as Avast, Comodo, Norton and compare the result before evasion and after evasion. Testing period might me include mathematical proof depending on evasion or obfuscation method.At the end of the each hypothesis‘ result, which is mean the method for concealing malware, will be reported properly. Each method will be another hypothesis, so we could be proof whether multiple or none successful hypotheses, yet the failed hypothesis could be crucial. There could be also many result which are too barque to proof them or explain their result, these cases could be observed on further work. For semantic knowledge, we could try to show relationship between evasion methods and hybrid approaches.

Consider involve
this line during

If we can find successful hypothesis for first question, we observe them in second question. Second question is depend on the first question‘s answer. Second questions methodology is actually exactly same circular. It start with defense hypothesis against evasion method. Dynamic and static detection methods must be both considered. The development of the counter algorithms could be proved mathematically, but it can be quite barque to formulate it. In order to prove it, we could design evasion and detection methods‘ Turing model, however; the main approach of our testing is involved with experimental solution. we will develop planned algorithms prototype. Proper test bed could be provided with lots of malware species.. We will test our algorithms‘ prototype with concealed or obfuscated viruses. If it is really necessary, we could prepare also control groups to prove trustworthiness of method, then we have to record result without any intervention. For semantic knowledge, we could try to show relationship between detection methods and hybrid approaches.

depending on th
evasion techniqu
odology could b
again

The last question is a matter of measuring and analyzing algorithms complexity. It is totally mathematical scientific methodology. We have to analyze worst, best and average complexity rate. There are also several Quantitative approaches like measuring resource usage. It could be efficient some system like network which there are lots of uncertainties in.

In this project, there five inevitable risks which we can face during development.

- The thesis is highly dependent on the hardware, and the cost of the hardware constitute risk on its own. Any case of hardware defect leads to comprise obstacle.
- Hardware dependency is also leads to logistical and time consuming risk which could result with latency on submit time.
- Firmware codes which we are planning to work on are mostly undocumented. We could discover their usage by proper reverse engineering and fuzzing process when required, however it is obviously manpower.
- Most important and highlighting risk is there isn't proper research on this particular area. That means there are strongly possibly hidden risks which could cause other mental and physical result.
- During testing and purification part, Anti-malware tools could come out with unreliable result. To analyze result properly, we may need to inspect mentioned tools with reverse engineering process which could violate proper usage agreement. To mitigate that kind of risks, we could request research agreement from companies.

1.3 Ethical and legal considerations

> "Virus don't harm, ignorance does."
> - VxHeaven

The content of this document could be used in order to malicious purpose, but any matter or information could be misused in the life and ignorance is not known well as a defense strategy. In this purpose, this thesis aims to enlighten security specialist and system developers against recent way of the possible attacks.

However, in order to act ethical responsibility, we tried to eliminate practice of tools and piece of codes which could leads malicious usage. In any case, there is no doubt that it is critical to discover and publish vulnerabilities which could cause deep impact before malicious people discover and abuse them.

2 Related Works

3 Malware Detection

3.1 Taxonomy

3.2 Scope of Model

3.3 Scope of Methods

3.4 Obfuscation Methods

4 Background Studies

4.1 Concurrent Programming

4.2 Caches

Solely, a cache is a small, fast, array of memory which is placed between lower level memory and higher one. It store a special block of information, in order to increase performance of computer systems. It is like a buffer area which has some logic to exploit locality features of programming logic. Today, with increasing of processing ability of computer systems, memory access is bottle neck.

The "cache" is originally french rooted with meaning "concealed place for storage."[34] We can move this definition basically to the computer science. The cache's design is definitely isolated from software layer, however; if you know your caches feature and how caches working you could program a lot more efficient codes easily.

4.2.1 Motivation of Caches and Principle of Locality

The main motivation of caches is indisputably performance. As we mentioned, Performance of high-speed computers is usually limited by memory bandwidth and latency. In order to increase, and turn around that, we use an small array of memory which is located close to the processors. The location of chip is important and there are many design decision (e.g On chip, out of chip), but more crucial properties of caches are their designs (e.g. Naive Capacitor , SRAM, DRAM) and their logic complexity[35].Due to physical constrains, the size of the memory is limited which we can locate close to memory. On the other hand, these design choices are decisive factor about prices of memories. Because of all these reasons, Multi-Layer Memory Hierarchy with several caches between processor core and main memory is well-known option in order to improve performance. Nevertheless, In multilayer memory hierarchy, it is hard to know where the particular data reside in, and whether it is coherent or not. It also add many layer between memory and processor and in some cases it even decrease system performance,especially because of logical complexity of the line.

The idea all the caches logic depending on is Principle of Locality. Principle of locality is actually a concern of information theory[36]. It a conjecture of data distribution and processing order. The phenomenon assume that the the same data and related document will be accessed more frequently than other data[37]. Today, it is the one of the corner stone of computer science. It was first developed with Atlas System with purpose to develop virtual memory systems work well[38]. Then, it spread from search engines optimization to hardware caches.

There are mainly two type of locality of reference:

Spacial Locality Spacial locality propose if there is a particular of memory which is accessed on memory, then it is more likely to accessing memory locations around of it in near feature. Especially arrays and instructions are exploiting this locality. Arrays, formed structure and instructions on memory are laid out lineally over

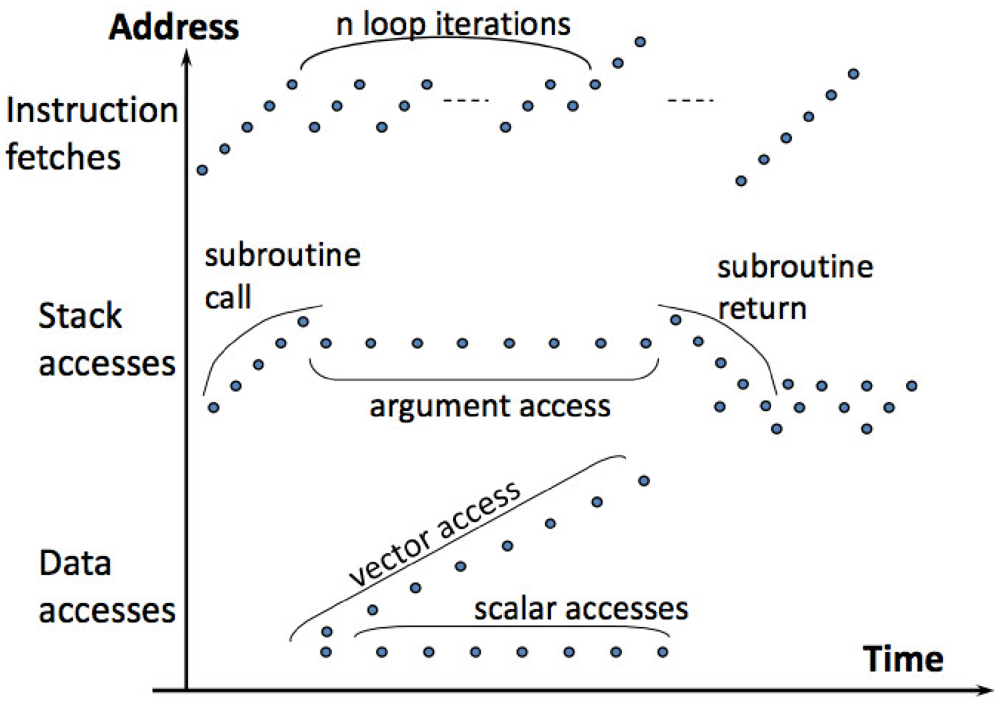

Figure 2: Principle of Locality
[39]

memory. On figure 2, we can see spacial locality simply. For example, during instruction fetches part on the figure, n loop iterations accesses same memory locations for many times. There are also subclass of spacial localities like Branch Locality and Equidistant Locality. They are designed locality types of indeterministic feature of program structure. Branch prediction and Special compiler designs aims to exploits this kind of locality more efficiently.

Temporal Locality Temporal locality propose if there is a particular of memory location which is accessed recently, it will be accessed again more likely than any other location. Especially, variables, subroutines of stacks or other calls exploits this feature of locality. On figure 2, it is obviously seen that the values accessed once is possible to accessed again.

4.2.2 The basic logic of caches

As we said in previous section, the basic logic behind caches is moving arranging caches with local data. In order to provide this feature as smooth as possible, we use a logic circuit called "Cache Controller". It does basic logic comparison and wiring the request and response into the right path. Thus, it intercept the write and read request from processor, replace its memory array with right scheduling method, and evict it safely and coherently. It processes with diving address of th request into three fields which are Index set field, tag field, and block field. In figure 3, these fields showed.

At the beginning of cache process after it divided address fields, It first request right cache line which is shown in figure 2. So if we have M byte memory and N byte

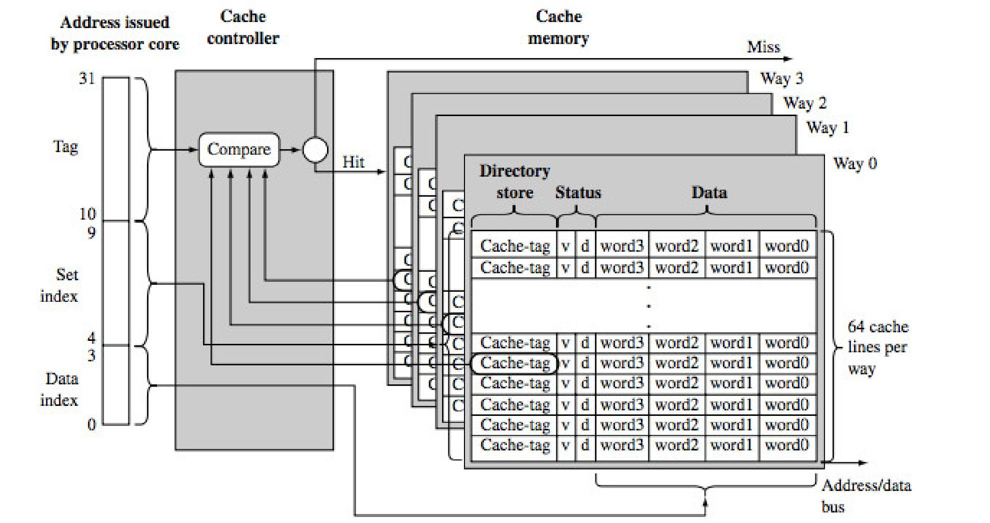

Figure 3: 4 KB 4-way set associative cache with 256 cache lines [34]

cache line, we must have $M/N = cacheline$, then we can represent it with p when $cacheline = 2^p$ Thus, cache controller just wire corresponding line with given set index.

In traditional cache convention, first field belongs to tag id. Tag id is determined depending on other field i.e. the remaining part after index field and block field calculated is tag id length. Tag id is using to verify the stored line is actually belongs to right location of memory. The cache controller has comparison circuit(XOR) and compare the requested address and the address which is in the pointed line by set index field. If they are matched with each other, then it check valid byte and hit or miss. There is a simple AND circuit between tag comparison.

Final field is called data index or block index field. It will point in the cache line the smallest addressable memory location. Therefore, when processor want to read a value, cache fetch the whole block, and that makes cache to exploit spacial locality linearly. However, it will limit the access speed remarkable, if we increase block size. The optimum block size is about 64 byte for many system. As we mention before each cache line includes cache-tag field, valid bit, dirty bit, and some coherency bits in some special systems. The length of the data index field is equal to r if $wordsize = 2^r$.

When we increase the set index count it increase basically temporal locality, but not always. The cache conflict could happened when two memory location which uses same cache line could be used concurrently or twisted. Highly trashing can reduce cache performance. For this reason, associative caches are developed. Set associative caches are represented by their way number e.g 3 way associative caches or full associative caches, and there are group of cache arrays corresponding to the same set index. So that decrease the set index count but increase the performance during conflict miss in some cases. However, because of the complexity of the comparison circuit, it must be carefully chosen the number of ways. The associative caches are showed in figure 3.

The computer architecture we uses today actually first formulated by John Von Neumann [40]. On the first design of computer it was a single cycle instruction machine without any pipeline or superscalar idea. Then Hardward Mark I machine is designed with proposing two type of caches which are one for instruction, and one for data. Icache and Dcache are specified for their own purpose, because data and instruction on memories have different deterministic properties. Instruction are more tent to be linearly accessed by memory and they has branch locality which can be predict earlier. Icache also could be located more close to decode and fetch parts of processors when Dcache are instead closer to memory fetch parts. Yet, the most significant benefit of Harward design is concurrently usage both caches during pipelined architectures.

4.2.3 Allocation, Write and Replacement Policies

There are three policy type determine a cache behaviours. They are write policy, read policy and reallocate policy. System's performance, coherency, and designs are determined depending on these rules.

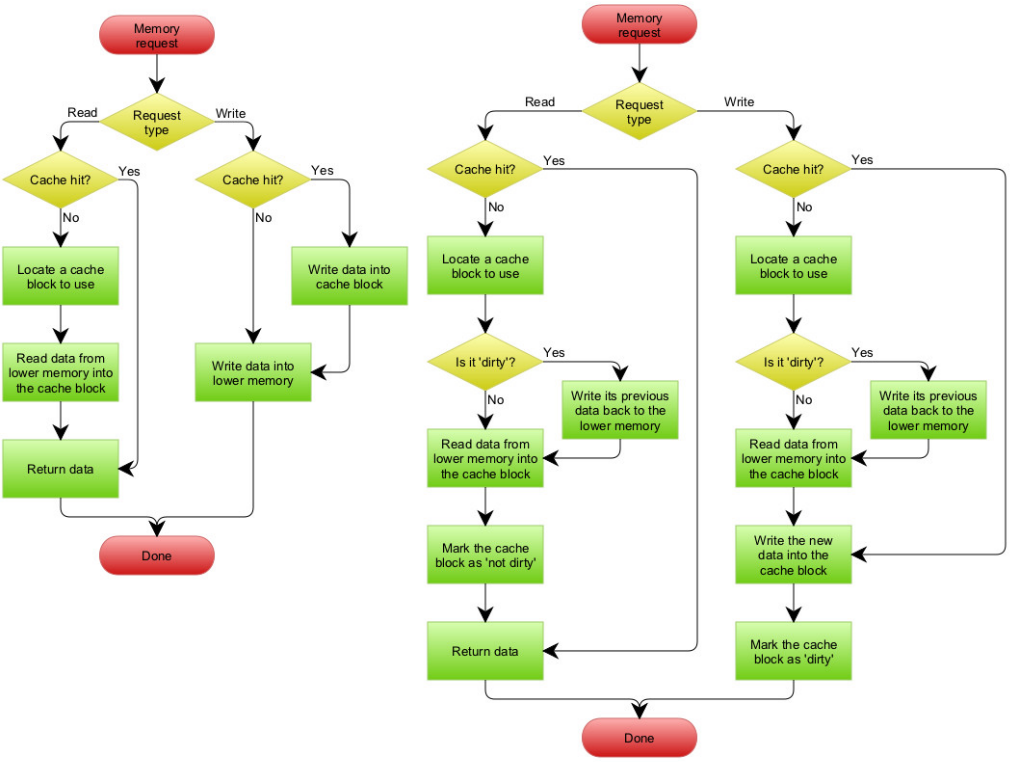
Memory request
Request type
Read
Write
Cache hit?
Yes
No
Locate a cache block to use
Read data from lower memory into the cache block
Return data
Write data into cache block
Write data into lower memory
Done
Is it 'dirty'?
Write its previous data back to the lower memory
Mark the cache block as 'not dirty'
Write the new data into the cache block
Mark the cache block as 'dirty'

Figure 4: A. A Write-Through cache with No-Write Allocation B. A Write-Back cache with Write Allocation

[41]

Write Policies

Write Through When the cache controller designed based on *writethrough* policy, it write the values into the memory and caches simultaneously, when the write request is arrived from processors. It does not depend on *writemiss* or *writehit*. It will reduce write performance, because writing data on memory is a lot slower, but it stay coherent all the times. It is performance could be increased a bit with write buffer memories between memory and cache.

Write Back The systems with that policies does not have same values in memory and corresponding cache line, so the coherency between memory and caches are provided by a trashing algorithm. Cache line always store more recent data, but if there is more than one cache it is hard to decide which one is more valid or whether there is a valid coherent one. However, it effects performance quite remarkable (e.g. in ARM 15 cpu writing cycle to memory is around 200 cycle, but caches is about 4.) . The system with limited register numbers can overflow to the memory to store loop variables and that could increase write memory usage. $WriteBack$ policy makes this kind of systems really effective. The dirty bit are stand for $WriteBack$ policy. If you write some value on any cache line, dirty bit must be set for eviction. During trashing process, you must first move dirt block back to memory.

Replacement Policies

Random Random policies are designed to evict a random line in the associative caches. It is not really random on implementation, but enough random to work with it. It sounds to weak and primitive approach but actually it could be really effective on highly associative caches.

Least Recently Used Least recently used replacement policies are actually implemented in two types. Fully most recently used and Not most recently used random. It is probably the most efficient algorithm to replace cache index sets, but it is really hard to implement on highly associative caches. You must record history of schedule and update it each attempt of access. It could be most effective and easy method on 2 way associative caches and it just need one bit to record who used last. It actually increase temporal locality, because it offers the most recently used one is more likely to be used again. The most recently used but random is a hybrid solution of least recently used and random policies. It just record who accessed last and replace one random set except most recently one.

First In First Out It is also known Round robin. It is also mostly using with highly associative caches. In its implementation, it has one one tail pointer of stack and in each attempt of access it evict tail pointers set, and increment the the tail pointer to next set.

Allocation Policies

Write Allocate $WriteAllocate$ policy is also known as $ReadWriteAllocate$ policy. It refers that during write miss process, cache controller allocates the cache line with related address, as like as normal read miss process. It is mostly using with $WriteBack$ policies, because it assume it is more likely to access same data which you write before.

No-Write Allocate $No - WriteAllocate$ policy is also known $ReadAllocate$. It is an exotic implementation of caches. It is generally seen with $WriteThough$ policy. This systems can be special to read privileged and they do not hope to read or write subsequent write(or even read after write.)

4.2.4 Miss Type and Advance Cache Optimization Methods

Miss Type

Cold Misses Cold misses are sometimes referred to as compulsory misses . If you never invoke related memory address and if you calling it first time, You will encounter with that misses. It is natural misses, and really hard to mitigate them. Spacial locality is the one of the method to avoid this misses. As we mention before, when we increase the size of block, it will increase spacial locality.Also before initializing memory, pre-fetching algorithms and branch prediction algorithms can be useful to eliminate this kind of misses. In addition to this, usage of large amount of caches will naturally reduce this misses, but it is side effect of it.

Conflict Misses Those misses are the one we are able to avoid. Conflict happens in systems set with lower associativity esp. with direct map systems. To reduce this you should increase associativity. In full associative caches, it all conflict misses are avoided. The change of conflict miss is $tagsize/memorysize$.

Capacity Misses They are also natural misses related with size of the caches. We can not store every information in memory into cache. Those misses are based by definition of caches. You can't solve it even with perfect replacement algorithm, but maybe you could decrease the rate of capacity miss with pre-fetching.

Advance Cache Optimization Methods

Pipelined Caches As we did in processors, we could divide cache organization in two separate stage which are decode and data. It will increase the writing efficiency because it will increase the bandwidth during subsequent requests. However the clock mechanism will decrease to hit time.

Write Buffers Write buffers are small fully associated buffer memories between caches and memories. They effects cache performance because the time between writing values to memory from cache, cache memories must lock if we do not use cache memories. Thus, Cache memories store values to buffer buffer will responsible with writing it. Buffer size is important, when consecutive write operation requested. When buffers is full, it will makes cache lock to get empty.

Multilayer Cache Multilayer caches are game changer optimization decisions, because when we have level 2 caches, then we could have faster level 1 caches, because it could be smaller and simpler. Namely, we are adding systems higher level caches, in order to, decrease lower level caches miss time penalty and increase the hit response time, but it will decrease lower level caches hit rate. Level 2 or higher caches could be also on-chip (i.e fast as possible) and SRAM, yet lower level caches must always be faster closer and simpler.

Victim Caches Victim caches are really useful and simple idea for decreasing miss penalty time. It is a buffer memory, fully associative and mostly 4 to 16 cache line. It stores recently evicted lines in it. It means it increase the associativity of recently used lines on other small buffer with cheap and flexible design.

Hardware Prefetching There are many theoretical pre-fetching method, but there are a few example implemented. The most well known is prefetch the most recently values incremental block line. That targets to increase most recently used ones spacial locality. It is really efficient to applying it, because increasing block depth is expensive job for caches and increase hit time. If you implement one buffer memory, which prefetch next block of block you need, it automatically increase spacial locality. Also compiler based branch prediction methods are good example of instruction prefetching, however, generally, prefethers for instruction caches load all branches to decrease miss rate.

Non-Blocking Caches

4.3 Cache Coherence and Consistency

Many modern computer systems with parallel processing ability have support of shared memory in hardware. Shared memory has lots of advantage over message based memory systems. Each processor could access same address space, read and write them simultaneously with using their own caches. This features has lots of benefit such as; low power consumption, higher performance and lower prices. However, without consistency between processors, parallel processing can not use many advantage of parallel programming. It could be also insecure to use a system without consistency between processors.

To provide better understanding of shared memory correctness, we defined it in two separate them in two definition, which are consistency and coherency. Consistency provide a definition of memory access rules and how they will act around computer system with store and load operations. When we compare it with coherency model, it must be more simple and easy to understand it. Therefore, it define a correct behaviours of the memory accesses of multiple threads by allowing or disallowing executions. On the other hand Coherency is a way of implementing a control protocol between memories and processors to support and provide consistency. Correct coherency provide a system which programmer or operator of the system can never determine behaviours (misbehaviours or correct behaviours) of caches[42].

As mentioned, Mention Consistency is try to define to correct shared memory behaviour between many processor in term of loads and stores. It does not have to concern specific hardware issues, such as hardware level pipelines, write buffers, caches, Out-of-order processing schemes etc. However, in the market, there is no hardware provide consistency perfectly, because the reordering store and load operations is regular optimization techniques in out-of-order processors. In addition to out-of-order processors, the multi layer memory architecture makes consistency vague and subtle. Yet, most of the programmers assume memories are completely consistent. There are several level between inconsistent and sequentially consistent memory.

Mention Consistency Application Summery.

Memory Coherency (a.k.a. Cache Coherency) is actually to impose a protocol between caches to provide a specific consistency model on shared memory systems. Unlikely consistency, it also concern hardware uncertainness and subtle part such as write buffer, pre-fetcher. Typical consistency protocol has features which include instruction caches, multiple-level caches, virtual-physical address transaction, and coherent direct memory access. However, it is not enough to ensure consistency(depending on consistency model) by itself. It tries to makes caches synchronization in shared memory systems invisible

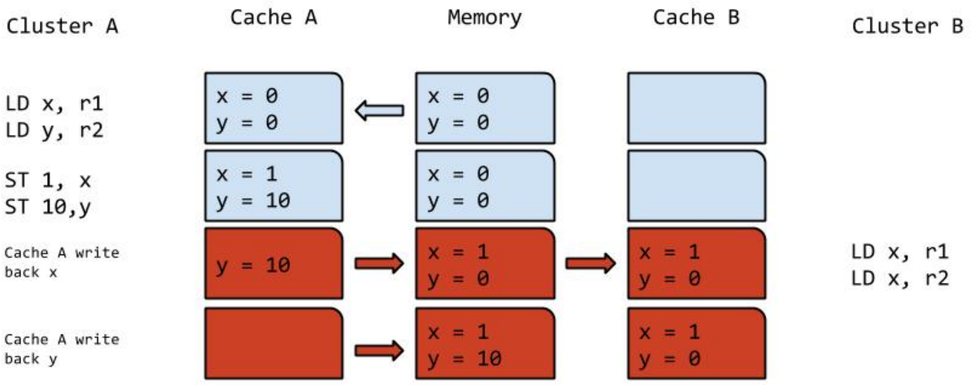

Figure 5: Write-back Policy Cache Memory Inconsistency

from software developer. However there are timing techniques to analysis cache architecture and coherency model of system.

In figure 5 and 6, the consistency issues on multi layer memory systems. Assume there is two cluster which has ability to process values with given instruction codes. LD and ST instruction refer to memory load and store request. In figure 5, there is a system with two caches which belongs each cluster and one shared memory block. x and y is represents a particular memory address. Contrast with figure 6, figure 5 uses write-back policy. In step 1, cluster A loaded x and y to the processor(it could be also pre-fetcher who load them to the cache block). In second step, somehow clusters stored 1 in memory location x and 10 in memory location y. In this step, memory is not consistent with memory but it is not hazard because they are not shared with cluster B. In step 3, caches evicted the block which include address x and later address x and y were loaded into the cache B. After this moment, they will never share the values which other cluster is actually using. Y was 10 at the end in the memory but it can't be seen by cluster B, even if it try to read it a million times.

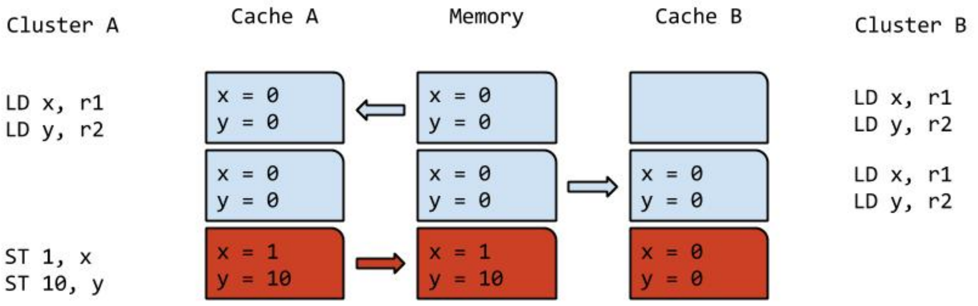

Figure 6: Write-through Policy Cache Memory Inconsistency

Write-through cache policy is intuitively perceived as solution of this problem, because it just write every values directly to the memory and it will be always synchronized with memory, yet it is not. In figure 6, write-through cache policy inconsistency showed. The problem with write-through policy, clusters use values which in their cache instead of memory, so even if memory is consistent with clusters, it is not consistent with each caches. In step 1 of figure 6, cluster A loaded x and y addresses into the its registers. Then, in step 2, cluster B loaded values of x and y addresses. In step 3, system got in inconsistent state, because cluster A write values through memory, but Cluster B uses the old values, and it will never reach never values, even if it try to load many times. For this reason, many of the large systems which has more than 64 core use this type of cache coherence.

In order to solve this problems, there are several coherency mechanisms and their protocols. Depending on the case and the number of cluster or processor in the system, system could use Snooping and Directory based mechanism. These each protocol have their own benefits and drawbacks. Snooping protocol is tent to use a lot of bandwidth, however, it is faster and more synchronous. Its logic is to broadcast each state to every node on the system. However, directory based mechanism work with request and response. There is interconnector to forward message to the right address and it makes directory based mechanism slower because of the increased latency, lighter because of the decreased bandwidth.

4.3.1 Consistency Models

4.3.2 Snooping Coherence Protocols

Snooping coherence (a.k.a. Bus Sniffing) is a technique to have caches to watch other processors caches and provide consistency depending on specified protocol. It basically implemented with external port to the system bus. Therefore, it implemented over cache controller which has feature to watch bus. It makes cache controller bigger and waste more power, so lower layer caches could use less complex coherency protocols and vice versa. There are many snoopy cache coherency protocol also depending on consistency model, but we can categorize them in two class which are Write update and write invalidate.

In this both protocol, we try to get rid of stall data which are in different caches, but it is provided with different logics. Write-update protocol is a broadcast write protocol that in every write attempt, it will write the values into the corresponding cache block but also it broadcast the write message to the every caches on the connected bus. Thus, everyone on the bus which has the ability of interpreting the message of write-update protocol will update stall values with new ones.

Secondly, Write-invalidate is whenever you write, you invalidate other cache copies and reduce to possibility usage of stall data. Instead of sending whole data block, it just send the tag number and state of the tag. It could effectively be successful, if you have limited bandwidth and power source. Most processor with coherency is today using write-invalidate protocol. However, it is efficient if there a few writer and many reader clusters or processors. Comparing with write-update protocol, if there is many writer, it could be less efficient because of invalidation process validate-invalidate-forward hops.[39]

There are many protocols for both write-invalidate and write-update to maintain coherence, such as MSI, MESI (aka Illinois), MOSI, MOESI, MESIF, write-once, and Synapse, Berkeley, Firefly and Dragon protocol. In this thesis, we will just focus on write-invalidate protocols because of their popularity, but basic principles are same as each other.

•	Clean/Dirtiy	Write?	Unique?	Silent Transition to
Invalid	Clean	No	No	-
Shared	Clean	No	No	Invalid State
Modified	Dirty	Yes	Yes	-

Table 1: MSI states' properties

Mention Serialization of buses

MSI - Basic

Basic write-invalidate snoopy cache control protocol is MSI (a.k.a Modified-Shared-Invalid protocol). In this model, each cache block has cache tag, and two status bit as same as standard caches, but instead of dirt and valid status bit, MSI cache line has state bits to refer in which state it is. MSI has three state in state machine and they are "Modified", "Shared", "Invalid". two bits can represent four state, so definitely represent three state. The main idea behind this protocol is that one writer and many reader states provide always consistent memory sharing. Therefore, every cache in the system has different responsibilities when they read or write.

Invalid Invalid state is exactly same state with standard caches' invalid state. When cache need to access a invalid block, it must act as cache miss, and be fetched this block again.

Shared When there is no writer processor on this line, and if a processor request this line with purpose of read it, it will be in shared state. It is read-only cache block, and processors are not allowed write without transforming state. The processor also can evict it without writing back to the upper layer memory, because that is for sure, it is clean block.

Modified It is modified and also modifiable cache block. In a memory coherent system there can be at most one modified cache and all other cache must be invalidated. It is responsible with writing back cache to the upper layer memory.

In figure 7, MSI protocol's state diagram is showed. Cache memory launch with invalid cache block, and when a read miss is comprised, cache controller will request memory block from memory. Then, the snooping control bus will broadcast the request of read. If there is a modified copy on the bus, it will abort request of memory block from memory. It will evict its line to memory, and change its state to shared state. Then, memory responds source of the request. After the fetching cache block to the source cache it, it sets the state as shared state. If there is a shared stated copies in the system, It does not matter who responds the request. In any case, It will fetch the memory block, and sets the state bits to shared.

When write miss is compromised in invalid state or shared, It will fetch the data as same as read miss cases, but the difference is it will invalidate other case's corresponding block which are shared or modified. Modified stated block must evict blocks properly. At the end, source cache block fetches the block.

Write hit can be compromise in modified state, and read hit can be compromise in shared state.

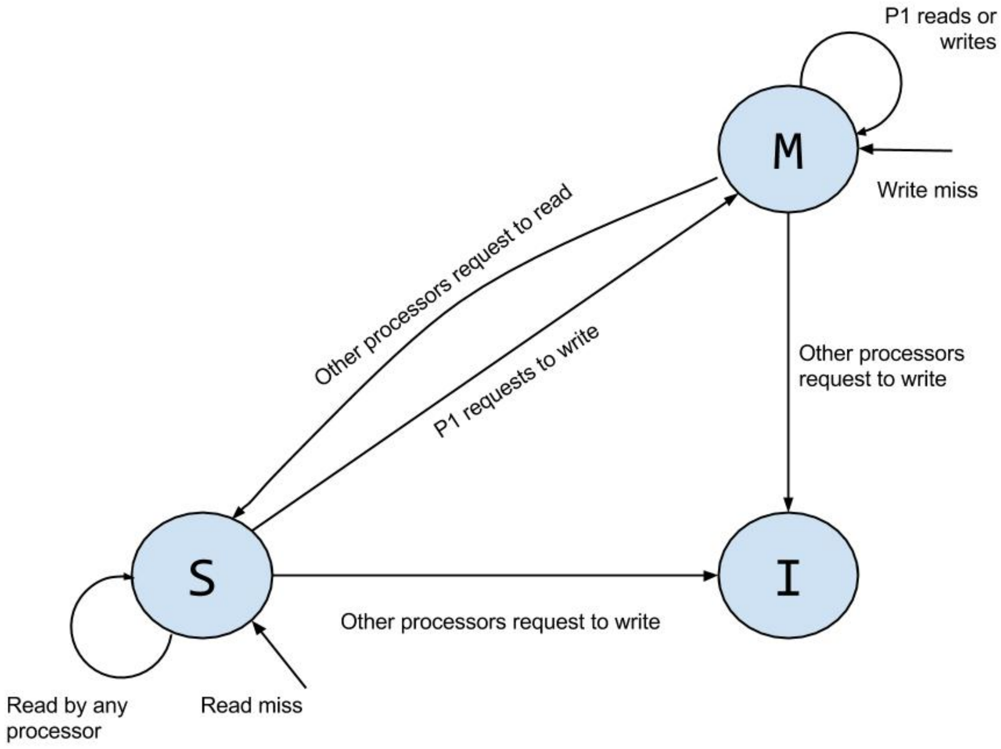
P1 reads or writes
M
Write miss
Other processors request to read
P1 requests to write
Other processors request to write
S
I
Other processors request to write
Read by any processor
Read miss

Figure 7: MSI State Diagram for processor P1

MESI - Exclusive

The MESI protocol (a.k.a Illinois protocol due to its development at the University of Illinois at Urbana-Champaign) is a widely used cache coherency protocol[43]. The idea behind the MESI is to use forth state we can use with 2 bits. In order to increase efficiency *exclusive* state is developed by JH. Patel et. al. in 1984[43]. As showed in MSI protocol, there is modified, shared and invalid states but also we have exclusive state. This exclusive states also known unmodified exclusive state, if we refer modified state as modified exclusive. This is very similar to the shared state in MSI, and in fact, Shared state is split in two different states. That is because of reducing the communication on the bus and increasing efficiency. In this case, there is exclusive cache blocks which are in read mode and they are unique i.e there is no other cache controller on the system has this cache block.

Exclusive The cache line is only present in current cache memory, and it has not modified yet. It is not a state to provide coherency, but it is state for increasing efficiency of bus bandwidth usage. When a cache line in exclusive state the cache controller can decide the transaction of the line without communicating with other caches. When a cache is requested with load operation, it is loaded in exclusive state, if there is no other cache controller has the cache block.

In figure 8, state transactions are showed. Bus usage is bottle neck, low performance behaviour in cache coherency. Silent state transactions are transactions in cache controller without communicating with other caches. For example, there is no need to broadcast

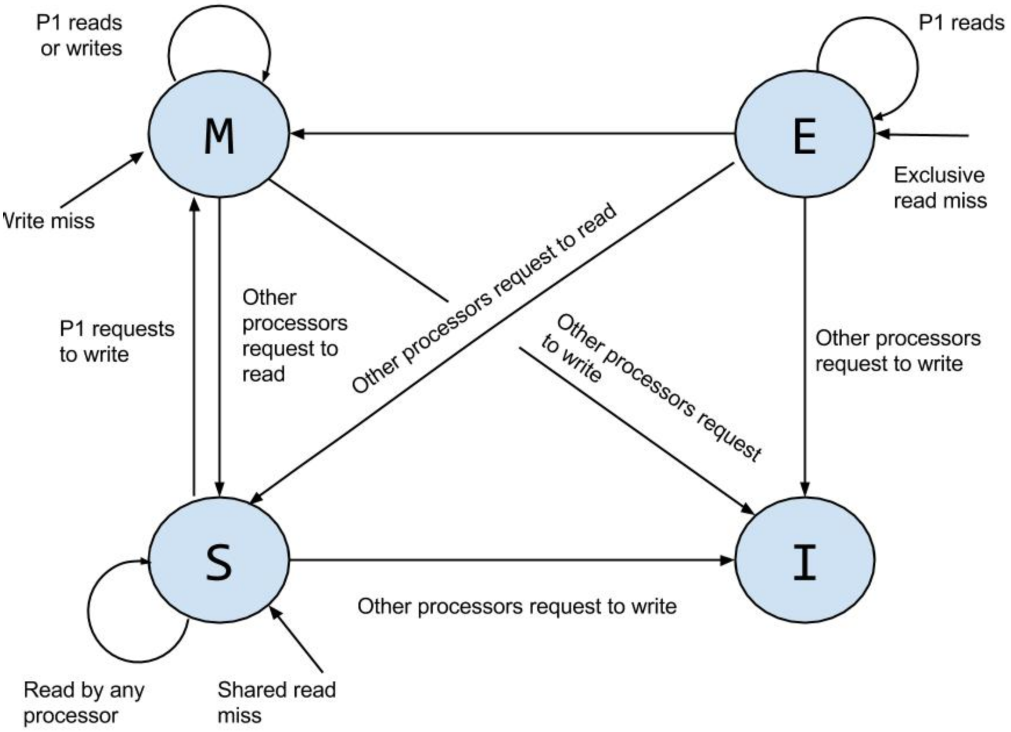

Figure 8: MESI State Diagram for processor P1

•	Clean/Dirtiy	Write?	Unique?	Silent Transition to
Invalid	Clean	No	No	-
Shared	Clean	No	No	Invalid State
Excursive	Clean	No	Yes	Shared Modified Exclusive States
Modified	Dirty	Yes	Yes	-

Table 2: MESI states' properties

and occupy bus for invalidating shared state in MSI protocol. If a cache controller is in shared state, the other cache controller can be shared or invalidate in MSI, so there is no dependency in the system transaction from shared to invalidate. Exclusive state is to exploit the salient transactions. When a load request arrive to cache controller from a processor, it request the line from upper level memory controller and other child caches controller. If any child controller send a shared state broadcast message, it load it in shared state. If there is a exclusive cache controller on the bus, it will degrade its state to shared and broadcast it. If there is no other shared state on the bus. It load the cache line in excluded state. Then, in case of store operation from processor, it will transact its state from exclusive to modified. It does not need to broadcast it, because we know it is unique in system. Contrast with modified state, due to be cleanness of the line, it does not need to evict line to upper memory, it can just invalidate it silently. The weakness of this protocol comparing with MSI, if there is many processor with the corresponding cache line, when it count the copies to test uniqueness, it occupy shared bus more in some cases. If there is n cache controller with corresponding cache line, it will send n broadcast message with this message, however, instead of sending whole line to upper

•	Clean/Dirtiy	Write?	Unique?	Silent Transition to
Invalid	Clean	No	No	-
Shared	Either	No	No	Invalid State
Excursive	Clean	No	Yes	Shared Modified Exclusive States
Owned	Dirty	No	Yes	-
Modified	Dirty	Yes	Yes	Owned

Table 3: MEOSI states' properties

memory it is mostly efficient to send this message.

MOESI - Owned Exclusive

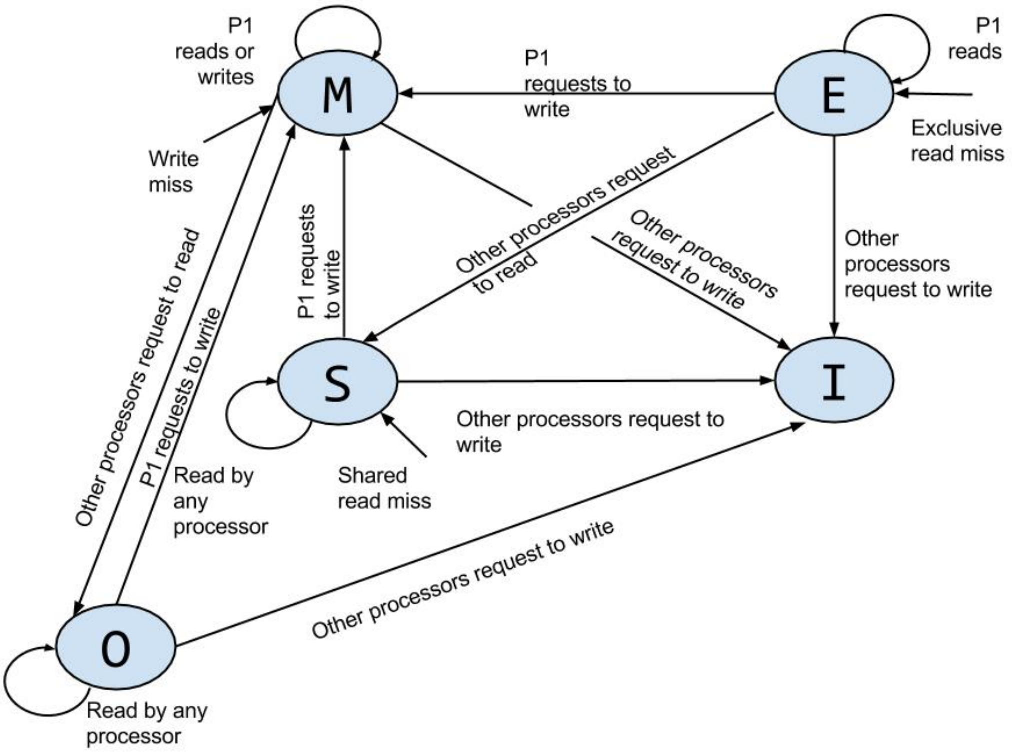
P1 reads or writes
M
P1 requests to write
E
P1 reads
Exclusive read miss
Write miss
Other processors request to read
Other processors request to write
Other processors request to read
P1 requests to write
P1 requests to write
Other processors request to write
S
I
Other processors request to write
Read by any processor
Shared read miss
Other processors request to write
O
Read by any processor

Figure 9: MOESI State Diagram for processor P1

Such processor producers AMD Opteron and Arm Cortex A are using MOESI protocol for cache sharing. In addition to the four states in MESI, a fifth state "Owned" appears here representing data that is both modified and shared. Using MOESI, instead of writing modified data back to main memory, it directly forward the dirty value from cache to cache before being shared, which could save bandwidth and gain much faster access to users to the cache.

Owned Owned state is a state if and only if a cache line can transact in it, when a read request message snooped from another processors when the cache line is in modified state. It allows dirty line sharing between caches, and reduce the latency which is arisen due to the communication between memories and processors. The line is read only by all processors, when it is owned state.

In figure 9, state transactions of MOESI protocol are showed. The relationships of states are almost same with MESI, but there is a state which supplants upper level memory with its own cache line. Hence, it is responsible with evicting lines and cleaning state. The cache line may be changed to the Modified state after invalidating all shared copies, or changed to the Shared state by writing the modifications back to main memory. If could increase efficiency sharply, if the line between upper memory and itself is long and bandwidth is limited. Mostly the L1 and L2 caches are located on-the-chip, and memory are located somewhere outside, the buses' bandwidth between in side and outside of chips are game changer. It can be efficient to use a chip as a forwarder in many system. However, in the MOESI protocol, it is not possible to forward the cache line which is not dirty but present on the chip. If there is a shared cache line in a cache, and if any other cache controller request to load the same cache line, it fetches it from memory.

4.3.3 Inter-connector Design

Switching

Routing

Topology

Flow Control

5 Deliberate Incontinence on Multi-Processing Architecture

6 Exploiting Deliberate Incontinence to Evade Malicious Code

7 Implementation for Hayward Computer Architecture

8 Case Studies

8.1 Simulation Tool

8.2 Samsung Exynos 5420

8.3 Comparison

9 Implication and Discussion

9.1 Theoretical Considerations

9.2 Practical Implication Issues

9.3 Summary

10 Conclusion

11 Further Works

Bibliography

[1] Von Neumann, J., Burks, A. W., et al. 1966. Theory of self-reproducing automata.

[2] Moser, A., Kruegel, C., & Kirda, E. 2007. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, 421–430. IEEE.

[3] Cavallaro, L., Saxena, P., & Sekar, R. 2008. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 143–163. Springer.

[4] Egele, M., Scholte, T., Kirda, E., & Kruegel, C. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2), 6.

[5] Marpaung, J. A., Sain, M., & Lee, H.-J. 2012. Survey on malware evasion techniques: state of the art and challenges. In *Advanced Communication Technology (ICACT), 2012 14th International Conference on*, 744–749. IEEE.

[6] Balakrishnan, A. & Schulze, C. 2005. Code obfuscation literature survey. *CS701 Construction of Compilers*, 19.

[7] Nachenberg, C. 1996. Understanding and managing polymorphic viruses. *The Symantec Enterprise Papers*, 30, 16.

[8] You, I. & Yim, K. 2010. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, 297–300. IEEE.

[9] Team, I. S. Bypassing anti-virus scanners. *Packet Storm Security*.

[10] Li, X., Loh, P. K., & Tan, F. 2011. Mechanisms of polymorphic and metamorphic viruses. In *Intelligence and Security Informatics Conference (EISIC), 2011 European*, 149–154. IEEE.

[11] anonymous. Polymorphic generators. In *VxHeaven*.

[12] Ferrie, P. 2008. Anti-unpacker tricks. In *Amsterdam: CARO Workshop*.

[13] Konstantinou, E. & Wolthusen, S. 2008. Metamorphic virus: Analysis and detection. *Retrieved on February*, 22, 2011.

[14] Rutkowska, J. 2006. Rootkits vs. stealth by design malware. *Black Hat, Europe*.

[15] Designer, S. 1997. Getting around non-executable stack (and fix).

[16] Shacham, H. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, 552–561. ACM.

[17] Roemer, R., Buchanan, E., Shacham, H., & Savage, S. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1), 2.

[18] Mohan, V. & Hamlen, K. W. 2012. Frankenstein: Stitching malware from benign binaries. In *WOOT*, 77–84.

[19] Chen, X., Andersen, J., Mao, Z. M., Bailey, M., & Nazario, J. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, 177–186. IEEE.

[20] Franklin, J., Luk, M., McCune, J. M., Seshadri, A., Perrig, A., & Van Doorn, L. 2008. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating Systems Review*, 42(3), 83–92.

[21] Chris, M. 2008. What is rootkit and how to detect and protect from rootkits.

[22] Ducklin, P. 1991. Tequila.

[23] Abdalla, A. 2011. Rootkits classification and their countermeasures.

[24] Reddy, D. K. S. & Pujari, A. K. 2006. N-gram analysis for computer virus detection. *Journal in Computer Virology*, 2(3), 231–239.

[25] Abou-Assaleh, T., Cercone, N., Keselj, V., & Sweidan, R. 2004. N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, 41–42. IEEE.

[26] Abou-Assaleh, T., Cercone, N., Keselj, V., & Sweidan, R. 2004. Detection of new malicious code using n-grams signatures. In *PST*, 193–196.

[27] Kendall, K. & McMillan, C. 2007. Practical malware analysis. In *Black Hat Conference, USA*.

[28] Saxena, P., Sekar, R., & Puranik, V. 2008. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 74–83. ACM.

[29] Smith, G. 2007. Principles of secure information flow analysis. In *Malware Detection*, 291–307. Springer.

[30] Lee, J., Jeong, K., & Lee, H. 2010. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM symposium on applied computing*, 1970–1977. ACM.

[31] Christodorescu, M. & Jha, S. Static analysis of executables to detect malicious patterns. Technical report, DTIC Document, 2006.

[32] Christodorescu, M., Jha, S., Seshia, S. A., Song, D., & Bryant, R. E. 2005. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, 32–46. IEEE.

[33] Tsopokis, C. V. Cuncurrent memory inspection for intrution detection. Master's thesis, University of Holloway, 2012.

[34] Sloss, A., Symes, D., & Wright, C. 2004. *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann.

[35] Hennessy, J. L. & Patterson, D. A. 2012. *Computer architecture: a quantitative approach*. Elsevier.

[36] Shannon, C. E. 2001. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1), 3–55.

[37] Denning, P. J. 2005. The locality principle. *Communications of the ACM*, 48(7), 19–24.

[38] Kilburn, T., Edwards, D. B., Lanigan, M., & Sumner, F. H. 1962. One-level storage system. *Electronic Computers, IRE Transactions on*, (2), 223–235.

[39] Wentzlaff, D. 2013. Computer architecture.

[40] Von Neumann, J. 1961. Collected works. *Oxford: Pergamon, 1961, edited by Taub, AH*, 1.

[41] wikipedia. 2014. Cache(computing).

[42] Sorin, D. J., Hill, M. D., & Wood, D. A. 2011. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3), 1–212.

[43] Papamarcos, M. S. & Patel, J. H. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. 12(3), 348–354.

A Cache Memory Simulation

B The Inconsistency Example