Anders Gaustad

Modelling Inclusive Cache Hierarchies in Multi-core Systems

Master's thesis in Computer Science
Supervisor: Magnus Själander
June 2022

Master's thesis

Anders Gaustad

Modelling Inclusive Cache Hierarchies in Multi-core Systems

Master's thesis in Computer Science
Supervisor: Magnus Själander
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

The performance of modern conventional computers are often limited by the memory latency of a system as the increase in processor performance has massively outperformed the increase in memory latency throughout the years. This causes the slow memory to act as a bottleneck, and is especially problematic for memory-intense applications. One common solution for reducing the memory latency is the use of caches, but these are only effective if they retain relevant data.

Deciding what data should be kept in a cache is decided by cache replacement polices, but these can often become complex and have non-trivial effects on other components in a memory system. Cache simulators are often used to quickly test the performance of replacement policies with varying cache sizes and memory hierarchy designs.

One interesting area of study is how inclusive cache policies affect multi-core systems – like Graphic Processing Units (GPUs) – with multiple levels of caches. While several open-source simulators exists, a literature review revealed no simulator that can simulate individual cores in a multi-core system with inclusive caches.

This thesis presents COCOASIM – a COncurrent Cache Operating Access SIMulator. The simulator is able to simulate advanced user-defined systems with multiple cores and L1 caches. COCOASIM also uses "Instruction Set Architecture (ISA)" agnostic memory traces – making it possible to define and test architecture independent systems. Experimental testing show that COCOASIM can more than 8000 Instructions per Second (IPS) for small traces (8192 operations), and more than 4000 IPS for larger traces (2097152 operations). The simulator also provides multiple fully customizable components like systems, caches, cache replacement polices, and also supports logging and dumping of statistics at certain checkpoints.

Sammendrag

Ytelsen til moderne datasystemer er ofte begrenset av hastigheten til minnet da hastigheten til prosessorer har økt mye mer enn hastigheten til minnesystemer over flere år. Dette har ført til at minnesystemer ofte blir en flaskehals for ytelsen til systemet da prosessoren må vente at minnet blir ferdig. Dette er særlig et problem for programmer med mange minneoperasjoner. En løsning for å redusere ventetiden på minner er å bruke hurtigbuffere, men disse er bare effektive hvis bufferne inneholder relevant data.

Hvis et hurtigbuffer blir fullt må noe kastes ut for å få plass til nye datablokker. Dette bestemmes av "erstatningspolicyer" ("cache replacement policies"), men disse kan fort bli avanserte siden det ofte ikke er enkelt å finne ut hva som er optimalt å kaste ut. I tillegg kan policyen påvirke (eller bli påvirket) av andre faktorer i systemet. Det er derfor vanlig å bruke simulatorer for å simulere prototyper av policyer med forskjellige parametere før de blir brukt i hardware.

Et interessant scenario det kan være verdt å se nærmere på er hvordan en inklusiv policy oppfører seg i et system med flere kjerner og nivåer med hurtigbuffere. Dette omfatter for eksempel graffik-prosessorer da disse ofte har noen private buffere med data og noen delte buffere mellom seg. Det finnes mange åpne akademiske simulatorer på nettet men virtuelt alle har begrensinger på som kan gjøres og hva som ikke kan gjøres. Ingen av de simulatorene som ble sett på i starten av dette prosjektet var i stand til å simulere minneoperasjoner per prosessorkjerne i et system med flere prosessorkjerner og nivåer med hurtigbuffere.

Denne masteroppgaven presenterer simulatoren COCOASIM – en "COncurrent Cache Operating Access SIMulator". Simulatoren er bygget helt fra bunnen av i C++, og er designet spesifikt for å simulere minnesystemer med flere prosessorkjerner og spesifiserte tråder til hver kjerne. Simulatoren tar filer med formatterte minneoperasjoner som input, og kjører disse gjennom et virtuelt system til alle operasjonene er ferdige. Siden simulatoren baserer seg eksklusivt på filer med minneoperasjoner er den helt uavhengig av dataarkitekturen som datamaskiner i den virkelige verden må forholde seg til. Eksperimentell testing viser at COCOASIM kan simulere mer enn 8000 instruksjoner i sekundet for små filer (med 8192 operasjoner) og mer enn 4000 instruksjoner i sekundet for store filer (med 2097152 operasjoner). Simulatoren har også flere nyttige funksjoner som f.eks. muligheten til å lage egendefinerte hurtigbuffere, policyer, eller systemer, og kan også logge alt som skjer i en simulasjon samt dumpe statistikk over tid under simuleringen.

Preface

This thesis marks the final submission required for completing a MSc degree in Computer Science at the Norwegian University of Science and Technology, and was written in Trondheim during the spring semester of 2022.

I would like to thank Magnus Själander for his valuable advice, feedback, and insight, that all were essential to the development of COCOASIM. I am also grateful for Magnus' patience and motivation for a project that quickly proved to be larger than expected.

Furthermore, I would also like to thank Ole Henrik Jahren at Arm, Trondheim, for his key role in this project. Ole made this project possible by donating essential memory traces used for simulation in addition to providing key insight and advice throughout the development process.

Lastly, I would like to thank my friends and family for all their support throughout the years.

Contents

Figures

Tables

List of Listings

Chapter 1

Introduction

In a modern and continuously technological advanced world, the demand for intelligent technology and novel solutions is higher than ever before. As humankind enters an age of world-wide digitization, smart devices and integrated components can be found virtually everywhere - from smartphones to sensor-equipped refrigerators and Bluetooth-compatible cars. Ignoring conventional "offline" devices and integrated systems, there are about 14.9 billion mobile devices in the world as of 2021 - each containing embedded processors and other smart hardware. Furthermore, the number is expected to only rise in the future to an estimate of over 18 billion by 2025 [1].

With an ever-increasing demand for higher performance, these devices are often equipped with multiple processor cores, fast specialized memory (known as "caches"), or/and accelerators like Neural Processor Units (NPUs) for machine learning or Graphic Processor Units (GPUs) for graphic workloads. Whereas accelerator conventionally were reserved for desktops and larger computer, they have become increasingly common in all types of computers. This makes the devices faster, smaller, and more efficient, but also increasingly complex as they require advanced hardware.

While the technological progress of the last decades is impressive, it also comes with challenges. To maintain a increasing performance in technological devices over time, new solutions or specialized components need to be developed. The increased performance that could be harnessed from continuously smaller transistors is no longer possible to use due to power constrains, so researchers need to find new ways make system more efficient. One solution that has become increasingly more common over the last few decades is to introduce multiple cores – allowing a device to perform logic in parallel. While it is no longer feasible to increase a system's clock rate as it will lead to a disproportionate increase in energy usage, it is possible to increase the number of cores while still using a lower clock rate. By *parallelizing* programs and workloads over multiple cores, the overall performance can be increased while keeping an acceptable energy usage. In the end, this leads to better energy efficiency than if using a single core with a higher clock rate. However this requires programs to be designed in a way that makes it possible to run operations in parallel over multiple cores. Though increasing core count is not a silver bullet, this approach is especially effective for highly parallelizable programs. Coincidentally, there are many prime examples of workloads that benefit greatly from parallelization – like machine learning, cryptography, and graphic processing.

Both performance and energy efficiency are important when performing graphic processing. Though Graphic Processor Units (GPUs) have high power usage, they are able to achieve high performance because of their innate parallelism. However, as a consequence of the "Memory Wall" [2], high-speed memory storage is needed to keep up with the speed of the computation for a workload to avoid being slowed by a memory bottleneck. This is often solved by introducing *caches* – a fast but expensive memory typically located close to the processor cores. The caches increase performance by decreasing the latency of looking up a memory address given that the requested data is present in the cache. As caches only have a capacity equal to a small fraction of the main memory, the data in caches are continuously replaced using *replacement policies* – acting as rules for what to place in or evict from a cache. Though caches often are associated with Central Processing Units (CPUs) rather than Graphic Processor Units GPUs, they are critical for keeping a high throughput and performance when dealing with graphic workloads and other GPU-compatible workloads.

As systems – both CPUs and GPUs – become increasingly more advanced, finding efficient replacement policies and other novel performance-increasing methods become harder to find. Testing new ideas on real-world hardware may be infeasible, slow, and expensive as custom components and logic need to be taped out physically. A feasible alternate to this is to test the behavior in software using a *simulator*. This enables a user to experiment with an implementation – like a cache replacement policy – and continuously make changes or variations by altering the code. Advanced accelerators and cache replacement policies do often have a non-trivial impact on the behavior of a program – making software simulations often the only practical way of testing new ideas.

Due to their high practicality, software simulators for computer architecture have been used and developed continuously since the 1990s [3]. Many simulators are created by computer system manufacturers for internal use, but several open-source simulators – like gem5 [4] and Sniper [5] – also exist. However, as a simulator only is a virtual, imperfect representation of real-world hardware, each simulator often have distinct strengths and weaknesses. Open-source simulators are often created to be as generic as possible, but do not act as a silver bullet for every use-case or experiment. As the list of possible modifications one could do to a computer system is virtually endless, the simulator often needs to be specialized or at the very least have rigid limitations. Consequently, the only practical way to guarantee that a specific use-case can be tested is either to make alterations to an existing simulator, or develop one from scratch.

As a consequence of the natural limitations of software representations of cache systems, a desired feature is not guaranteed to be supported in every simulator. When experimenting with a new accelerator or replacement policy, a few requirements therefore need to be fulfilled:

- The simulator needs to support the desired feature – be it a new accelerator component, a custom replacement policy, or something different.
- The simulator needs to represent an underlying architecture that the feature should be tested with.
- The simulator needs to be easily configurable and scalable so that different variations, architectures, and programs can be tested with the desired feature.
- The simulator must have an acceptable scaling for large inputs in both memory usage and time so that inputs of all sizes can be appropriately tested.

This thesis presents the custom-made simulator COCOASIM, which meets the aforementioned requirements when testing coherent memory accesses to distinct caches in a system. The simulator accepts realistic trace representations of memory operations supplied by Ole Henrik Jahren from Arm, Trondheim, and mimics the behavior of the memory hierarchy of a multi-core GPU. Opposed to other common simulators, COCOASIM accepts memory traces directed to individual cores/caches instead of a signle input/output interface. As the simulator accepts memory traces rather than emulating an executed application, the systems can be simulated independently of the underlying Instruction Set Architecture (ISA) that other simulators need to consider. Thus, other simulators that can generate memory traces from ISA-specific applications can be used to generate input for COCOASIM. Additionally, the simulator is highly configurable, and accepts a wide array of user-defined cache configurations with pre-made or custom cache replacement policies. The simulator was also developed with performance in mind, and uses an event-based programming paradigm to simulate cache accesses as distinct events. The following chapters consider the motivation, design, and performance of COCOASIM, but also presents results of replacement polices and cache configurations tested using the simulator.

One thing that makes this thesis special is that it focuses on two different topics rather than a single one: 1) The software paradigms and design choices used when developing the simulator, and 2) The architectural design of computer systems – including components like caches and replacement policies – that are used in simulation. As COCOASIM is a software simulator used to simulate hardware, these subjects often overlap with each other throughout the thesis.

Chapter 2

Related Work

2.1 Introduction

Simulators for computer systems and cache behavior have been developed and published free of charge in the past, and new simulators still appear from time to time. Many simulators – especially the more popular ones – are maintained and expanded upon as scalable open-source projects, and continuously gain more features and increased compatibility over time.

This section presents three existing cache simulators:

1. The popular gem5 simulator is presented in Section 2.2.
2. GPGPU-Sim – a simulator designed to analyze Nvidia CUDA workloads – is presented in Section 2.3.
3. Lastly, the Sniper simulator that uses analytical modeling when emulating programs is discussed in Section 2.4.

2.2 gem5

The gem5 simulator [4] is described as a merge between the ISA- and CPU-focused M5 simulator [6], and the memory- and coherency-focused GEMS simulator [7]. Combined, they form the gem5 simulator, which is able to simulate entire computer systems – including CPU, caches, and bus activity. This makes gem5 a full system simulator rather than just a cache simulator, and the memory and cache behavior is just one part of many components being simulated. Out of the box, gem5 provides ready-to-use CPU, memory, and ISA models that all can be used with relative ease. Additionally, the simulator is highly configurable – including customizable architectures, CPUs, memories, and interconnects. Furthermore, gem5 includes an advanced cache coherency interface that can be utilized to create a wide array of coherence protocols [8]. This coherency protocol – while advanced and highly flexible – is meant to describe cache states and coherency rather than replacement policy.

Every component in gem5 that can be used in a simulation is defined as a "SimObject". In practice, all components are built on top of the C++ class which the components with a minimal interface so that gem5 is able to call certain functions on them. For example, every SimObject has to be initialized by the gem5 simulator and thus needs to have methods such as `init()` or `load_state()`. Since every component derives from the same super-class, gem5 is able to use a common way of communicating with *all* of them – despite every component being unique behavior.

Though gem5 simulator was initially designed for CPU and memory emulation, it does currently also support simulation of GPU systems thanks to new features being added over time. This is made possible due to the generic and scalable development interface of gem5 that makes it possible for new, custom components to communicate with the rest of the system. There are currently two ways of simulating GPU behavior in gem5: using the bundled GCN3 GPU model [9, 10], or with the help of the gem5-gpu fork [11].

In addition to several bundled CPU models, gem5 also includes AMD's Graphic Core Next 3 (GCN3) GPU model. This model is able to accurately simulate GPU behavior for workloads, but only for GPUs built on the GCN3 architecture. In return, the model comes with a powerful software stack that can be used by developers to, i.e., create advanced GPU cache coherency models. The GCN3 model is able to simulate a program in gem5's system-call mode, but requires the emulated program to be built and compiled using AMD's Radeon Open Compute platform (ROCm). In the end however, when the GCN3 GPU model is used in a simulation that runs a binary compiled with the ROCm tool-chain, gem5 is able to accurately simulate GPU behavior.

There is also a fork of the gem5 repository named gem5-gpu that extends gem5 by integrating it with the GPU capabilities of the GPU simulator GPGPUSim [12]. In practice, this is a merge between the two simulators where gem5 sends and receives data to and from GPGUSim. Operations - such as loads and stores - are initially handled by the gem5 simulator, but are propagated to GPGPUSim through gem5's generic port interface. In short, this allows gem5 to run programs as normal and re-route requests to GPGPUSim through an interface while still having control over timing and the overall state of the system. The GPGPUSim itself is discussed further in Section 2.3.

2.3 GPGPUSim

The GPGPU-Sim simulator [12] is a GPU simulator released in 2009 made to analyze general-purpose GPU workloads. The simulator uses Nvidia's CUDA programming model [13] to run instructions using the same data as parallel threads distributed over the GPU's many cores in what is called a "single instruction, multiple thread" (SIMT) model. While the simulator originally was created to analyze the behavior of CUDA workloads, GPGPU-Sim can run any compiled CUDA or OpenCL [14] executable. An overview of the design of GPGPU-Sim as well as the logic flow of simulating applications can be seen in Figure 2.1.

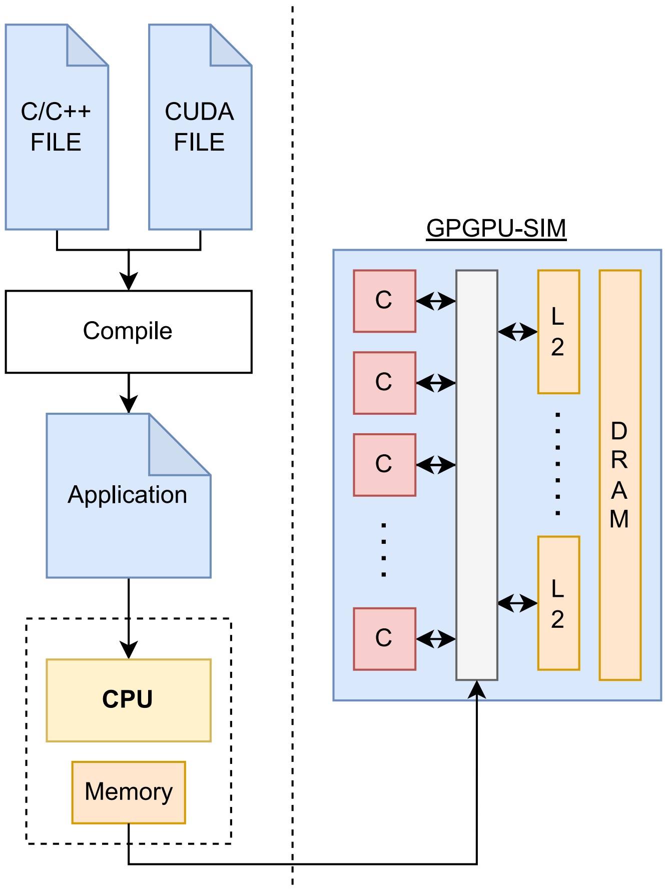


Figure 2.1: The design and logic flow of GPGPU-Sim.

In short, GPGPU-Sim works by defining the simulator as a highly parallell processor in the CUDA model. This results in the CUDA instructions being sent to the simulator which in turn can simulate their behavior. Note that GPGPU-Sim is not primarily a cache simulator, but still provides the option of experimenting with different cache sizes. Otherwise, it is also possible to alter several other properties like the number of maximum threads or registers per core or the DRAM latency.

While GPGPU-Sim is designed to simulate the behavior of GPUs executing non-graphic workloads, it does only accept application compiled along with CUDA or OpenCL libraries. In the same way as gem5's GCN3 GPU only accept applications built with ROC platform, the programs simulated by GPGPU-Sim must be created using CUDA.

2.4 Sniper

Sniper [5] is a simulator that can be used to emulate behavior for systems that use multiple cores. As with gem5, Sniper can run emulate complete computer systems, but has also the option of exclusively simulating cache behavior.

What makes Sniper unique is that it simulates system using *interval simulation* [15] rather that conventional emulation. In short, Sniper uses an analytical model to estimate the delay caused by various misses like branch mispredictions, and cache- and Translation Lookaside Buffer (TLB) misses instead of tracking the individual instructions. For example, on a cache miss Sniper does not actually perform a propagating request to the lower memory but rather estimates when the requested data is fetched. For example, if a cache miss occurs Sniper does not propagate the request to lower level memory, but rather estimates when the referenced data block arrives in the cache. Likewise, the simulator also uses analytical modeling to estimate how many cycles are spent on branch mispredictions and TLB misses. This makes the system less cycle-accurate as the analytical model is based on prediction and not actual simulation, but also result in a notably higher performance as the interval simulation enables Sniper to simulate the cores in parallel. The developers of Sniper also shows that interval simulation actually increases the overall accuracy of a simulation compared to the conventional one-instruction-per-cycle approach. This happens primarily because the approach used for comparison only commits one instruction per cycle regardless of the amount of cores – thus entirely ignoring Instruction Level Parallelism (ILP).

Sniper can simulate the behavior of a program either through emulating an application or by parsing an instruction trace. This is made possible by using the Graphite simulator [16] and Intel's Pin tool [17] as a base. In short, Pin is able to extract instructions and addresses from a `x86`-binary. Sniper uses Pin as an interface to capture instructions in a given application which it then simulates.

Chapter 3

Background

This section introduces the required background knowledge to fully understand how COCOASIM works. Note that because of the interdisciplinary nature of this thesis, both computer architecture – including caches and replacement policies) – and software development – including classes, polymorphism, and event-driven designs – is covered in this section. In summary, four important concepts are introduced:

1. **Caches, Section 3.1** – Explaining how caches work and why they are used.
2. **Cache Simulation, Section 3.2** – Motivates why performing simulation of cache systems is feasible, and how it is achieved.
3. **Inheritance & Subtype Polymorphism, Section 3.3** – Introduces important software concept used to create the simulator.
4. **Discrete-event Simulation, Section 3.4** – Describes a simulation paradigm of "discrete-event simulation", and why it is useful.

3.1 Caches

3.1.1 Motivation

While memory performance has increased significantly over the past years, it severely lags behind the increase in CPU performance. This concept is often referred to as the *memory wall* [2], and leads to the memory acting as a bottleneck for overall system performance. To counter this, small but fast memories called *caches* are added close to the CPU. The caches are designed to be significantly faster to access than the main memory, but are larger and more expensive. As a practical consequence, caches have a limited capacity and needs to maintain its content intelligently.

There are two forms of modern computer memory: Static Random-Access Memory (SRAM) and Dynamic Random-Access Memory (DRAM). Both of these forms enable a computer to read and write data as it pleases, but are designed in different ways. One "memory cell" of Dynamic RAM can be built using only two components – a capacitor and a transistor – while a Static RAM cell requires six.

This in turn makes DRAM smaller and cheaper to produce. However, there are primarily two disadvantages with DRAM cells: 1) The cells need to be continuously refreshed to retain its state as the capacitor discharges over time, and 2) They are significantly slower than the SRAM counterpart. While the refresh takes up time and somewhat affects the performance of a DRAM cell, the main reason for the slow access is because of the small transistor size. In short, this happens because smaller transistors take longer to switch a bit than larger transistor. SRAM cells on the other hand have larger transistor – making them bigger but also slower. Additionally, SRAM cells do not require refreshing as it is not reliant on a capacitor. This leads to DRAM being highly preferred for memory where capacity is important – like the main memory – and SRAM only being reserved for memory that need to be exceptionally fast - like registers and caches.

3.1.2 Design

As SRAM cells are larger and more expensive than DRAM cells, cache capacity is typically only a fraction of the total main memory capacity. While the RAM of an average desktop computer often is in the range of 8 GiB to 32 GB, caches are typically of sizes between 256 KiB and 8 MiB. This makes it practically impossible to store any substantial amount of data in the caches perpetually. Instead, data in caches are continuously replaced to ensure that only the most relevant data stays in the cache at any time. More specifically, caches want to retain data that is likely to be requested multiple times in the future, while evicting data that is not being used. How to do this naturally depend on what program is being run, but virtually all computers follow a pattern known as "Locality of Reference" that makes it easier to predict what data is going to be reused in the future. The phenomenon of "Locality of Reference" dictates the following:

1. Data with addresses *close to previously accessed addresses* are more likely to be accessed in the future. This is called *spatial locality* or *locality in space*. As caches fetch more than a few bits at a time, the surplus data fetched have a high probability of being used. Caches may also use this property to perform an operation known as *prefetching*, but this is outside the scope of this project.
2. After an address *has been referenced once*, it is likely to also be referenced some time in the future. This is called *temporal locality* or *locality in time*. This is widely used in caches as it always tries to store the referenced address in the cache. Additionally, efficient replacement policies – discussed later in this section – like LRU attempt to keep relevant data in the cache for as long as possible.

If requested data is present in the cache, the system saves a substantial amount of time by only having to access the quick cache instead of the slower main memory. This is often referred to as a *cache hit*, while requesting data that is not present is called a *cache miss*.

```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
    char* string = argv[1];
    int as = 0;

    char c;
    int i = 0;
    do
    {
        c = string[i];
        printf("%c", c);

        if (c == 'a')
            as++;   // Increment number of as

        i++;
    }
    while (c != '\0');

    printf("\nThere are ");
    printf("%i", as);
    printf(" as in the input");

    return 0;
}
```

Listing 1: Example of a program that can be exploited by caches

Consider the example presented in Listing 1. This simple program takes a single string argument, reads it, count the number of times the letter a occurs, and then prints every letter to standard output. Keep in mind that the cache itself does not know how the program looks, but instead reacts on the memory requests issued by the CPU. Note the following:

- There are two arrays in the code: `argv` – the list of arguments – and `string` – the list of characters making up a string. While `argv` is only used once, the `string` array is used in the `while`-loop and thus accessed multiple times. Since the elements of the array are placed next to each other, this results in `string` having *spatial locality*. This should cause the cache to miss once the first time it is accessed, but hit on the other loops as a large part of the array (often 64 Bytes) exist in the cache.

- Note that `c` and `as` are accessed repeatedly inside of the loop. Technically, the `c` may be inlined by the compiler, but the `as` needs to be present as it is accessed at run-time. This implies that the `as` variable has *temporal locality* as it is accessed multiple times in succession.

Note that in both of these cases the caches do not know that these addresses are reused, but rather "discover" it as the program is executed. Data put in the cache is often reused either because an address in an already fetched range next is used, or because the exact same address is reused multiple times.

In Listing 1, it is easy to see that `as` should remain in the cache, while the other data like `argc` can be safely evicted. However, for most programs, no simple answer exists on what should be cached and what should be not. When a cache miss happens in a full cache, the system needs to choose one entry to evict from the cache (or more specifically the cache *cache set*, more on this later). To figure out what to evict and what to keep, the cache uses an algorithm known as a *replacement policy*. When something misses – or hits in the case of some replacement policies – the cache queries the replacement policy what to evict. There are many different replacement policies, but some include *Random*, *First In First Out (FIFO)*, and *Least Recently Used (LRU)* [18]. These works as follows:

- **Random** will simply choose one entry to evict at random.
- **FIFO** will put the requests in a queue, and evict the one at the front when missing, while putting the new entry in the back.
- **LRU** will tag each entry with a counter, where higher values indicates that the entry has been more recently used. These counters are updated when a request hits in the cache, and the entry with the lowest counter is evicted upon a cache miss.

In addition to cache replacement policies, the cache also uses a *placement policy* to place a data block depending on its address. While replacement policies tell the cache what to replace upon a cache miss, cache placement policies tell the cache where in the cache to place the new entry. There are three categories of cache placement policies:

- **Direct-mapped**: When given a memory address, the cache knows exactly where to place the data block. Each memory address is assigned exactly one slot, but multiple addresses may map to the same slot.
- **Fully associative**: Any memory address may exist anywhere in the cache. The cache treats every memory address the same, and all entries share the cache capacity.
- **Set-associative**: Memory addresses are filtered to different *sets* in the cache, but otherwise share the capacity of each set. This is seen as a trade-off between the direct-mapped policy and the fully associative policy.

Each of the three placement policy categories have their own advantages and disadvantages:

Direct mapped caches are simple, energy efficient, and do not need any replacement policy. The cache can check if an access leads to a hit or miss easily, as it simply needs to check the content of the direct mapped slot – which in turn also requires less energy. If this leads to a miss, it simply needs to evict the old block and insert the new – eliminating the need for a replacement policy. However, direct-mapped caches often have a lower hit-rate than the other variants, as every new memory access mapped to the same slot will cause a miss.

Fully associative caches work in the opposite way of direct-mapped caches by enabling every access to be mapped to *anywhere* in the shared cache. This enables full utilization of the cache as no "slots" are unused since the cache will eventually fill up completely. This approach also generally achieves the highest hit rate of all variations, as all recent memory access are cached regardless of their address. The disadvantage to this design is that every entry of the cache needs to be checked on a new access to determine if the access is a hit or miss. This process also has a high power consumption, as the cache needs to go through all of its content before it can fully determine if an access is a miss. Additionally, the metadata overhead is large as every entry must use a long bit-tag to properly distinguish each address – making the hardware requirements for these caches expensive.

Set-associative caches work as a trade-off between the two opposites. Instead of directly mapped "slots", set-associative caches divide the cache into sets. Depending on the size of each set, the cache is said to be n-way associative - where n indicates how many "slots" exist in each set. As implied, this makes direct-mapped caches "1-way associative", while fully associative caches only have a single set and thus the maximum number of slots. Set-associative caches tries to balance the performance of fully associative caches with the cost/energy-efficiency of direct-mapped caches. Though no objectively optimal balance exist, many caches use an associativity of either four or eight. Larger caches – primarily L2s and lower – may also have an even higher associativity to create large enough sets.

In practice, the "slots" mentioned in the paragraphs above are commonly referred to as *cache lines* or *cache blocks*. Whenever a cache miss occurs, the cache (often) fetches more data than needed in order to fill an entire cache line. As mentioned earlier, the reason for this is to enable spatial locality as other data in that line is likely to be used in the future. While the size of cache lines theoretically may differ from cache to cache, a cache line is more often than not 64 Bytes.

3.1.3 Implementation

Though early computer systems would simply connect a single cache between the main memory and the CPU, the implementation of caches have greatly changed since then. Caches are now placed in a *cache hierarchy*, with multiple levels of different caches that support each other. The first level of caches are typically called L1 caches, and is what all memory requests from the CPU initially access. If a memory request miss in the L1 cache, it is forwarded to the level below it, which in this case is the L2 caches. This continues until the request eventually hit or there are no more levels of cache. If the request miss in the final level of cache (often called the Last Level Cache or LLC), the request is instead forwarded to the main memory. Though there are no limits on how many levels of cache that may exist, most system operate with two, three, or rarely even four levels of cache as of 2022.

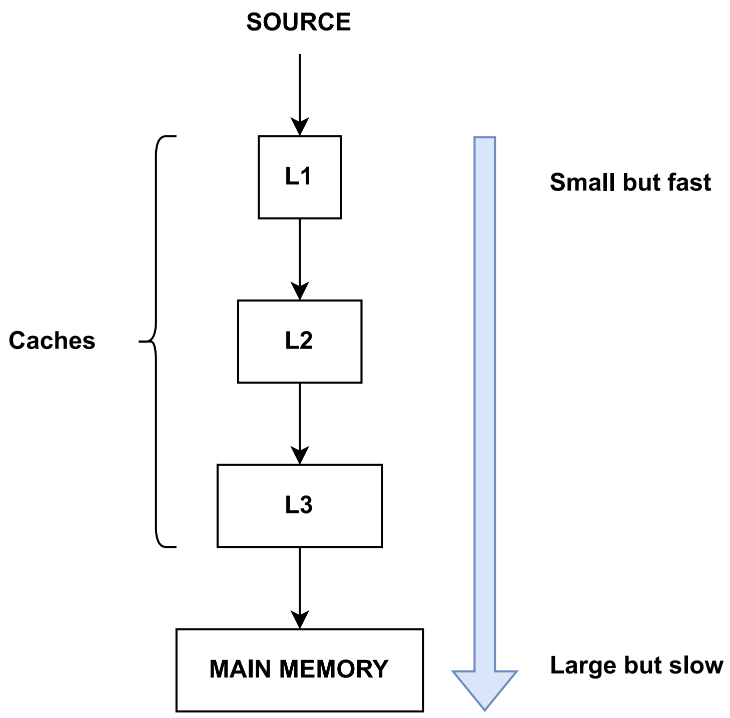


Figure 3.1: A basic cache hierarchy with three levels.

An example of a basic cache hierarchy with three levels can be seen in Figure 3.1. Note the following:

1. The L1 cache is shown as a single entity. While this was normal for early computer systems, it is now much more common to split the L1 cache into an *Instruction cache* (L1-I) and a *Data cache* (L1-D). As the names suggest, the L1-I stores instructions while the L1-D stores data. With that in mind, figures and examples in this document will only display one singular L1 for the sake of simplicity.
2. The hierarchy consists of only a single cache on each level. While this works for single-core processor systems, multi-core processors – which dominate the modern market – usually have a L1 cache dedicated to each core. This often makes the design of the cache hierarchy resemble an inverse tree, as seen in the example in Figure 3.2.

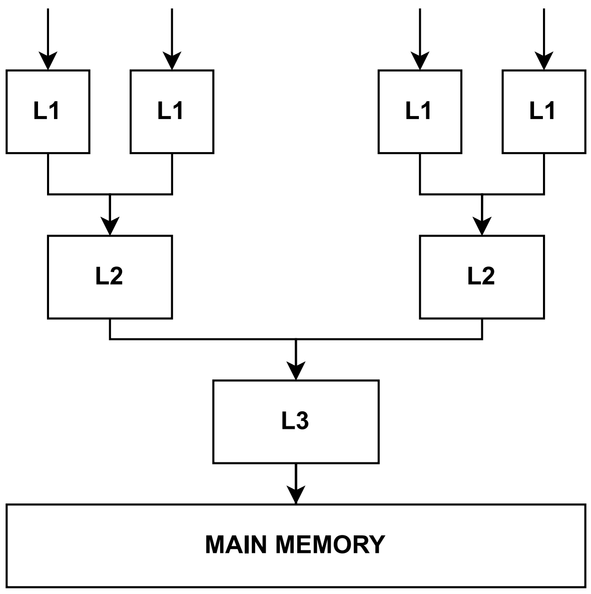


Figure 3.2: A cache hierarchy with four cores/L1s.

In addition to having a placement and replacement policy, caches also operate with an inclusion policy. This policy indicate what relationship there should be between instances of the same data in different caches in a cache hierarchy. In other words, can a data entry in the L2 affect the content of a L1 cache – and if so, how? There are three options of implementing this: either by including or excluding data in the lower levels, or by not enforcing any restraints at all.

There are three types of inclusion policies:

1. **Inclusive Policy** dictates that every entry in a L1 also exist in the caches of lower levels. This means that every entry of a L1 is guaranteed to also exist in a L2. Note that data may exist in the L2 without being in any L1 caches, and that if an entry is removed from the L2,2 it must also be removed from the L1 to ensure that the rule holds. When a data block is loaded from the main memory, it is stored in all levels of the cache - ensuring that every cache has their own copy.
2. **Exclusive Policy** is practically the opposite of the inclusive policy. When data is loaded from the main memory, it is exclusively placed in the L1. Whenever something is evicted from the L1, it is moved to the L2 instead of disappearing from the cache hierarchy.
3. **Non-Inclusive Non-Exclusive (NINE) Policy** is – as the name suggests – not inclusive nor exclusive. It does not impose any rules on where a data block may exist, but does install a data block in all cache levels when loading an address.

3.1.4 Non-blocking caches

As discussed earlier in this section, the significant advantage of caches is the decreased memory access latency when accessing data present in the cache. Caches lead to an increase in overall performance regardless of its implementation due to caches and SRAM simply being faster than the DRAM of the main memory. However, this is all under the assumption that the program hits in the cache. When a memory request misses in the cache - and all other caches in the hierarchy - it needs to fetch the data from the main memory. The caches want to maximize its hit rate to minimize the times the data has to be retrieved from the slower memory. For simplicity, the rest of the section will assume the following:

1. All requests are sent by a CPU to a single L1 cache.
2. The cache contains `A` and `B` (meaning these will hit), but not `X` (meaning this will miss).
3. The CPU sends three requests to the cache for the addresses `A`, `X`, and `B` - as shown in Table 3.1.

Cycle	Request	In cache?
1	`A`	✓
2	`X`	✗
3	`B`	✓

Table 3.1: Example of three requests accessing a cache.

Depending on the design of the cache, every cache miss may act as a bottleneck. Whenever a memory request results in a cache hit, it may simply return the data on a read or store the new data on a write. After responding to the cache hit, the cache is immediately ready for any new requests. However, this is not the case for cache misses. Every miss causes the cache to perform replacement logic, an eviction, and a request for the data from a lower level cache or the main memory. Depending on the implementation, the cache may need to wait and stall for the requested data to appear before returning the data to the CPU. The CPU has no knowledge of if the requested data is present or not, so it simply waits for the data until it is ready. This type of cache is called a *blocking* cache, as misses causes the pipeline to stall as it is blocked.

Consider the example of Figure 3.3. As request `A` hits in the cache, its data can be returned immediately, and the CPU is able to continue as normal. However, when request `X` misses, the CPU must stall until the data of `X` is fetched and forwarded. Note that request `B` hits, but the data cannot be returned before *after* the miss logic of request `X` has finished.

The opposite of a blocking cache – a *non-blocking* cache – is designed to circumvent the penalties of cache misses. Note that while the miss logic of request `X` was pending in Figure 3.3, the cache could in theory handle another request in the meantime. A cache hit – as with request `B` – makes it possible to return the data of another request while simultaneously waiting for the return of re-

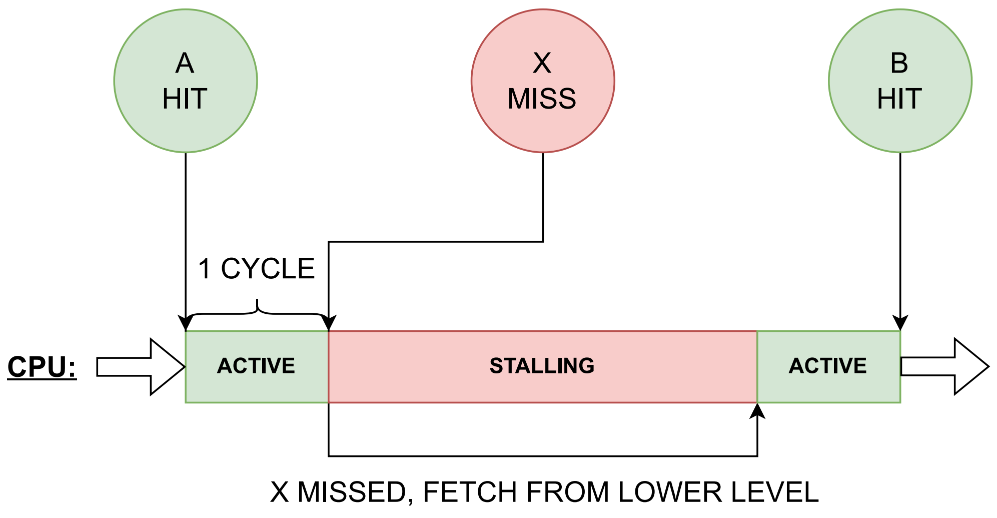


Figure 3.3: A blocking cache - request X stalls the cache pipeline

quest X, but would require support for Out-of-Order (OoO) processing. Another cache miss would cause the cache to issue a new request in parallel with the old one – ultimately hiding the delay. Though non-blocking caches are more advanced than their blocking counterpart, this implementation circumvents the problems of blocked pipelines and only stalls when the CPU signals that it needs certain data to continue. The same example using a non-blocking cache is shown in Figure 3.4.

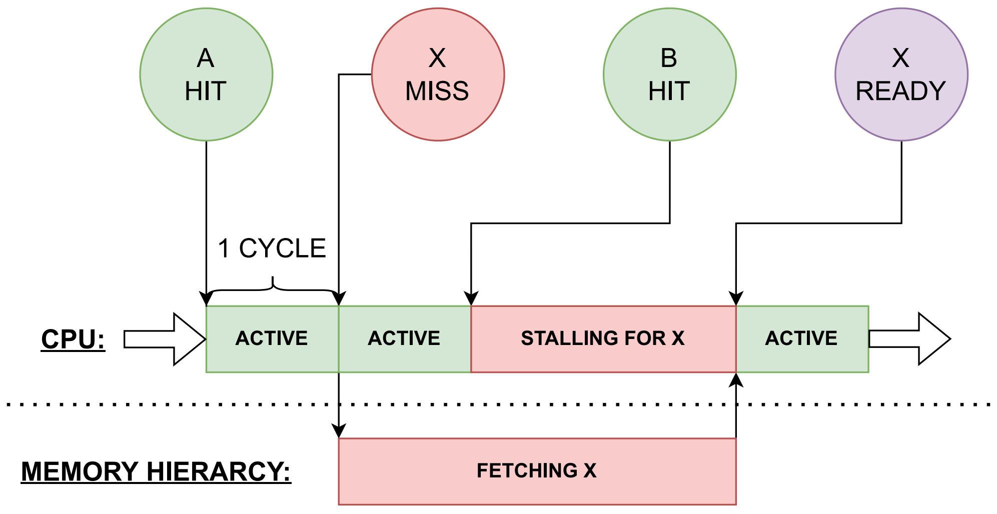


Figure 3.4: A non-blocking cache – reducing the stall from request X.

3.2 Cache Simulation

While caches are fast and efficient, they can only keep a finite amount of data at a time due to their limited size. Thus, it is important to only retain the relevant data while evicting unused addresses. How software simulators can be used to experiment with virtual caches is discussed in Section 3.2.1, while different types of cache simulation is discussed in Section 3.2.2.

3.2.1 Features of Cache Simulation

Creating simulators to test a desired behavior has become increasingly common for both the industrial and scientific community. As accelerators and replacement policies become increasingly complex and specialized, testing new functionality is practically impossible without a simulation tool. Emulating a system in software rather than hardware makes it easy to implement a feature, test it, and adapt and make changes. Though the simulated system is an imperfect representation of real world computer systems, new features can be tested using code without the need of actual hardware. Though this section will focus on simulators for caches in particular, many of the mentioned concepts apply to all simulators in general.

In addition to simulator tools used by the industry for internal projects, there are multiple open-source simulators cache that available for the general public. The scope of these simulator varies greatly in what they are meant to do – with some simulators testing entire computer systems while others being limited to smaller subsystems. When designing simulators in software, many aspects may make it feasible to limit the simulator in one way or another. One natural reason for this is time as it is virtually always possible to add new features to a simulator. However, software projects in general tend to have a diminishing return for every new feature added. For a cache simulator, that means that major components like CPUs or caches bring much more functionality than specialized components like Write Buffers (WBs) or Translation Lookaside Buffers (TLBs). As such, a prototype with limited functionality can be created rather quickly, while an advanced, realistic simulator may take a long time to develop. However, it may still be feasible to limit a simulator despite given infinite time as to curb the complexity. The number of features a simulator needs to have for it to be a viable representation of a real world system is ultimately decided by scope of the development process. A system that is meant to simulate cache behavior essentially only needs to consider the memory hierarchy, and not other logical components that may be part of a larger computer system. Simpler systems may also be easier to use, and can represent a broader array of computer architectures at the cost of decreased realism and less accurate results. The balance between simplicity and complexity is ultimately decided by the developer – making sure the simulator is advanced enough while being simple to use.

As each simulator is uniquely designed and has its own strengths and limitations, there is no "silver bullet" simulator that can do everything. Instead, a simulator may have a feature not available in another, and vice versa. The choice of what simulator to use is therefore driven by the requirements or features needed to conduct an experiment. Though many of the essential features of a simulator is communicated clearly, limitations and assumptions are often vague. When surveying 28 different CPU cache simulators, Brais et al. classifies simulators using a list of properties [19]:

- **Type** – Either *Functional* – with no notion of time - or *Timing* – keeping track of time by counting cycles.
- **Mode** – The way the simulator works – either by *Executing* or *Emulation*, or through the use of a *Instruction Trace* or *Memory Trace*.
- **Level** – Either on an *Application* or *Full System* level – indicating if simulator supports instructions from a single program or the entire operating system.
- **Scope** – The confinement of the simulator – essentially if the simulator is for caches only, or support entire computer systems.

3.2.2 Simulator Classes

Type

Functional simulation is relatively simple as the simulator only need a notion of order and not time. The input memory requests interact with the same logical components as in a timing simulation, but the simulator does not know when events happen. A functional simulator may know that some logical operation will happen in a memory hierarchy – like a cache hit or miss – but not know when this operation starts, ends, or what happens in the meantime. Thus, a developer only needs to consider the logical behavior of a component when receiving a memory request – like the replacement logic of a cache miss – and not the convoluted logic of correctly timing each event. This makes functional simulation generally much simpler than timing simulation, but also less realistic. However, functional simulation has several uses – e.g., verifying that the components work as intended or getting a rough overview of important events like number of accesses to a cache.

Timing simulation works in many ways as an opposite to functional simulation as it is more complex to develop but also a more realistic representation of a real system. The simulator needs to keep track of time by knowing when something is happening, how long some operation takes, and how many simulated cycles have passed since the simulator started. Time is often measured in a number of "cycles" – where each cycle represent a tiny unit of time. The simulator can then schedule the effects of each logical operation using a delay in cycles to emulate a passing a time. For example, the simulator may receive a memory access from the CPU that is meant to access a L1 cache. If this happens 20 cycles after the simulator began and the access to the L1 cache is going to take 5 cycles, the simulator needs to execute the actual access in cycle 25. Many other events may happen in the meantime

– e.g., another request accessing the same L1 cache in cycle 23. Furthermore, an independent access to the main memory may happen simultaneously – forcing the simulator to keep track of every operation to every component all the time. This makes designing timing simulators considerably more complex than functional simulators, but allows for more realistic and advanced interaction between events in the system.

Ensuring that every logical operation happens at the right time in a timing simulator can be done in mainly two different ways. One solution is to simulate every component for every cycle in case something affects it. For example, a component – e.g., a L1 cache – may know that something going to access it in five cycles. As the simulator emulates the behavior of every component the next cycles, the L1 cache has no change in the first four cycles. On the fifth cycle however, the simulator executes the logical operation of the cache access. In this way, the simulator has accurately represented the desired logic as it took five cycles from the access was scheduled until it was executed. In practice, this could be achieved by telling a component to perform some behavior on cycle 5 and simply wait for a function call. A basic example of this is shown in Listing 2.

```
#include ...

#include <cstdint>
#include <vector>

class Component;

int main()
{
    std::vector<Component*> components = get_components();

    uint64_t cycle = 0;
    while (!simulation_is_finished())
    {
        for (auto * component : components)
        {
            component->simulate(cycle);
        }

        cycle++;
    }
}
```

Listing 2: Simulating a system by performing some behavior on every component every cycle.

However, this approach is inefficient – especially for systems with many components or long delays between scheduling an operation and executing it. Consider a system with several cores, and that a L1 cache is assigned to each core. Regardless of how many of the caches actually are active in a given cycle, they all need to simulated to account for an access taking place. Each added cache in a configured system corresponds with another component that needs to be simulated every cycle – meaning that the performance will suffer disproportionately with the number of caches. Likewise, a simulation with long execution delays will lead to a lot of cycles with nothing happening. If the simulator is fed a single instruction that takes a very long time to execute – e.g., a load that takes 10 000 cycles – the system is forced to simulate every cycle until something eventually happens.

An alternate approach to representing time can be done by keeping track of the events themselves. Instead of simulating every component every cycle, it is possible to only simulate the events on the cycles when some sort of logical operation executes. As every event – like memory accesses – knows where in the system it happens and how long it takes to execute, the system may simulate the events as they happen. Instead of iterating over every component and simulating them every cycle, the simulator may instead over a queue of events and simulate them at certain moments. This is much more efficient than the previously mentioned approach as the system is agnostic to the actual components in the simulated environment and the delay between issuing and executing operations. Consider the same examples mentioned above using this approach. As the number events – rather than components – dictate the performance, an increase of simulated components do not directly increase the time spent. As many components may be inactive most of the time – i.e., not have any logical operation manipulating it – the simulation of these can be skipped. Likewise, the simulator does not need simulate every cycle for operations with long delays. Given a theoretical program of a single load using 10 000 cycles, the simulator may only simulate the initial cycle and the one 10 000 cycles after it as nothing happens in the meantime. The gem5 simulator uses this technique, and calls it *event-driven simulation*. This is further discussed in Section 3.4.

Note that very few simulators are actually 100% cycle-accurate as virtually every simulator makes some of assumption that makes the software an imperfect representation of real hardware. In summary, timing simulation is more accurate (but also slower) than functional simulation, but will still deviate from an actual real-life system. There are also varying degrees of tradeoffs between accuracy and performance within timing simulators – like the analytical modeling and interval simulation done by Sniper discussed in Section 2.4. It might be possible to create sub-categories for how timing simulators perform simulation, but this thesis will only focus on the timing and functional simulation types for the sake of simplicity.

Mode

Every simulator has the same base goal – i.e., simulating the logical operations in a computer system – but may often differ in the way they achieve this. Though all simulators are uniquely designed, one major reason for this difference is that the simulators uses and manipulates input distinctly. While some simulators are able to run another application natively, other simulators may only accept a trace-representation of a program. For example, gem5 can run a simulation through *emulation* or by feeding it an *instruction trace*. Note that *mode* refers to more than input as the way the simulation is conducted is vastly different for e.g. emulation and execution than a memory or instruction trace.

A simulator that works by execution will run a given program natively while capturing its behavior. The executed application then communicates with the simulator through the low-level software interface known as the Application Binary Interface (ABI). However, for this to work the run application needs to have a compatible interface with the simulator. In return, the application runs fast and accurately.

Emulation is reminiscent of execution, but is more general and scalable for the executed application. Instead of running the program natively, the application runs in an encapsulated environment. The benefit of this approach is that any application can be used – including programs without a compatible Application Binary Interface. The trade-off for this is that emulation requires a higher overhead in form of environment management. Additionally, emulating is done by performing system calls for both the application and OS. When testing cache behavior for a program, the OS system calls may not be relevant or desired.

Lastly, simulators accepting instruction and memory traces are often far easier to develop than ones using execution or emulation. Instead of running a program and having it communicate with the simulator, the system is instead given a file with instructions or memory accesses. In its simplest form, this may be no more than a text file where each line represents an address to access. The format of these input trace files may differ greatly among simulators based on what information the simulator needs. During simulation, the trace files are read and converted into operations that manipulate the system. Though this approach is simple and contains no additional overhead, large traces may not fit in the working memory and lead to slow I/O operations and following reduce performance. Additionally, the trace files themselves need to be generated in some way for the simulation to be able to run. While execution and emulation modes simply need an application, trace-based modes require the program to be converted into a trace. This is usually done by other software tools – e.g., functional simulators. As with emulation, trace-based simulators are not reliant on any specific interface as the instructions of the traces are independent of the target architecture.

Level & Scope

In addition to type and mode, Brais et al. also considers simulators to be of a certain level and scope.

The level of a simulator is either classified as "application" or "full system", and represent if the simulator is limited to only simulating applications or whole operating systems. Among these, application-level simulators are easier to implement and often faster as the simulator only needs to consider operations done in a single application. On the other side is "full-system"-simulators that capture every operation – including those that are either directly or indirectly made because of another operation in the application. As such, these are often more accurate, but also slower and more complex.

Lastly, simulators may either be confined or complete. As discussed in Section 3.2.1, this is determined by what the simulator wants to be accomplish. Though many cache simulators are just meant to simulate the cache behavior, some simulators attempt to simulate whole computer system.

3.3 Inheritance & Subtype Polymorphism

When developing a simulator, it is often beneficial to use the Object-Oriented Programming (OOP) paradigm which represents abstract concepts as "objects". The "objects" in OOP are often abstract representations of real-world entities, and contain data and logic local to each object. Programs written in an object-oriented programming language often create and manipulate objects to solve some sort of problem.

An object-oriented programming language follows five basic design principles:

- It features **Objects** or/and **Classes** that may be initialized to create distinct entities which contains data, variables, and logic unique to every entity.
- It allows for **Information Hiding** – meaning that members of a class may be configured to restrict access of its data or logic. In practice, this is often defined by the keywords *public*, *protected*, and *private*.
- It includes **Interfaces** or **Prototypes** – which essentially is the ability for some code to inform other code of the *signature* of methods and functions. In practice, this means that code may use a function defined elsewhere without implementing the function itself as long as it knows its signature. The signature includes properties like output type, parameter types, and parameter count.
- It allows the objects to have **Inheritance** – meaning that objects can be created as "extensions" of other through the use of "sub-classes". A "subclass" contains all the data and logic fields of its "super-class", but may come with new additions or even override existing logic.

- It has **Subtype Polymorphism** – i.e., the ability to swap an object with an "sub-class" object. This means that the programming language knows that a "sub-class" object is an extension of a parent object, and that they thus share the same type.

Defining new classes as sub-classes allows developers to create a hierarchy of classes that expand on each other. Paired with polymorphism, it is possible to create wide definition in base classes that are narrowed down to specialized sub-classes through leveled inheritance.

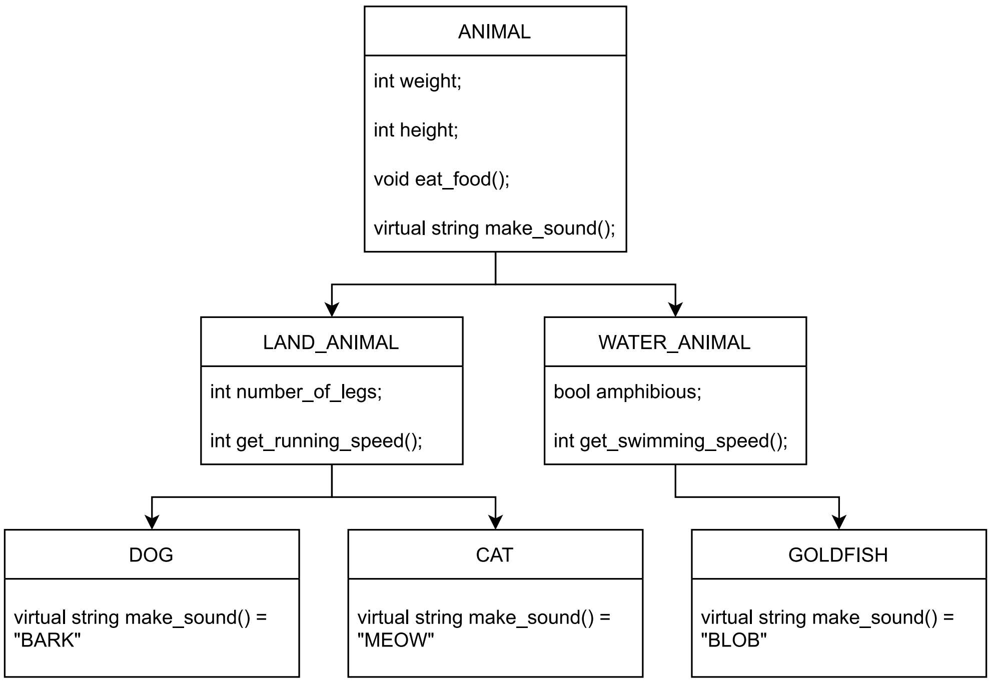

ANIMAL
int weight;
int height;
void eat_food();
virtual string make_sound();
LAND_ANIMAL
int number_of_legs;
int get_running_speed();
WATER_ANIMAL
bool amphibious;
int get_swimming_speed();
DOG
virtual string make_sound() = "BARK"
CAT
virtual string make_sound() = "MEOW"
GOLDFISH
virtual string make_sound() = "BLOB"

Figure 3.5: Hierarchical inheritance with 'Animal' as the common super-class.

Consider the example in Figure 3.5. All sub-classes derive from the same "Animal" class despite that the dog, cat, and goldfish classes are different from each other. However, all the animals have a weight, height, and a common method for eating food. The animals are all also able to make a sound, but each animal will say something different based on what type of animal they are. Note that all land animals are animals, and that since dogs are land animals they are implicitly animals as well. In other words, the dog class is an extension of the land animal class which in turn is an extension of the base animal class. This means that the "dog" class has all the attributes of land animals – like the number of legs – as well the data and logic of animals - weight and height.

As a programming language knows of every class – including super-classes and sub-classes – at compile-time, it is able to determine what classes derive from other. An example of a C++ header files for the "dog" class of Figure 3.5 can be seen in Listing 3.

```
#include <string>

class Animal
{
    int weight;
    ...

    virtual std::string make_sound();
    ...
}

class LandAnimal : public Animal
{
    ...
}

class Dog : public LandAnimal
{
    virtual std::string make_sound() override;
    ...
}
```

Listing 3: The "Dog" class and its parent classes

When analyzing the type of attributes and variables in the code, the programming language is able to perform sub-typing by accepting a subclass attribute in place of a super-class attribute. In the aforementioned example, the language knows that "Dog" objects may be substituted with "Animal" objects as all dogs are animals. This enables the behavior shown in Listing 4.

Note that the code in Listing 4 calls the *virtual* method `make_sound()` on the animal. This is *defined* in the Animal class, but *overridden* in each subclass. This implies that every animal makes some kind of sound, but this is up to each animal themselves to implement.

Lastly, it is worth considering that while the "dog" class can be linked to a real world entity – i.e., a physical dog – other classes may not. For example, the "land animal", "water animal", and "animal" classes are all ambiguous as there is no single animal that can represent all these. During programming, it may not make sense to initialize a "land animal" object as the class is just meant as an interface to extend upon. This makes "land animal" – and transitively also "animal" – an *abstract* class. For C++, an abstract class is any class that contain a "pure virtual function". Compilers will usually prevent developers from creating abstract objects, and fail when compiling code with pure virtual functions that are not overridden elsewhere.

```
#include <iostream>
#include <vector>

...

int main(int argc, char **argv)
{
    // All dogs are animals, but not the other way around:
    Animal dog_as_animal = Dog();                   // OK; dog is an animal
    // Dog animal_as_dog = Animal();                // Not OK

    std::vector<Animal> animals;                    // A list of animals
    animals.push_back(Dog());                       // OK to add dog
    animals.push_back(Cat());                       // OK to add cat

    for (auto & animal : animals)                   // For each animal:
    {
        std::cout << animal.make_sound() << "\n";   // - make animal sound
    }
}
```

Listing 4: Subtyping with Animal sub-classes - C++

3.4 Discrete-event simulation

Discrete-event simulation is a way of simulating a system where events manipulate the internal this As mentioned in Section 3.2.2, an effective way of simulating systems is by *discrete-event simulation*. This is an efficient way of simulating behavior for scenarios where certain events change the state of the system at certain moments in time, but no change happens between these events [20]. The events may be deterministic, but are often added dynamically – e.g., by other events. There are multiple way of performing discrete-event simulation, but the most common is by having the software "jump" from event to event. Once an event has been simulated, the simulator may jump to the next event as it is guaranteed that there will be no change in between.

The gem5 simulator uses this approach when simulating computer systems. The main logic of the simulator is performed by a single, infinite loop that continuously fetches and fires events. Each event may spawn another event at a later point in time, so the simulator does not necessarily end when after the initial events are executed. Instead, it fires and and processes the effects of each event as they happen, and only stops when it cannot fetch any new events.

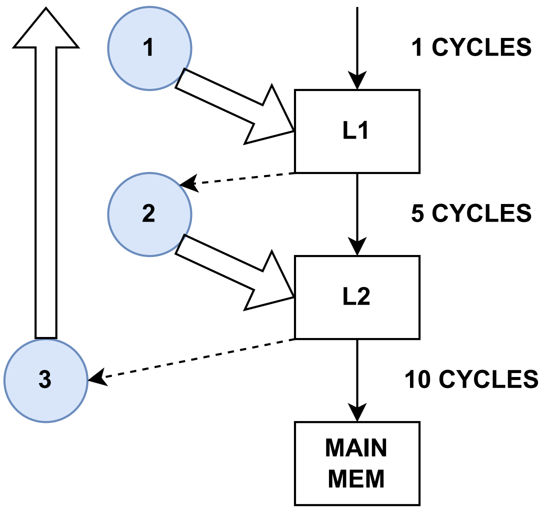


Figure 3.6: A cache access event to L1 leading to two other events

An example of a very simple cache simulation with a single access to can be seen in Figure 3.6. Assume that the simulator has just started, and that the current cycle/time counter is set to 0. Also assume that the requested data is available in the L2 cache, but not the L1 cache. This may lead to the following:

0. Event 1 is set to access the L1 cache. As the time to access the L1 cache is 1 cycle, the simulator schedules the access for the next cycle.
1. At cycle 1, event 1 fires – accessing the L1 cache. Given the assumption that the data is only available in the L2 cache, this will lead to a miss. Thus, event 1 will create a new event – event 2 – and schedule this five cycles into the future at cycle 6.
2. At cycle 6, event 2 fires the memory request event which results in a hit. This should trigger the creation of another new event. Depending on the implementation, this may be a message of acknowledgment sent to either the L1 or the source. For simplicity, assume that the message is sent directly to the source. This takes a total of 6 cycles, so event 3 is scheduled for cycle 12.
3. Lastly, event 3 fires at cycle 12 – telling the system that the message has been received. This is the last event in the simulation, as this doesn't spawn any new events. Thus, the simulation ends at cycle 12.

Chapter 4

Motivation

4.1 Goal

As mentioned in Chapter 1, the use of a cache simulator is essential to efficiently research new and advanced accelerators, configurations, and replacement policies. Open-source simulators may be extremely useful in testing new ideas, as they allow researchers to focus on getting results quickly instead of worrying about the underlying logic of a simulator. Together, many different open-source simulators cover a large amount of features and platforms that can be experimented on. Additionally, some of the larger, long maintained simulators – like gem5 [4] – has a wide array of features and configurations. In the end, however, these simulators are only meant as assisting tools, and not every problem is automatically compatible with any simulator.

Though many powerful CPU cache simulators exist, there are relatively few simulators focusing on GPU and multi-core behavior. The aforementioned simulators capable of this behavior – i.e., gem5 and its gem5-gpu fork, GPGPU-Sim, and Sniper for multi-core simulation – are exceptions rather than the rule. Additionally, these come with various assumptions – like GPGPU-Sim requiring the application to be built on the CUDA framework. The lack of GPU simulator is not directly surprising however as memory latency is a dominant bottleneck for CPUs rather than GPUs because of the aforementioned "Memory Wall" [2], and the demand to reduce CPU latency has been (and still is) an area of focus. Additionally, GPUs often value throughput over memory latency, as the number of operations – i.e., throughput – is viewed as more important than the latency of each individual operation. Still, with GPU systems being increasingly more advanced and utilized more for non-graphical workloads, GPU caches have become more important. This does not mean that cache behavior in GPU systems is unexplored, but rather that there exist a myriad of different implementations and ideas with have non-trivial solutions that can be tested. One such idea may be the effect of an inclusive cache policy in multi-core cache hierarchies, and how this is affected by different graphic workloads.

As mentioned in Section 3.1.3, inclusion policies may increase cache hit rate for certain implementations. In particular, an inclusive cache policy generally performs well for multi-core CPUs. Recall that every entry of a L1 also exist in the L2 when following an inclusive policy. This works well for multiple cores, as the threads of a program often reference the same addresses in memory. If one thread installs a data block in its local L1 cache, it will also forward the data to the L2 (and L3 and so on). This makes it possible for individual threads to hit in the L2 (or L3) cache when requesting addresses that another thread has loaded. This behavior should also carry over to GPUs, as the cache hierarchy structure stays the same despite the change in workload.

The last few paragraphs create the basis for the *goal* of the project this thesis is based upon:

Explore the effects of memory hierarchy consisting of coherent inclusive caches on programs divided between multiple cores.

As mentioned above, the motivation for this is that data shared across a program should lead to hit rates in the lower level caches. Assuming that multiple threads use the same data, only one miss is needed by one of the threads before it is available in the L2 or L3 cache.

For example, consider the matrix multiplication function of Listing 5. Note that except for a small sequential part at the start of the function, the matrix multiplication are parallelized for each row. Though most of the variables refer to a unique address, the variable `b` uses the same access pattern across all threads. In other words, after one thread requests a variant of this – say, `matB[0,0]` – it should be available in the shared L2 or/and L3 cache due to the inclusive cache policy.

Another example can be seen in Figure 4.1. Each colored circle represent a data block that is derived from a unique address. Note that each L1 caches some data that is shared between both L1s and some data that is unique to each cache. However, as the hierarchy consists of inclusive caches the textitall of the data is present in the L2. This is beneficial as it is quite likely that one L1 receives a request for data that is present in the other. For example, a request for the red block in the left L1 will initially miss, but hit when propagating to the L2 cach.

4.2 Requirements

4.2.1 Simulator Requirements

To test the previously described behavior, two things are needed: a data set with GPU-compatible memory operations, and a simulator that mimics the behavior of a GPU cache hierarchy while accepting the format of the data sets. Though feasible data sets may be hard to find, a large batch of generated memory traces from GPU emulation was generously provided by Ole Henrik Jahren of ARM Trondheim.

This section will focus on the 11 requirements presented in Table 4.1, and discuss the reasons for these in the following subsections.

```
using ...

// Example borrowed from:
// https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/
// how-to-write-a-simple-parallel-for-loop#matrix-and-stopwatch-example
// Code is slightly altered to fit better
public class MatrixSharedDataExample
{
    static void MultiplyMatrices(double[,] matA, double[,] matB, double[,] result)
    {
        // Get rows, cols from matrices
        int matACols = matA.GetLength(1);
        int matBCols = matB.GetLength(1);
        int matARows = matA.GetLength(0);

        // Parallelize the outer loop:
        Parallel.For(0, matARows, i =>
        {
            // Each thread will have the same amount of iterations
            for (int j = 0; j < matBCols; j++)
            {
                double temp = 0;
                for (int k = 0; k < matACols; k++)
                {
                    // a is unique for all thread because of i...
                    double a = matA[i, k];
                    // But b is shared between all threads!
                    double b = matB[k, j];

                    temp += a * B;
                }
                result[i, j] = temp;
            }
        });
    }
}
```

Listing 5: Example of a matrix multiplication in C# where data is shared among threads.

The requirements of Table 4.1 are divided into three groups for the sake of readability. The requisites are grouped as follows:

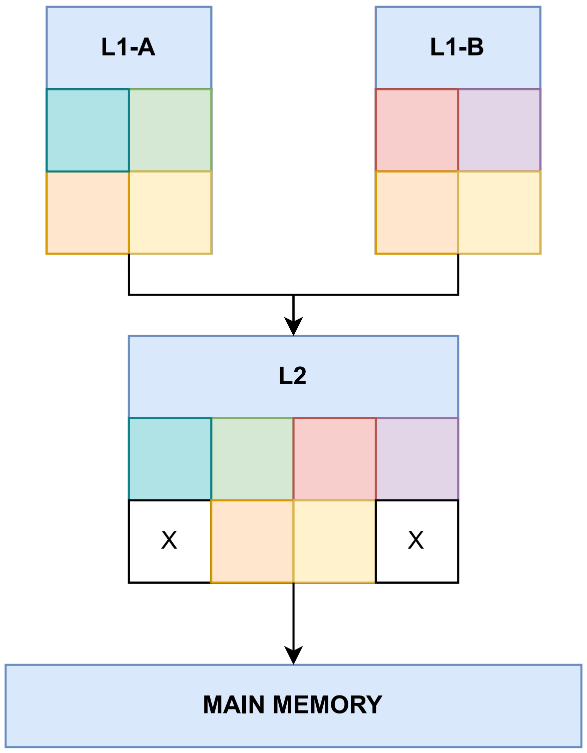


Figure 4.1: A cache hierarchy sharing some of the same data across cores.

- **Group A** indicates the provisions of the simulator in terms of its minimum base features – i.e., what the simulator must be capable of to be able to simulate the problem in the first place. Though a simulator may feature an advanced coherency or inclusion policy protocol, it cannot be used if it lacks an interface for memory traces per core.
- **Group B** consists of the requirements of features in the simulated system itself. These include what components are possible to implement in the simulator, and how advanced the configuration of these needs to be. For example, the system must not only support caches that replace entries on cache misses, but rather a configurable cache replacement policy. As with group A, it is not enough for the simulator to accept the memory trace format if it cannot test the desired systems due to, e.g., non-configurable caches.
- **Group C** is closer to strong recommendations than absolute requirements, but important nonetheless. For practical reasons, the simulator should strive to achieve a high performance as to limit simulation times for larger programs. Traces consisting of millions or more of operations should also be handled in a way to limit their resource footprint.

The requirements of category A and B as well as their implications are further discussed in the Section 4.2.2 and Section 4.2.3.

#	Requirement
A1	The simulator must emulate cache behavior in a GPU environment
A2	The simulator must be able to simulate distinct caches and cache hierarchies
A3	The simulator must accept a memory trace **per core** – i.e., each individual L1 must be able to read from its own trace
A4	The simulator must accept memory traces in a custom binary Google Protocol Buffer format
B1	The simulator must be scalable and configurable
B2	The simulator must support highly configurable caches in terms of size, associativity, cache line size, etc
B3	The simulator must allow for an *inclusive* cache policy
B4	The simulator must allow for a cache being able to handle a configurable number of requests *without blocking*
B5	The simulator must be able to handle dependencies between memory operations
B6	The caches in the simulated cache hierarchy must have a configurable cache coherency protocol, cache placement policy, and cache replacement policy
C1	The simulator should use an appropriate amount of resources (especially memory usage) under simulation
C2	The simulator should attempt to scale its performance as optimally as possible – i.e., the time spent simulating a program should scale as linearly to its size as possible

Table 4.1: Requirements

4.2.2 Requirements – Group A (Simulator Features)

The requirements of group A are based on the following reflections:

- **A1** – The main goal of the simulator is to simulator the behavior of a GPU system with caches, so the system cannot be designed exclusively for CPU behavior. Some simulators may be built on the assumption that the caches receive requests from the CPU, and these cannot be used here. In other words, a CPU cache simulator should not be used to solve a GPU cache simulation problem.
- **A2** – The simulator must treat every cache distinctly and make it so that the caches can be connected through a hierarchy. The simulation behavior cannot be limited to one cache, and the caches need to communicate seamlessly with each other e.g. when forwarding a cache miss or when eviction of a dirty block leads to a write-back. The simulator must also keep track of an user-defined number of caches and how these are connected while leaving the configuration of the system to the user.

- **A3** – Each individual L1 cache must be able to access a separate input of memory operations assigned exclusively for that L1 cache. Instead of a single memory trace file, there are multiple smaller files consisting of operations where each file is meant for one (and only one L1 cache). This is needed as each core creates their own separate set of instructions independently from the other cores, and only the L1 assigned to each core receives this trace.
- **A4** The memory trace files of requirement A3 is provided in a custom-made binary format based on Google's Protocol Buffer. The simulator needs to provide an interface for reading the files this format. The details surrounding the trace files will be discussed later in Section 5.3.3.

4.2.3 Requirements – Group B (System Features)

- **B1** – The simulator must be scalable and easily configurable so that different configurations and options may be tested with relative ease. For example, a user should easily be able to manage the design of a cache hierarchy as well of the number of caches, or what caches read from which files. Additionally, the simulator should not (as far as it is possible) put any limits on the size of the memory, number of caches, how caches are connected in a memory hierarchy, etc.
- **B2** – The caches of the simulator need to have several configurable options – like size and associativity – so that different configurations can be easily tested. Thus, the caches should have as few hard-coded assumptions as possible.
- **B3** – A large part of the main goal of the simulator is to explore the behavior of inclusive caches of a GPU system. To achieve this, the simulator must know the rules of the inclusive cache policy. The details of inclusive cache policies are discussed in Section 3.1.3.
- **B4** – In order to generate realistic results, the caches of the simulated system should be non-blocking – meaning that the cache should be able to handle new incoming requests continuously without stalling. This should also be configurable – i.e., it should be possible to indicate how many operations a cache can process while still continuing without blocking. Non-blocking caches are discussed further in Section 3.1.4.
- **B5** – The operations of the memory trace might be dependent on another operation before it being completed before being allowed to execute. Thus, the simulator should be able to recognize and buffer certain operations so that these are only executed once all dependencies are completed.

- **B6** – In addition to size, associativity, etc., the caches of a system need to have configurable coherency protocol as well as placement and replacement policies. All these factors may have a significant impact on performance, and different combinations or variants of these should be possible to test. Perhaps the most important of these is the cache replacement policy, so it should be possible for a user to define a new replacement policy and compare its results to other policies.

4.2.4 Discussion

Note that the requirements of Table 4.1 varies in how specific they are. The requirements of group B are quite general, as the majority of these (with the possible exception of B3) might persist for *any* simulation – not only this experiment. However, the support for an inclusive cache policy is naturally required as it sets the basis of the project, but this is not necessarily a rare feature for a simulator to have. Not all open-source simulators may meet all requirements of group B, but most have at least a way of configuring cache size and manipulating the cache replacement policy.

The requirements of group A are however more specific to this project in particular. Recall that the group A is meant to describe the requirements of the simulator itself, and that the goal of this project is to explore the behavior of inclusive caches in coherent multi-core cache hierarchies. Also recall that the project itself is made possible by custom memory traces provided by ARM, but that these also requires a specific interface to be interpreted correctly. This leads to a list of requirements for the simulator in general rather than the underlying system, and results in the items of group A. To define these, the goal of the project as well as the trace format can be used as a starting point. The first three requirements may be derived directly from the project goal itself: The simulator needs to emulate GPU behavior, simulate a hierarchy of distinct caches, and allow each individual L1 to read from a separate file representing a thread. Likewise, the last requirement may be described using the trace format itself: The simulator must know how to translate the information from the trace into internal representations that it can use to simulate a system. All in all, these requirements are much less generic than the ones of group B, and are less likely to be supported in any open-source simulator.

Though multiple open-source simulators exist freely on the web, the largest and most maintained of these is arguably gem5 [4]. The gem5 simulator is discussed further in Section 2.2, but is in short a full-system simulator capable of emulating cache behavior. As mentioned earlier, there is no guarantee that gem5 is compatible with every single problem or project, but it still has a large array of features and configurable options.

#	Short requirement	Supported?
A1	Multi-core GPU environment	N/A*
A2	Multiple caches and hierarchies	✓
A3	Memory trace per core	✗
A4	Accept memory traces in a custom format	✗
B1	Scalable + configurable simulator	✓
B2	Configurable caches	✓
B3	Inclusive cache policy	✓
B4	Non-blocking caches	✓
B5	Request dependencies	✗
B6	Cache coherency, cache placement policy, cache replacement policy	✓
C1	Acceptable resource usage	✓
C2	Scalable for larger programs	✓

Table 4.2: Desired features supported by gem5.

Consider the checklist of Table 4.2. Note that while gem5 is highly configurable and covers a lot of the requirements, it still fails on at least two of the items of group A and one item in group B. Though gem5 supports advanced features like non-blocking cache, inclusive caches, and coherency protocols, it is not enough to simulate the desired problem. The simulator fails to meet the following requirements:

- **A3**: The gem5 simulator may simulate programs in two ways: either by emulating a given application, or by providing a `TraceCPU`-component with a custom generated trace. However, the latter is for practical reasons only meant for replaying simulations after generating the trace from emulation. Nevertheless, gem5 does not support manual routing of memory traces to exclusively L1s/cores.
- **A4**: As the provided traces come in a custom binary format, it is not surprising that gem5 does not know how to translate the input to an appropriate representation. Additionally, the component responsible for reading traces – `TraceCPU` is: 1) a CPU, 2) only capable of replaying traces in a specific format, and 3) made for reading instruction traces rather than memory traces. This makes `TraceCPU` unfit for accepting the custom trace files of this project.
- **B5**: One special requirement for this project was to put dependencies between the completion and initialization of certain memory operations. As of May 2022, there is no easy way to force a such dependency in a trace format. Though dependencies probably are resolved automatically when gem5 emulates an application, there does not seem to be support for this when simulating from traces. However, this is not really surprising given that gem5 is meant for simulation through emulation and not through memory traces.

Additionally, there are a few other points that should be addresses: A1 – support for GPUs – is technically achieved, but gem5 only simulates the architecture of either a Nvidia system (through gem5-gpu) or an AMD system (through the bundled GCN3). For the purposes of this project, this requirement is not adequately fulfilled. Secondly, B3 is fulfilled through gem5's advanced cache coherency interface. Though gem5 is bundled with several pre-made coherency protocols, the protocols with inclusion would probably have to be altered to fit the project. Nevertheless, this should be possible, so the requirement is marked as fulfilled.

Lastly, it might be interesting to note that the `TraceCPU` actually uses Google Protocol Buffer to create custom traces when generating and replaying traces. However, Google Protocol Buffer is only a framework for creating binary-formatted input files. The base of these custom traces may be similar, but their contents are not.

4.2.5 Conclusion

As discussed, the specificity of the goal of this project makes it hard to simulate the desired behavior out-of-the-box using an open-source simulator. While gem5 is the only simulator discussed and compared against the requirements, it is highly likely the simulator with the most features of all cache simulator available. Though other simulators – like Sniper [5] or GPGPUSim [12] – may have unique features that gem5 lack, the simulation is still hindered by several very specific requirements. This is especially true for A3 and A4, as these require a custom-made interface to be fulfilled. In the end, this results in a limited number of potential options:

1. **Test the implementation on real hardware**: This is not a practical option and should not be considered. As motivated earlier, this makes it practically impossible to tweak parameters and test different systems. While the use of FPGAs might be slightly more feasible, this option brings a massive technical overhead for a project that is limited to an exploration of cache behavior.
2. **Modify existing simulators to meet the requirements**: This option is much more realistic. Though gem5, Sniper, or GPGPU-Sim (to name a few) do not have any compatible interface for, e.g., the custom trace format, this might be still possible to develop. In fact, one argument for this option is that code is made open-source partly for this very reason – enabling developers to add new features over time. However, the complexity of this may vary greatly. In the case of gem5, this might change a lot of the underlying logic of the simulator. Recall that gem5 runs applications through emulation, and that the requests are generated by a CPU. All this would need to be modified or expanded upon to first allow gem5 to read from *any* memory trace, and then further changes would be needed to allow it to read from this memory trace in particular. This might be easier for other simulators accepting

memory traces in the first place, but would still be significantly complex for the A3 requirement. Furthermore, these types of simulators might miss functionality gem5 has. Though this option may be theoretically possible, it is still a great endeavor.

3. **Develop a specialized simulator from scratch**: This may sound like an even greater undertaking than the option above, but does not necessarily have to be. Any new simulator would have to meet all the requirements of Table 4.1, but does not need additional advanced features like interconnects or CPU logic that gem5 has. This will effectively allow for tailoring the software to the exact needs of the project and ensure that all needed configurations and features are present. However, this approach should still take a lot of development time before the actual problem of the project could be tested. While any advanced non-essential features could be safely ignored, basic functionality like caches, memory, request representation, and I/O-interfaces would all need to be developed from scratch. There is also a high chance of encountering bugs or errors along the way that existing simulators already have identified and fixed. Ultimately, though, this is the only realistic option for ensuring that all the requirements are met. Recall that group A of Table 4.1 is reserved for requirements of the simulator itself, and the only way to guarantee that these are met is to have full control of the simulator.

While developing a simulator is hard and a time-consuming process, it seems to be the most feasible of the options above. The rest of the thesis introduces COCOASIM: a Concurrent Cache Operating Access Simulator capable of simulating multi-core systems using an inclusive cache policy. The simulator is custom-made specifically to solve the problems described in this chapter, but is able to simulate highly configurable systems.

Chapter 5

Simulator Overview

The decision to develop an entirely new cache simulator from scratch eventually culminated in COCOASIM – a COncurrent Cache Operating Access SIMulator. The simulator itself is written almost entirely in pure C++, but also includes Makefile code to simplify building the COCOASIM binaries, and also uses Python to continuously integrate and test new features as they were added in the development process. Building the source of COCOASIM using the Makefile results in three binary files: `cocoasim`, `cocoasim-fast`, and `cocoasim-fast`. All of these simulate a user-defined system in the same way, but varies in terms of performance and debugging support.

In summary, COCOASIM comes with multiple practical features. While the simulator was created to solve a very specific problem, many of the features are problem-agnostic and can be useful for any simulation. Some of the things COCOASIM is capable of are listed below:

- Simulate in three different "modes" – each with varying performance and debugging support.
- Direct individual traces to individual cores/L1s –allowing for fine-grained input configurations.
- Read from highly compact binary traces through streams. Additionally, the streams are only read from when needed – thus minimizing memory usage.
- Use highly configurable user-defined systems: either through creating the system in C++ or by providing the simulator with a configuration file in JSON.
- Perform logging at a user-defined level – ranging from nothing to extensive logging.
- Create dumps of each simulation with logs and statistics. The simulator assigns either a unique ID or a user-defined name to the dump – making it easy to organize the results of multiple simulations.

Using the simulator classes discussed in Section 3.2.2, COCOASIM can be defined as the following:

- **Type: Timing** - The simulator keeps track of time in cycles. Events in the simulator are first initialized, and then scheduled for whenever the operation finishes. COCOASIM keeps track of what events finishes when and what how many cycles have passed since the simulation started. The simulation ends only when all events are finished.
- **Mode: Memory Trace** - As mentioned earlier, COCOASIM reads from traces consisting of memory instruction at runtime to simulate a system. These traces tell COCOASIM things like the operation executed, the address accessed, any dependencies on other requests, and more.
- **Level: N/A** - Since COCOASIM only simulates the provided operations, it does not need to consider the level like simulator that rely on execution or emulation do. This can be viewed as a double-edged sword; The simulated system does not to run other applications, but in turn require generated memory traces. These traces are typically generated from other functional simulators.
- **Scope: Cache simulator** - COCOASIM is limited to being only a cache simulator as the main motivation for its development is to explore cache behavior.

While COCOASIM is designed specifically to meet the requirements of Table 4.1, the simulator is also designed to be scalable and have a generic and configurable interface. This is done to ensure that new features can be added without making major changes, and make it possible to test different variants of systems and components. For example, COCOASIM allows for *any* number of cores in a simulated system. This makes it just as easy to test a traditional single-core system as a GPU with 20 streaming multiprocessors or improbable designs with thousands of L1 caches. Furthermore, every component in a simulated system is represented by an object, and may be modified or replaced as needed. New objects can also be added with relative ease as long as they inhibit an interface that COCOASIM recognizes. For example, new replacement policies can be created by a user as long as the simulator know what to do on cache hits and misses.

While the interface and internal logic of COCOASIM greatly differs from that of other cache simulators, the simulator is inspired by some of gem5's design decisions. Most notably, COCOASIM simulates the passing of time in the same way as gem5 – i.e., through discrete event simulation. Though the implementation differs between the two simulators, both build on the same concept of jumping between events at set timestamps. Discrete event simulation is discussed further in Section 3.4.

A quick comparison between gem5, Sniper, and COCOASIM is presented in Table 5.1. All simulators have a notion of time, but use different input when simulating. Note that gem5 and Sniper provide a larger scope of features, while the main focus of COCOASIM is to simulate a specific type of problem.

Simulator	Level	Type	Mode	Scope
gem5	Timing	Application / Full System	Emulation / Instruction Trace	Full system simulation
Sniper	Timing	Application	Execution / Instruction Trace	Configurable
COCOASIM	Timing	N/A*	Memory Trace	Cache simulator

Table 5.1: Comparison of gem5, Sniper, and COCOASIM.

5.1 Software requirements

5.1.1 Simulation

In theory, simply executing one of the COCOASIM binaries should be possible without the need of any other library or external program. The simulator only needs to know what program it should run in what type of system. To achieve this, COCOASIM needs:

1. a collection of traces in the custom Google Protocol Buffer Format.
2. a representation of the system to simulate.

For technical reasons, it is highly recommended that the simulator binary is run from within the COCOASIM repository. There are two reasons for this:

1. The simulator will attempt to write to and move files during execution. A dump containing statistics and logs will be dumped in the same location as the binary.
2. The simulator needs to use another software module usually bundled with the COCOASIM repository called `pbtrace`. The code of COCOASIM will attempt load a header from this project during compilation. Though the external repository can be extracted along the binary and still work, it is recommended to leave things as they are.

5.1.2 Developing

While the code of COCOASIM tries to have as few dependencies as possible, a developer working with the source code of the simulator needs to have some software installed. These prerequisites are shown in Table 5.2.

5.2 Running COCOASIM

Simulations in COCOASIM can be run by executing a binary along with a list of parameters from the command line. The format of a hypothetical configuration using every available (and compatible) option is shown in Listing 6.

Requirement	Needed to	Used for
C+11 (or newer)	Compile	Code, especially for simplifying `for`-loops and creating objects through the use of the `auto` keyword
Boost (C++ library)	Compile	Various code, logging, JSON parsing, algorithms, etc.
GNU Make	Compile	Run Makefile, build project
Python	Test	Run tests

Table 5.2: Requirements for compiling and testing COCOASIM

```
./<binary>
>   -L
>   -l LOG_SETTING
>   -i INPUT_DIR
>   [-c CONFIG_FILE] OR [-s SYSTEM_NAME]
>   -o OUTPUT_DIR
>   -F
```

Listing 6: How to run COCOASIM – featuring the available options.

An overview of the possible arguments that COCOASIM can use can be seen in Table 5.3. Note that while the simulator only needs a single argument telling it what system to use, more options can be used for increased configurability. Note that while only one argument is needed, the simulator needs to actually have access to the collection of traces and the configuration file. In other words, if COCOASIM is run using `./cocoasim -C` it assumes that the directory `traces` – as well of as its files – and `configs/test_system.json` actually exist.

The flags of Table 5.3 have the following effects:

- **-L / –loud**: Causing the simulator to be "verbose" when logging. All log messages are written to standard output in addition to the log file.
- **-l / –log-level**: Takes an integer argument in the range 0-3 which determines the amount of logging. The numbers represent the following levels:
  0. **NONE**: Might be slightly misleading as COCOASIM logs only the absolute essential – like if the simulation completed with or without errors. In other words, *almost* nothing is logged.
  1. **BASIC**: Logs things like setup logic and confirmations of the chosen configuration, but nothing else. This is the default option.
  2. **INTERNALS:** Logs a lot of internal logic – like memcopy requests accessing caches, or eviction of tags in cache sets – but not details.

Flag	Long option	Argument	R?	Default	C
-L	–loud	-	✗	False	L
-l	–log-level	int [0-3]	✗	1	L
-i	–input-dir	string/path	✗	"traces"	I
-c	–config	string/path	*	-	S
-C	–config-default	-	*	"configs/test_system.json"	S
-s	–system	string	*	-	S
-S	–system-default	-	*	"TestSystem"	S
-o	–output-dir	string	✗	PID of COCOASIM	O
-F	–full-simulation	-	✗	False	D

R?: Required?, **C**: Category, **L**: Logging, **I**: Input. **S**: System, **O**: Output, **D**: Debug

One – and only one – of the options marked with * must be specified

Table 5.3: The arguments that can be parsed by COCOASIM.

3. **ALL:** Logs virtually every event and operation happening during execution. Dumps a lot of data of components as they change state. Used primarily in debugging as much of the information at this level is just the simulator checking that everything works correctly.

- **-c / –config**: Tells the simulator to load the system configuration file at a given path. The file must be in the custom JSON format shown in Figure 5.7. The details regarding configuration files are presented in Section 5.3.4.
- **-C / –config-default**: The same as `-c configs/test_system.json`. Separated from *-c* to avoid optional arguments.
- **-s / –system**: Tells the simulator to use a system defined by a C++ class. Assumes that the class exists and is added to a custom catalog interface that the simulator has access to. This is discussed further in Section 5.3.4.
- **-S / –system-default**: The same as `-s TestSystem`. Separated from *-s* to avoid optional arguments.
- **-o / –output-dir**: Specifies the name of the directory which COCOASIM logs messages and statistics to. If the `-o` flag is absent, the simulator will use the current Process ID (PID) as the directory name.
- **-F / –full-simulation**: Toggles full data simulation. Since hit and miss rates are fully independent of the data in the cache, the simulator only considers the actual addresses when handling memory accesses. However, enabling full data simulation makes COCOASIM also keep track of the data at any given address. This is primarily meant as a secondary feature to be used to ensure that the correct data is being read and written as it is resource-demanding for large traces.

A couple of examples on different argument configurations are shown in Listing 7.

```
# 1
# Run simulator with already existing files
# Uses "traces" as default input directory
# and existing C++ class TestSystem as system
./cocoasim -S

# 2
# Run simulator with custom trace directory
# and custom configuration file
# Use fastest version
./cocoasim-opt -i memcpy -c configs/my-8-core-config.json

# 3
# Run simulator with maximum level of logging
# and log verbosely to terminal output
# Output statistics and results to new directory
# named "results" in addition to terminal output
./cocoasim -L -l 3 -C -o results

# 4
# After adding a new system class in C++,
# run simulator with test traces
./cocoasim-fast -s MyNewCPPSystem -i test_traces

# 5
# Run multiple simulations (in sequence), but map
# output to different dumps - allowing for easy
# comparison between the two of them
./cocoasim-opt -s MyNewCPPSystem -o sim1 &&
./cocoasim-opt -c configs/my_config_system.json -o sim2
```

Listing 7: Examples of configurations for running COCOASIM

5.3 Configurations & Arguments

While COCOASIM has several features that are not visible to the user, much of the underlying logic of the simulator can be linked directly to the arguments given to the simulator during run-time. This section will use the options of Table 5.3 to discuss how these are supported through the design of COCOASIM.

This section discusses the six available arguments and explains what implication they have on a simulation:

1. **Binary Variants, Section 5.3.1** - How the binary choice affect debugging and performance, and why.

2. **Logging, Section 5.3.2** - How logging works in COCOASIM.
3. **Inputs & Traces, Section 5.3.3** - The traces used by COCOASIM, their format, and why these are used.
4. **Systems, Section 5.3.4** - How to specify, modify, and ultimately run a system.
5. **Output, Section 5.3.5** - What output a simulation in COCOASIM generates.
6. **Full Data Simulation, Section 5.3.6** - The full data simulation option.

5.3.1 Binary variants

The simulator can be run using one of the three binaries – i.e., either `cocoasim`, `cocoasim-fast`, or `cocoasim-opt`. The binaries have the same interface and uses the same input parameters, but `cocoasim-fast` and `cocoasim-opt` will ignore any logging options. In short, `cocoasim-fast` runs significantly faster than `cocoasim`, and `cococasim-opt` is slightly faster than `cocasim-fast`. Stable versions of COCOASIM should produce the same results regardless of the binary used, but the `cocoasim` binary is recommended for testing new features as this provides the best debugging support.

Version	Performance	OF	S?	AT?	L?
cocoasim	Good	-O0	✓	✓	✓
cocoasim-fast	Better	-O3	✓	✓	✗
cocoasim-opt	Best	-O3	✓	✗	✗

OF: Optimization Flag, **S?**: Statistics?, **AT?**: Assertion Testing, **L?**: Logging?

Table 5.4: Comparison of the three variants of COCOASIM

A comparison of the three binary variants is shown in Table 5.4. As seen, while `cocoasim-opt` is the fastest of the three it also has the least features. The features of the table are briefly explained in the following list:

- **OF / Optimization Flag** - The optimization flag option provided to `g++` when compiling the COCOASIM source. Ranges from `O0` (zero optimizations) to `O3` (heavy optimizations).
- **S? / Statistics?** - If the simulation produces results/statistics. As shown, all version do this – meaning all can be used to gather results from a simulation.
- **AT? / Assertion Testing?** - The simulator contain a high number of *assertions* for certain statements that should be true. These are scattered all over the source code, and are meant to check that the simulator behaves as expected. When an assertion test fails, the simulator exits with an error code. For example, an assertion in COCOASIM is that all cache lines contain data and not waiting for any load from e.g. the main memory. These tests do not impact the actual simulation in any way, but use some computational time to verify that the simulator is working correctly.

- **L? / Logging?** - The unoptimized version of COCOASIM logs a lot of messages describing the internal state of the system. The level of logging can be configured in `cocoasim`, but is ignored in the other versions. The reason for this is that the I/O operations of writing the logs take a lot of time, and is thus only used in practice when debugging.

Lastly, a small technical detail about COCOASIM is that the Makefile can either build all binaries at once using `make -j ‘nproc‘`, or a specific version through `make <binary-name> -j ‘nproc‘`.

5.3.2 Logging

Logging in COCOASIM is done through a custom C++ function that logs a given message if certain conditions are met. The logging function itself does not impact the simulated system in any way, and – as shown in Table 5.4 – is completely removed by the compiler when building either `cocoasim-fast` or `cocoasim-opt`. When running the `cocoasim` binary however, the simulator will consider certain conditions to decide if it should log a message. Furthermore, if COCOASIM chooses to log a message, it must also consider *where* to log. The simulator will always log to a log file, but also prints the message to standard output if the `-loud` flag is set.

It should also be noted that COCOASIM may log to an additional file depending on the log condition. This is done to have a split the main log file into certain categories depending on what was logged. The goal of this is to make it possible for a user that is only interested in logs for e.g. the event logic of the simulator to open a filtered log file with only events.

```
// Logging function, slightly altered for readability
void log(
    std::string& string,                // 1
    LogChannelName& channel,            // 2
    LogConfiguration required_level,    // 3
    std::string& source,                // 4
    LogSeverity log_severity            // 5
    );
```

Listing 8: The prototype of the logging function in COCOASIM

While this section will attempt to refrain from going into the technical details of the logging function, the header prototype of it is shown in Listing 8. As shown, there are a total of five arguments:

1. **string** - The string/message to be logged. Can be any string.

2. **channel** - The *channel* to log to. The `LogChannelName` type is only an alias of the `string` type, but is encapsulated as it needs to be equal to a predefined string. Some options for this include *default*, *setup*, and *events*. This makes it possible to write the log message to certain fragments depending on the message being logged.
3. **required_level** - What level must be specified by the `-log_level` flag for this message to be logged. This is the main condition of the log function, and uses an enum in the following range: *NONE*, *BASIC*, *INTERNALS*, *ALL*. The simulator only logs the message if the log level is at least the enum in the function. In other words, a log level of 2 will allow all messages marked with *NONE*, *BASIC* or *INTERNALS* to be logged, but not the ones marked with *ALL*.
4. **source** - The string that should be listed as the source of this log message. For example, if a log message originates in a cache, the source should be something akin to "Cache". This is done to make it easier to filter the logs when debugging later.
5. **log_severity** - An enum used to indicate if something is a standard message (INFO) or something that probably shouldn't happen (WARNING). For the vast majority of logs, this is set to INFO.

An experienced programmer might wonder why COCOASIM uses string aliases in place of enums for the channel argument. The answer to this is to ensure *scalability*. The logger itself is a class that is designed to be expanded by a subclass if a user finds it necessary. Simply explained, new string-based channels can be added without problems, but there is no easy way to do this with enum-based channels.

```
// Code is slightly edited for readability
std::string log_entry = (boost::format
    ("@%lu: <%s> %s%s\n")    // Format
    % engine->get_cycle()    // 1
    % source                 // 2
    % severity_tag           // 3
    % string                 // 4
).str();
```

Listing 9: How a log message is constructed inside the log function.

The format of the log messages is shown in Listing 9. The messages are always formatted in this way – i.e., `@cycle: <source> <optional_tag><message>` – but the vast majority of messages leave the `<optional_tag>` blank. The cycle is determined by the simulator, while the source, severity tag, and message string are fetched from the arguments. A simple representation of the log-function's logic flow is visualized in Figure 5.1.

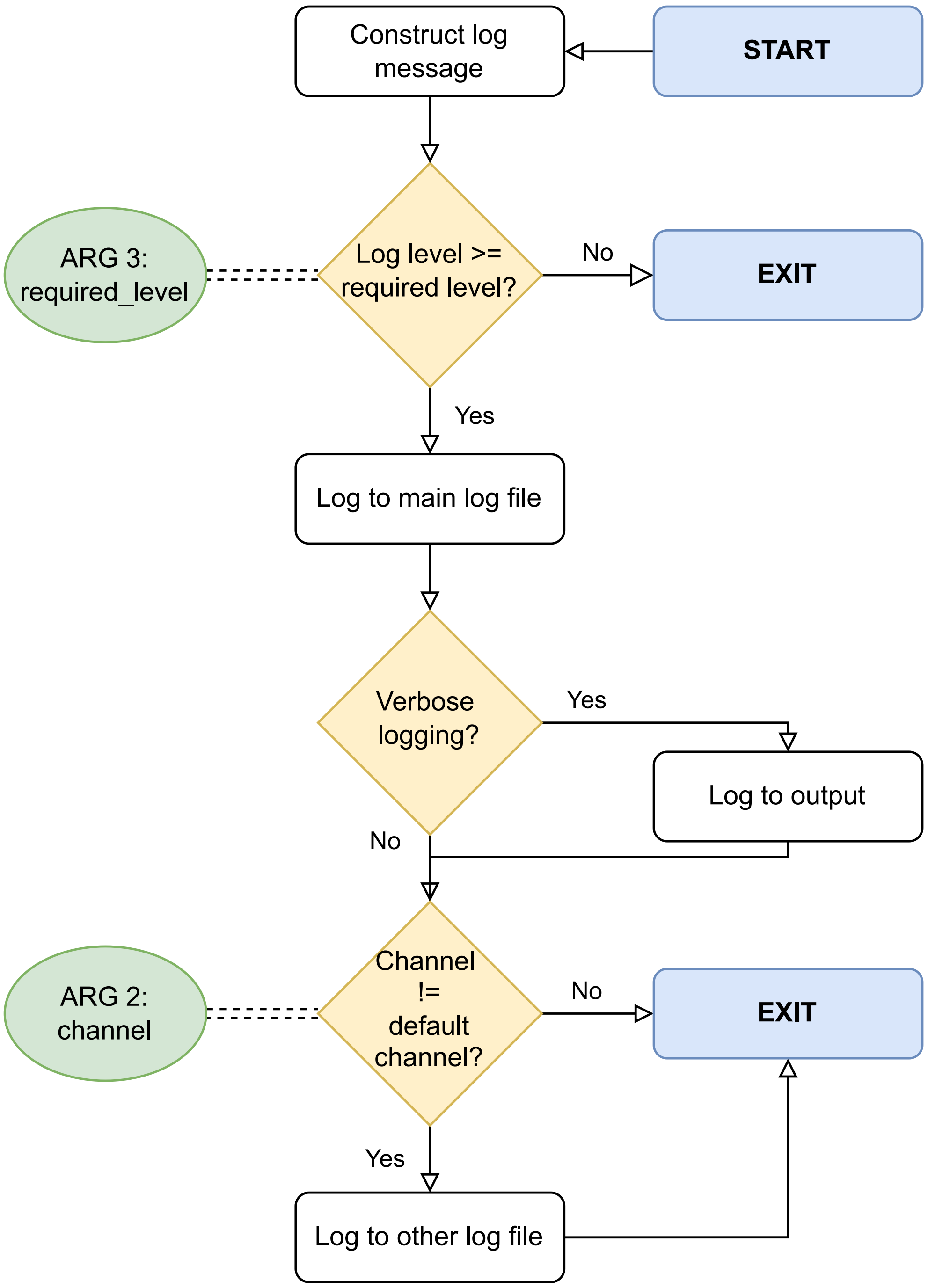


Figure 5.1: Logic flow of COCOASIM's log function

5.3.3 Inputs & Traces

Google Protocol Buffer

As mentioned earlier, the simulator uses custom Google Protocol Buffer [21] files to represent a collection of operations that access the simulated system in order.In short, Google Protocol Buffer – also called *protobuf* – is a framework architecture for structuring serialized data. Protobuf's own developer web-page [21] describe it as "*[...] XML, but smaller, faster, and simpler.*". The reason for this is that protobuf – contrary to XML – is encoded to a binary wire format. While XML files can be altered directly through editing the file, the encoding of protobuf files make them impossible for humans to read. However, this is also a much smaller and more compressed format that eliminates the need of text parsing.

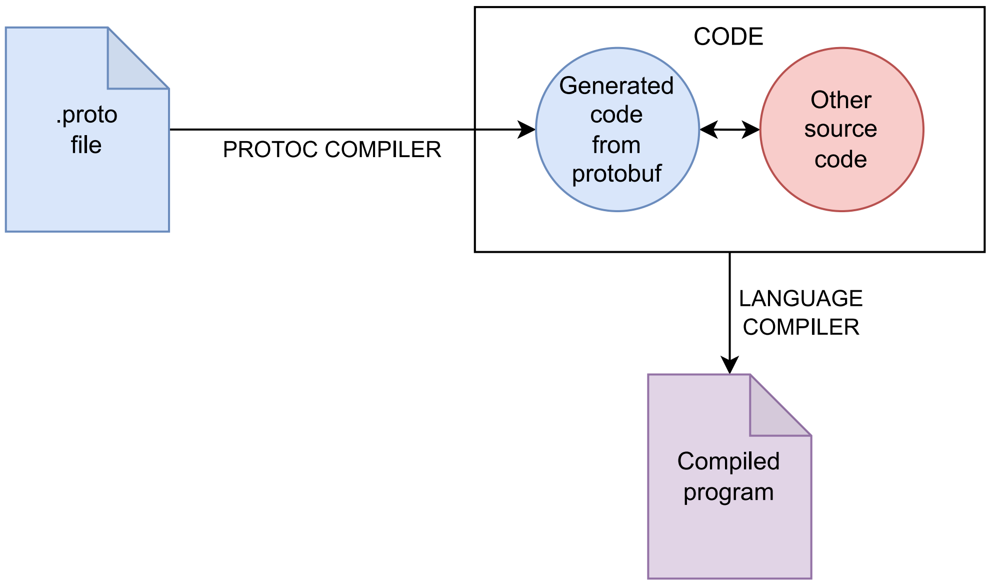


Figure 5.2: A simplified explanation of how protobuf works.

The basic workflow of protobuf is shown in Figure 5.2. First, a developer defines the format of custom messages in the `.protobuf` file. This is written in a custom but single language to define things like the fields of a message, their types, and if it is required or optional. Secondly, the `protoc` compiler parses the file and generates code in a language of the developers choice – e.g., C++. The generated code then contains all of the necessary information for it to send and receive information. This makes it possible for other code in the project to easily translate, read, or write data using the generated code. Lastly, a language specific compiler – e.g., `g++` – can compile can combine the source code and the generated into a single program.

This project uses a bundled `.proto` file and its generated classes to represent memory operations in C++. When reading from a protobuf trace, the generated code decodes the binary wire format of the protobuf files into object representations. Afterwards, other code in COCOASIM can access the decoded data through the generated code's class interface. An example of how this is done in practice is shown in Figure 5.3. The same example can also be reversed to visualize how encoded protobuf files are created, as seen in Figure 5.4. Both of these examples convert to and from the class `Request` which has a single variable called `Address`. While most real-world `.proto` files are more advanced than this one, the same base principle persists when reading and writing to and from encoded files.

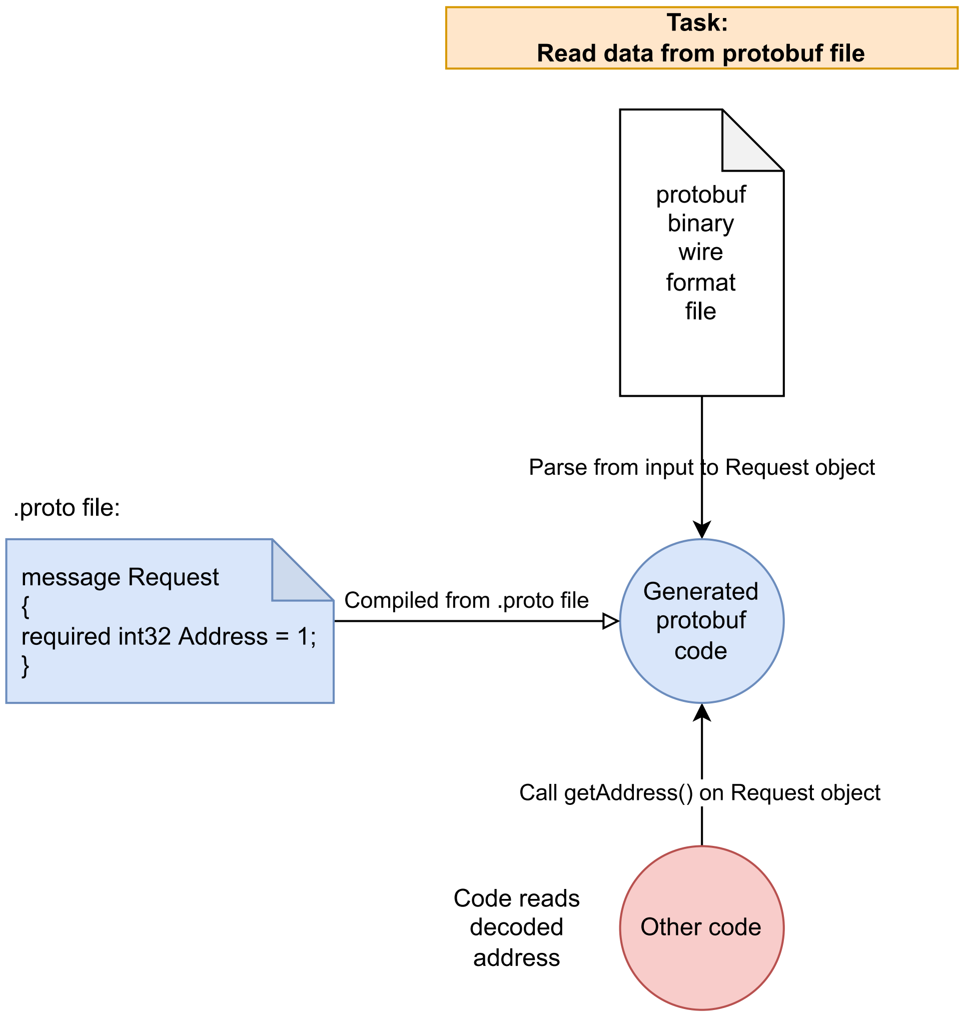

Task:
Read data from protobuf file
protobuf binary wire format file
Parse from input to Request object
.proto file:
message Request
{
required int32 Address = 1;
}
Compiled from .proto file
Generated protobuf code
Call getAddress() on Request object
Code reads decoded address
Other code

Figure 5.3: An example of how protobuf reads from encoded files.

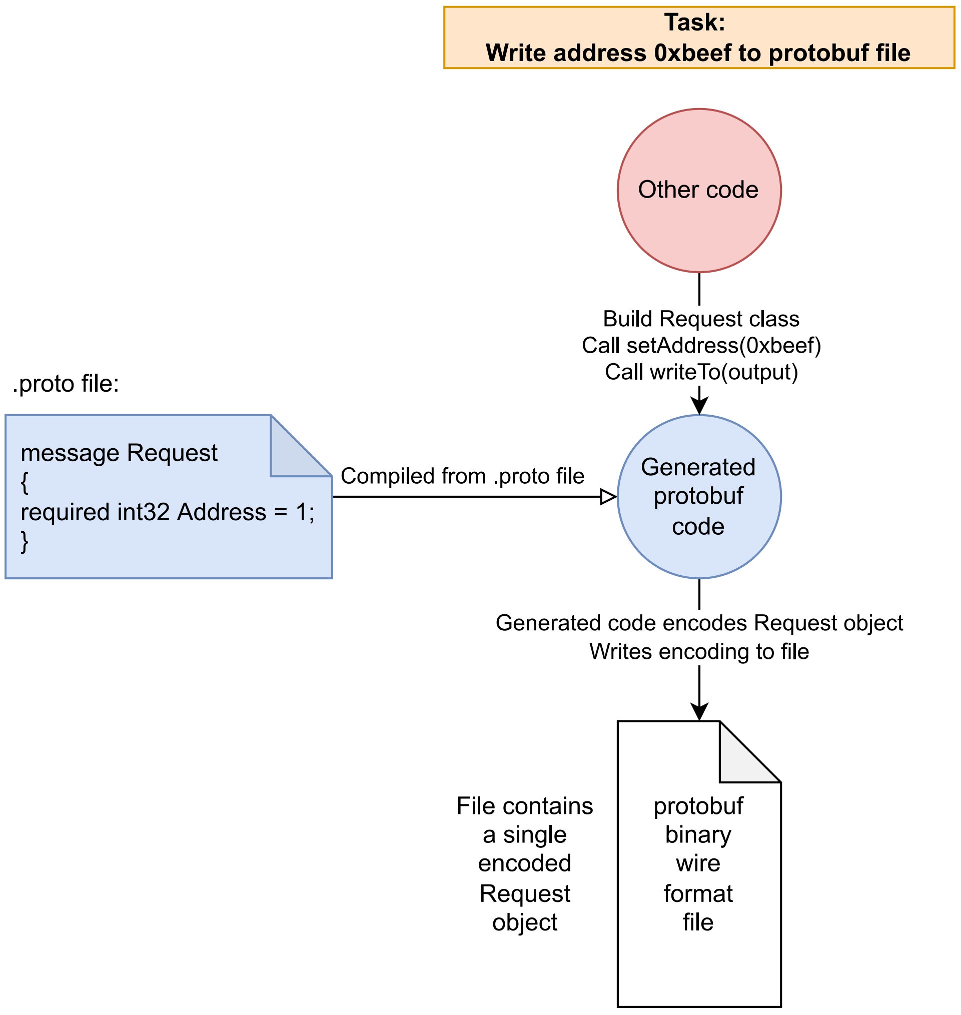

Task:
Write address 0xbeef to protobuf file
Other code
Build Request class
Call setAddress(0xbeef)
Call writeTo(output)
.proto file:
message Request
{
required int32 Address = 1;
}
Compiled from .proto file
Generated protobuf code
Generated code encodes Request object
Writes encoding to file
File contains a single encoded Request object
protobuf binary wire format file

Figure 5.4: An example of how protobuf creates the encoded protobuf files.

Note that COCOASIM only *reads* from protobuf files, and does not write or create new protobuf files. Instead, all files used for simulation is provided externally by Ole Henrik Jahren of ARM, Trondheim. However, the source repository of COCOASIM is bundled with another module called `pbtrace` that is capable of converting files in a custom JSON format to the protobuf format. This is also what is used to generate the traces used in this project. The process of generating traces is shown in Figure 5.5.

There are two major benefits of using protobuf to represent the memory traces:

1. As the traces are encoded using a compressed binary format, the simulator can quickly handle input without parsing string based formats like XML or JSON.

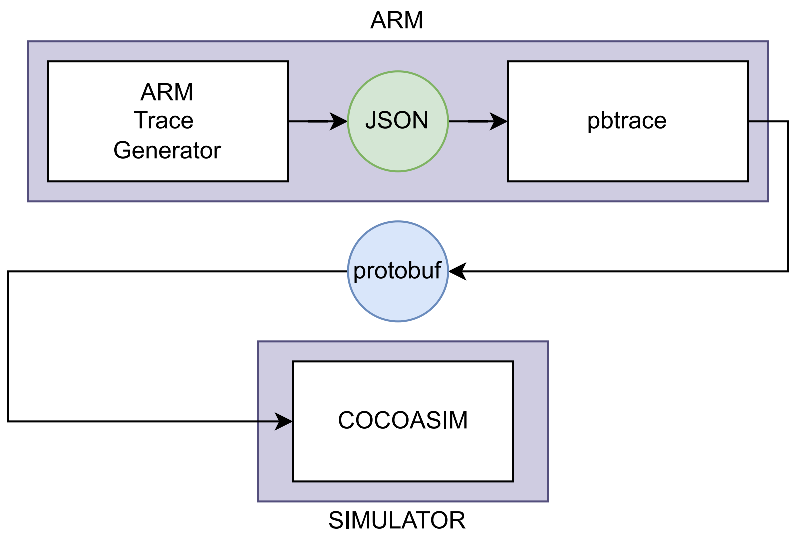


Figure 5.5: How traces are generated and used.

2. Since the data of the protobuf is *serialized*, items can be read one by one – ultimately minimizing the memory usage. The simulator does not need to read the entire trace, but can instead read individual items separately. This is highly beneficial for cache simulators in particular as the traces often consist of millions of operations. COCOASIM does not need to read new items from the trace if it already knows that the simulated system cannot handle more requests – e.g, when the pipeline stalls. Instead, the read can be paused and continued when appropriate.

Reading Traces

As seen in Table 5.3, COCOASIM does not take a filename as the input argument but rather a directory path. The reason for this is that the simulator is designed to handle multiple traces simultaneously, and the simulator treats each trace as a distinct core which communicates with its own exclusive L1 cache. An input directory may contain any number of trace files, but there must be an equal number of L1 caches and traces. Furthermore, each trace must be named in the format of `X.pbtrace`, where $0 <= X < n$ and n is the number of traces. Traces not named in this format will simply be ignored. When running a simulation, COCOASIM will assign `0.pbtrace` to the first L1, `1.pbtrace` to the second L1, and so on. A simple example of how this is done in a system with with four L1 caches can be seen in Figure 5.6. Note that while the example connects all L1s to a common L2, any hierarchical structure can be used as long as there are four L1 caches.

The very first access of each trace is scheduled (through the use of discrete-event simulation, see Section 3.4) for cycle 1 – meaning that all accesses happen in the same cycle. This should always result in a cache miss as every cache is empty, and accesses will propagate to a L2 that usually is shared by at least two L1s.

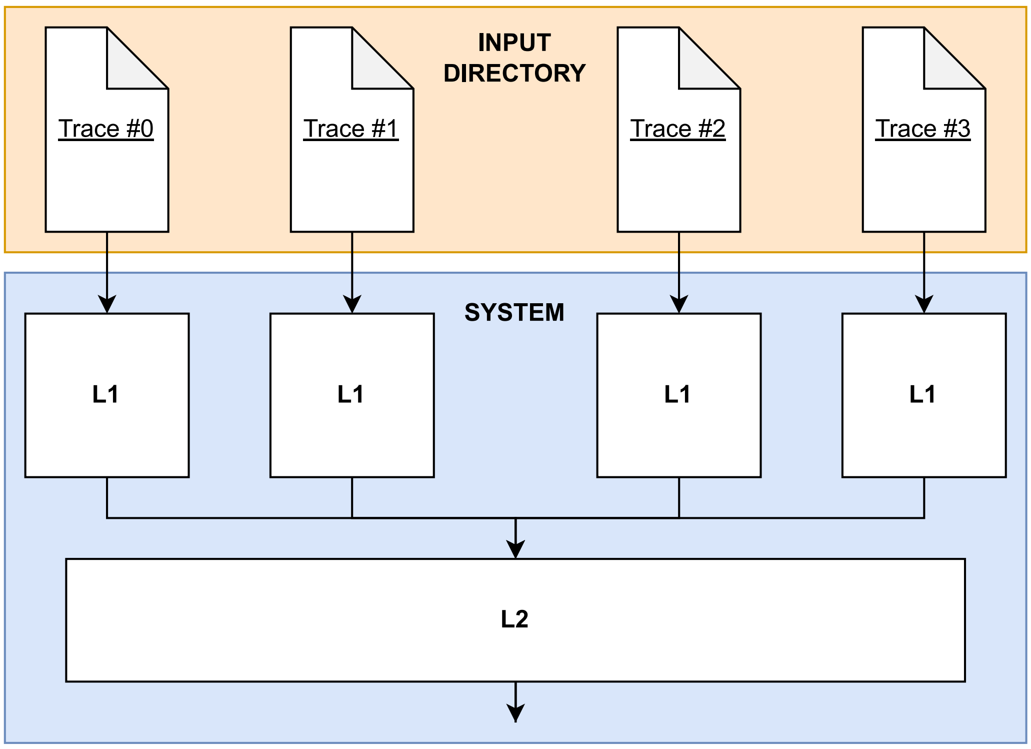


Figure 5.6: COCOASIM assigns each trace in the input directory to a separate L1 cache.

Assuming that the delay of every L1 is the same, this will cause multiple request to arrive in the same cycle. In these cases, whatever request was *scheduled* first gets priority while the other requests are stalled for one cycle. For all practical purposes, the requests with lower indices are scheduled first while higher indices are scheduled later. This means that if a user wants a certain trace to have priority, it should be assigned a lower index – e.g., `0.pbtrace`.

As an example, assume that COCOASIM has just started simulating the system of Figure 5.6, that the delay of each L1 is 1 cycle, and that the delay of the L2 is 10 cycles. This should result in the behavior shown in Table 5.5.

Trace Content

As mentioned in Section 5.3.3, the format of a protobuf traces is defined using a `.proto` file. This means that there are no predefined members of a protobuf trace, and that the traces can be custom-tailored to the needs of an application. For this project, the protobuf traces contain serialized memory requests with the following fields:

Cycle	Trace #0	Trace #1	Trace #2	Trace #3
0	Schedule access to L1#0	Schedule access to L1#1	Schedule access to L1#2	Schedule access to L1#3
1	Access L1#0	Access L1#1	Access L1#2	Access L1#3
11	Access L2	Stall	Stall	Stall
12	-	Access L2	Stall	Stall
13	-	-	Access L2	Stall
14	-	-	-	Access L2
...	...	...	...	...

Table 5.5: Accesses happening in the same cycle if possible – if not, lower indices are prioritized.

- **cycle** - The cycle that the request was issued. In the majority of cases, this is set to 0 as the entire program is set to execute instantly. It is important to note that a cache can only handle one operation each cycle, so the simulator will schedule requests to prevent them from all accessing the same cache on the first cycle.
- **op_id** - Each request/item has a unique id. This is used to resolve dependencies and to simplify debugging.
- **group_id** - What *group* a request belongs to. Groups are used to group collections of requests that use the same operation. The motivation behind this is that specialized caches can continue unblocked – see Section 3.1.4 – for a limited amount of unique groups, but must block if exceeding this limit.
- **op_type** - Type of operation – i.e., if the operation is a load/store/etc.
- **mem_type** - Memory type determining the memory type of the physical address mapping. Defines if the mapped memory is cacheable/coherent/etc.
- **paddr** - Physical address.
- **asid** - Unique address space ID of the address mapping.
- **vaddr** - Virtual address.
- **msg_type** - Message type indicating what type of memory operation should be performed. For example, if the aforementioned *op_type* is a read, this determines what type of read it is.
- **byte_en** - Byte enable. A 64-bit integer determining which of the 64 bytes of data is affected by the operation.
- **reason** - The reason for the operation (if any).
- **alloc** - Allocation hint. One request may contain multiple allocation hint – one for each level of caches. Determines if the cache should attempt to allocate the tag and data in a cache.
- **dep_id** - Dependency ids. IDs that this operation rely on. All requests with ID present in this list must be completed before this request can be scheduled.
- **flag** - Special flag associated with the request, if any.

5.3.4 Systems

Using a System

As seen in Table 5.3, there are two *modes* of defining what system COCOASIM should simulate. If the `-c` flag is specified, the simulator will load a configuration from a custom JSON file, and translate this into a system object. If the `-s` is specified however, the simulator will search through a catalog of existing systems and use it if it exists. As it only makes sense to simulate a single system for each simulation, only one of these flags can be present at the same time. Note that each of these options have their own default alternative – i.e., `-C` and `-S` – that can be used in their place. When running a simulation using the `-C`, note that this flag will simply default to the filename/path of `configs/test_system.json`. Thus, the file must exist for the simulation to work. The `-S` flag will however use the bundled `TestSystem`, and is guaranteed to work.

Feature	Configuration Mode	System Mode
Systems are defined in ...	JSON	C++
System created by ...	Building blocks	Code
Architecture limitations?	Few	None
Configurable cache parameters?	✓	✓
Can use any replacement policy?	✓	✓
Control over cache hierarchy design?	✓	✓
File-location agnostic?	✗	✓
Can be changed without rebuilding?	✓	✗

Table 5.6: Comparison between "Configuration" mode and "System" mode.

A comparison of the "Configuration" mode – used by the `-c` option – and the "System" mode – used by the `-s` option – is shown in Table 5.6. Note that the "System" has virtually no limitations on the system (as long as COCOASIM knows how to use it), while the "Configuration" mode is limited by the configuration file. Still, most system without any radical changes to the caches or the cache hierarchy should be possible to represent using the configuration mode. While both modes are perfectly fine for setting up systems, the "Configuration" mode is recommended for most users. The reason for this is twofold: 1) Users can define systems in a simple JSON configuration file without having to touch the code of the simulator, and 2) Multiple variations in size, associativity, etc. can be done repeatedly without rebuilding COCOASIM.

Systems in COCOASIM consists of two components: the *cache hierarchy*, and the `main memory`. In addition, the simulator also uses a `I/O interface` component to forward the memory requests from input to the system itself. The three components are described below:

1. **The I/O interface** - Every request COCOASIM handles originates from an interface that reads and forwards the content of the protobuf trace files. In addition to decoding the protobuf format, the interface is responsible for sending each memory requests to the appropriate L1 cache. In the COCOASIM source code, this interface is called `DataOutput` as every request returns here after performing a read or a write. The nearest equivalent of this in other simulators is perhaps gem5's memory bus connecting the CPU and L1 cache, but this is only a rough analogue as the `DataOutput` is also responsible for e.g., decoding requests and pausing and continuing reads from input.
2. **The cache hierarchy** - The main component of a system in COCOASIM is the cache hierarchy. This includes every cache in the system as well of their sizes, replacement policies, latencies, and info on how they are connected to each other. The cache hierarchy is not explicitly described in a system, but rather implicitly as every cache know of the cache – or main memory – below it. A large number of properties – including how the caches are connected to each other, how many L1 caches there are, how many levels there are, the cache sizes, the associativity of each cache, and the replacement policies used in each cache – are fully configurable, and can be changed using either of the two modes.
3. **The main memory** - The last component in every system is the main memory. The simulator assumes that all data is present in the main memory at all times, and any memory request that misses in the lowest level cache (LLC) will eventually hit here. While most cache configurations – like associativity or block size – don't make sense for the main memory, it is possible to change the memory access latency.

System Mode

When using the `-s` flag, the simulator uses a system already defined in C++. There is no need – or even possible for that matter – to provide COCOASIM with the properties of the system as these are already defined in the system. For example, loading `TestSystem` will simply use the system as it is – including its cache hierarchy, cache sizes, replacement policies, and memory access latencies. Different traces can naturally be used for the same system, but the system itself cannot be changed easily. For example, if a user wants to change size of the caches in a system this way (say, from 128kB to 256kB), this would have to be changed in the code itself. For most cases, this would only amount to changing a couple of lines in a single file, but would still require rebuilding the source.

As with most abstractions in object-oriented programming, systems in COCOASIM are represented by a *class*. In this way, a system is treated as a separate object when initialized, and has access to its own variables and methods. More specifically, systems are represented by sub-classes derived from the `CacheSystem` super-class. While individual sub-classes may differ greatly in their designs and available features, they must all derive from `CacheSystem` to ensure that COCOASIM has a minimum interface. For example, COCOASIM must have access to the system's L1 caches to connect the input from the traces to each individual cache.

```
// The class template of CacheSystem.hh

class CacheSystem : public BaseSystem {
public:
    CacheSystem();
    ~CacheSystem() override;

    TraceNumberSize get_number_of_l1_caches() const;

    std::vector<Cache*> l1_caches;

protected:
    void do_setup() override;
    void do_integrity_check() const override;
    void ignite() override;

    void destroy_l1_caches();

};
```

Listing 10: The CacheSystem class.

The `CacheSystem` class used as the template for all other systems is shown in Listing 10. Note that the class has a couple of public members – i.e., `l1_caches` and `get_number_of_l1_caches()` – that will persist for all classes that derive from `CacheSystem`. This way, COCOASIM is able to access the `l1_caches` of *any* system that derives from the super-class. Additionally, the class has several methods that override basic functionally, and thus also implicitly can be overridden themselves. Of the listed methods, *do_setup()* is the most interesting as this initializes the system. This is useful as classes that derive from `CacheSystem` may override this method to define how that specific system is initialized.

As an example, consider two of the simulator's bundled systems – `TestSystem` and `MinimumSystem`. As its name suggests, `TestSystem` is used to test that the very basics of COCOASIM work as intended. As such, it only contains a single L1 cache that is connected directly to the main memory, and does only read from one protobuf trace – i.e., `0.pbtrace`. For simulator tests that use multiple traces, `MinimumSystem` can be used. While this also only have a single level cache hierarchy, the system contains up to a total of eight L1 caches. As all of the provided trace sets have a total of eight trace files, this system is the minimum that can be used to handle every memory operation. Fragments of the C++ header files of `TestSystem` and `MinimumSystem` can be seen in Listing 11.

```cpp
// TestSystem derives from CacheSystem...
class TestSystem : public CacheSystem {
    ...
protected:
    // Unique setup override for TestSystem
    void do_setup() override;
    ...
};

// ... and so does MinimumSystem
class MinimumSystem : public CacheSystem {
    ...
protected:
    // Unique setup override for MinimumSystem
    void do_setup() override;
    ...
};
```

Listing 11: The CacheSystem class.

While both `TestSystem` and `MinimumSystem` derive from `CacheSystem`, they are distinct systems with different designs. Thus, each system overrides the *do_setup()* individually with their own code.

The overridden `do_setup` function for `TestSystem` and `MinimumSystem` can be seen in Listing 12 and Listing 13 respectively. Whereas `TestSystem` only initializes a single cache, `MinimumSystem` uses a `for`-loop to set up as many as there are traces. Note that while the systems use different associativity and replacement policies, all caches and cache line are of the same size. This is done purely for simplicity, and different variations of the systems can easily be created by simply changing e.g. the `l1_size` variable.

After defining the system itself in C++, the simulator also needs to know of the system's existence. In other words, the command `-s MinimumSystem` must lead to the `MinimumSystem` class actually being used. This is done through a simple mapping between a string – e.g. "MinimumSystem" – and the actual object in a simple helper class. The simulator will simply call a function named `get_system(argument)`, and the catalog will return the appropriate system if it exists.

```
// The do_setup() method of TestSystem.cc
void TestSystem::do_setup()
{
    CacheSystem::do_setup();

    Size main_memory_size = Size(1, memory_size_order::gigaByte);

    delete output;
    output = new DataOutput(this);

    delete main_memory;
    main_memory = new MainMemory(this);
    main_memory->initialize_main_memory(main_memory_size);

    // Cache size
    const Size l1_size = Size(8, memory_size_order::kiloByte);
    const Size block_size = Size(64, memory_size_order::byte);
    const unsigned associativity =
        Cache::calculate_full_associativity(l1_size, block_size);

    // Scale cache number to number of traces
    destroy_l1_caches();

    auto * cache = new Cache(this);
    cache->initialize_cache(l1_size,
                            associativity,
                            new LRU(),
                            block_size);
    cache->connect_to_child_memory(main_memory);

    l1_caches.resize(1);
    l1_caches[0] = cache;
}
```

Listing 12: The setup method of TestSystem.

In summary, the "System" mode makes it possible for a user to create their own system in C++, and then use the -s flag to use the system in a simulation. As discussed in Table 5.6, this comes with the advantage of being able to define any possible design. On the other hand, this approach requires frequent rebuilding of COCOASIM for every change done to the system. Every property – like cache sizes – needs to be hard coded into the system file as there is no way to communicate changes at execution-time.

```
// The do_setup() method of MinimumSystem.cc
void MinimumSystem::do_setup()
{
    CacheSystem::do_setup();

    Size main_memory_size = Size(1, memory_size_order::gigaByte);

    delete output;
    output = new DataOutput(this);

    delete main_memory;
    main_memory = new MainMemory(this);
    main_memory->initialize_main_memory(main_memory_size);

    // Cache size
    Size l1_size = Size(8, memory_size_order::kiloByte);
    Size block_size = Size(64, memory_size_order::byte);

    // Scale cache number to number of traces
    destroy_l1_caches();
    l1_caches.resize(number_of_traces);

    for (TraceNumberSize i = 0; i < number_of_traces; ++i)
    {
        auto* cache = new Cache(this);
        cache->initialize_cache(l1_size,
                                4,
                                new RandomReplacementPolicy(0),
                                block_size);
        cache->connect_to_child_memory(main_memory);
        cache->set_access_latency(1);
        cache->set_name("Cache", static_cast<unsigned>(i));

        l1_caches[i] = cache;
    }
}
```

Listing 13: The setup method of MinimumSystem.

Configuration Mode

When using the `-c` flag, the simulator will attempt to load a configuration from a file and translate it into a system. Contrary to "System" mode, this approach requires the file to contain all the required information to represent a system while COCOASIM only parses it. As implied, the advantage of this is that the configuration file is completely independent of the simulator. Making changes in the configuration file has no impact on the simulator – allowing systems to be defined at run-time rather than compile time. Note that loading `test_system.json` will not load a single static system as with "System" mode, but rather a dynamic system based on the contents of the configuration file.

When discussing systems defined in C++ in the previous segment, it was emphasized that changes to a system would require recompiling code as these are statically declared. As there are no way to communicate changes to a system at run-time, this would have to be done by changing the code itself. While this is still true, there is one system that circumvents this requirement: `ConfigurableSystem`.

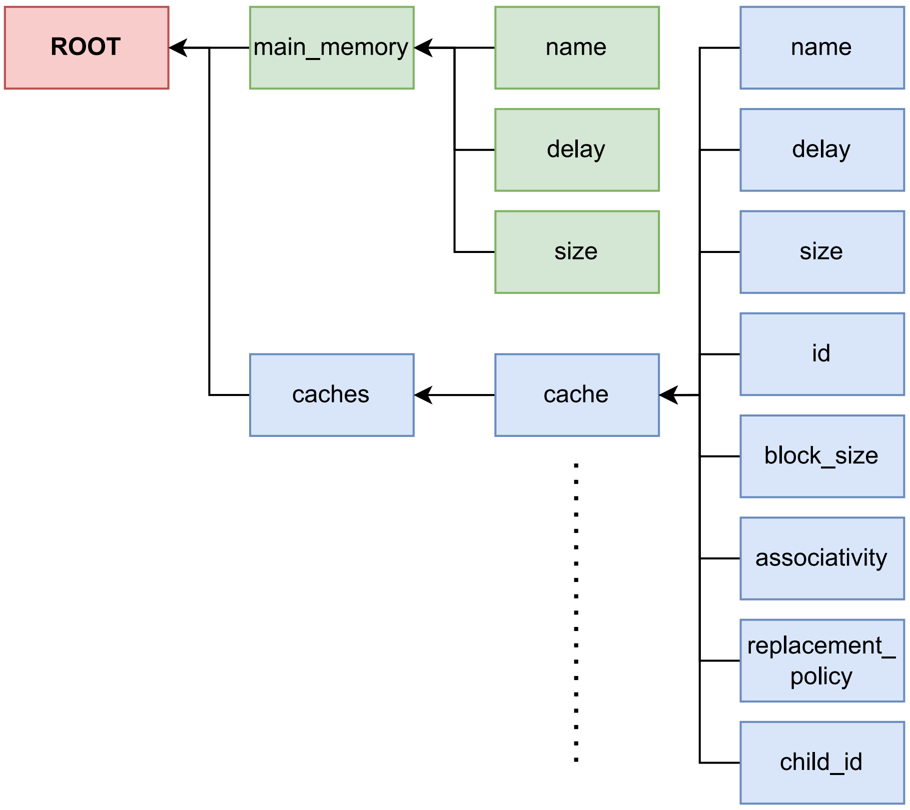
```
// A fragment of ConfigurableSystem.hh
class ConfigurableSystem : public CacheSystem {
public:
    ConfigurableSystem();
    ~ConfigurableSystem() override;

    void load_system(const boost::filesystem::path& path);
    ...
};
```

Listing 14: The header of the ConfigurableSystem class

The header of `ConfigurableSystem` is shown in Listing 14. There is two aspects of this header that are worth noting: 1) It inherits from the `CacheSystem` class – allowing it to expose its L1 caches to COCOASIM and ultimately allow for simulation, and 2) It contains the method `load_system()` and does not override the `do_setup()` method that `TestSystem` and `MinimumSystem` do. As the system never sets up any logic in the `do_setup()` function, it is completely empty. Thus, it makes no sense running system using "System" mode. When running a simulation using "Configuration" mode however, COCOASIM will initialize a (empty) `ConfigurableSystem` and invoke its `load_system()`. The system will then attempt to load the content of the given path, and translate it into components like caches and replacement policies. The `load_system()` assumes that the provided file is a JSON file in the correct format, and iterates through its content while translating strings to mapped objects. While the function itself is quite long and complex, the external `boost` library [22] is used to simplify the parsing process. The data hierarchy of the JSON configuration is shown in Figure 5.7.

All possible configurable options are listed in Table 5.7 and can described as the following:

- **main_memory** - The main memory. Every system *needs* to have this component, so if this is specified in the configuration the simulator will initialize a default main memory instead.



The only required property is *caches* – all others are optional.

Figure 5.7: The data structure of the configuration file.

- **caches** - A list of cache objects. The list itself is required, but it can contain any number of caches. However, an array of zero caches would not make any sense and cause the simulator to crash.
- **cache** - A cache. Any number of caches can be added in the "caches" array this way. The caches themselves do not require any data, so an empty cache object will instead use default values. However, it is recommended to at least provide each cache with a distinct name as to distinguish them from each other in the logs. Also note that all caches are added in the same way, and that the `id` and `child_id` is used to connect the caches to each other.
- **name** - The name for either a main memory or cache. Used for logging behavior, and can be ignored if the plan is to use either `cocoasim-fast` or `cocoasim-opt`.
- **delay** - The access latency in cycles it takes to access this components. Defaults to 1 cycle for caches, and 100 cycles for the main memory.
- **size** - The size of either a main memory or cache. Has no practical use for the main memory, but will limit the capacities of caches. The simulator accepts sizes in the format of `<number><order>B` - like "128kB" or "8MB".

Property	Type	In ...	Optional?
main_memory	Object	Root	✓
caches	Array<Object>	Root	✗
cache	Object	caches	✓
name	string	main_memory / cache	✓
delay	int	main_memory / cache	✓
size	string	main_memory / cache	✓
id	int	cache	✓
block_size	string	cache	✓
associativity	int/string	cache	✓
replacement_policy	string	cache	✓
child_id	int	cache	✓

Table 5.7: The types and members of the configuration file.

- **id** - Every cache has an id. This can either be explicitly assigned by the id property, or automatically assigned if the id field is absent. Ids are used to connect caches to each other, and may be ignored if simulating a flat cache hierarchy of only a single level. For any cache hierachy with multiple levels, it is recommended to explicitly declare all ids. Unsurprisingly, the ids also need to be unique for each cache.
- **block_size** - The block size determines how large a single cache line is. This can either be declared in the same format as the `size` property, or left absent to use the default value of 64 bytes.
- **associativity** - The associativity of a cache. This can either be set to any number (above zero), or the string "full". Any associativity above the maximum possible will abort the program. The simulator will use a full associativity if the property is absent.
- **replacement_policy** - The replacement policy to use. Any replacement policy can be used as long as COCOASIM knows of it. The simulator translates the provided string to a replacement policy in the object in a similar way to how it fetches systems in "System" mode – i.e., through a catalog mapping the name and the actual object. This is handled by the `CacheReplacement-PolicyCatalog` class which contains all available replacement policies in COCOASIM. New replacement policies can easily be added to this catalog in C++, but requires recompiling the source code. If no replacement policy is present in the configuration, the simulator will default to using Least Recently Used (LRU).
- **child_id** - The child id tells the simulator to place the cache with that id below this one in the cache hierarchy. In other words, a L1 cache will set its *child id* to the *id* of the L2 cache. On a cache miss, the cache will forward its request to the cache specified by the child id. A cache without a child id is assumed to be Last Level Cache (LLC), and will instead forward any misses to the main memory.

```json
{
  "caches": [
    {
      "name": "L1-Cache#1",
      "id": 0,
      "size": "8kb",
      "block_size": "64b",
      "associativity": "full",
      "replacement_policy": "lru",
      "child_id": 2
    },
    {
      "name": "L1-Cache#2",
      "id": 1,
      "size": "8kb",
      "block_size": "64b",
      "associativity": "full",
      "replacement_policy": "lru",
      "child_id": 2
    },

    {
      "name": "L2-Cache",
      "id": 2,
      "delay": 10,
      "size": "32kb",
      "block_size": "64b",
      "associativity": "full",
      "replacement_policy": "lru"
    }
  ]
}
```

Listing 15: Example of a JSON configuration with two L1 caches and one L2 cache.

An example of a valid configuration consisting of two L1 caches and a single shared L2 cache is shown in Listing 15. Note that the main memory is not defined – causing the simulator to use the default option. Also note how the L1 caches uses the id of the L2 cache as their child id – implying that the L2 is lower in the cache hierarchy.

As all of the fields of Table 5.7 needs to be supported for a system to be loaded, `load_system()` is a large method with complex logic. However, the function essentially performs three operations:

1. Reads the content of the provided JSON file, and parses each keyword to a separate object.
2. Uses the `id` and `child_id` to connect the caches to each other and ultimately the main memory.
3. Analyzes the user-defined configuration to identify the L1 caches and main memory, and exposes these so that the system can be simulated as any other `CacheSystem`.

In summary, the "Config" mode allows a user to define systems in JSON that can be changed quickly and with relative ease. Having a dynamic configuration file and a static interpreter eliminates the need of continuously rebuilding source code at the cost of limiting vocabulary of the configuration file to the keywords of Table 5.7. While this approach only allows for altering properties that can be parsed by the loading function, it is still the recommended mode for most use cases.

5.3.5 Output

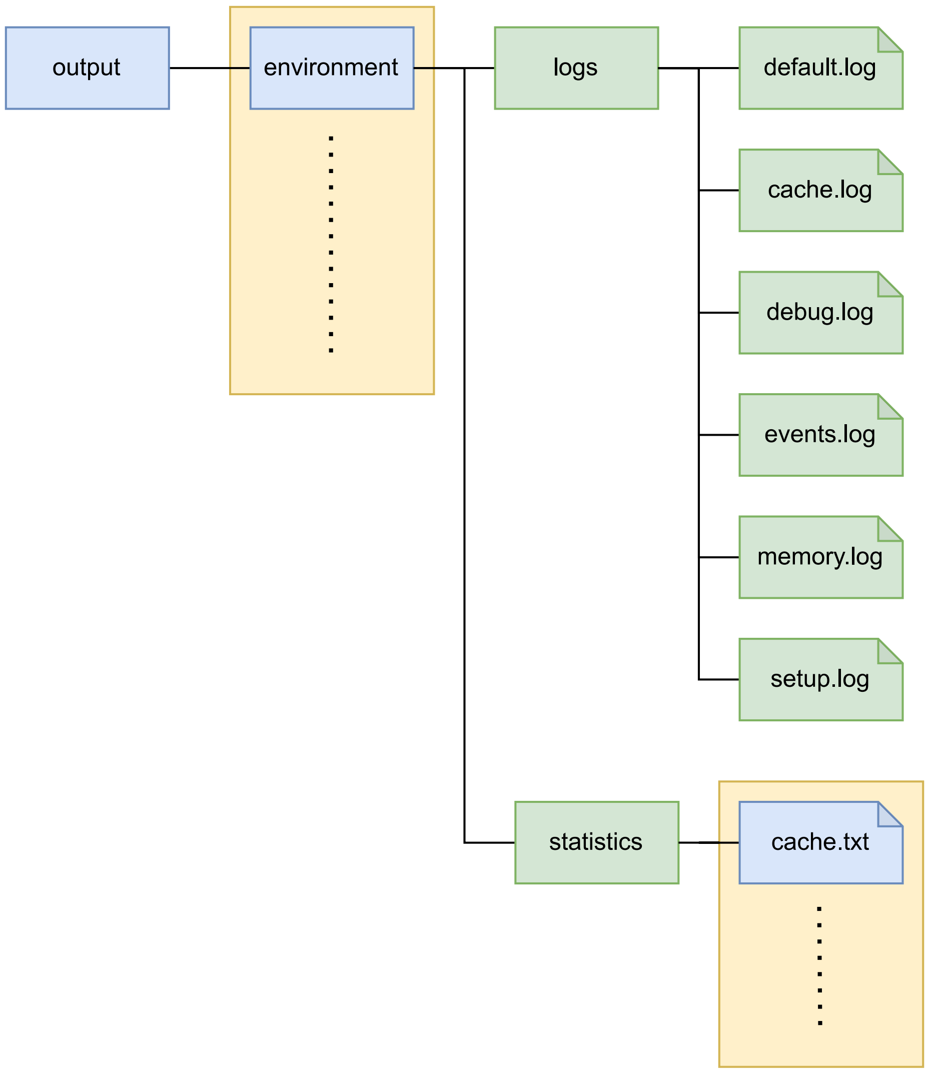
After the simulation has finished, COCOASIM outputs two thing: logs and statistics. Depending on the binary version and run-time flags the logging may also be written to the terminal output, or not at all. More information on how the simulator handles logging are discussed in Section 5.3.2. However, COCOASIM will always produce statistics as long as the simulation is successful.

Every simulation, COCOASIM writes logs and statistics to a custom directory that is created at run-time. As mentioned earlier, the name of the directory is either specified by the `-o` flag or the process id of the program if the flag is absent.

The structure of a typical generated output is shown in Figure 5.8. As explained in the figure, the blue and green colors indicate that the name is either dynamic or static, and the yellow boxes indicate that multiple sub-directories/files exist. The `logs` directory will only be used if the simulator actually logs something – meaning this will only be used when running the base `cocoasim` binary.

The directories that COCOASIM creates can be described as the following:

- "**output**" - The base directory which name is specified by the `-o` flag. Contains all other directories and files.
- "**environment**" - Used to separate what "environment" produces what logs and statistics. An environment may be seen as a "phase" of the simulation that performs distinct logic not present in other phases. Environments allow for advanced behavior when simulating, and is discussed more in detail in Section 6.1.4. However, the output-generating component of COCOASIM simply uses the name of the environments to separate logs and statistics from different phases into distinct directories. In most cases, this typically results in the environment directories of `setup` and `default`. The `setup` directory contains logs of the initialization process, while the `default` directory contains logs and statistics from the rest of the simulation.



Color codes: BLUE: Dynamic/configurable file names, GREEN: Static names, YELLOW: Variable number of files depending on system/configuration.

Figure 5.8: The hierarchy of the output directory.

- **logs** - Contains all logs split into different "channels". All messages are logged to `default.log`, and may also written to another specific log file depending on the nature of the log message. More on channels and logging behavior is discussed in Section 5.3.2.
- **statistics** - Contains all available statistics for every cache in the simulated system. Statistics are added when an event manipulates a cache in any ways that COCOASIM recognizes – like accessing it, hitting, or missing. Additionally, more custom statistics can by extending existing logic, but this must be done in "System" mode rather than "Configuration" mode. More on

the input modes of COCOASIM is discussed in Section 5.3.4. At the end of the simulation, COCOASIM will write the statistics to the output folder in the format of `<counter>-<name>-performance.txt` – like `0-L1-Cache#A-performance.txt` or `3-L2-performance.txt`. As implied, every cache writes statistics to a unique file.

By default, the simulator outputs the following statistics per cache:

- Accesses
- Successful Accesses (should be the same as Accesses in a successful simulation)
- Request Accesses ("Normal" requests)
- Info Accesses (Eviction messages, etc)
- Other Accesses (Should be 0 by default, but will recognize accesses that are neither Requests or Info if implemented)
- Loads
- Load Hits
- Load Misses
- Stores
- Store Hits
- Store Misses

5.3.6 Full Data Simulation

Compared to the other features of the simulator, the full data simulation is relatively simple. If the `-F` flag is set, COCOASIM will keep track of the actual data traveling through the cache hierarchy. In addition to storing metadata like tags, dirty/clean flags, and LRU counters, full data simulation will make COCOASIM also store data in the cache. The data itself has no effect on the simulator statistics, but makes it possible that the correct data is indeed being read and written. However, this comes at the cost of managing the data content of: 1) Every request in the system, 2) Every cache line in every cache, and 3) Data stored in the main memory.

When running a simulation without full data simulation, COCOASIM will simply pretend that data is read and written from the caches and main memory. However, to ensure that the data behaves as expected, an object is actually used to represent the data. The object – named `AbstractDataBlock` has no purpose and is completely empty, but is carried by memory requests as to pretend they have data. This is done to ensure that two following properties always hold:

1. Memory requests *reading* **must not** have data when requesting data from either a cache or the main memory, and likewise **must** have data upon returning.
2. Memory requests *writing* **must** have data when writing to any memory.

While the data itself doesn't exist, the abstract objects represent if the data *should be* present or not. This is primarily done to catch bugs as the simulator will exit with an error if the assertions above don't hold. Memory reads that either hit in a cache or make their way to the main memory will be given an empty class regardless of what address they requested. Other caches in the cache hierarchy will recognize that the returning read has data, and install their own "fake" copy as to mimic an inclusive cache hierarchy. This behavior can be seen in Figure 5.9.

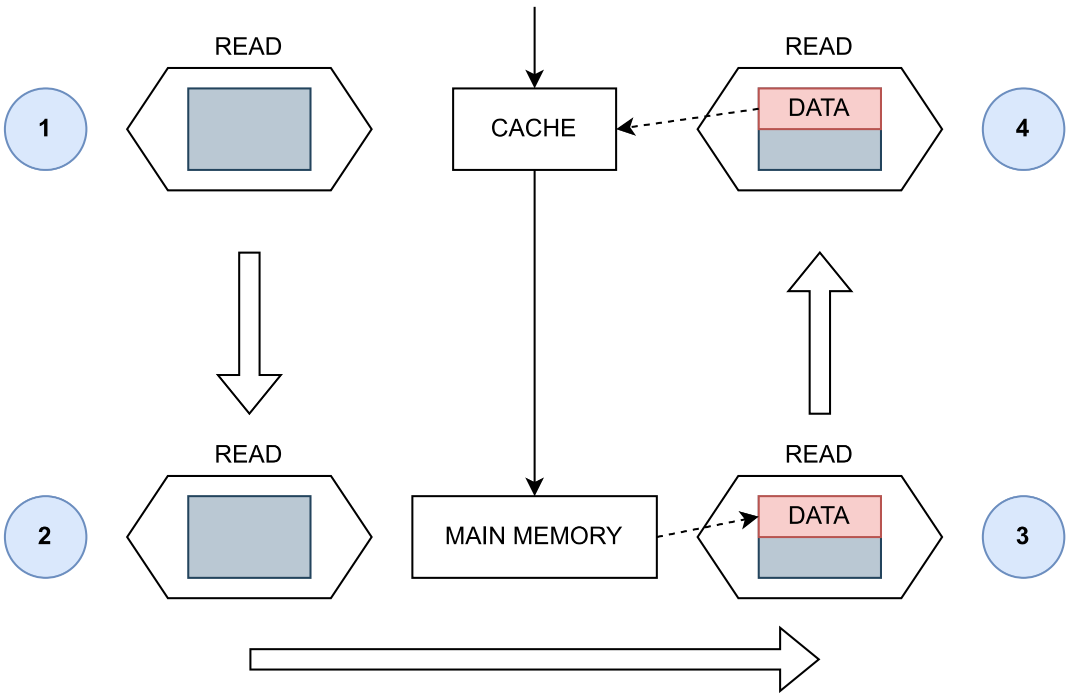


Figure 5.9: Typical read behavior with data simulation off.

Much of this logic persists when performing a full data simulation, but the simulator will also remember what data is read and written. Instead of an abstract data object, full data simulation will instead use an object encapsulating real data. This is possible as the object used in this mode – named `DataBlock` – inherits from the `AbstractDataBlock` used in a "normal" simulation. Additionally, the main memory will return distinct data blocks based on what address is requested. The main memory also keeps track of all data that is written to it through a mapping between the addresses and the data – making the data persist in the memory. This expanded behavior can be seen in Figure 5.9. Note that the request still contains a data object, but that it is expanded to also contain raw data.

Full data simulation is exclusively meant to verify that data behaves correctly, and not to perform an actual simulation with data. As such, the main memory does not contain any actual data when the simulation starts, and the only way to populate the memory with data is to perform write operations that propagate all the way to the main memory. This happens rarely and is only for the largest

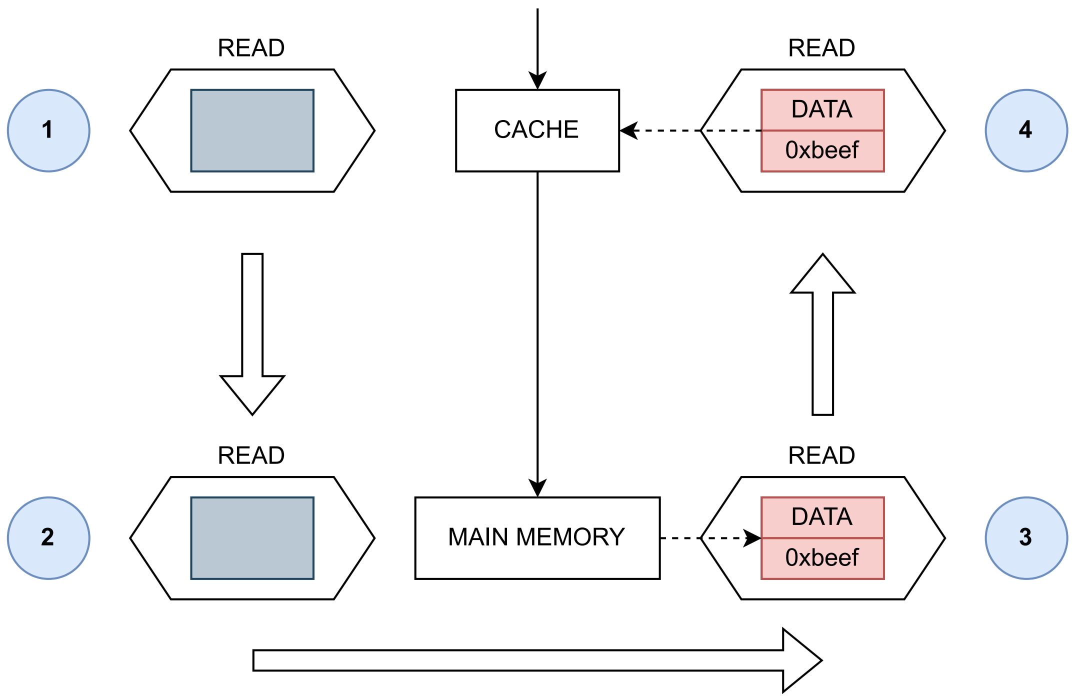


Figure 5.10: Typical read behavior with data simulation on.

traces as the cache hierarchy uses a write-back policy rather than a write-through policy. Additionally, while the input traces often include write operations, they do not specify *what* data it should write. The simulator instead creates dummy data for every write operation. Thus, full data simulation is primarily meant as a test feature. However, several things can be added to expand upon the full data simulation feature:

1. Support writing specific data in the trace format, and make the simulator use this instead of dummy data.
2. Expand the data object wrapper to use data of variable bit size. The current version of COCOASIM will use a 32-bit unsigned integer to represent the data, but this could be configured to be larger and more realistic.
3. Separate "normal" simulation and full data simulation at compile-time rather than run-time. As the `-F` flag can be specified as an argument, the simulator doesn't know if a simulation uses full data simulation before at run-time. If this is decided at compile-time instead, the compiler may optimize the binary by removing conditional branches that check if the simulation uses full data simulation or not.

Chapter 6

Simulator Design

While many of COCOASIM's features can be derived from the available configuration shown in Table 5.3, a lot of the underlying logic is completely invisible to the end user. For typical users, this is often adequate as the simulator provides an interface for making changes by either minimally changing the code – e.g., "System" mode of Section 5.3.4 – or by not touching the code at all – e.g., through use of arguments, inputs, and the "Configuration" mode of Section 5.3.4. Thus, understanding the underlying logic is not strictly necessary for users that simply wants to run a simulation. However, to fully understand COCOASIM, there are a couple more concepts that requires explaining. The purpose of this chapter is to describe the key components and programming paradigms that make simulation possible.

6.1 Key Concepts

The five most essential key concepts of COCOASIM are described below. These can be summarized as the following:

1. **The Event Engine, Section 6.1.1** - The "engine" of COCOASIM responsible for running a simulation.
2. **Events, Section 6.1.2** - The cornerstone of the "discrete-event simulation" concept that COCOASIM is built upon.
3. **SimulatorObjects, Section 6.1.3** - How almost every object in COCOASIM derives from a common super-class, and why this was done.
4. **Systems / Environments, Section 6.1.4** - An extension of Section 5.3.4 explaining the underlying logic of systems, and how these fit in an "environment".
5. **MemorySignals, Section 6.1.5** - The signals / packages / requests that represent information that move between two components in a system.

6.1.1 The Event Engine

While there a lot of critical components that together make simulation possible, the one that perhaps is the most important is the *Event Engine*. The engine is responsible for two key aspects:

1. Firing every "event" that occurs in a simulation at the correct times.
2. Keeping track of time in the simulation, and moving time forward when appropriate.

Every single thing that happens during a simulation is a consequence of the engine firing an event that causes some behavior. Events happen at determined times, and the event engine is responsible for triggering all events that happen when the simulation reaches one of these moments in time. For example, every cache and main memory in a simulation has a configurable variable named *latency* which indicates how many cycles it takes to access that memory. This latency can be exploited by the event engine to determine at which time the actual memory access happens. In this case, the memory access itself is an event that should happen after a certain delay has passed. The engine recognizes this, and sets the memory access to trigger in $n = latency$ cycles in the future.

As an example, assume that the simulator has just experienced a miss in a L1 cache. The simulator then goes through the following logic:

1. A miss just happened in the L1. Since the requested data is not present, the request should propagate to the L2 cache.
2. The simulator **creates a new event** representing an access to the L2. However, the code cannot execute this event just yet as the access is to happen in the future.
3. The simulator check how long it takes to access the L2 cache from the L1 using the L2's *latency* variable. Recall that this is fully configurable by the user, but for the purposes of this example assume that it is set to 10 cycles.
4. After knowing the delay, the simulator tells the event engine to **fire the memory access 10 cycles in the future**.
5. The event engine receives this information, and calculates what cycle the event should be fired. Recall that the event engine also is responsible for keeping track of what time it currently is inside the simulation. For simplicity, assume that the simulation is currently in cycle 5. The event engine sets the memory access to **trigger when the simulation enters cycle 15**.
6. After the event has been created and scheduled, the simulator continues as normal. While this happened as a result of a miss in a L1 cache, there may be other events happening in the same cycle – i.e., cycle 5.
7. After all events of cycle 5 has finished, the event engine continues to the **next cycle where any event triggers**. This method is called *discrete-event simulation*, and is explained in Section 3.4. How this works in practice is explained later.

8. At cycle 15, trigger the memory access event logic. This causes an access to the L2, and may result in either a hit or miss. Either way, this will the scheduling of another new event – either a hit causing a "return to L1 with data"-event, or a miss causing a "memory request to main memory"-event.

Note that most events cause the creation of another event. As mentioned in the last item of the list above, a memory access event will issue a new event regardless of a hit or miss as it either needs to propagate or return. As events are removed after fired, many events – like most memory accesses – replace themselves with a new event happening later. Thus, there are only two scenarios that cause the event engine to "run out of" events:

1. A deadlock where events are not fired as they are dependent on other events that never trigger. Needless to say, this should *not* happen in stable versions of COCOASIM.
2. The simulation runs out of events as every memory request has finished executing – implying a successful simulation.

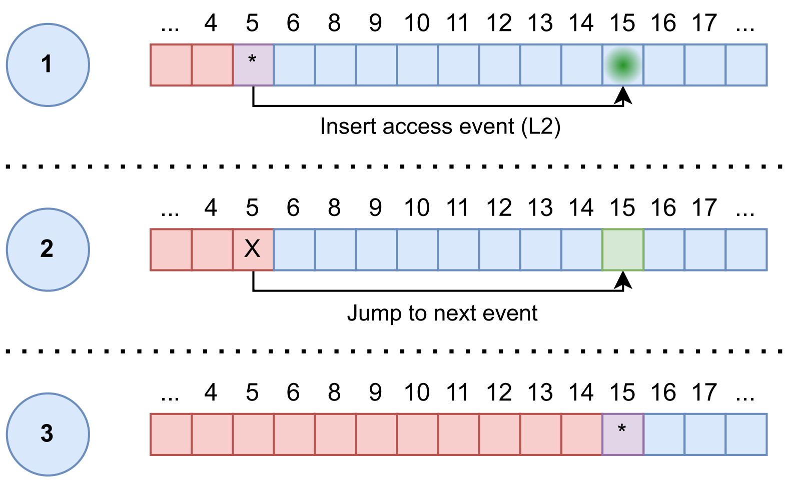
Note that events are continuously added by the input until *all* of the specified pbtrace files have been read. While memory accesses usually replace themselves, they do not issue any new event when returning to the aforementioned `DataOutput` interface as this indicates that the read or write has finished. The logic flow of memory request events are described in further detail in Section 6.1.5.

As mentioned, the event engine will only simulate cycles where it knows that something will happen. This is possible as the engine knows what the next event is at all times. While this may change during a single cycle, it will remain static after all events of a single cycle has been simulated. The simulator exploits this by finding the next event at the end of each cycle, and simply fast-forwards to that event and cycle while skipping all cycles in between.

The behavior of the aforementioned example is visualized in Figure 6.1. The example is split into three segments:

1. The miss in the L1 cache at cycle 5 causes the event engine to schedule the memory access event to the L2 10 cycles in the future – i.e., cycle 15.
2. The simulator exits cycle 5, and finds the next cycle where something happens. For this example, this is cycle 15 as the engine just added the L2 memory access event here. The simulator simply skips all the cycles in between cycle 5 and 15 as nothing happens in these cycles. In practice, this is done by simply updating the cycle counter to 15 and readying the L2 access.
3. The event engine fires the event accessing the L2 cache.

Note that in the simple example of Figure 6.1, there are no other events than the L1 and L2 access in the relevant time frame. However, if there was another event between cycle 5 and 15 – say another event accessing another L1 in cycle 8 – this would need to be handled before. In this altered example, the event engine would instead fast-forward to cycle 8, fire the new event, and then possibly fast-forward to cycle 15.



PURPLE: Current event/cycle, BLUE: Empty cycles / no events, RED: Past cycles

Figure 6.1: The event engine uses discrete-event simulation – allowing fast-forwarding past "empty" cycles.

The source code of the `run()` method responsible for simulating events and the passing of time is shown in Listing 16. Despite being only a few lines long, this method is responsible for running all logic of COCOASIM after initializing the simulator. In its simplest form, the main logic of the `run()` method can be reduced to: 1) Get events, 2) Simulate all fetched events, 3) Move to the next cycle, and 4) Repeat.

Lastly, one might perhaps wonder what happens if: A) There are multiple events firing in the same cycle, and if B) This happens frequently. The answer to question B is: yes, especially for memory access events. Recall that one of the main motivations behind COCOASIM is to simulate cache behavior concurrently. As discussed in Section 5.3.3, each individual "core" sends memory requests to its respective L1 cache for every cycle until the cache cannot handle more requests. As each of these events happen simultaneously – as seen in Table 5.5 – the first cycles of every simulation typically has a number of events each cycle equal to the number of L1 caches. This continues for a while, but becomes more asynchronous as some requests hits and other misses. All in all, it is somewhat rare for a cycle to only contains a single event, and more common to either have none or multiple events. Returning to the first question, the answer to A is: depending on the order they were issued. As seen in Listing 16, the `next_events` variable is a *vector* – meaning that new items added to the container are put in the back. When iterating over a vector, elements will be listed in the order they were added. In other words, events in the same cycle are handled by the in a First-In-First-Out (FIFO) principle – meaning events scheduled first will have priority despite all events happening

```cpp
// Slightly modified run() method of EventEngine
void EventEngine::run()
{
    CycleStruct<std::vector<SimulatorEvent*>> next_events_struct;
    std::vector<SimulatorEvent*> next_events;

    do
    {
        // Get next cycle and events
        next_events_struct = pass_to_next_events();

        // Ready next cycle and events
        cycle = next_events_struct.cycle;
        next_events = next_events_struct.data;

        // Fire all events in that cycle
        for (auto e : next_events)
        {
            e->fire();
            delete e;
        }

        // Mark cycle as finished, delete data
        events.erase(cycle);

    } while (!next_events.empty());
}
```

Listing 16: The run() method of the event engine

at the same time. At one point during the development this ordering was actually configurable, but would often create deadlocks or often unpredictable behavior as there often was a complex indirect dependency between events. To simplify the simulator, this feature was removed.

6.1.2 Events

Whereas the previous section focused on how events are scheduled, this section will focus on the events themselves. In its simplest form, an event in COCOASIM is simply an object-encapsulated function call that can be fired at any time. As hinted in Listing 16, all events used in by the simulator inherit from a super-class named `SimulatorEvent` which also inherits from another super-class named `ITriggerable`. In short, the classes can be described as the following:

- **ITriggerable** – A minimal *interface* telling its sub-classes to implement a single *pure virtual function* called `fire()`. The compiler forces developers to override this function before it can be used, or throw an error.
- **SimulatorEvent** – An *abstract class* deriving from the `ITriggerable` interface. This class also contains other simulator-specific methods – like a "what happens when this is scheduled by the event engine"-function. This is an abstract class as it has *does not* override the `fire()` method and thus cannot be instantiated.

```
// The ITriggerable header
class ITriggerable : public Nameable {
public:
    // Simply waterfalls to the protected
    // do_fire() method that can be safely overridden
    inline void fire() { do_fire(); }

protected:
    // Note that this is a pure virtual function
    virtual void do_fire() = 0;

};
```

Listing 17: The ITriggerable interface header

The header of the `ITriggerable` interface can be seen in Listing 17. Note that the interface *defines* the `do_fire()` method, but does not *implement* it. As `SimulatorEvent` does not override the method, it is not possible to create nor call `fire()` on either of the classes. However, it is possible to call `fire()` on a *subtype* of `SimulatorEvent` that has implemented the `do_fire()` method as seen in Listing 16.

By default, COCOASIM has seven types of event classes that derive from the `SimulatorEvent`super-class. However, about half of these are special variations of other events – leaving the following four as the most essential:

1. **MemoryAccessEvent** – For requests accessing memory like caches and the main memory.
2. **StreamReadEvent** – For reading memory operations from a pbtrace file.
3. **SignalTransferEvent** – For returning memory requests along with their potential data back up in the cache hierarchy.
4. **EntryInvalidationEvent** – For communicating the invalidation of an address between caches so that an inclusive cache policy can be simulated.

The following segment will attempt to explain how these events are used in practice by the simulator. While the `MemoryAccessEvent` might sound like a good example, the logic of memory accesses are actually somewhat complex as it needs to account for multiple scenarios like what to do if the cache is blocked and needs to stall. Thus, the next segment will review the cause, behavior, and effect of a `SignalTransferEvent` as this type is quite simple.

In summary, a `SignalTransferEvent` is scheduled when a memory request is ready to return after it either hits in a cache or reads from the main memory. It is then set to trigger in a number of cycles equal to the latency of the current memory the request returns from. For simplicity, this example will assume that a request returns from the L2 to the L1. For Assuming that the L2 cache has a latency of 10 cycles, it will also take 10 cycles to return to the L1 cache.

First, the event is constructed using three parameters: the signal/request, the source (i.e., the L2), and the destination (i.e., the L1). These values are saved in the event object so that they can be used later. The header of the `SignalTransferEvent` class can be seen in Listing 18.

```
// SignalTransferEvent.hh
class SignalTransferEvent : public SimulatorEvent {
public:
    SignalTransferEvent(MemorySignal* signal,
                        SignalInteractive* source,
                        SignalInteractive* destination);

protected:
    void do_fire() override;

    MemorySignal* event_signal;
    SignalInteractive* event_source;
    SignalInteractive* event_destination;
};
```

Listing 18: The SignalTransferEvent class header

This causes the event to act as a container. However, note that the logic of the actual return is not simulated before the event is fired. This could be achieved by simply calling the `fire()` method, but would lead to incorrect behavior as this cannot happen before in 10 cycles. Instead, the simulator passes the object to the event engine and tells it to trigger it in 10 cycles. Assuming that the simulator currently is in cycle 10, the event engine schedules the event to happen in cycle 20. At cycle 20, the engine finally calls the `fire()` method on the `SignalTransferEvent` object – indicating that the memory request has arrived at the L1.

```
// The SignalTransferEvent implementation of do_fire()
void SignalTransferEvent::do_fire()
{
    event_destination->receive_signal(event_signal, event_source);
}
```

Listing 19: The SignalTransferEvent class locally overrides the `do_fire()` function.

Firing the event will "waterfall" into the object calling `receive_signal()` on the L1 object. Using the saved memory request, source, and destination, the event is able to provide the function with the correct target and parameters. The `receive_signal()` method itself will update the content if the L1 cache to match whatever data was fetched before creating a new event to return it from the L1 cache to the "core"/`DataOutput`. A summary of the scheduling and firing the event can be seen in Figure 6.2 and Figure 6.3 respectively.

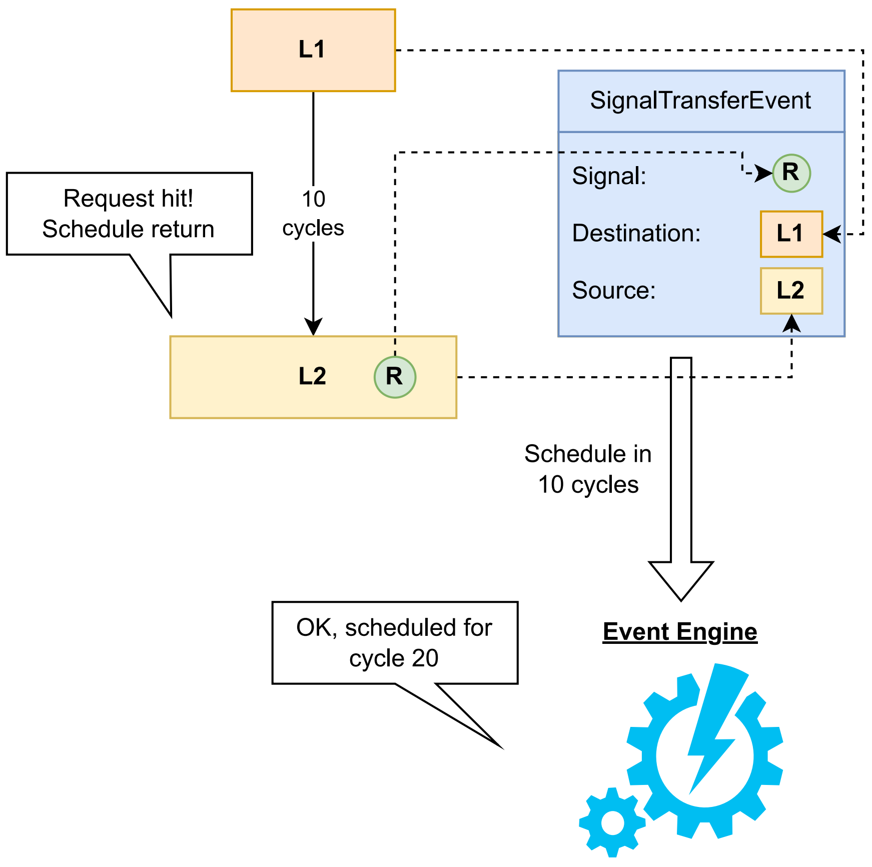
While this example described the behavior of `SignalTransferEvent` in particular, the same concepts applies to all event types. In total, the scheduling and firing of events make up most of the logic of COCOASIM.

6.1.3 SimulatorObjects

While COCOASIM often use a myriad of different objects with vastly unique behavior – like caches, a main memory, replacement policies, and Miss Status Holding Registers (MSHRs) – most derive from one common super-class: the `SimulatorObject`. In fact, there are only a few selected classes that *do not* derive from this – being mainly events, loggers, or systems. Broadly speaking, any object that can be represented as something physical is a `SimulatorObject`.

The decision to making the vast majority of objects deriving from a common class is inspired from the "SimObjects" of gem5. SimObjects – while different from the SimulatorObjects of COCOASIM – allow any object deriving from it to have access to a basic interface. This allows gem5 to recognize and interface with the object. For example, gem5 will call the method `init()` on all SimObject when the system is fully initialized and ready to simulate. Users may have a custom object derive from the SimObject class and then override the `init()` function to trigger custom behavior.

The `SimulatorObject` follow the same base principle as SimObjects, but has only a small number of inherited methods and variables. While SimObjects have several virtual methods that can be overwritten for custom behavior, `SimulatorObject` instead have a collection of regular methods and one pure virtual function – named `do_get_system()`. Note that as this function is pure virtual, any class that derive from `SimulatorObject` must implement this method. The `do_get_system()` method is a simple "getter"-function that return what *system* the object is a part of. While systems and environments are discussed more in



R = Request

Figure 6.2: After hitting in the L2 cache, COCOASIM creates a SignalTransfer-Event to the L1.

detail in Section 6.1.4, the main motivation behind this decision is to make all objects able to access the same pseudo-global object. Through this common system, objects can access data and functions global to the simulation – like the logging interface, event engine, or even other objects in the same system.

The header of the `SimulatorObject` is shown in Listing 20. Note that in addition to enforcing every simulator object to contain a pointer to its owning system, each object also inherits the other "get"-functions and the `set_name()` method. Furthermore, every simulator object also inherits from the `Nameable` class that provides each object with a name used for identification. The `set_name()` function of `SimulatorObject` overrides this name – allowing users to name the object anything. In short, this makes it possible to, e.g., call `set_name("L1")` on a cache or `set_name("LRU")` on a replacement policy.

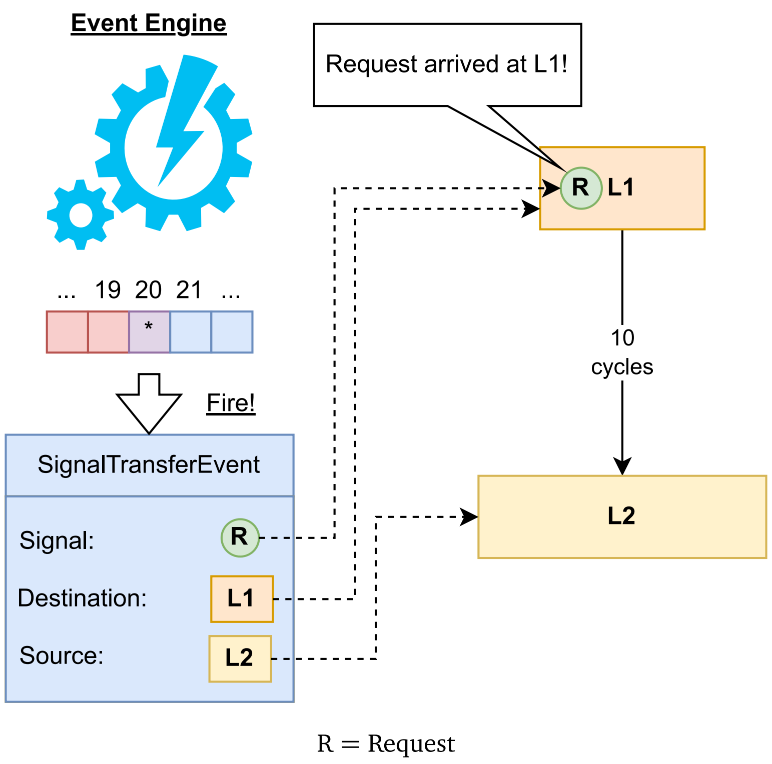


Figure 6.3: At cycle 20, the event engine fires the scheduled SignalTransferEvent.

The concept of designing classes and objects in this way is highly advantageous for complex object-oriented programming as objects can be built incrementally on top of other existing code. For example, the logic of `SimulatorObject` needs only to be defined once but are used by almost all other objects. Likewise, even even broader logic can be used to define a class for all objects that can be named – as the `Nameable` class does. Note that there exists several objects in COCOASIM that aren't SimulatorObjects, but still benefit from having a name – like for example events. While it is possible to add a name logic to every class individually, this method is redundant as it requires adding the same code to multiple classes. Instead, a class deriving from an existing class with this logic already implemented may simply add new functionality instead of worrying about basic functionality. This is naturally also true for objects deriving from `SimulatorObject` as well – as for example with the `Memory` class. This class inhabits all of the logic defined in the `SimulatorObject`, but also provides an interface for memory accesses. Consequently, the logic defined in the `Memory` class is used by the `Cache` and `MainMemory` classes as both these have access interfaces. As shown, this creates a hierarchy of classes where each class adds a little functionality on top of the existing. As subclasses branch out from each other, they gain increasingly specific behavior, but still share the same parent class.

```
class SimulatorObject : public Nameable {
public:
    SimulatorObject();

    inline BaseSystem* get_system() const { return do_get_system(); }
    DefaultLogger* get_logger() const;
    EventEngine* get_engine() const;

    void set_name(std::string new_name);
    ...

protected:
    // Pure virtual
    virtual BaseSystem* do_get_system() const = 0;
    ...

};
```

Listing 20: The SimulatorObject class – slightly simplified.

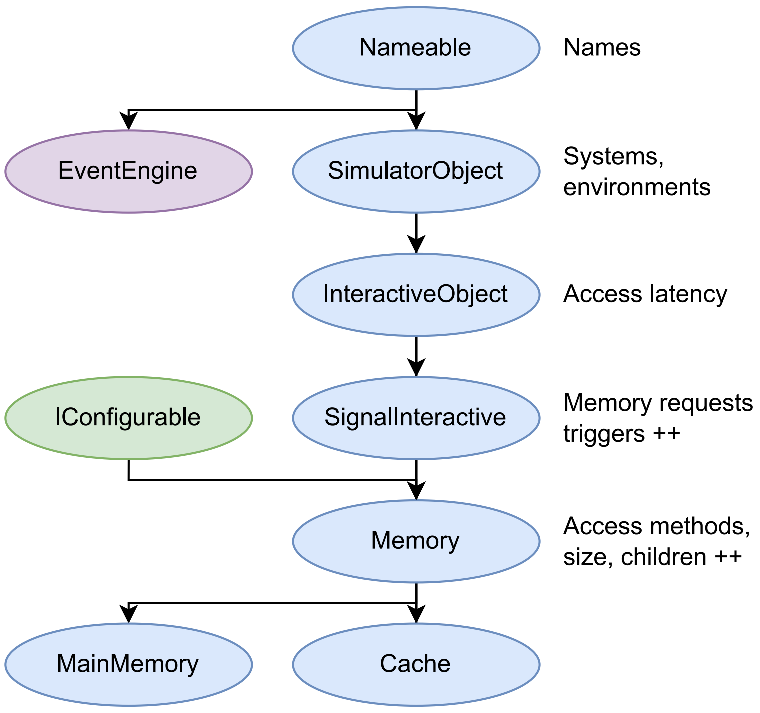
A class hierarchy featuring the `EventEngine`, `MainMemory`, and `Cache` classes can be seen in Figure 6.4. Note that while the `EventEngine` is not a `SimulatorObject`, it still has a common super-class with `MainMemory` and `Cache`. Also note that the `Memory` class also inherits from the `IConfigurable` interface – telling the simulator that memories can be configured in "Configuration" mode. While `MainMemory` and `Cache` both are memories, the classes themselves are vastly different.

6.1.4 Systems & Environments

As mentioned in the previous section, every simulator object exist in a system. As shown in Listing 20, all simulator objects have a method for returning the system it is a part of. In the vast majority of cases, this means that the objects have access to a shared class – i.e., `BaseSystem` – with data and methods that all simulator objects can access. This approach can be compared to the use of a *global variable*, but has two major advantages:

1. The simulator circumvents the need of an omnipresent global variable.
2. There can be *multiple* systems active at the same time.

The first advantage is grounded in the fact that the use of global variables is generally discouraged. The main reason for this is that global variables break the conventional rules of access and decoupling of variables and methods in object-oriented programming. In standard object-oriented programming, classes and objects have exclusive ownership over a variable or method, and may determine if that value should be exposed to external code or not. The latter property is called



The text on the right indicate the new features added by each derived class.

Figure 6.4: The hierarchical inheritance of some of COCOASIM's classes.

access specification, and is implemented in C++ through the keywords of *public*, *protected*, and *private*. Though a public value may be manipulated by external code, the value is still exclusive for the class/object. For a global value however, the value can be changed by any code at any time. In other words, every single file has the ability to access a variable even though the variable only should be accessed by a few classes. This can quickly lead to messy code and hard-to-find bugs – especially for larger projects.

The other advantage to this approach is that it is possible to have multiple "global" objects. This allows the simulator to instantiate and manipulate distinct variables and still maintain the illusion that these are globally accessible. Simulator objects in one system will point to *their* global version of a system, while other objects belonging to another system will instead reference that one. This also clearly separates the systems – i.e., objects of system A has access to other objects in system A, but not those of system B. Though the system, a simulator may also access other "global" non-system objects affiliated with that system. The collection of system-specific shared data is handled by an *environment* that can be accessed through the system. As shown in Figure 6.5, contains three objects: the system, the logger, and the event engine.

While the three components are described earlier, they can be summarized as the following:

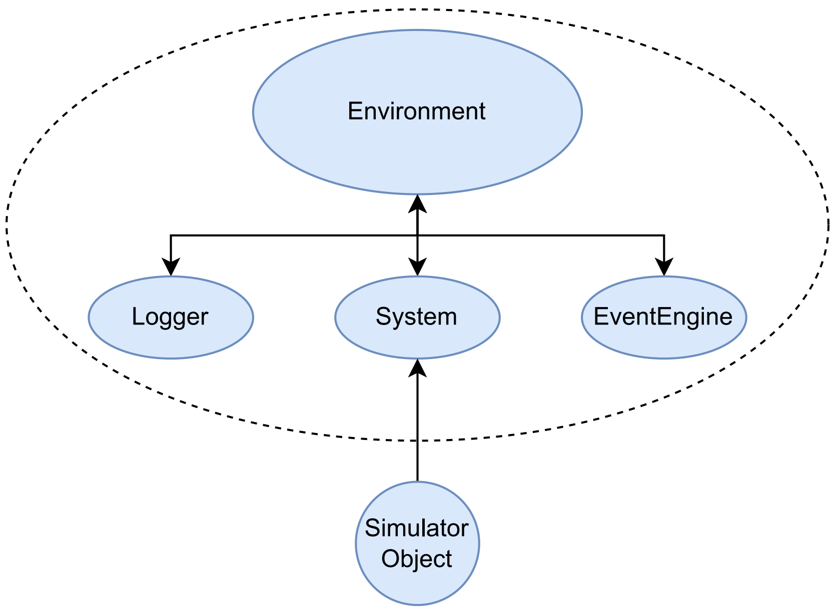


Figure 6.5: The design and content of an environment.

- **System** – A class/object representing the system to be simulated. May be defined either in C++ or through the use of a JSON configuration file as discussed in Section 5.3.4.
- **Logger** – The logger used by the system and all objects. The logger itself is not modifiable, but relies on the event engine that changes state during the simulation.
- **Event Engine** – The event engine responsible for handling events in the system.

Every environment knows of its members – i.e., logger, system, and event engine – and every member knows of its environment. This makes it possible for a simulator object to indirectly access the logger or event engine. In this way, the simulator may access all three of the shared objects to cause different behavior:

- **System** – All simulator objects may directly access the system containing them, but do rarely manipulate the system itself. However, it is possible for any object to access the L1 caches this way by casting the system to a `CacheSystem`.
- **Logger** – Simulator objects wanting to log anything must access the shared `DefaultLogger` object.
- **EventEngine** – Objects may create and schedule new events by accessing the event engine.

6.1.5 MemorySignals

So far, this document has called the objects read from the protobuf traces for *requests*, and explained how these are forwarded to the cache hierarchy. In practice, this is achieved by translating the read data into a special class/object called a *memory signal*. In simple terms, a memory signal represent any signal or packet that is sent between two components – including requests to a cache. In very general terms, the logic flow of any given simulation goes like this:

1. A request is read from the protobuf trace and translated into a memory signal.
2. The simulator creates an event that tells the memory signal to access a memory – either a cache or the main memory.
3. The event is fired, and the memory signal checks if it hits in the target memory. If it misses, step 2 is repeated.
4. When the memory signal hits, the simulator creates an event to return to the previous cache and move towards the `DataOutput`.
5. Step 4 is repeated until the memory signal reaches the `DataOutput` and is terminated – implying a completed operation.

When reading from the protobuf traces, every memory signal created is either a read or write request that is scheduled through a special `MemoryAccessEvent` for one of the L1s. However, memory signals are also used to represent special messages sent from one cache to another, and write-backs to the lower levels of memory. While most memory signals are used in events that move a signal from one place to another, entries in the Miss Status Holding Register (MSHR) or stalled requests are also represented by memory signals.

An overview listing what is read and what is used in a memory signal is presented in Table 6.1. As seen, not all of the available fields in the protobuf trace is used, and the memory signal adds the data- and two flag fields. The reasons for not including some of fields available in the protobuf trace are listed below:

- **cycle** – All traces used for testing COCOASIM begins their memory operation in cycle 0. Thus, the simulator reads as many requests as feasible at cycle 0, and schedules these to access the L1 caches in the next few cycles.
- **mem_type** – The memory type determines the type of coherency the mapped memory type has, and if the request is cacheable. This is not used as the simulator assumes that all requests are cacheable and uses the coherency pattern as it is.
- **asid** – Address space identifier is not used for any trace. Since mem_type was dropped, this is not used either.
- **vaddr** – All provided traces use the same physicall address and virtual address. To simplify the cache behavior, the physical address is used as the *de facto* address.

Data	pbtrace	Memory Signal
cycle	✓	✗
op_id	✓	✓
group_id	✓	✓
op_type	✓	✓
mem_type	✓	✗
paddr	✓	✓
asid	✓	✗
vaddr	✓	✗
msg_type	✓	✗
byte_en	✓	✗
reason	✓	✗
alloc	✓	✗
dep_id	✓	✓
flag	✓	✗
data	✗	✓
subtask_flag	✗	✓
return_flag	✗	✓

Table 6.1: Comparison between the data read versus the data used when reading from protobuf.

- **msg_type** – As mentioned, msg_type determines what type the memory operation is. As op_type already defines what operation is being executed – i.e., read/write/etc – this is only used when separating operations from the same type from each other. None of the provided traces do this. Though future traces may use this feature, it is outside of the project's scope and thus not implemented.
- **byte_en** – By default, this is set to 18446744073709551615 – i.e., all 64 1-bits enabled. Additionally, writing data to a cache line where only some of the bytes are enabled would lead to the *dirty* flag only applying for some bits – requiring a more advanced design. The simulator assumes that every byte is enabled at all times.
- **reason** – The simulator does not use the reason for an operation for anything, and thus ignores it.
- **alloc** – COCOASIM ignores any hint, and instead assumes that every request should be cached regardless of the cache level.
- **flag** – There are no special flags at the present time, so the simulator does not use this field. Instead, COCOASIM uses some internal flags – like request, info, and subtask – but these are created by the cache states rather than the pbtrace file.

In addition to the selected fields from the trace, a memory signal contains some metadata, a signal type classifier, data, and a few flags. The purpose of the data field is explained in Section 5.3.6, and the type classifier can be briefly summarized as classifying if the signal is a request, acknowledge, or info signal. The signal type and flags are slightly more complex however. In short, there are three types of signal types – *request*, *acknowledge*, and *info* – and two flags of interest – i.e., the *sub-task* flag and the *handle-on-return* flag.

As the name suggests, all operations read from the protobuf trace are *requests*. Note that requests can be both reads and writes, and all attempt to access an address in a cache – leading either to either a hit or miss. When a rad request misses in a cache, it will put itself in the Miss Status Holding Register (MSHR), and create a clone of itself in form of a request that is forwarded to the next level of memory. Meanwhile, the original request waits until the cache receives an *acknowledge* signal. Note that a read request *allocates* a spot in a cache (or hits), while a acknowledge *fetches* the relevant data. On the other hand, a write request first evicts a block, but needs to load updated data as well before writing its content. The reason for this is explained in further detail later in this section. Nevertheless, the write request waits until it receives a read acknowledge before it can write its content.

Whenever a request hits in either a cache or the main memory, it transforms into the *acknowledge* type. Note that whenever a request transforms this way, it *loads* a data payload if it is a read, but *stores* the data if it is a write. Regardless of if it being a read or write, the acknowledge then returns to the previous cache/"core" and tells any waiting request that the operation has been completed. This causes the waiting access to re-perform their operation before returning as an acknowledge themselves. The `DataOutput` representing the "cores" above the L1 caches makes sure that the number of created requests is equal to the number of acknowledges at the end of the simulation to verify that every request has been completed.

In some special cases, the caches/memories communicate internally using *info* signals. These are primarily used to communicate eviction logic in inclusive cache hierarchies, but can also be extended to perform various logic by attaching a `SimulatorEvent` to it. Contrary to requests, info signals do not attempt to "hit" in a cache, but rather tells a cache to perform an internal operation by firing the attached event. This can be seen by comparing the specific functions for each signal type in Listing 21 and Listing 22 respectively.

As seen in Listing 22, the simulator executes the event payload attached to the signal itself. In this way, custom behavior can be added by a developer by adding events rather than doing changes to the cache itself. While this logic realistically should be performed by the cache, delegating this behavior to an event makes it easy to configure what logic should be executed. A developer simply needs to create an event defining what should happen when a signal accesses the cache, and attach this event to an info signal.

```cpp
// Simplified version of Cache::handle_cache_access_request()
// If found_entry is null, no matching address has been found
// implying that the access was a miss
void Cache::handle_cache_access_request(MemorySignal *signal,
                                        CacheRowEntry *found_entry)
{
    // If found, handle on hit logic - else, handle on miss logic
    CacheRowEntry* context_entry =
            found_entry != nullptr
            ? replacement_policy->on_cache_hit(signal, found_entry)
            : replacement_policy->on_cache_miss(signal);

    ...
}
```

Listing 21: A simplified version of the request handling function of Cache.cc.

Moving on to the flags, the sub-task field that determines if a signal is created for the sole purpose of solving a sub-task of another operation. When a signal is spawned by a cache this way, its sub-task flag is set to true. This makes the signal not attempt to return to `DataOutput`, and the signal is also filtered when a cache produces statistics on hits and misses. Sub-task signals are only spawned in a few specific cases:

1. When evicting a dirty block, the cache will begin a write-back to the next level of memory. Thus, the write-back itself will be a regular memory write request marked as a sub-task as to not return to the `DataOutput`.
2. If a write causes an eviction in a cache, the address of that operation must be fetched from lower memory before overwriting the block with the new data. This is required as the write operation may read only specific bytes, and the cache hierarchy must ensure that the data is synced correctly. For example, if address `A` causes address `B` to be evicted, several sub-tasks may be created:
    a. If address `B` is dirty, a sub-task will be created to perform a write-back. If `B` is clean, this step is skipped.
    b. To ensure that the data in the cache is up-to-date with the lower caches, a sub-task read is created for address A *before* the signal writes to the cache.
    c. Lastly, after receiving the read sub-task the original signal can finally perform the write.

```
// Simplified version of Cache::handle_cache_access_info()
void Cache::handle_cache_access_info(MemorySignal *signal)
{
    assert(signal->signal_class == SignalClass::INFO);

    SimulatorEvent* event = signal->signal_behavior_info;
    assert(event != nullptr);

    // Fire custom behavior
    event->fire();
    delete event;
    signal->signal_behavior_info = nullptr;
    ...

}
```

Listing 22: A simplified version of the info handling function of Cache.cc.

3. Info signals are as a general rule always sub-tasks as these are communication messages between two caches. Thus, these are exchanged within the cache hierarchy, and does not attempt to return to the `DataOutput`. A typical example of an info signal is an eviction caused by the inclusive cache policy. When accessing another cache, the sub-task flag tells the cache that this is not a normal request.

The first scenario of the list above is shown in Figure 6.6. Assume that the read signal requests the purple block, and that the orange block is dirty and the target of the next eviction. As seen, this will cause the following two things to happen:

1. The request misses in the L1, and the cache needs to evict one of its members.
2. The orange block is evicted – causing a write back sub-task to be created. The write-back is forwarded to the next cache followed by the propagating read request the next cycle.

The "handle-on-return" flag behaves somewhat similarly to the sub-task flag as it also tells the simulator that a signal has special behavior. This flag is set by default, and only switched off on signals that also are sub-task. In other words, all signals created by protobuf stream readers have the return flag enabled. The reason that this flag is needed in some sub-tasks is to prevent incorrect behavior in components that react on signals that return to a cache. For the most part, this can be simplified to making sure that the MSHR does not react on certain messages. As an example, consider the following scenarios:

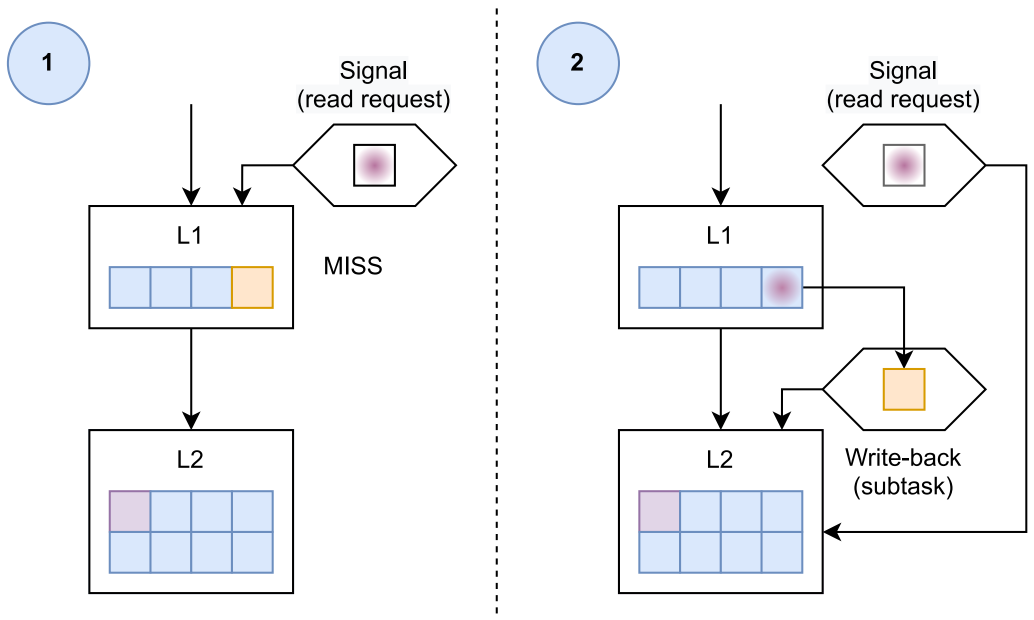


Figure 6.6: The initial read request results in two propagating requests: one write-back sub-task and one forwarded read.

1. A *write* causing an eviction of a *clean* block. This will require stalling the write until the new data is fetched from the lower memory levels by a sub-task read. The sub-task has the "handle-on-return" flag set to true so the write is informed when the read arrives with the updated data.
2. A *read* causing an eviction of a *dirty* block. This will cause a write-back as well as a propagating read to fetch the requested data. While the data request should trigger other signals upon its return, the write-back should not. For consistency's sake, the cache should receive an acknowledge that the write back is successful, but this message should not trigger, e.g., the MSHR or any other component. For all practical purposes, the write back is a "fire-and-forget" sub-task where the cache does not need to wait for its response. Thus, the "handle-on-return" flag is set to false.

An interesting aspect of the simulator design is that all signals of the *info* type is *never* returned after accessing a memory, and thus is a true "fire-and-forget" signal. However, as a write-back is a write for all practical purposes, it is classified as a write request rather than an info signal.

In summary, the "sub-task" flag is added to separate between primary and secondary requests, while the "handle-on-return" is introduced to make exceptions for requests that should be ignored for special reasons. While there may be many other ways to design the simulator and still retain this effect, this is done to prevent enforcing a "type" to a predetermined behavior. As mentioned, info signals are automatically deleted after their logic has been executed, and making a write-back an info signal could have circumvented the need of a "handle-on-return" flag. However, the design decision in this case was to rather add another flag as a write-back in form of a write signal fits much better in the architectural design of COCOASIM.

6.2 Logic Flow

While the previous section focuses on understanding the key concepts of COCOASIM, this section will focus on how the simulator translates the protobuf input into results. This section explains how memory signals move through a cache hierarchy by following it from a trace and a `StreamManager` until it eventually reaches the `DataOutput`. Additionally, this section discusses what happens when a requests accesses a cache and how the replacement policy reacts on hits and misses.

6.2.1 StreamManagers

When initializing a system, the simulator goes through two phases. First, the system is configured – either by the aforementioned "System" mode or "Configuration" mode – through the use of the `setup()` and `load_system()` functions. This readies all of the components that are going to be used during the simulation like caches, the main memory, replacement policies, and the data output. This section also connects the components and verifies that everything is set up correctly. Afterwards, the system is "ignited" by calling the `ignite()` method. While this may sound like a strange choice of words for a phase, it is meant to virtually represent the "ignition" of an engine. While the previous may represent the *construction* of a machine or engine, this phase is meant to represent actually starting the system. This is done by initializing a `MultiStreamReader`, and must be done *after* initializing the system to ensure that every cache is ready.

The `MultiStreamReader` is simply a container class for a dynamic number of `StreamManagers`. The multi-stream reader is created when the system is ignited and creates as many stream managers as there are traces. If the number of traces and caches are not equal, the simulator will log a warning and instead limit the number to whatever number is the lowest. For example, if there are eight traces but only four caches, the multi stream reader will only open four of the traces. This is emphasized as to limit COCOASIM to only use as much resources as necessary, as well as ensuring that every component works correctly. While the multi stream reader encapsulates multiple stream readers, each individual stream manager encapsulates a raw stream. Recall that the pbtrace is *serialized* meaning the requests can be fetched one by one instead of reading the entire file. This is a major advantage as the stream manager can read the items in order instead of reading all of them at once. As mentioned in Section 5.3.3, each trace can instead be paused and continued as needed.

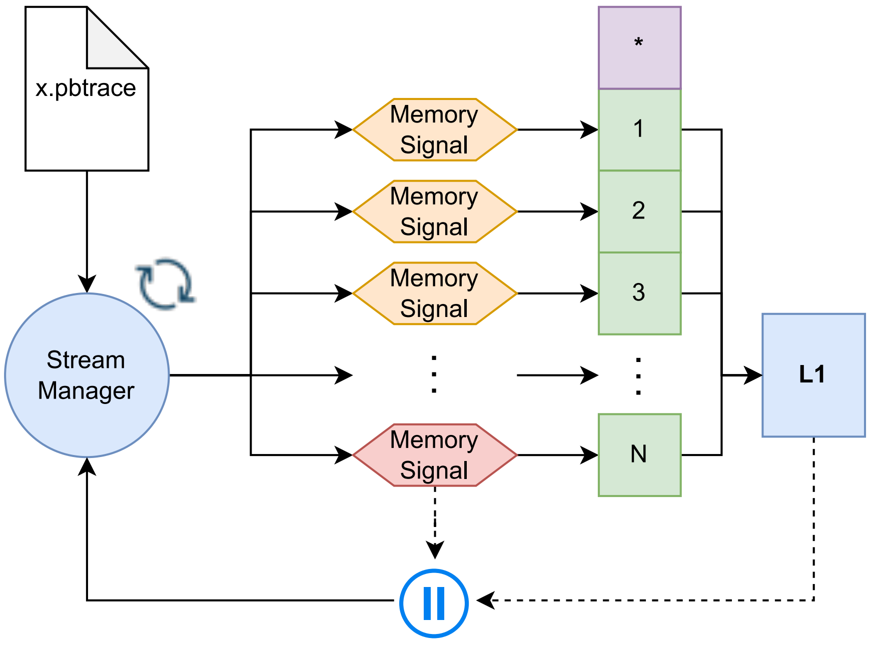


Figure 6.7: The initial stream read event reading requests from the pbtrace until the cache cannot handle more requests.

After the system has been "ignited", the simulator creates multiple `Stream-ReadEvents` (one for every trace/file/cache) that executes immediately – i.e., at cycle 0. Every stream read event performs on a unique pbtrace file, stream manager, and cache. The behavior of a single `StreamReadEvent` is shown in Figure 6.7 performing the following behavior:

1. First, the stream read event checks if the entire stream has been read. If all requests has been read, the stream read is completed and nothing else needs to be done. While it is theoretically possible to exhaust an entire pbtrace the first time a `StreamReadEvent` is fired, this example will assume that there is a high number of requests in the trace.
2. If there are unread requests, the stream read event reads the first one of these and saves it temporally.
3. The stream read then checks if the target cache is able to handle more requests at the time. There are two reasons that makes a cache not ready to handle new request: either 1) because the cache is stalling, or 2) because it cannot handle more requests of a certain group. The simulator can easily check if the cache is stalling on a given cycle, but the group limit make it possible to handle some requests but not others. Thus, the target cache cannot handle the read request if it cannot handle the group id of that request. This is also the reason that the request is read and temporally saved before determining if the cache can handle more requests or not.

4. If the cache cannot handle more requests, the read item is buffered and the stream read paused. Note that the simulator at this knows that there are no reason to read more requests as they cannot be handled. Thus, COCOASIM conserves resources by instead pausing the read.
5. If the cache is able to handle the request, the simulator creates a memory signal and creates a special `MemoryAccessEvent` – telling the signal to access the cache in n = (delay to L1) + (number of signals created by the stream read event excluding this) cycles. The special access event is a slight variation of a normal memory access event that also tells the cache to reserve the group of the incoming request. Thus, other signals read by the stream read event also need to consider the newly added group when checking if they can access the cache.
6. After the signal has been created and scheduled, the stream read repeats this process by jumping to item 1.

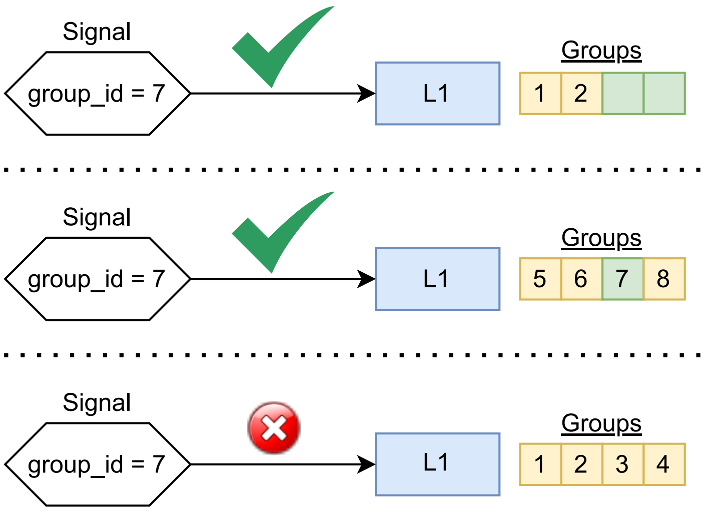


Figure 6.8: Each cache has a configurable number of "groups" that it can handle while still being non-blocking.

After a `StreamReadEvent` has finished – either because no more items exist in the pbtrace file, or as a consequence of the cache not being able to handle more requests – the event will be deleted and disappear. However, the state of the stream – e.g., if any items remain – are kept in each individual stream manager. Note that this also allows the traces to be as unbalanced as desired, and a stream can finish much earlier or later than the others without any problems. While the initial stream read events fire in cycle 0, the simulator continues reading from the traces when it suspects that the L1 cache may accept new memory accesses. There are two scenarios where this happens:

1. The cache goes from a "stalling" or "buffered" state to a "ready" state.
2. The cache frees one of its groups – making it possible for a request of another group to access the cache non-blocking.

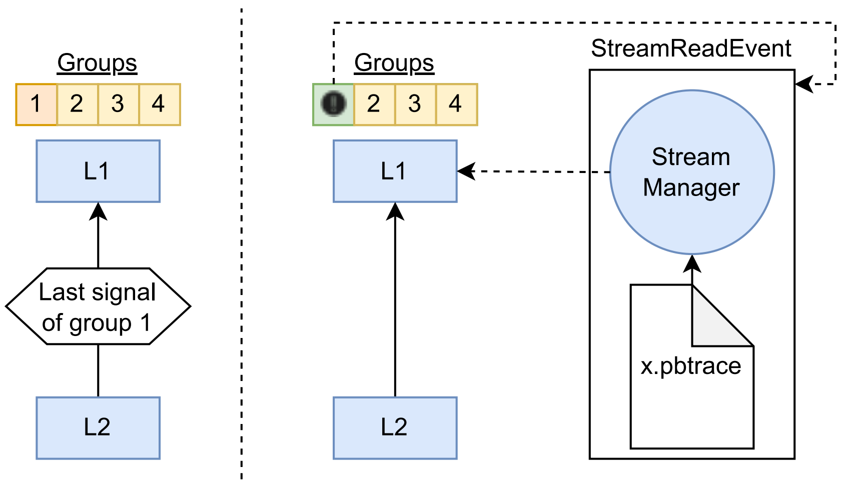


Figure 6.9: An "acknowledge"-signal may free a group – leading to a new `StreamReadEvent`.

The latter scenario is shown in Figure 6.9. When a group is freed, the next request of the protobuf trace may suddenly be able to access the L1 cache. For example, if the next request has a group id of 7, this group can now be allocated. Note that it is not guaranteed for a freed group to cause a successful stream read as the cache might be in the "buffered" state as it handles stalled requests that have been buffered. In this scenario however, the cache will retry firing another `StreamReadEvent` when it finishes up handling all buffered signals and entering the "ready" state.

As mentioned, the main motivation for implementing stream readers and stream managers is to reduce the memory resource usage to the minimum. As explained, the simulator will only read requests that it knows can be handled at the time into working memory. In the meantime, the stream is paused – which is possible thanks to the serialized nature of the input traces.

6.2.2 Caches

The main component of every simulation in COCOASIM is the cache. Caches are simulator objects, but also inherit from the `Memory` class – allowing them to be accessed my memory requests. The `Cache` class is the largest file of the COCOASIM source, and is even split into three smaller abstractions to simplify the use and development of cache objects. The three sub-components are:

1. **CacheInternals** – A class managing the content of a cache. The simulator separates the content from the interface and other logic of a cache in an attempt to reduce complexity.
2. **CacheSets** – The cache internals contain a number of sets that divide the cache into segments. More on cache sets and associativity is discussed in Section 3.1.2.

3. **CacheRowEntries** – A cache set contains a number of "slots" often called cache lines. Cache lines are represented by the `CacheRowEntry` class.

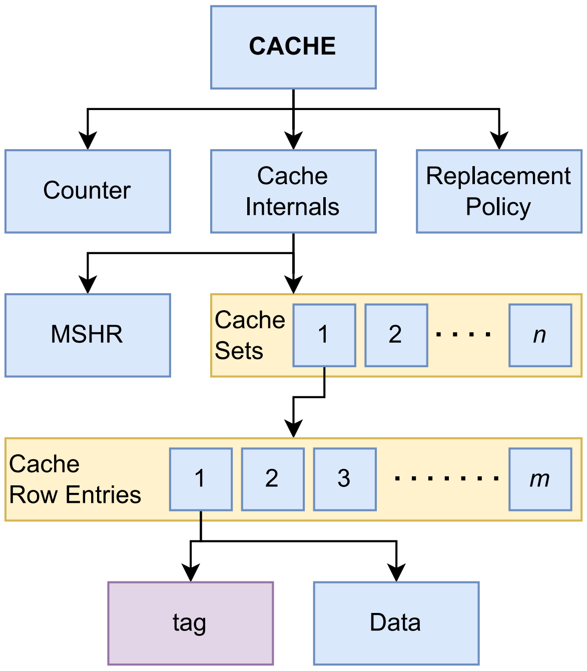


Figure 6.10: The content of a cache and its members.

The complete structure of a cache can be seen in Figure 6.10. Note that this overview includes all objects in a cache, but not values like associativity or memory latency – with one exception: the tag of a cache line. Additionally, only the objects *owned* by the cache (and its members) are shown. In other words, the data output is not shown as the cache does not own this object despite having a reference pointer to it. Each component is described in Table 6.2.

Component	# per cache	Function
Cache Internals	1	Manage cache content
Counter	1	Track statistics
Replacement Policy	1	Perform replacement logic
MSHR	1	Manage missed requests
Cache Set	n	Represents a cache set
Cache Row Entry	$n \times m$	Represents a cache line
tag	$n \times m$	Represents the cache tag
Data	$n \times m$	Manage data of this cache line

Table 6.2: The content of a cache and what they are used for.

Memory requests are often read in large bulks by a `StreamReadEvent` before the requests are sent in sequence to a cache. As mentioned in the previous section, the stream reader will only send a memory request to a L1 if it thinks that the cache can handle more requests. However, due to a delay between issuing the memory requests and the request actually accessing the cache, the stream reader may issue requests that cannot be handle. This should never happen because of the group limit (as this is handled before the request arrives), but may happen for other reasons. While there are no formal cache states in the COCOASIM source, caches can fit into one of three categories:

- **Ready** – The cache can handle new requests.
- **Buffered** – The cache *cannot* handle new requests at this time, but might be ready after finishing a number of buffered memory request.
- **Stalling** – The cache *cannot* handle new requests, and needs to wait for a signal outside of the current cache to continue.

Buffering of requests may happen in any cache, but are especially common in L2s. When multiple L1s tries to access the L2 in the same cycle, the L2 must handle one of them and place the rest of the requests in a buffer. Given eight requests, the L2 cache spends a total of eight cycles executing all of them. Thus, one of the request are handled instantly, while the rest are scheduled (FIFO) for the few next cycles. However, if everything goes without problems, the cache is ready to handle new requests after the eight cycles have passed. This behavior is summarized in Figure 6.11.

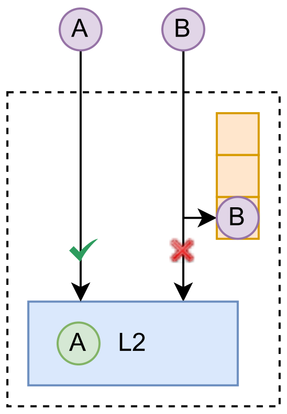


Figure 6.11: The cache buffers additional requests if it is already handling one.

A stalling cache is the result of a buffered request not being allowed to perform a desired operation, and thus must stall the pipeline. Note that it *is* possible to send more requests to the cache in this state, but the requests will not have any effect and are put at the end of the FIFO queue. The stream readers will recognize this, and will stop reading and creating memory signals until the operation can execute. There are two scenarios that might cause a stalling cache:

1. A request misses in a cache, but is unable to evict any entries. This happens when a cache line block is allocated by another read request, but not yet fetched from lower memory. As the cache cannot ignore the eviction and allocation using the new tag, the request is instead stalled until it is possible to evict a cache line – i.e., when a returning read acknowledge carrying data changes the state of the cache line from "allocated" to "ready". This behavior is more common for systems with small caches, low associativity, and long memory latencies.
2. A request misses in a cache, but the Miss Status Holding Register (MSHR) is already full. As with the previous scenario, the operation must wait until a returning acknowledge frees up space in the MSHR. This is more common for caches with small MSHRs.

Note that the only component that reacts to the "state" of a cache is the stream reader. If a L1 attempts to send a request to a non-ready L2 cache, the request will instead be stopped as it tries to access the cache. On the other hand, the stream reader will not send request to its respective L1 cache if it isn't ready. This approach has the advantages of: 1) limiting the amount of resources used, and 2) being relatively simple as requests are automatically stalled if needed. However, one disadvantage with this is that the simulator assumes that every cache is able to buffer an infinite number of requests. This either assumes that each cache comes with a large hardware buffer – which is improbable – or that the interconnects connecting the caches are pipelined – which is more plausible. However, the latter alternative should only allow for a number of requests equal to the latency of each cache. To preserve the simplicity of the buffering design, this is ignored by the simulator.

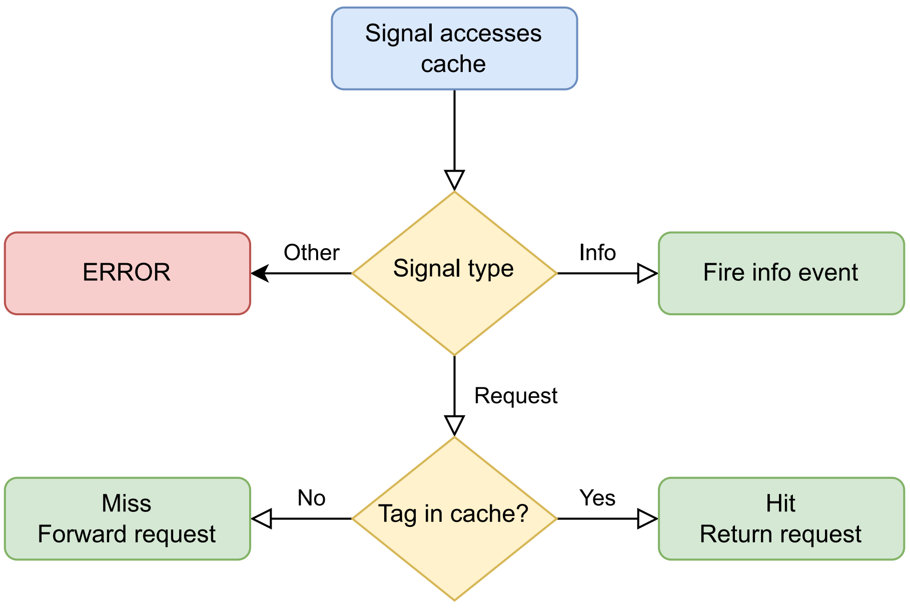
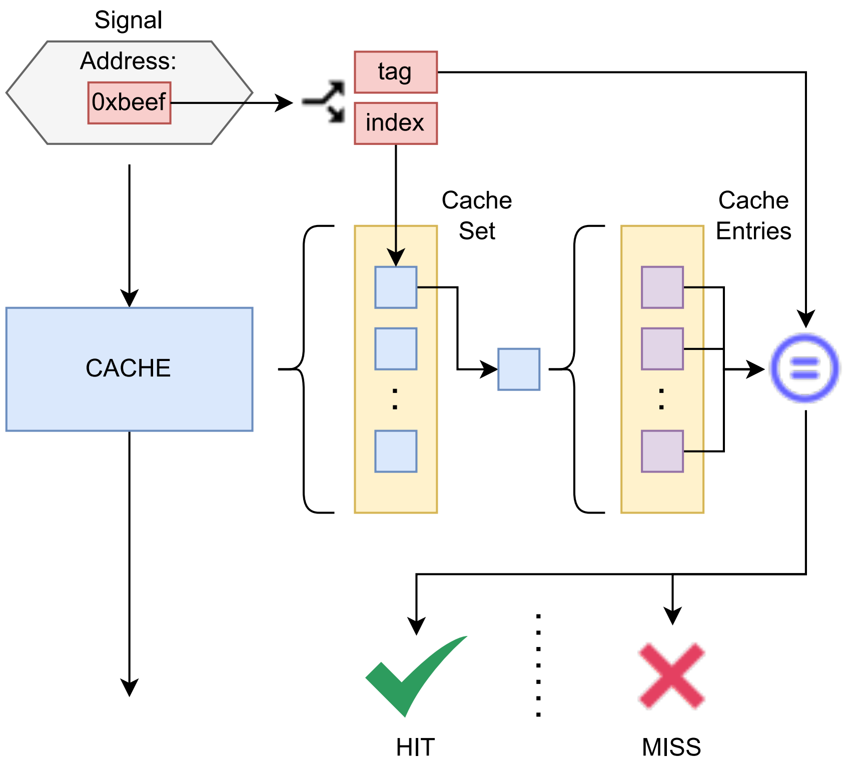


Figure 6.12: The logic flow performed when a signal accesses a cache.

Signals that successfully access the cache will follow the behavior shown in Figure 6.12. Note that "acknowledge" signals should never access the cache this way, but rather through another interface. As shown, requests may hit or miss, while info signals are handled and then simply discarded. To check if a request hits or misses, the cache uses the address of the signal and translates it to a tag, index, and block offset using the associativity/sets and cache line size of the cache. The cache then fetches the correct set using the index, and compares the content of every cache line of that set with the new tag. If one of the cache lines contain the tag, the request is a hit and the respective data can be returned. If not, the request is a miss. This behavior is visualized in Figure 6.13.



Cache is used to describe both the actual "cache" object as well as the "cache internals" object.

Figure 6.13: How a cache checks if an address exist in the cache.

6.2.3 Replacement Policies

When a cache checks if a request hits or misses, it goes through the following three steps:

1. First, the cache checks if the address already exists in the cache by performing the logic shown in Figure 6.13.

2. If the request misses, the cache needs to fetch the data from lower memory. Thus, the cache first checks if it can replace any of the invalid entries with the new address. Invalid cache lines often exist in the beginning of a simulation when the cache set is either partly or entirely "empty", but may also appear later as a consequence of a line being invalidated by another cache through the inclusive cache policy.
3. If no invalid entries exist, the cache needs to replace one of the existing entries with the new address. This is done by listing all the replaceable cache lines in the current set and telling the replacement policy to choose one of them to evict.

While the two first steps are performed automatically by the cache, the decision of what entry to replace is delegated to the cache replacement policy. Note that the cache will only nominate entries that are possible to evict, as invalid entries of entries that are allocated but not fetched cannot be evicted from the cache. Thus, it is possible to encounter scenarios where *none* of the cache lines can be evicted – causing the cache to stall as described in Section 6.2.2.

As seen in Figure 6.10, the cache objects are designed in a way to separate the replacement policy from the cache itself. Thus, a cache can use any type of replacement policy as long as it contains an interface that the cache understands. In practice, this means that any sub-class of the `ReplacementPolicy` class can be used.

The base `ReplacementPolicy` class can be seen in Listing 23. Note that the class contains multiple methods with the `virtual` keyword – hinting that these can and should be overridden in a sub-class. Also note the `on_cache_hit()` and `on_cache_miss()` functions, and recall that these are called in Listing 21 when a cache access either hits or misses respectively. The main idea here is that every cache replacement policy may perform their own logic on cache misses (and hits). The methods of `CacheReplacementPolicy` functions in the following way:

- **do_get_system()** – Returns the system that this replacement policy exist in, and should not be overridden. Note that `CacheReplacementPolicy` derives from `SimulatorObject` which enforces it to have a method for referencing its appropriate system, and is discussed further in Section 6.1.3. While this is not something that sub-classes of `CacheReplacementPolicy` need to consider, it is still worth noting as to understand how a replacement policy knows what system it inhabits.
- **do_create_new_instance()** – Used for creating another replacement policy of the same type as this. A sub-class needs to override this to return another instance of itself as the method is made pure virtual. This method is primarily used when a simulation in "Configuration" mode needs to spawn instances of a replacement policy present in a replacement policy catalog.

```
// Condensed version of CacheReplacementPolicy.hh
class CacheReplacementPolicy : public SimulatorObject {
public:
    CacheReplacementPolicy();
    virtual ~CacheReplacementPolicy();
    ...

protected:
    BaseSystem * do_get_system() const override;

    virtual CacheReplacementPolicy* do_create_new_instance() const = 0;
    virtual CacheRowEntry* handle_replacement(MemorySignal* signal) = 0;
    virtual CacheRowEntry* handle_on_cache_hit(MemorySignal* signal,
                                               CacheRowEntry* hit);
    virtual CacheRowEntry* handle_on_cache_miss(MemorySignal* signal);
    virtual CacheRowEntry* handle_find_and_fill_unused_cache_slot(
                           MemorySignal* signal);
    virtual void do_connect_to_internals(CacheInternals* internals_to_manage);
    virtual CacheRowEntryFlags* do_create_required_flags();

};
```

Listing 23: The CacheReplacementPolicy header – featuring the many virtual functions.

- **handle_replacement()** – Called when the cache needs the replacement policy to make an eviction, and differs for every cache replacement policy. A random replacement policy should simply choose one of the replaceable entries at random, while a LRU policy should find the entry within a set with the lowest counter value. The function returns the appropriate candidate for replacement, or alternatively a null value of no entries can be replaced. This is arguably the most important method for replacement policies, and the rest of the functions should only be overridden if needed.
- **handle_on_cache_hit()** – As seen in Listing 21, this is called when a request hits. By default, this method will execute logic like writing or reading to the cache line. However, it is also possible to perform additional logic when this method overridden. The best example of this is the LRU policy which also increments the access counter on the cache line that experienced a hit. Note that this method can be *expanded* upon, but must not be *replaced* by other code.

- **handle_on_cache_miss()** – As with `handle_on_cache_hit()`, this is also called by Listing 21 but instead for cache misses. By default, this performs the logic associated with cache misses like allocation of cache lines, adding entries to the MSHR, and forwarding requests to the next level of memory. It is also possible to expand on this with additional logic, but no existing replacement policy does this.
- **handle_find_and_fill_unused_cache_slot()** – This is called by the `handle_on_cache_miss()` function *before* an eviction to check if the cache has unused/invalid cache lines. If it does, the cache does not need to evict any cache line, and can use the empty slot instead. However, in the vast majority of cases this method will return a null value as no entries can be replaced. Nevertheless, this can also be expanded on by a sub-class if desired. The LRU actually does this by updating the access counter on cache lines that are successfully filled by the base function.
- **do_connect_to_internals()** – Connects the replacement policy to the contents of the cache. Called once when initializing the replacement policy, and may be overridden to contain additional logic.
- **do_create_required_flags()** – Whenever a cache is initialized, it will ask the replacement policy if it needs any special flags for the cache lines. By default, this will return two flags: the *valid* flag, and the *dirty* flag. However, more values – not only 1-bit flags – can be added if needed. For example, the LRU policy will tell the cache that it also needs a field for an access counter for it to perform its replacement logic. An example of how this works is shown in Figure 6.14.

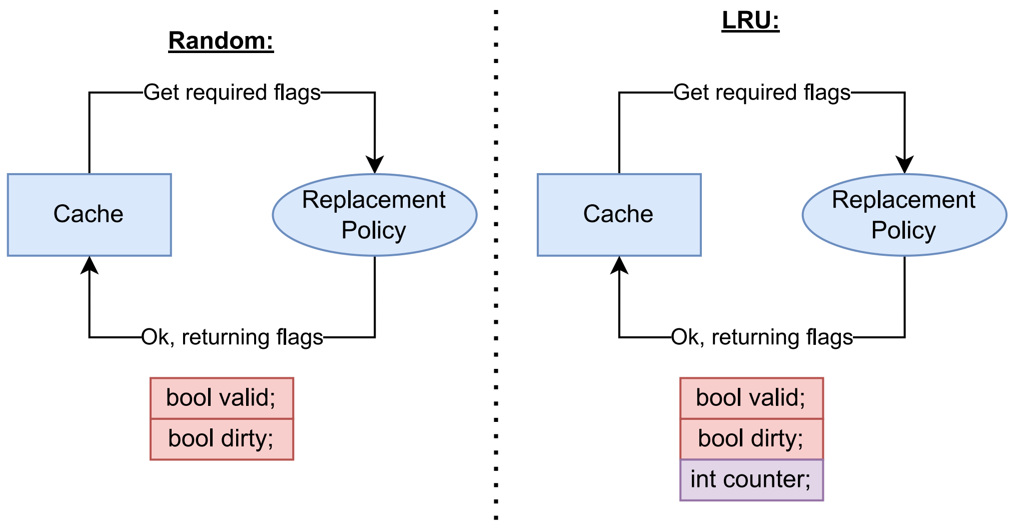


Figure 6.14: The cache queries the replacement policy what flags it should use.

In summary, cache replacement policies are designed to be highly configurable and easy to expand upon. By using the base class of `CacheReplacementPolicy`, the simulator knows what behavior to do on hits and misses. While there are several methods that can be overridden and expanded, all a new replacement policy really needs is an algorithm in place of the `handle_replacement()` method. Here, the algorithm for deciding what to evict can be as simple or complex as needed. As the cache and the replacement policy are clearly separated, any replacement policy should be compatible with any cache – making experimenting with different policies simple.

6.2.4 DataOutput

After hitting in a cache or the main memory, a memory signal will eventually return to the `DataOutput` In the same way as every cache has a reference to the memory *below* it, every memory also know of every "signal interactive" object *above it*. Note the distinguishing between a memory and a "signal interactive" object: every memory is a signal interactive object, but the reverse is not true. As explained in Section 6.1.3 and Figure 6.4, both of these inherit from the `SimulatorObject` class, but the `SignalInteractive` class is closer to the `SimulatorObject` root than the `Memory` class. This means that while caches and other memories have an access method and set size, the data output does not. However, all acknowledge signals can *return* to "signal interactive" objects – including the data output. As the data output represents the output/CPU/core in a simulated system, memory signals returning here are marked as completed and deleted. Additionally, the data output counts the returning signals as they arrive to make sure that all signals have been received at the end of a simulation.

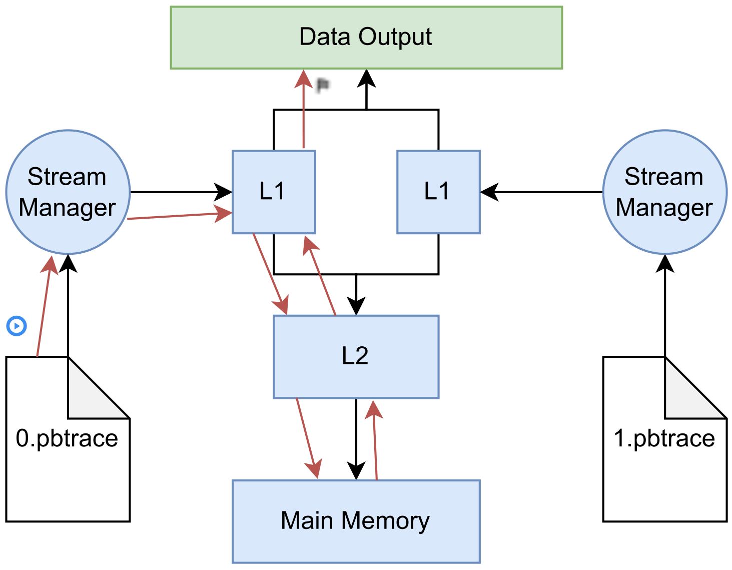


Figure 6.15: A request's journey through the cache hierarchy

The typical behavior of an early request traveling through the cache hierarchy is shown in Figure 6.15. Note that while the example signal travel to and from every memory, it does only travel *from* the stream manager, and only *to* the data output. Whenever a request misses in a cache, the simulator creates a new request that is told to access the next level of memory, and return to the current cache when acknowledging. However, when a stream manager creates a request, the simulator tells it to access one of the L1 caches, but return to the data output instead when acknowledging. This is possible as each signal knows what it is supposed to access – i.e., the *destination* – and where it should return to when finished – i.e., the *source*.

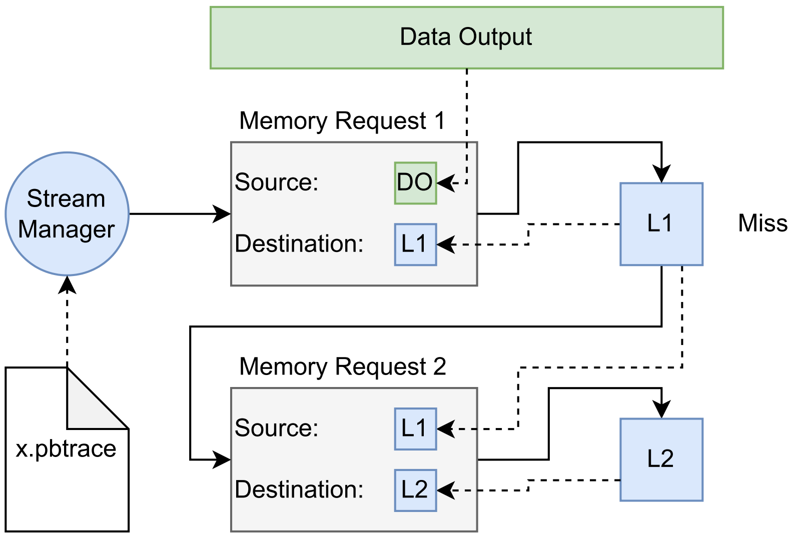


Figure 6.16: The source and destination of two memory requests.

The concept of a signal's "*source*" and "*destination*" is shown in Figure 6.16. Note that the source of request 1 is set to the data output despite being spawned by the stream reader, and that the L1 creates a new request with itself as the source. This design method make it easy to keep track of the state of each request. If the initial request had hit in the L1 cache, it would instead returned directly to the data output. However, since the signal missed it does not get to return to the output before the new request has completed. If request 2 misses in the L2, this behavior is repeated – with the L2 creating a new signal bound for the L3 / Main Memory and request 2 waiting for an acknowledge. Note that this also leads to a signal only existing between two components, e.g., the L1 and L2 or the L1 and the data output. This allows a source to safely delete any signal that return to it while returning the original waiting signal. This makes it easy and to orderly manage signals and states as the simulation continues, but comes at the slight cost of increased memory usage as some redundant metadata would have to be reused across all signals. A previous iteration of COCOASIM attempted to attach all metadata to a single signal as it traveled through the cache hierarchy, but this proved to be complex to implement and quite messy to work with.

In short, the data output is responsible for the following:

1. Registering signals returning from the L1 and marking them as finished.
2. Managing dependencies between requests, and firing new memory access events when a returning signal makes another waiting request ready.
3. Continuing stream reads if caches changes state to "ready" from "buffered" or "stalling".

Note that contrary to the stream readers, there is always only a single data output. This means that every signal returning from a L1 cache will arrive at the same data output. However, the data output is able to distinguish between what cache each signal comes from – making it possible to choose the correct stream reader when continuing stream reads.

Chapter 7

Validation & Results

While the previous chapters describes the theory and basis of COCOASIM, this chapter will focus on the practical functionality of the simulator. First, Section 7.1 will discuss how testing is used to validate that the simulator works as intended. The next section – Section 7.2 – will show how COCOASIM can be used in practice by presenting an example on how a relatively simple custom variation of the LRU replacement policy can be created. Lastly, the results of the simulator as well as the replacement policy variant will be presented in Section 7.3.

7.1 Testing

7.1.1 Tests

COCOASIM contains multiple tests to validate that the simulator behaves as expected in different scenarios. Contrary to the simulator itself, all tests are written in Python. It should be noted that COCOASIM does not perform *unit testing*, but rather something closer to *integration testing*. In summary, this means that the individual parts – like caches and replacement policies – of COCOASIM is not tested by the testing interface. Instead, the tests run specific simulations featuring different systems and traces and validate that the simulations are successful. As mentioned in the earlier chapters, the source code of COCOASIM contains multiple assertions that continuously check that the simulation is in a valid state. If any of these assertions fails, the testing interface is notified and returns a failure. A test is only a success if every single assertion across all possible configurations is a success.

Tests in COCOASIM can be put in one of two categories:

1. **System Tests** – Used for testing that configured systems behave correctly. Normally tests each system on every available trace. Great for testing that new systems work as intended, and that new changes to COCOASIM don't break any of the existing successful assertions.

2. **Configuration Tests** – Primarily used to validate that configurations / run-time arguments work as expected. Usually only uses a single trace with different configurations

In summary, system tests perform the heavy integration testing while the configuration tests validate that certain configurations work as intended. The simulator currently uses four system tests – each testing a system on all available traces – and a single configuration test. All tests can be seen in Table 7.1.

Tests	Description	Typical duration*
Advanced System Test	Test the "bulldozer-inspired system" configuration	47m 1s
LRU+ Test	Tests a system using the "LRU+" policy	43m 26s
Scaled System Test	Tests a simple system with 8 L1 caches connected to a main memory	43m 52s
Single L1 System Test	Tests a very simple system with a single L1	6m 58s
Configration Test	Tests that "System Mode" and "Configuration Mode" works as intended	57s

*: The duration varies for every simulation. The duration for every test used here is from the automatic testing configuration of GitHub's "Build and Test #35". This is discussed futher in Section 7.1.4

Table 7.1: The tests run when running the integration test suite.

The complete structure of the "testing" directory the simulator uses for testing is shown in Figure 7.1. Note the following:

- Before any testing can be performed, the `test_setup.py` file must first be executed. The setup file will essentially only prepare the traces by copying from the "traces" directory and decompressing them in a temporary directory that is deleted after the tests have finished. The simulator does not want to upload the files to a version control due to their large sizes, but still needs the traces to perform integration testing. The simulator solves this by retaining a permanent compressed version that is uncompressed by the setup file when needed. To quickly prepare the simulator for testing, it is recommended to use the `make test-setup` command provided by the Makefile.
- The testing itself is managed by the various test runners. Executing the test runner at the root will simply run both of the other test runners located in `configuration_tests` and `integrity_tests` respectively. Both of these will run every test that is present in their respective "tests" directories. Due

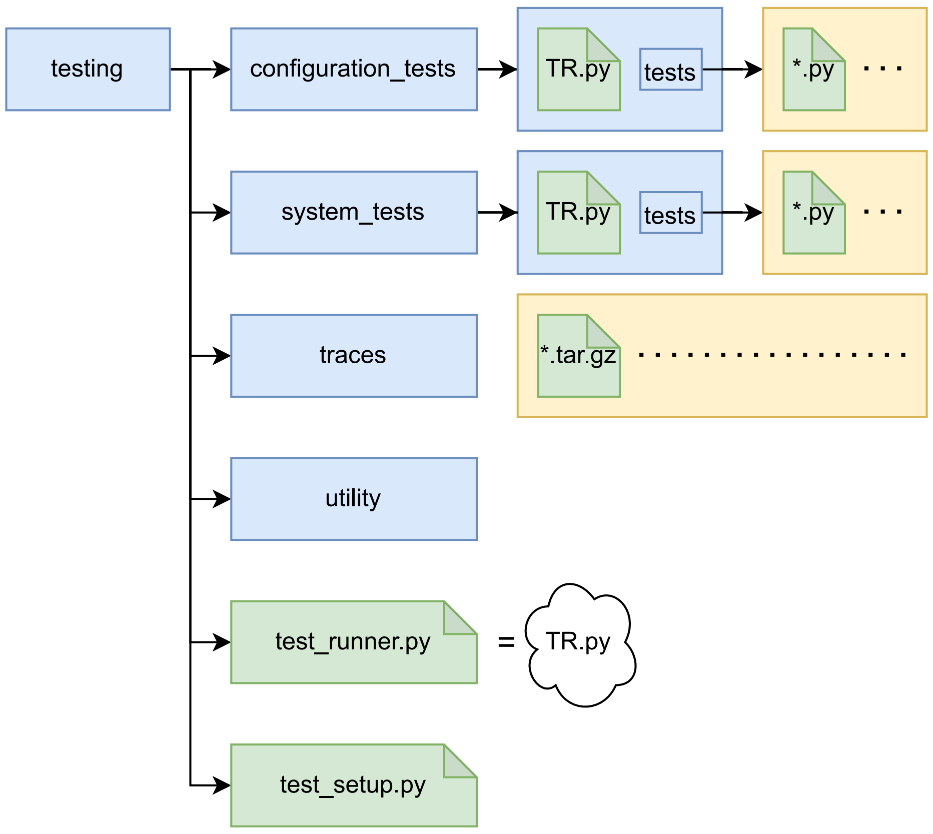


Figure 7.1: The content of COCOASIM's testing directory.

to the long execution time of its tests, the system test runner also has the option of specifying a single test to perform. To quickly test every single test, simply running the `make test` command is adequate. This should also run the setup file automatically and prepare the required traces.

- To add a new test, simply adding it to either "testing" directory should be enough. The test runners should automatically recognize new tests as long as they end with the `".py"` post-fix.
- Likewise, new (compressed) traces can be added to the "traces" directory and automatically be recognized. System tests will automatically test each system on every trace collection in the "traces" directory.
- While not explicitly shown, the "utility" directory contains helper functionality for the tests. This is useful for reusing functions or classes across tests. This currently contains some simple logic for traversing file paths, and is used by the tests to choose the correct trace directories.

7.1.2 Traces

While COCOASIM is able to use any input traces in the correct format, the simulator contains a total of 21 bundled trace collections in the "traces" directory. As mentioned, these are compressed but can be used like any other trace directory once decompressed. All of these traces were generously provided by Ole Henrik Jahren from Arm, Trondheim, and is used extensively for the system tests as well as regular experimenting. The trace collections are summarized in Table 7.2.

Collection	Operations per trace	Total operations
8c-64eg-256KiB-memcpy	1 024	8 192
8c-64eg-2MiB-memcpy	8 192	65 536
8c-64eg-64MiB-memcpy	262 144	2 097 152
memcpy-16KiB-x32-8c-64eg	16 384	131 072
memcpy-16KiB-x512-8c-64eg	262 144	2 097 152
memcpy-512KiB-x8-8c-64eg	16 384	131 072
memcpy-512KiB-x128-8c-64eg	262 144	2 097 152
memcpy-4MiB-x32-8c-64eg	16 384	131 072
memcpy-64MiB-x1-8c-64eg	262 144	2 097 152
load-32KiB-x32-8c-64eg	16 384	131 072
load-32KiB-x512-8c-64eg	262 144	2 097 152
load-1MiB-x8-8c-64eg	16 384	131 072
load-1MiB-x128-8c-64eg	262 144	2 097 152
load-8MiB-x1-8c-64eg	16 384	131 072
load-128MiB-x1-8c-64eg	262 144	2 097 152
store-32KiB-x32-8c-64eg	16 384	131 072
store-32KiB-x512-8c-64eg	262 144	2 097 152
store-1MiB-x8-8c-64eg	16 384	131 072
store-1MiB-x128-8c-64eg	262 144	2 097 152
store-8MiB-x1-8c-64eg	16 384	131 072
store-128MiB-x1-8c-64eg	262 144	2 097 152

Table 7.2: The traces used by COCOASIM

The traces of Table 7.2 – with the exception of the first three – are formatted in the same way – i.e., **<operation>-<memory-size>-<repeats>-<number-of-traces>c-<executing-groups>eg**. These fields can be explained as the following:

- **operation** – What kind of operation is performed – i.e., load, store, or memory copy.
- **memory size** – How large memory area is manipulated.
- **repeats** – How may times the operation is repeated in total. A *x1* means the operation only happens once, while *x512* means the operation happens a total of 512 times.
- **number-of-traces** – How many L1s are needed to execute the entire trace collection. Set to 8 for all traces.

- **execution-groups** – How many groups are needed to run the trace collection non-blocking. Set to 64 for all traces.

The format of the first three traces is reversed. Though not included in the name, these do only perform the operation a single time. Lastly, an observant reader might notice that the memcpy operations use half the memory area as some of the loads and stores, but still contain the same number of total operations. The reason for this is that memcpy performs both a read and a write for every address whereas loads and stores only perform one of them.

7.1.3 Test Design

There are no strict rules on how to create new tests in COCOASIM, but the test must: 1) be written in Python and have a `.py` file post-fix, and 2) exit with an error code if the tested simulation fails. It is also recommended to write the behavior or the test to standard output, and keep track of how long time the simulation uses. Virtually all tests should use the `cocoasim-fast` binary when performing a simulation. As mentioned, the `cocoasim-opt` ignores every assertion and is consequently an ill-suited choice for testing. It is possible to use the base `cocoasim` binary, but this will in practice only be a slower version of `cocoasim-fast`.

A simplified version of the test responsible for testing the advanced "bulldozer" system can be seen in Listing 24. In short, the test performs the following logic:

1. Get the next collection of traces available.
2. Start a timer.
3. Begin a simulation with the "bulldozer" configuration using the fetched collection of traces. Use the `cocoasim-fast` binary for increased performance. Note that setting the `"-l 0"`-option – i.e., setting logging to 0 – here has no practical effect as `cocoasim-fast` is sued, but is done for the sake of consistency.
4. After the simulation has ended, stop the clock and check if the simulation was successful.
5. If the simulation succeeded, print "OK" and the time used. If not, exit with an error code.
6. If there are more traces left, repeat this process by returning to step 1.

7.1.4 Continuous Integration

As briefly mentioned in the caption of Table 7.1, the "Github Actions" [23] interface have been used during the development of COCOASIM to continuously test that the simulator is behaving as expected. In summary, the testing interface automatically builds and tests COCOASIM every time a new major feature is added. In practice, the GitHub Action job simply runs the the `test_runner.py` scripts in the "system test" and "configuration test" directories with different arguments. This could in theory be done manually as well without the need of continuous testing interface, but the automation has especially two major advantages: 1) Testing is

```python
def main():
    main_dir = util.get_cache_sim_directory()
    temp_test_trace_dir = util.get_tmp_test_traces_directory()

    ...
    sub_dirs = temp_test_trace_dir.glob("**/*/")

    for sub_dir in sub_dirs:
        if sub_dir.is_dir():
            test_name = "verify_advanced_" +
            str(sub_dir).replace(str(temp_test_trace_dir), "")[1:]
            test_dir_name = main_dir / test_name
            print("Testing " + test_name, end=".....", flush=True)

            os.chdir(main_dir)
            start = timeit.default_timer()
            run = subprocess.run(
                ["./cocoasim-fast", "-l", "0", "-i", str(sub_dir),
                "-c", "configs/bulldozer_inspired.json"],
                encoding="utf-8",
                stdout=subprocess.PIPE,
                stderr=subprocess.PIPE
            )
            end = timeit.default_timer()

            try:
                run.check_returncode()
            except subprocess.CalledProcessError as e:
                print(e)
                exit(e.returncode)

            print("OK: " + str(end-start) + "s")
```

Listing 24: A simplified version of verify-advanced-system.py

done without any user intervention – making it easy to "fire-and-forget" changes and thus eliminating the need for waiting for the test to complete to complete, and 2) Testing is done automatically – thus making it impossible to forget to test features.

The continuous integration of the simulator is handled by the `build_and_test_job.yml` file located in the COCOASIM source files. While this can be configured in many different ways, the current version of COCOASIM tells GitHub's servers to perform a build and test suite whenever one of two things happens:

1. Code is *pushed* to the remote `main` branch – indicating that the simulator has received an update. The `main` branch should never contain faulty code, so any errors detected by testing here should be handled immediately – either by fixing whatever caused the bug or through reverting the simulator back to a non-faulty version. Automated testing for changes to the `main` branch is highly desirable as the continuous integration can test if any new changes break the simulator.
2. Code is *requested* to be pushed to the `main` branch through a "pull request". This is a safe way of adding new features to the simulator as the code is tested before being part of the `main` branch. Fixing errors detected by automated testing on a pull request is not urgent as the code is not yet a part of the main branch, but a pull request can only be approved if the automated testing show no errors.

During an automated test, the server starts by building the binaries of COCOASIM. If this completes without errors, the GitHub server continues by using the binaries to run multiple tests in parallel. This is another major advantage by using automated continuous integration as each tests can be executed simultaneously in separate environment. As shown in Table 7.1, the tests vary greatly in length – with the longest tests currently completing in about 45 minutes.

7.2 Example: Creating LRU+

Though earlier chapters and sections like Chapter 5 have discussed the design of COCOASIM as well as how to use it in general, these have primarily explained how the simulator works on a theoretical level. The focus of this section however will be how a user can create and test a custom cache replacement policy on different systems and with different programs. The following paragraphs will present the entire process from an initial idea to the implementation and eventually the collection of results. The aim of this example is to experiment with an idea for a replacement policy and see if it has any effect – positive or negative – on a cache's hit rate.

7.2.1 Idea

Consider the cache hierarchy of Figure 7.2. Note that this the caches in this hierarchy is inclusive – meaning that every block in a L1 also exist in the L2. As seen, the A block is used in both of the L1s while the D block is used in none. Assume that the reason for the presence of block A in both L1s is because that address is used abnormally often by the program, and that the caches "recognizes" this because it uses an intelligent replacement policy like LRU. For simplicity, assume that each L1 cache receive their own improbable series of requests: Table 7.3 for the left cache, and Table 7.4 for the right cache.

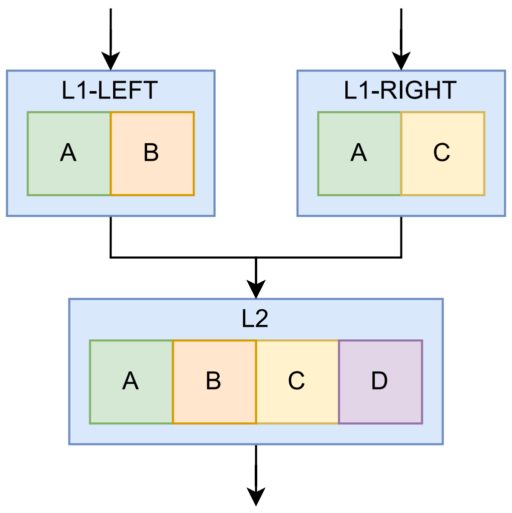


Figure 7.2: A typical cache hierarchy where address A is used by two L1s.

#	Address	Hit?
1	A	✓
2	B	✓
3	A	✓
4	C	✗
5	A	✓
6	D	✗
7	A	✓
8	E	✗
9	A	✓
+	+	+

Table 7.3: Future requests for L1-LEFT.

As seen in the two tables, there is a clear pattern for every cache. The left cache will alternate between a hit and a miss every two cycles as it either requests A or something that is not in the cache. On the other hand, the right cache on the other hand will hit on every single access.

Moving on to the actual simulation, assume that every cache uses LRU, and that the cache uses the following access counters for all the blocks: `A:3`, `B:2`, `C:1`, `D:0`. Recall that when a block hits its value becomes the highest in the set, and when a cache miss happens the replacement policy evicts the block with the lowest counter. While the right cache will hit on every access, the left cache will trigger some unusual behavior.

Consider the simulation flow of Table 7.5. Note that something unexpected happens in cycle 10: address `A` is evicted from the L2 cache. Recall that this cache hierarchy also is inclusive – meaning that both L1s are told to evict their own copy of `A` as well. Note that the aftermath of cycle 10 – marked `!10` – results in address

#	Address	Hit?
1	A	✓
2	C	✓
3	A	✓
4	C	✓
5	A	✓
6	C	✓
7	A	✓
8	C	✓
9	A	✓
+	+	✓

Table 7.4: Future requests for L1-RIGHT.

After #	R	L1 Hit?	L1 (Eviction Order)	L2 Hit?	L2 (Eviction Order)
1	A	✓	AB (BA)	-	ABCD (DCBA)
2	B	✓	AB (AB)	-	ABCD (DCBA)
3	A	✓	AB (BA)	-	ABCD (DCBA)
4	C	✗	AC (AC)	✓	ABCD (DBAC)
5	A	✓	AC (CA)	-	ABCD (DBAC)
6	D	✗	AD (AD)	✓	ABCD (BACD)
7	A	✓	AD (DA)	-	ABCD (BACD)
8	E	✗	AE (AE)	✗	ABCE (ACDE)
9	A	✓	AE (EA)	-	ABCE (ACDE)
10	F	✗	AF (AF)	✗	CDEF (CDEF)
!10	-	-	-F (F-)	-	CDEF (CDEF)

R: Requested address

Table 7.5: A functional simulation of the L1-LEFT and L2 when reading the requests of Table 7.3

A being invalidated in all caches. This is highly unfeasible as block A is accessed every two cycles and is by far the most referenced address in the program. An optimal replacement policy should recognize this and attempt to retain A in all caches permanently. However, in cycle 10 the L2 cache needs to evict an entry, and at that point address A has the lowest access counter. In other words, the algorithm works as intended despite resulting in the eventual eviction of A.

One key observation on the way of identifying the problem is that this would not happen if the L2 did not exist. Signal requesting A exclusively hit in both L1 caches, and the LRU makes sure that A is pushed back in the eviction order when a request hits. In other words, this problem is caused by a curious paradox: A *never* hits in the L2 because it *always* hits in the L1 and thus never need to access lower memory. Since no requests for address A reach the L2 cache, its access counter is never updated.

The idea on how to solve this problem is quite simple: inform the L2 cache of hits that happen in the L1s. In Table 7.5, this would cause every hit in the L1 also cause a "hit" in the L2. Consequently, A would have been put in the back of the eviction order every two cycles – preventing it from ever being evicted. Another advantage of this approach is that the info can be sent to L2 while the original request returns to the data output. This base idea of this behavior is shown in Figure 7.3.

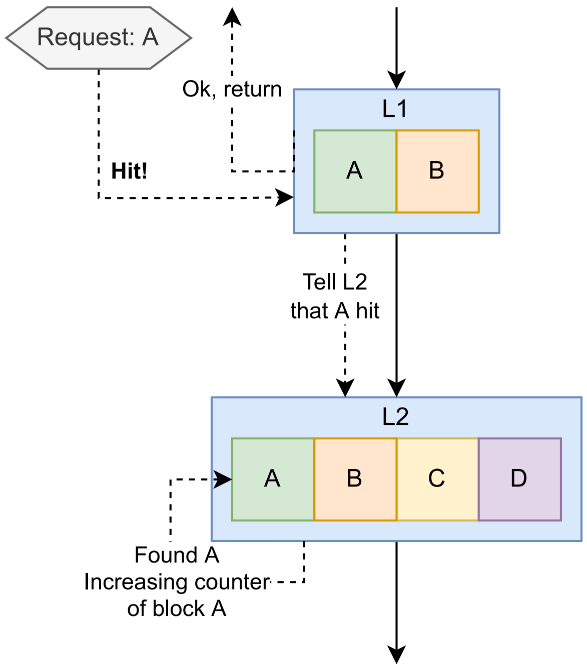


Figure 7.3: Upon hitting in the L1, info on the hit is sent to the L2.

Note that this solution only makes sense for replacement policies that perform some logic upon a cache hit like the LRU policy. While it is technically possible to mimic this behavior for, e.g., a Random replacement policy, it would have no effect on what is being evicted. As LRU is the most obvious candidate, this section will focus on an expansion of the LRU policy that also sends metadata signals on hits in caches to the lower levels of memory. As the info signals "reinforce" existing blocks in the inclusive cache hierarchy on hits, this variant is named "LRUWith-HitReinforcement" – or "LRU+ for short. Note that this policy is an extension of LRU, the same base concept should apply for all replacement policies that perform some sort of logic on a cache hit.

7.2.2 Implementation

To implement LRU+ in COCOASIM, only two new components need to be created:

1. The actual LRU+ replacement policy. This should behave in the exact same way as LRU on cache misses and eviction, but needs additional logic on cache hits.
2. A way of telling a cache to "reinforce" a block as to make it less prone to being evicted. In LRU, this can be done by simply increasing the access counter of the cache line containing the appropriate address.

Recall that COCOASIM already supports logic for sending communication messages between caches in the form of info signals – as explained in Section 6.1.5. In summary, a info signal simply performs some sort of behavior determined by an event upon accessing a memory. Thus, the info signal can be used as it already is, but it needs to carry an event that makes the cache reinforce the desired address. The resulting event – named the `HitEmulatingInfoAccessEvent` – able to do this is shown in Listing 25, while the logic of the `do_fire()` method is shown in Listing 26.

```
class HitEmulatingInfoAccessEvent : public SimulatorEvent {
public:
    HitEmulatingInfoAccessEvent(MemorySignal* info_signal,
                                Memory* target_memory);
    ...

protected:
    void do_fire() override;

    MemorySignal* event_info_signal;
    Memory* event_memory;
};
```

Listing 25: The header of the new HitEmulatingInfoAccessEvent class.

Note that the event derives from the `SimulatorEvent` class – meaning that the simulator recognizes it and knows how it is fired. As seen, the event needs parameters for what signal should be used and what memory the info signal is going to access. The reason for this is the event is entirely separated from the signal carrying it, as any possible event can be fired. Note that the `do_fire()` method goes through two checks before it actually executes any logic. First, as the info signal may access *any* type of memory, the event checks if it is accessing a cache. Then, the event checks if the address is in the cache at all. While an inclusive cache policy should enforce that every address in a L1 also is in the L2, the signal only triggers after a delay. It is possible for the relevant address to be evicted in

```
void HitEmulatingInfoAccessEvent::do_fire()
{
    auto* event_cache = dynamic_cast<Cache*>(event_memory);
    if (event_cache == nullptr) return;

    CacheRowEntry* entry = event_cache->in_cache(event_info_signal->
                                                 trace_metadata->
                                                 address);

    if (entry != nullptr)
    {
        CacheReplacementPolicy* replacement_policy =
            event_cache->get_cache_replacement_policy();

        replacement_policy->on_cache_hit(event_info_signal, entry);
    }
}
```

Listing 26: The logic that is fired by the info signal once it arrives at a cache.

the time period between an info signal is sent, and it is received. If it is a block with the appropriate address, the event will manually call the `on_cache_hit()` method with the correct signal, address, and cache line. There are two advantages to calling the `on_cache_hit()` instead of simply increasing the access counter:

1. This allows a L1 cache using LRU+ to communicate with *any* L2 cache – regardless of the replacement policy used. Any policy that use the `on_cache_hit()` method will reinforce the address in their own way, while replacement policies that do nothing on hits – e.g., Random – simply ignores the message. On the other hand, accessing the access counter would *enforce* the target replacement policy to be LRU (or derived from LRU).
2. This behavior may propagate further to the next level of memory. For example, when a regular LRU is "reinforced" it will simply register a "fake" hit and then retire the info signal. However, if an LRU+ is "reinforced" in the same way, it will also tell the next level of memory that it experienced a hit.

The header of the LRU+ class can be seen in Listing 27, while the custom implementation of the `on_cache_hit()` method is shown in Listing 28. While some parts of the code in both excerpts are removed for readability, most of the entire LRU+ definition fits in these two listings. Most of the logic that LRU+ uses already exists in the LRU super-class, and all LRU+ needs to do is to send the aforementioned `HitEmulatingInfoAccessEvent` signal to lower levels of memory once it hits. This can be seen in Listing 28 where the `on_cache_hit()` method first calls the base implementation before creating and sending an info signal with the hit emulating event as its payload.

```
class LRUWithHitReinforcement : public LRU {
public:
    LRUWithHitReinforcement();
    ~LRUWithHitReinforcement() override;

protected:
    // Interface start
    CacheReplacementPolicy * do_create_new_instance() const override {
        return new LRUWithHitReinforcement();
    }

    CacheRowEntry * handle_on_cache_hit(MemorySignal *signal,
                                        CacheRowEntry *hit) override;

};
```

Listing 27: The header of the LRU+ replacement policy.

This is all that is needed in order to add the LRU+ replacement policy to COCOASIM. After this, LRU+ can be used by any cache in the same way as any other replacement policy. Note that if the LRU+ is to be recognized by the simulator when running in "Configuration Mode", the policy must also be added to the `CacheReplacementPolicyCatalog`. However, this can be done in a single line by adding "`register_policy(new LRUWithHitReinforcement(), "lru+");`" to the catalog's constructor.

7.3 Results

7.3.1 Methodology

There are two kinds of results that are relevant to this project: the results of the simulator itself, and the results of simulations done using COCOASIM. While several policies may be of interest, it might be especially valuable to compare the LRU+ policy from Section 7.2 to LRU for different configurations. All configurations use an inclusive cache policy, but different types of systems are tested. All simulation will run on a standard laptop connected to power with the specifications shown in Table 7.6. Unless otherwise specified, the performance tests of COCOASIM will use a configuration file with the values listed in Table 7.7.

```
CacheRowEntry *LRUWithHitReinforcement::handle_on_cache_hit(MemorySignal *signal,
                                                            CacheRowEntry *hit)
{
    CacheRowEntry* super_result = LRU::handle_on_cache_hit(signal, hit);

    Cache* cache = internals->get_owner();
    Memory* target_memory = cache->get_next_level_in_memory_hierarchy();

    MemorySignal* hit_info_signal = signal->create_info_offspring();

    auto * hit_info_event = new HitEmulatingInfoAccessEvent(hit_info_signal,
                                                            target_memory);
    hit_info_signal->signal_behavior_info = hit_info_event;

    // Begin propagating access with new signal
    cache->begin_propagating_access(hit_info_signal);

    return super_result;
}
```

Listing 28: The only change to LRU+ vs LRU: overriding the `on_cache_hit()` method.

7.3.2 Simulator Performance

This section will list the results of empirical experimenting with different configurations for COCOASIM and show how the performance of the simulator is affected. This section includes the figures between Figure 7.4 and Figure 7.14 as well as Table 7.9. The following box plots feature 10 distinct experiments with that configuration, and the orange line represent their average. The plot of Figure 7.12 shows the average, maximum, and minimum operations per second among 10 experiments as well. Lastly, experiments testing the performance of LRU+ is presented in Figure 7.13 and Figure 7.14, but the times used here are only the result of a single simulation per variation. A summary of all experiments can be seen in Figure 7.9.

7.3.3 Cache Performance

While the previous section focused on performance of the simulator, this section will explore the results of cache performance for different combinations of cache replacement policies and programs. More specifically, the following figures shows how the hit rate of L1s and L2s changes over time. Unless otherwise specified, each experiment will use the methodology of Table 7.10.

Table 7.6: Platform specifications.

	Type
OS	Ubuntu 18.04.6 LTS
OS Type	64 bit
L1 Cache	32 kB
L2 Cache	256 kB
L3 Cache	8192 kB
Memory	15.3 GB
Processor	Intel Core i7-10510U
Clock	1.80 GHz
Cores	8

Property	Value
# of L1s	8
# of L2s	1
Replacement policy (all)	Random
Associativity (all)	4
Block size (all)	64 B
L1 size	32 KiB
L2 size	1 MiB
L1 delay	1 cycle
L2 delay	10 cycles
Main memory delay	100 cycles

Table 7.7: The default system used for performance testing

The data of Table 7.11 and the figures of Figure 7.15 through Figure 7.22 visualize several interesting behavioral patterns:

- **In general** the replacement policies have little effect on the final hit rate of a cache. Additionally, the Random replacement policy actually outperforms the more advanced policies in some cases. This may sound counter-intuitive, but recall that the traces being tested are pure memory copy operations and primarily consists of reading and writing an address space. Thus, "every" address is reused in cycles, and the hit rates depend primarily on how much data the caches can hold.
- **Figure 7.15** shows that one of the threads spends the first $\sim$5000 cycles on fetching distinct memory addresses. However, as shown in Table 7.7, each cache has a size of 32 KiB. Thus, all referenced data is present in the cache after a certain point in time – causing every following request to hit. This causes a 100% hit rate after cycle 5000, and an overall hit rate of over 98% 100%.

Figure(s)	Trace	Configuration
Figure 7.4, Figure 7.5	8c-64eg-256KiB-memcpy	Table 7.7
Figure 7.6	memcpy-16KiB-x32-8c-64eg	Table 7.7
Figure 7.7, Figure 7.8	memcpy-16KiB-x512-8c-64eg	Table 7.7
Figure 7.9	[memcpy-load-store]-[64MiB-128Mib-128Mib]-x1-8c-64eg	Table 7.7
Figure 7.10	memcpy-[16KiB-512kB-64MB]-[x512-x128-x1]-8c-64eg	Table 7.7
Figure 7.11	memcpy-16KiB-x512-8c-64eg	Table 7.7, with Random, LRU, and FIFO
Figure 7.12	8c-64eg-256KiB-memcpy, 8c-64eg-2MiB-memcpy, memcpy-64MiB-x1-8c-64eg	Table 7.7
Figure 7.13, Figure 7.14	memcpy-16KiB-x512-8c-64eg	Table 7.7 with LRU+

Table 7.8: A summary of various performance tests.

- **Figure 7.17** show the memory copy behavior for one of the L1s. Note that the "pattern-searching" replacement policies of FIFO, LRU, and LRU+ result in a 0% hit rate, while simply choosing a random target to evict does much better. As mentioned above, the reason for this is probably that the newly added blocks are kept in the cache as long as possible. However, since the data is copied in cycles, the newly added are never accesses before it is marked for eviction. This way, all entries are kept moderately long in the cache, but none stay there long enough to result in any hits. The Random replacement policy however does not consider how when a entry was added, and simply evicts one at random. This probably causes some blocks to remain in the cache for a long time while others are replaced almost immediately. This is actually a benefit for this kind of program as some of the entries stay long enough in the cache to result in a hit.
- **Figure 7.18** shows the shared L2 cache connected to the L1 mentioned above. Note that just in the same way as Figure 7.15, the L2 eventually contains all of the referenced data. As shown in Table 7.7, the L2 cache has a size of 1 MiB which is larger than the memcopy of 512 KiB.

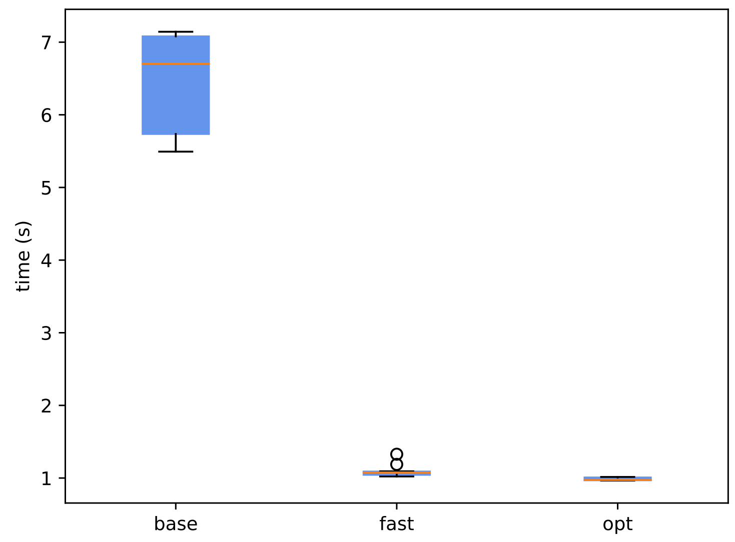


Figure 7.4: Testing the simulator on a small collection of 8192 operations.

- **Figure 7.19 and Figure 7.20** shows the same behavior as the figures above, but this time repeated 128 times - ultimately extending the length of the program. As seen in the two figures, repeating the operations simply extend the existing trend - e.g., making the hit rate of the L2 converge to 100%.
- **Figure 7.21** has a hit rate of 0% in all L1s, but manages to achieve a ∼30% hit rate in the L2 – indicating that the cache manages to keep some relevant data in the cache. However, note that Random and the other policies behave experience the same hits in the same intervals - indicating that the cause of the hits is rapid reuse of an address within a trace rather than intelligent replacement. There are slight variations however – implying that FIFO and LRU recognizes this behavior shortly before the rapid reuse occurs.
- **Figure 7.22** is the last experiment of the collection, and acts as an extension of the behavior in Figure 7.21. This is actually slightly surprising, but are probably due to the same reuse as in the previous figure. The system only manages to keep a small fraction of the overall memcpy in the L2 cache, but at least reuses data frequently enough to achieve a notable hit rate.

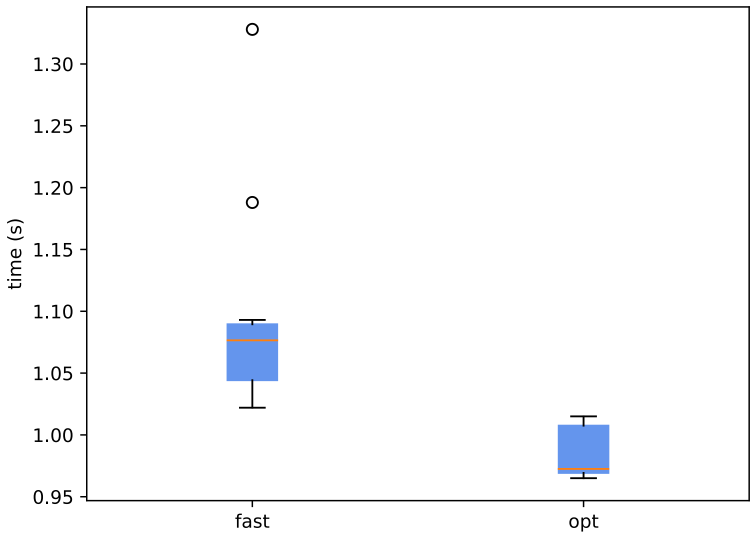


Figure 7.5: Same as Figure 7.4, but only with `fast` and `opt`.

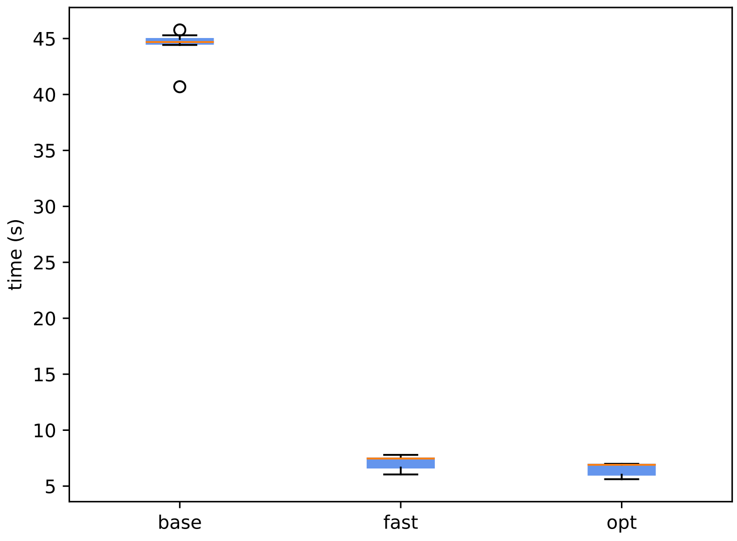


Figure 7.6: Testing the simulator on a medium collection of 131 072 operations.

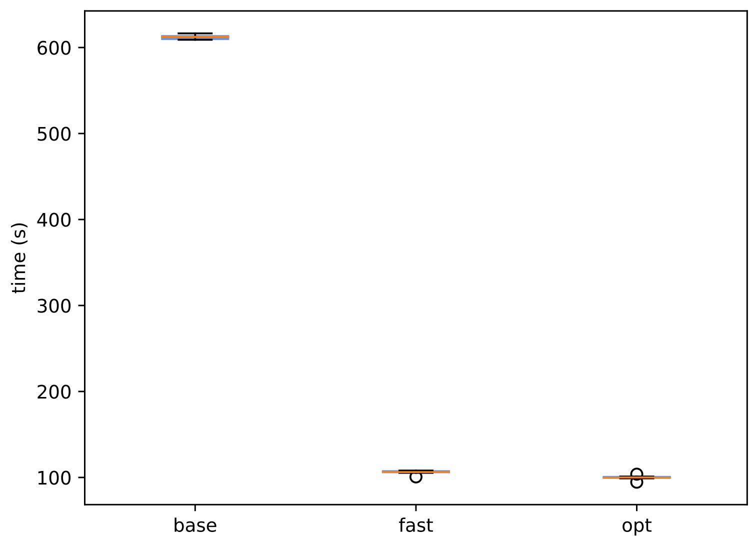


Figure 7.7: Testing the simulator on a large collection of 2 097 152 operations.

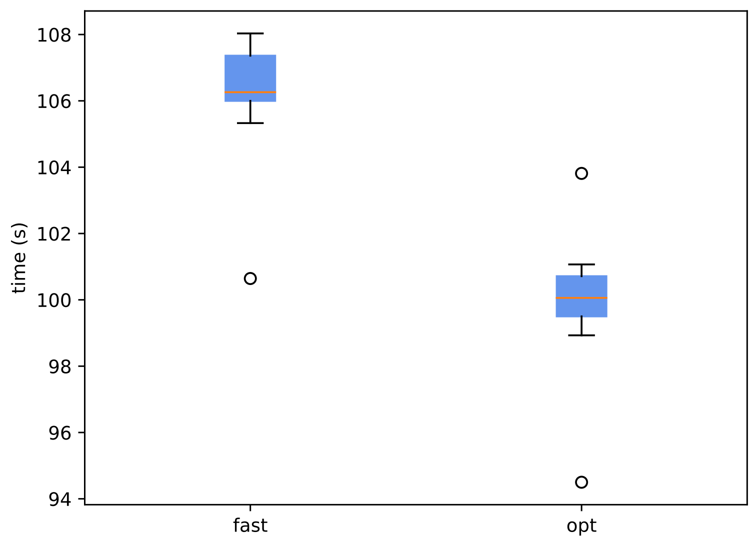


Figure 7.8: Same as Figure 7.7, but only with `fast` and `opt`.

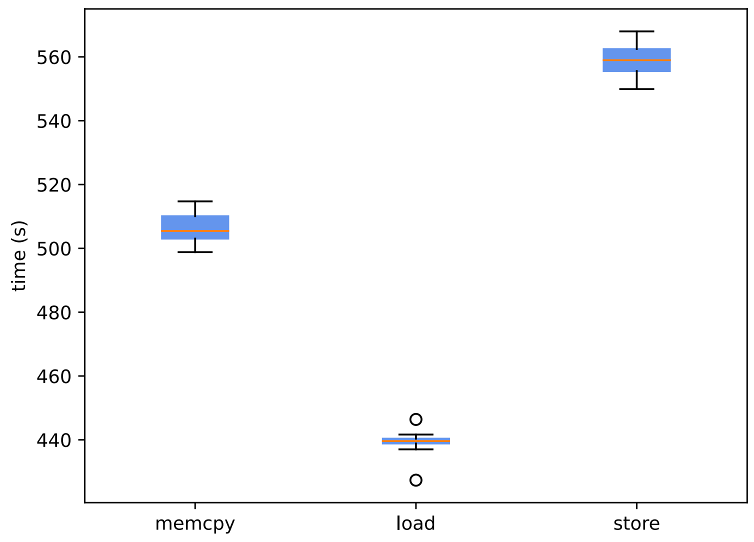


Figure 7.9: Comparing performance for memcpy, load only, and store only.

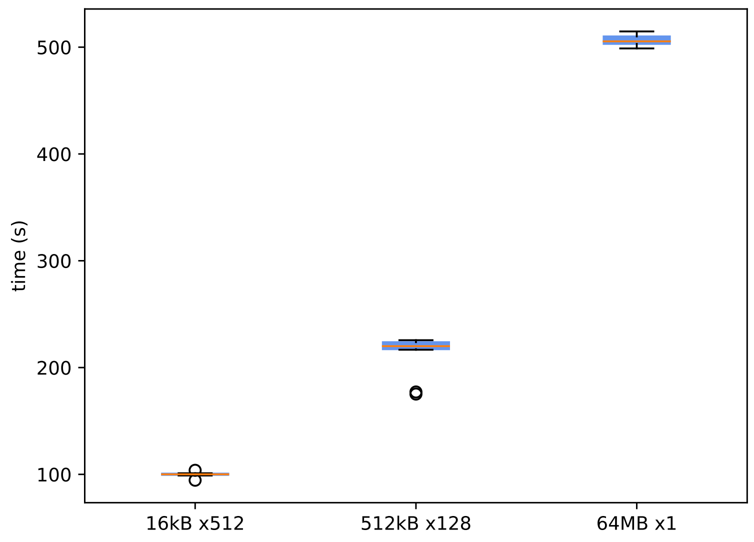


Figure 7.10: Comparing memory copies for different combinations.

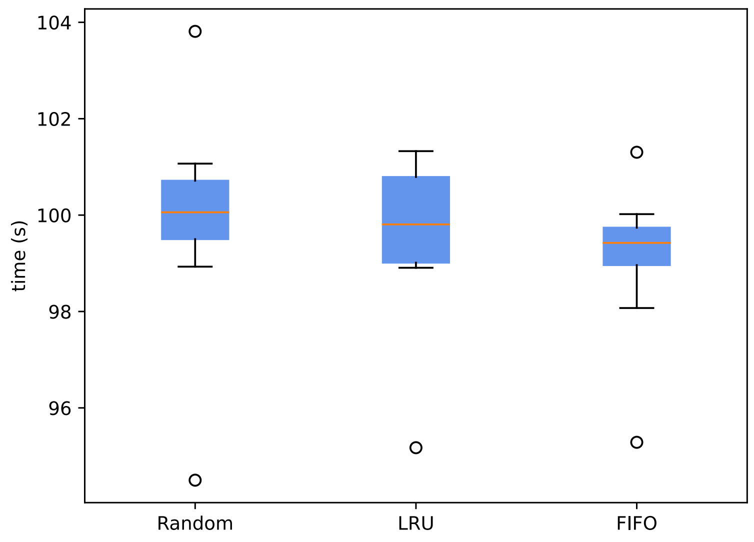


Figure 7.11: Replacement policies have low impact on performance.

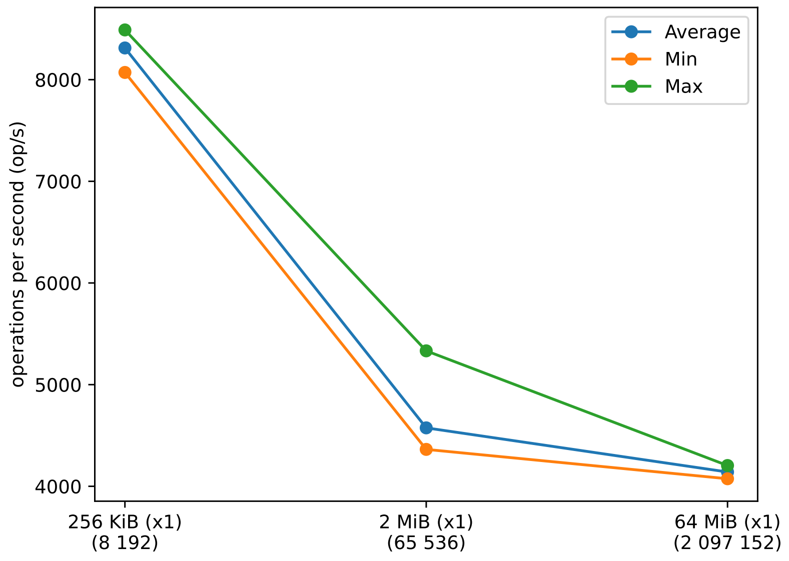


Figure 7.12: Memory copy operations per second versus trace sizes.

Reinforcements skipped	Time used
Basic LRU - ∞	1m 41s
50	1m 42s
40	1m 45s
30	1m 47s
20	1m 54s
10	2m 31s
7	3m 17s
5	4m 46s
3	9m 03s
2	15m 55s
1	37m 41s
Full LRU+ - 0	171m 59s

Note: Times represent the result of a single experiment for each variation.

Example: Three skipped reinforcement signals represent 1 out of 4 hits being "reinforced" by forwarding an info signal.

Table 7.9: Execution times for LRU+ versus the times of reinforcement signals ignored on hit.

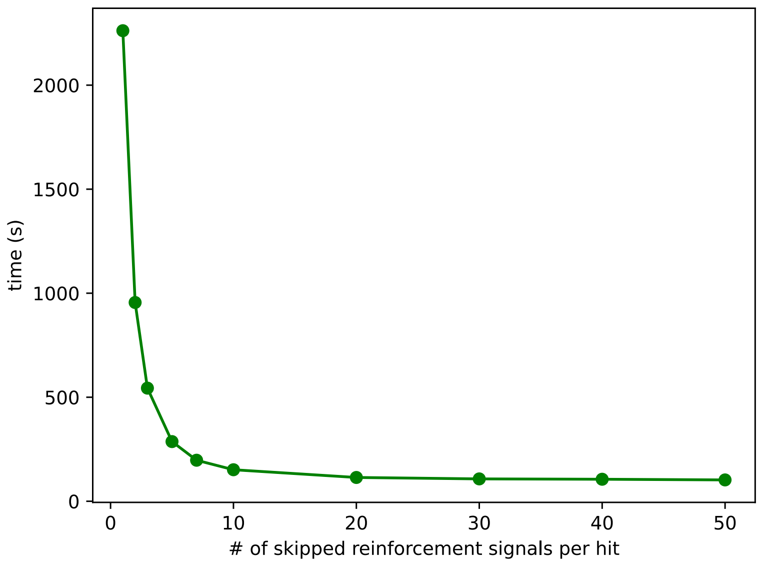


Figure 7.13: The execution times of Table 7.9 from 1-50 skipped cycles.

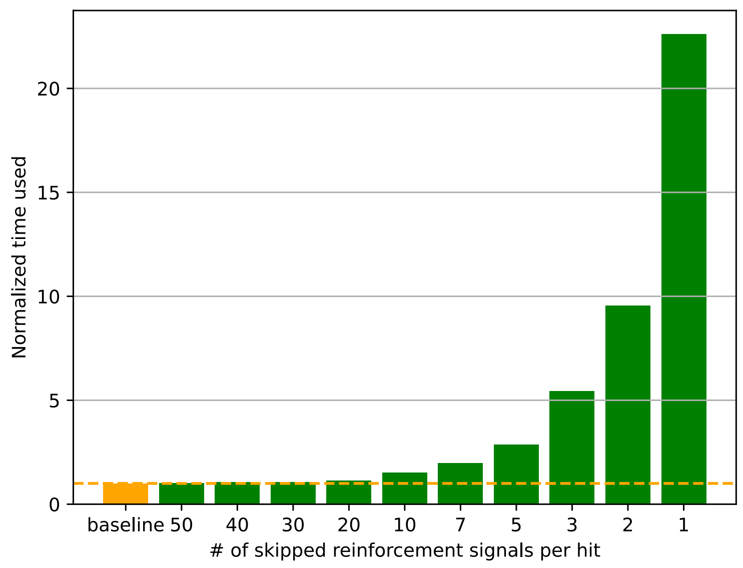


Figure 7.14: Each simulation in Table 7.9 normalized against basic LRU.

Property	Value
Checkpoints	Every 100 cycles
Cycle limit	500 000
Replacement policy	Random, FIFO, LRU, LRU+
LRU+ timeout	10

Table 7.10: Methodology for testing various cache configurations.

Group	Figure(s)	Size	Repeats	L1 Hit	L2 Hit
1	Figure 7.15, Figure 7.16	16 KiB	512	~99%	~0%
2	Figure 7.17, Figure 7.18	512 KiB	8	0-~20%	~90%
3	Figure 7.19, Figure 7.20	512 KiB	8	0-~20%	~95%
4	Figure 7.21	4 MiB	1	0%	~30%
5	Figure 7.22	64 MiB	1	0%	~30%

Table 7.11: A summary of various cache performance tests.

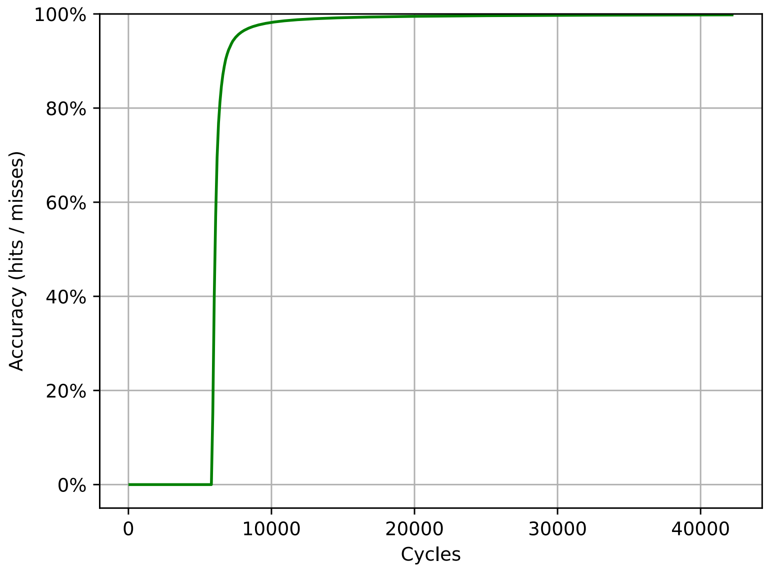


Figure 7.15: A L1's behavior over for a 16 kB memcpy repeating 512 times.

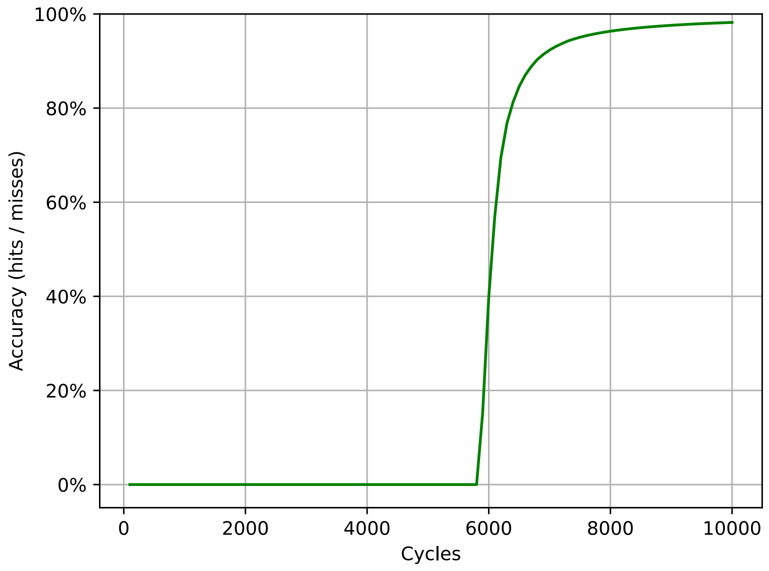


Figure 7.16: A close-up of Figure 7.15 featuring the first 10 000 cycles.

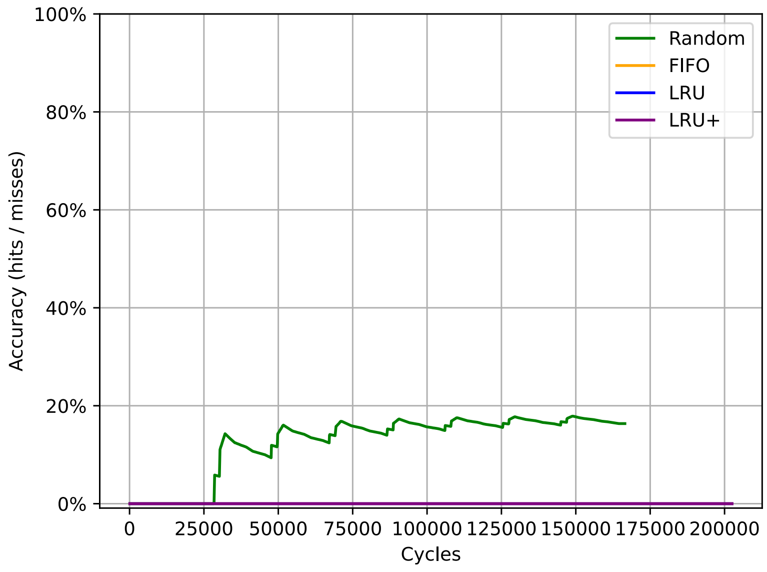


Figure 7.17: A L1's behavior over time for a 512 kB memcpy repeating 8 times.

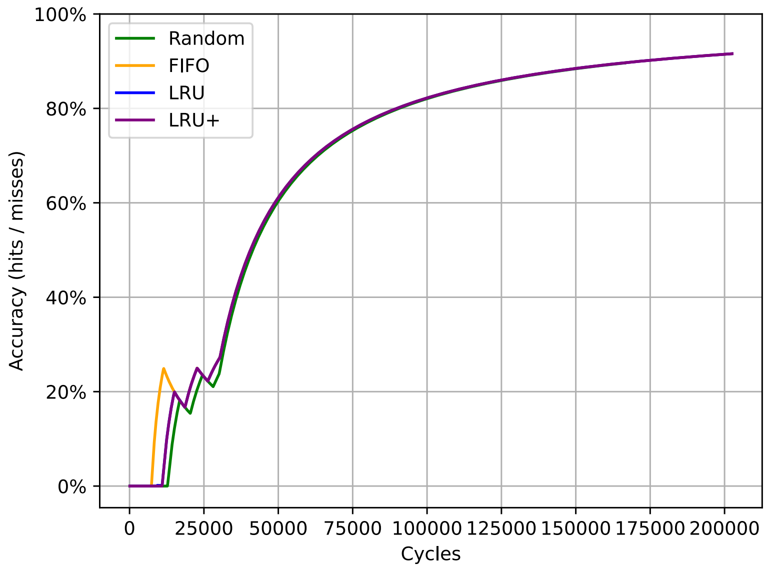


Figure 7.18: The behavior of the L2 connected to the L1 in Figure 7.17.

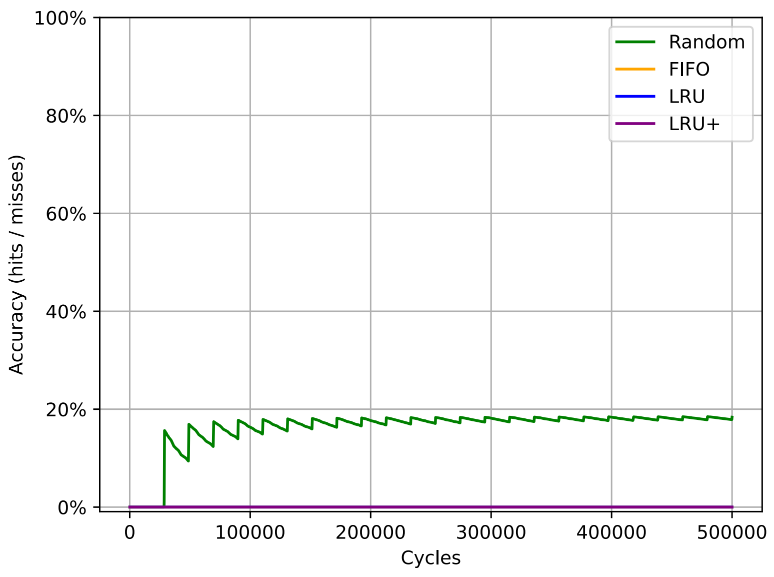


Figure 7.19: Same as Figure 7.17, but repeated 128 times instead of 8 (capped at cycle 500000).

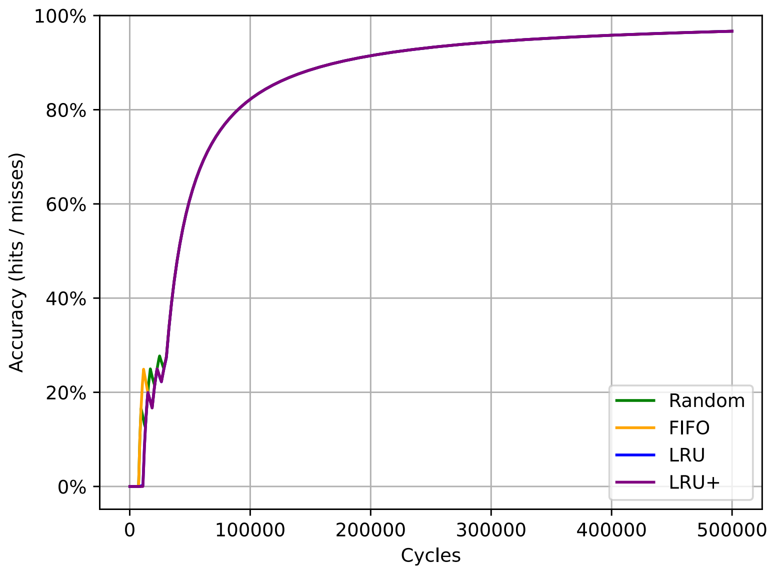


Figure 7.20: The behavior of the L2 connected to the L1 in Figure 7.19 (also capped at cycle 500000).

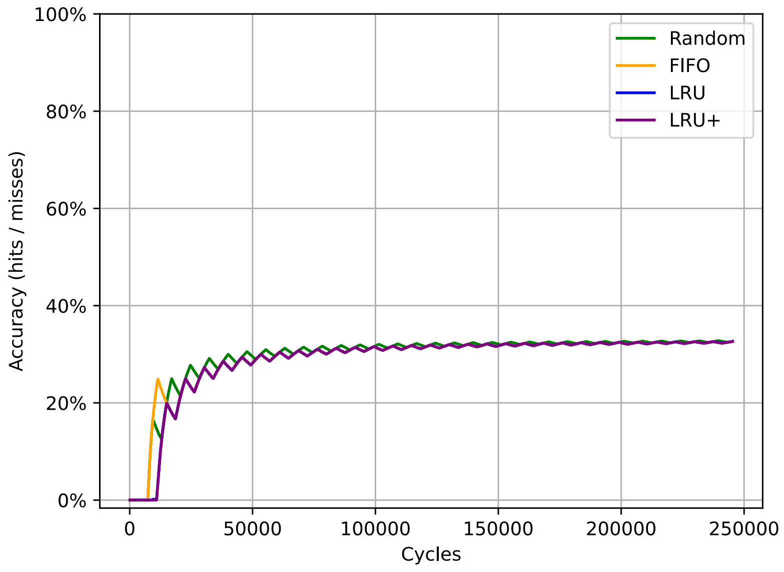


Figure 7.21: The behavior of a L2 over time for a single 4 MiB memcpy.

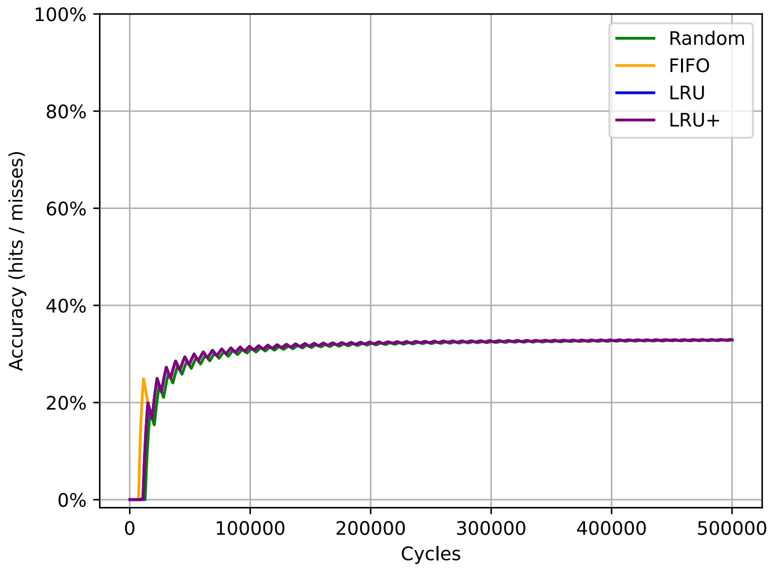


Figure 7.22: The behavior of a L2 over time for a single 64 MiB memcpy.

Chapter 8

Future Work

While the simulator is able to simulate advanced configurations of concurrent cache accesses per core, it does naturally have fewer features than, e.g., gem5. The obvious reason for this is that COCOASIM was created to meet the minimal requirements of a specific goal in a limited span of time. The result is a simulator that achieves the goal of the project and provides a high degree of customization, but also has room for additional features in the future. This chapter will list a couple of features that could have been added to COCOASIM, but were dropped due to time constraints.

8.1 The Oracle Replacement Policy

One feature that actually was under development but eventually got scrapped was the "oracle replacement policy" – an "optimal" replacement policy that would result in the highest possible hit rate for any given cache. It is naturally impossible to create a perfect replacement policy in the real world, but this should in theory be possible in simulation. While a cache only reacts on individual memory request misses as they happen, the simulator knows of *all* requests in the system – including future requests. When a replacement policy knows of requests that happen in the future, it should be able to figure out what eviction leads to the fewest misses in the future and thus the highest hit rate.

Consider the example in Figure 8.1. Assume that request 1 has just missed in the L2 cache, and the replacement policy needs to choose a block to evict. A normal, realistic replacement policy may make this decision in multiple ways – i.e., either choosing at random, using LRU, etc. – but can ultimately only consider the content of the cache and the request causing the miss. However, the simulator can "cheat" as it knows that request 2 is going to access the red block in the future. In this way, a replacement policy could in theory communicate with the simulator and be told not to evict the red block. Furthermore, the replacement policy could ask for the next requests until it knows which of the four possible evictions leads to the highest possible hit rate. While only two requests are shown in Figure 8.1, the replacement policy should be able to know of every single request that is ever

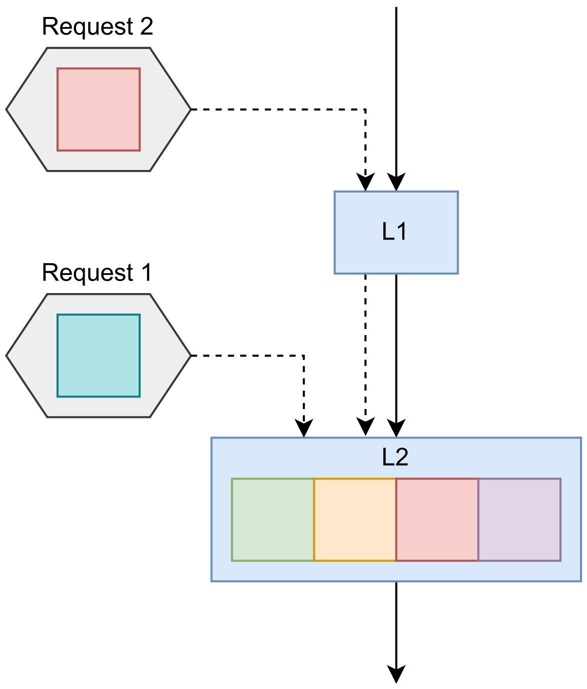


Figure 8.1: Evicting the red block will result in request 2 missing.

to access the target cache. The oracle could also read from the traces themselves to fetch requests that have not yet been loaded into the system. However, the oracle would only need to read enough request until it knows what block it should evict. Three examples of how the oracle replacement policy works can be seen in Figure 8.2, Figure 8.3, and Figure 8.4.

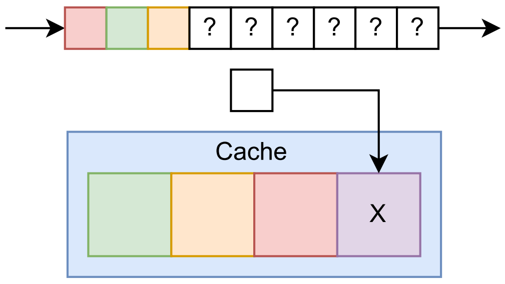


Figure 8.2: The oracle sees that red, orange, and green is going to access the cache in the near future, and decides to evict the purple block.

Note that while this works great for L1s, it quickly becomes very complex for lower level caches. Since L1s only have one possible source – i.e., a stream reader – it knows that the order of every incoming request:

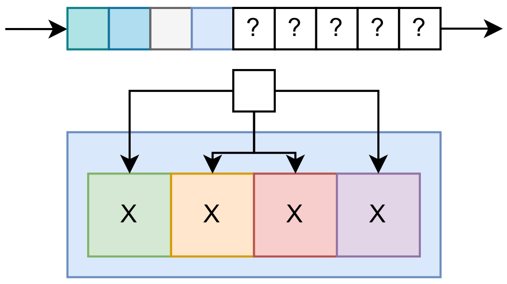


Figure 8.3: The oracle sees that next four accesses are going to miss regardless of what is evicted and may evict one of the blocks at random.

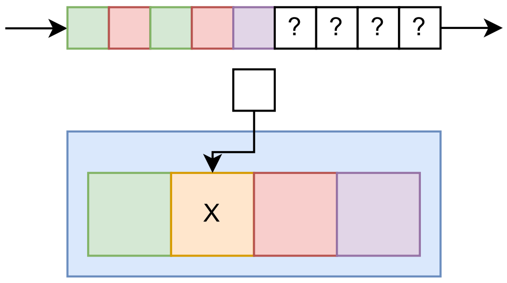


Figure 8.4: The oracle quickly sees that red and green should not be evicted. However, it does not what remaining block to evict before finally seeing the purple block – resulting in an eviction of the orange block.

1. First, every signal "in-flight" targeting the relevant cache are fetched in order. In practice, this can be done by iterating over every future `MemoryAccessEvent` in the event engine.
2. If there are too few "in-flight" signals to perform an eviction, the stream reader can be used to read the as many signals as needed. Note that the signals are only read in this process and not actually created or forwarded to the L1.

Note that the order of every request is already defined when the oracle is used for a L1 cache. While the cache may stall in the meantime – either due to all "groups" being used, replacement being impossible, or the MSHR being full – every fetched request will arrive in the given order. If the cache stalls, requests accessing the cache will reserve spots in order – thus preserving the order. However, this is not the case when the order of the requests depend on the order of multiple higher level caches. In this case, multiple problems appear:

1. Multiple sources send requests to the same cache. At first, this might not look like a problem in itself as the same process as before – i.e., iterating over `MemoryAccessEvents` in the engine and fetching their order – is still possible. However, in this case *some* requests can be stalled by the cache while other not as this can depend on their assigned *group*. It is possible to check each request signal, but this approach would require the simulator

to keep track of every group and the number of groups in the future until figuring out what to evict. Additionally, groups can be freed by returning acknowledge signals from lower memory – making the number of groups virtually impossible to manage.

2. The requests sent depend on what request *miss* in the higher level caches – making it necessary to figure out what hits, misses, and is evicted in these caches as well.
3. This approach also assumes that no other external factors manipulate the content of the caches. For example, the "LRU+ reinforcement" signals are not requests, but still affect what blocks are kept in the cache.

This insight led to the feature being dropped for this project as it proved to be a quite complex implementation that had to developed in a limited scope of time. However, a couple of alternatives were considered during the development:

1. **Attempt to create a virtual representation of every request that access a cache** – As discussed, this could be possible for L1s, but becomes too complex and scales terribly for caches nearer the main memory. As mentioned, every single operation that *may* have some effect on the content of a cache needs to be considered.
2. **Brute force check instances of the results of every eviction** – The number of unique combinations of blocks a cache can have explodes after only a few brute force eviction – making simulating systems using this approach to virtually use infinite time before completing.
3. **Performing sub-simulations representing the behavior of the next requests leading to an eviction** – This is by far the most promising alternative. Instead of trying to keep track of every request manually, a new special system is instead initialized to simulate the specific behavior of this cache for the next few requests. For example, a special cache with blocks in a kind of "superstate" could be used – where incoming requests that hit would confirm that blocks should remain in the cache. The "environments" of COCOASIM described in Section 6.1.4 might be able to do this with some more work as they are designed to handle independent instances of systems and event engines.

8.2 Parallel Programming

It should be possible to make the simulator run segments of the program in parallel to increase performance. Note that many of the events in a simulation operate on entirely separate components and do not alter any shared variables. For example, consider the L1 caches of the default configuration used for performance testing shown in Table 7.7. The first cycles, each L1 cache receive a requests that only changes the state of that specific cache. Thus, it should be possible for each of the `MemoryAccessEvents` to execute at the same time. The only two thing to watch out for in this case is that 1) the logging may happen out of sync, and 2) there needs to be some extra logic that ensures that new events created by the parallel accesses are inserted into the event engine in a consistent order. An example of this behavior can be seen in Figure 8.5. Events that *do* modify the same component however would need to execute in sequence.

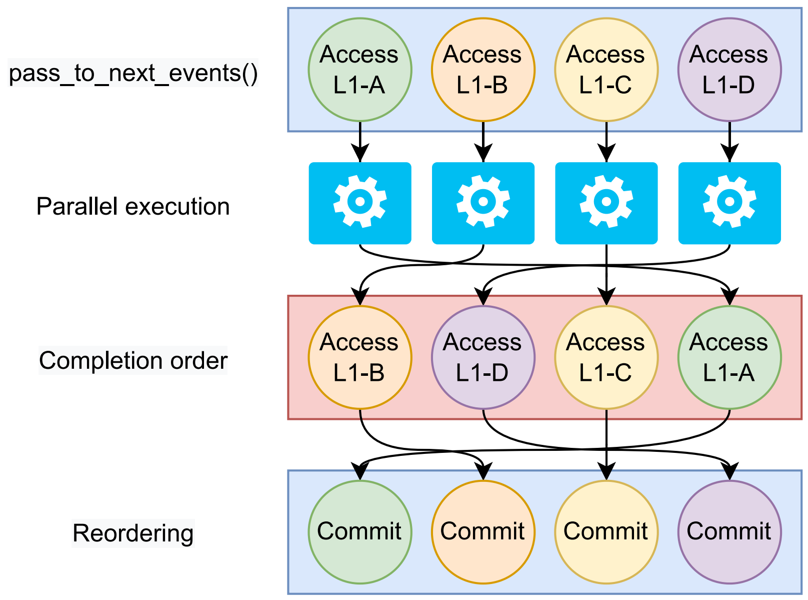


Figure 8.5: Four access events are executed in parallel as they access separate caches.

As this simulator is built on the principle of discrete-event simulation, it should always be safe to execute independent events simultaneously as long as they happen in the same cycle.The simulator guarantees that no change happens between events, so no new events should be added in the same cycle as events fire. However, events are scheduled to execute in an unknown later cycle. Since which cycles events are added to is not known before the events complete, the simulator needs to wait for all events to finish before moving on to the next cycle. This would mean a lot of "forking" and "joining" threads on every cycle, but would still allow for the simulator to execute events in parallel within a cycle. This behavior is shown in

8.3 Interconnects

Recall that the event engine works by simply managing what events happen and what times - including when memory requests access a target memory. Note that in the time period between a `MemoryAccessEvent` is scheduled and the event is executed, the signal only exists in a state of limbo. When the event engine fires

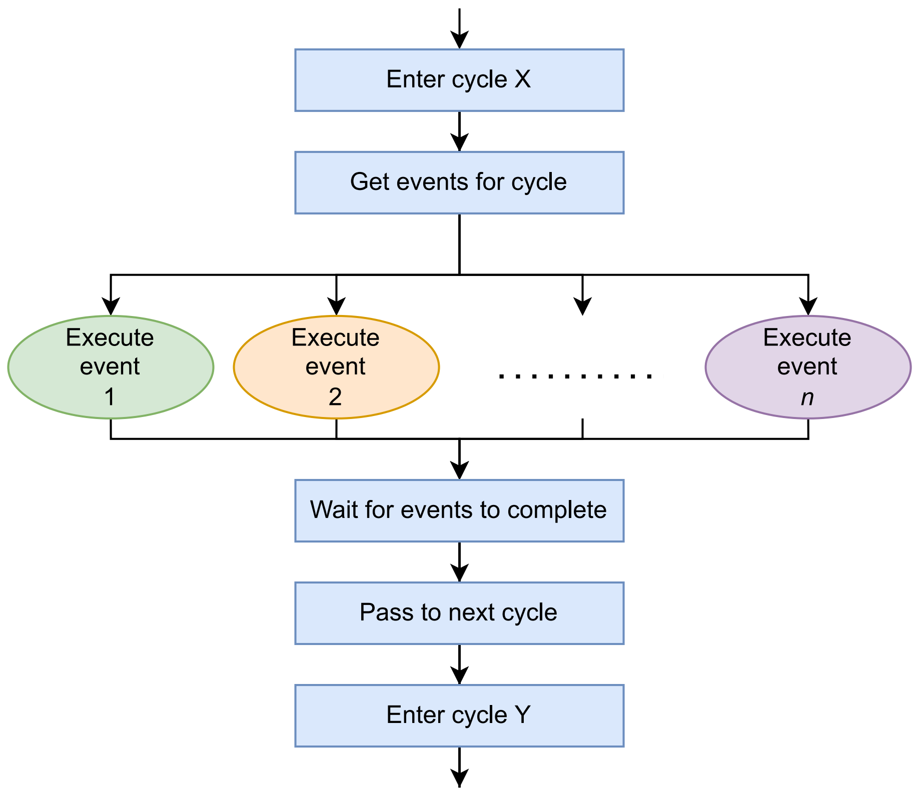


Figure 8.6: The simulator must "fork" and "join" threads for every cycle.

the event causing the signal to access the memory, the signal reappears. Once the signal accesses the memory, the simulator will make sure that it is buffered correctly if it the memory is busy. A simple representation of how this works is shown in Figure 8.7.

Note that while this design approach is simple and effective, it is not fully realistic as it "cheats" by hiding the signals. Thus, the simulator makes a couple of assumptions that may or may not be true:

1. Every request is pipelined when sent over the virtual interconnect - meaning multiple requests can exist on the same bus simultaneously. Each component makes sure that it doesn’t schedule more than one signal on a single bus in a cycle, but this may still happen if multiple components share an interconnect. For example, a L1 will not schedule multiple requests to a L2 in a single cycle, but multiple L1s may schedule requests to the L2 at the same time. In these cases, the component on the other end of handles the incoming requests one by one. For example, if a L2 receives two signals in the cycle, it will handle the one that was scheduled first and put the other in a buffer and prioritize it for the next cycle instead. This way, the simulator creates an illusion of the signals being on the interconnects.

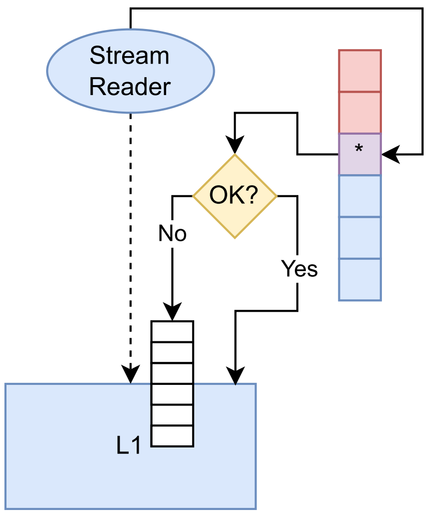


Figure 8.7: Memory access events cause signals to disappear and reappear when the events are fired. The signal may then be put in a buffer local to the cache if the cache is busy.

2. The local buffer on each component has no max size - meaning an infinite number if requests can be buffered if the cache is busy. While the requests still access the cache in the correct order once the cache is ready, there may be more queued signals than physically possible. For example, a delay of 10 cycles between two caches implies that there may be at most 10 pipelined signals on the interconnect simultaneously. The simulator ignores this - making it possible for components to queue a request or signal regardless of the number of waiting signals.
3. The simulator uses different interfaces for signals traveling "up" and "down" in the cache hierarchy. This means that e.g. signals accessing the L2 ignores signals returning to a L1. The simulator is designed this way to prevent deadlocks, but this design assumes that there are separate interconnects for signals depending on which direction they are traveling.

8.4 Breakpoints

By default, COCOASIM simulates a system until it either completes or fails an assertion. In most cases, this is sufficient as the simulator aims to either perform an entire simulation or detect bugs. However, it might be feasible to simulate a system until a certain condition is satisfied. For example, it might be interesting to simulate a large program but stop after a certain number of cycles have passed. Furthermore, if the state of the simulator could be saved on a breakpoint, it should be possible to start the simulation on a certain cycle as well. This could be valuable as it would make it possible to, e.g., test the behavior of a cache replacement policy on certain parts of a program.

8.5 Full Data Simulation

The current version of COCOASIM rarely uses full data simulation, and only to verify that everything works as expected. While this is discussed in Section 5.3.6, possible future alteration of full data simulation can be summarized as the following:

1. Support writing user-specified data instead of dummy data.
2. Use larger sizes for data read and written. Also support the size to be configurable by a user.
3. Optimize performance by specifying full data simulation at compile-time rather than run-time.

8.6 Additional Configurable Values

While COCOASIM offers a fairly wide array of configurable values, there are still many values that are not yet implemented that could be interesting to tweak. Perhaps the most interesting missing feature is to toggle the inclusive cache policy. As the goal of this project was to explore cache behavior when using an inclusive cache policy, the source code of COCOASIM assumes that every cache is inclusive.

Additionally, some features are only configurable when using systems in "System mode" rather than "Configuration mode". For example, the number of maximum concurrent executive groups of a cache can only be altered in "System mode" as the simulator has no support for "executing group"- or MSHR-keywords in the JSON configuration file. More advanced options – like configuring how many hits a `LRU+` replacement policy should "reinforce", or when the simulator should dump cache statistics – can also be added to the configuration file in the future.

Chapter 9

Conclusion

In conclusion, this thesis has presented and described a custom-made cache simulator created to simulate advanced memory behavior. The simulator stands out from other existing open-source simulator by being able to handle individual and concurrent threads per core/L1. This makes it possible to observe how threads performing memory operations on individual cores and L1 caches affect the cache hierarchy as a whole. As with many other simulators, COCOASIM is highly configurable and includes a wide array of tweak-able options - including cache content, memory hierarchy structure, replacement policy logic, and more. Furthermore, the simulator is created with scalability in mind so that it is easy to add or modify components in COCOASIM's source code. While this project is limited in regards of time and thus is not as large as e.g. gem5, the simulator still allows the simulation of advanced systems with a large amount of configurable values and options.

#	Requirement	gem5	COCOASIM
A1	Multi-core GPU environment	N/A	✓
A2	Multiple caches and hierarchies	✓	✓
A3	Memory trace per core	✗	✓
A4	Accept memory traces in a custom format	✗	✓
B1	Scalable + configurable simulator	✓	✓
B2	Configurable caches	✓	✓
B3	Inclusive cache policy	✓	✓
B4	Non-blocking caches	✓	✓
B5	Request dependencies	✗	✓
B6	Cache coherency, cache placement policy, cache replacement policy	✓	✓
C1	Acceptable resource usage	✓	✓
C2	Scalable for larger programs	✓	✓

Table 9.1: Final comparison between gem5 and COCOASIM using the requirements of Table 4.1.

A final comparison of gem5 and COCOASIM based on the requirements of Table 4.1 is shown in Table 9.1. As shown, the simulator fulfills all of the requirements needed to simulate the custom inclusive cache structure that was used as a motivation for this project. Summarized, the simulator can accept multiple traces, decode memory signals encoded in the custom Google Protocol Buffer, and enforce dependencies between memory requests if present in the trace. Additionally, COCOASIM also inhibits features found in other simulators - like highly configurable replacement policies, coherency protocol, and non-blocking caches. While Section 7.3 already shows multiple experiments of loading, storing, and copying data, the simulator is also able to simulate advanced programs as long as they are in the correct pbtrace format. In this sense, COCOASIM meets the goal of being a specialized simulator able to model advanced multi-core systems with inclusive cache hierarchies.

Bibliography

[1] P. by S. O'Dea and S. 24, *Number of mobile devices worldwide 2020-2025*, Sep. 2021. [Online]. Available: `https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide/`.

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, pp. 20–24, 1 1995.

[3] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, pp. 13–25, 3 Jun. 1997. DOI: `10.1145/268806.268810`. [Online]. Available: `https://doi.org/10.1145/268806.268810`.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 1–7, 2 May 2011. DOI: `10.1145/2024716.2024718`. [Online]. Available: `https://dl.acm.org/doi/10.1145/2024716.2024718`.

[5] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: Scalable and accurate parallel multi-core simulation," in *International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, Abstracts*, High-Performance, Embedded Architecture, and Compilation Network of Excellence (HiPEAC), 2012, pp. 91–94. [Online]. Available: `http://hdl.handle.net/1854/LU-2968322`.

[6] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The m5 simulator: Modeling networked systems," vol. 26, pp. 52–60, 4 Jul. 2006. DOI: `10.1109/MM.2006.82`.

[7] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, pp. 92–99, 4 Nov. 2005. DOI: `10.1145/1105734.1105747`. [Online]. Available: `https://doi.org/10.1145/1105734.1105747`.

[8] *Gem5: Ruby cache coherency description langueage*. [Online]. Available: https://www.gem5.org/documentation/general_docs/ruby/.

[9] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 608–619. DOI: 10.1109/HPCA.2018.00058.

[10] *Gem5: Gcn3 gpu model*. [Online]. Available: https://www.gem5.org/documentation/general_docs/gpu_models/GCN3.

[11] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "Gem5-gpu: A heterogeneous CPU-GPU simulator," *IEEE Computer Architecture Letters*, vol. 14, pp. 34–36, 1 Jan. 2015. DOI: 10.1109/LCA.2014.2299539.

[12] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2009, pp. 163–174. DOI: 10.1109/ISPASS.2009.4919648.

[13] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, Jul. 2010.

[14] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL Programming Guide*. Pearson Education, Jul. 2011.

[15] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, Jan. 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416636.

[16] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, Jan. 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416635.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, pp. 190–200, 6 Jun. 2005. DOI: 10.1145/1064978.1065034. [Online]. Available: https://doi.org/10.1145/1064978.1065034.

[18] A. Jian and C. Lin, *Cache Replacement Policies*. [Online]. Available: https://doi.org/10.2200/S00922ED1V01Y201905CAC047.

[19] H. Brais, R. Kalayappan, and P. R. Panda, "A survey of cache simulators," *ACM Computing Surveys*, vol. 53, pp. 1–32, 1 May 2020. DOI: 10.1145/3372393. [Online]. Available: https://dl.acm.org/doi/10.1145/3372393.

[20] S. Robinson, *Simulation: the practice of model development and use*. Bloomsbury Publishing, 2014.

[21] *Protocol buffers*. [Online]. Available: `https://developers.google.com/protocol-buffers`.

[22] *Boost c++ library*. [Online]. Available: `https://www.boost.org/`.

[23] *Github actions*. [Online]. Available: `https://github.com/features/actions`.

Anders Gaustad

NTNU
Norwegian University of
Science and Technology