Behavior characterization of the shared last-level cache in a chip multiprocessor

Author: Pedro Benedicte

Directors: Jose Maria Llaberia Computer Architecture Department Teresa Monreal Computer Architecture Department

June 27, 2014
Abstract
Since the stop in the frequency increase in processors, chip multiprocessors have become more relevant. A key aspect of chip multiprocessors is their memory hierarchy, specifically the shared last-level cache.

This project consists in analyzing different aspects of the memory hierarchy and understanding its influence in the overall system performance. The aspects that will be analyzed are cache replacement algorithms, memory mapping schemes and memory page policies.

Abstract
Desde que la frecuencia en los procesadores dejó de incrementar, los chips multiprocesador han ganado relevancia. Un aspecto clave de los chips multiprocesador es la jerarquía de memoria, específicamente el último nivel de cache compartida.

Este proyecto consiste en analizar diferentes aspectos de la jerarquía de memoria y entender su influencia en el rendimiento del sistema. Los aspectos que se analizaran son los algoritmos de reemplazo, los esquemas de mapeo de memoria y las políticas de página de memoria.

Abstract
Des que la freqüència als processadors va deixar d’incrementar, els chips multiprocessador han guanyat rellevància. Un aspecte clau dels chips multiprocessador és la jerarquia de memòria, específicament a l’últim nivell de la cache compartida.

Aquest projecte consisteix a analitzar diferents aspectes de la jerarquia de memòria i entendre la seva influència al rendiment del sistema. Els aspectes que s’anàlitzen són els algorismes de reemplaçament, els esquemes de mapeig de memòria i les polítiques de página de memòria.
Contents

List of Figures 6

List of Tables 8

1 State of the Art 9
  1.1 Cache memory hierarchy ........................................ 9
  1.2 Chip multiprocessors ........................................... 9
  1.3 Replacement algorithms ........................................ 11

2 Replacement algorithms 12
  2.1 Least Recently Used ........................................... 12
  2.2 Non Recently Used ............................................. 13
  2.3 Least Recently Reused ......................................... 15
  2.4 Non Recently Reused ........................................... 18
  2.5 Static Re-Reference Interval Prediction ....................... 20
  2.6 Set Dueling ................................................... 22
  2.7 Dynamic Re-Reference Interval Prediction ..................... 22
  2.8 Dynamic Re-Reference Interval Prediction with Protection ... 23
  2.9 Thread-Aware Dynamic Re-Reference Interval Prediction ...... 23

3 Simulation 24
  3.1 Simulator ................................................... 24
    3.1.1 SIMICS .................................................. 24
    3.1.2 GEMS: Ruby .............................................. 24
    3.1.3 DRAMSim2 ............................................... 24
  3.2 Simulated system ............................................... 25
    3.2.1 Caches and DRAM specification ............................ 25
    3.2.2 DRAM mapping and page policy ............................ 26
  3.3 Benchmarks .................................................. 27

4 Results 29
  4.1 Replacement algorithms ....................................... 29
    4.1.1 Performance ............................................. 29
    4.1.2 Miss distribution ....................................... 30
    4.1.3 Energy consumption ..................................... 31
### List of Figures

1. Cache hierarchy ............................................ 9
2. Frequency in Intel microprocessors .......................... 10
3. Cache hierarchy in a chip multiprocessor .................... 10
4. LRU algorithm ............................................... 12
5. LRU example ................................................ 13
6. NRU algorithm ............................................... 14
7. NRU example ................................................ 15
8. LRR groups .................................................. 16
9. LRR algorithm ............................................... 17
10. LRR example ................................................ 18
11. NRR algorithm ............................................... 19
12. NRR example ............................................... 20
13. SRRIP algorithm ........................................... 21
14. SRRIP example .............................................. 21
15. Set Dueling .................................................. 22
16. Simulation tools used ....................................... 25
17. Simulated system ............................................ 25
18. L3 mapping .................................................... 26
19. DRAM mapping ............................................... 27
20. Performance of the different replacement algorithms over LRU ........................................... 30
21. Relation between the performance and the misses of DRRIP ........................................... 30
22. Miss distribution by L3 bank ................................ 31
23. DRAM energy reduction over LRU ......................... 32
24. CPI and MPKI with Open Page ............................... 33
25. CPI and MPKI with Close Page .............................. 33
26. CPI and MPKI with Mapping 1 .............................. 34
27. CPI and MPKI with Mapping 2 .............................. 34
28. Gantt chart .................................................... 41
29. Memory hierarchy description ................................ 47
30. Memory system hierarchy .................................... 50
31. L3 states and transitions .................................... 53
32. GETS and GET_INSTR transitions ......................... 54
33. GETX and UPGRADE transitions .............................. 55
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>PUTF and PUTX transitions</td>
<td>55</td>
</tr>
<tr>
<td>35</td>
<td>DRAM DIMMs</td>
<td>56</td>
</tr>
<tr>
<td>36</td>
<td>DRAM banks</td>
<td>57</td>
</tr>
<tr>
<td>37</td>
<td>DRAM arrays</td>
<td>58</td>
</tr>
<tr>
<td>38</td>
<td>DRAM cell</td>
<td>59</td>
</tr>
<tr>
<td>39</td>
<td>DRAM states and transitions</td>
<td>60</td>
</tr>
<tr>
<td>40</td>
<td>ACTIVATE-PRECHARGE current through time</td>
<td>63</td>
</tr>
<tr>
<td>41</td>
<td>READ current through time</td>
<td>64</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table No.</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cache and DRAM characteristics</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>Benchmarks</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>Task estimated time and dependencies</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>Human resources costs</td>
<td>43</td>
</tr>
<tr>
<td>5</td>
<td>Hardware resources costs</td>
<td>43</td>
</tr>
<tr>
<td>6</td>
<td>Total costs</td>
<td>44</td>
</tr>
<tr>
<td>7</td>
<td>L2 to L3 requests</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>L2 requests and L3 replies</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>Directory requests to L2</td>
<td>51</td>
</tr>
<tr>
<td>10</td>
<td>Directory requests and L2 replies</td>
<td>51</td>
</tr>
<tr>
<td>11</td>
<td>L3 stable states</td>
<td>52</td>
</tr>
<tr>
<td>12</td>
<td>L3 transient states</td>
<td>52</td>
</tr>
<tr>
<td>13</td>
<td>DRAM commands</td>
<td>59</td>
</tr>
<tr>
<td>14</td>
<td>DRAM states</td>
<td>59</td>
</tr>
<tr>
<td>15</td>
<td>DRAM latencies</td>
<td>61</td>
</tr>
<tr>
<td>16</td>
<td>DRAM same bank accesses delays</td>
<td>62</td>
</tr>
<tr>
<td>17</td>
<td>DRAM different bank same rank accesses delays</td>
<td>62</td>
</tr>
<tr>
<td>18</td>
<td>DRAM different rank accesses delays</td>
<td>62</td>
</tr>
</tbody>
</table>
1 State of the Art

1.1 Cache memory hierarchy

One of the main issues in computer architecture is the time it takes to transfer an
element from the memory to the processor. This happens because the memory access
time is directly proportional to its size [1], so big memories will be slow, and fast
memories will be small.

This issue has been mitigated with the usage of caches. A cache is a small memory
that takes advantage of spatial and temporal locality in order to reduce the number
of accesses to memory, and thus the latency of an access. Nowadays, most processors
include 3 levels of cache: from L1 which is the smallest and fastest to L3 which is the
biggest and slowest (Figure 1).

![Cache hierarchy diagram]

**Figure 1:** Cache hierarchy. The lower levels of the cache hierarchy have smaller access
time than the higher levels.

1.2 Chip multiprocessors

Since 1970, the performance of processors has been increased mostly because of the
frequency increase, as seen in Figure 2.
Figure 2: Frequency in Intel microprocessors

Around 2003 the commercial processor’s frequency increase stopped growing at the same rate. This was because of the increment in the static power in each generation. Since it was not affordable to dissipate the thermal heat generated by the processor if they kept increasing the frequency, chip manufacturers chose to design chips with more than one processor in order to keep the performance growing each year.

In a chip multiprocessor the memory hierarchy has some private levels per processor and a shared level. In most chips the L1 and L2 caches are private per processor and the L3 is shared. (Figure 3)

Figure 3: Cache hierarchy in a chip multiprocessor. The two lower levels of the cache hierarchy are private per processor and the last level is shared.
1.3 Replacement algorithms

Since 2003, one important topic in computer’s architecture research has been the replacement algorithms for the Shared Last-Level Cache (SLLC). A replacement algorithm is in charge of deciding which line has to leave a cache set when it is full and there is a miss. If the cache evicts a line and in a short period of time accesses the same line, it will have to request the same line again, thus increasing the total execution time.

Replacement algorithms try to predict which lines are going to be referenced more so they are not evicted.

The classic approaches are Least Recently Used (LRU) [4] and Non Recently Used (NRU). Some new alternatives are Least Recently Reused (LRR) [5], Non Recently Reused (NRR) [6] or Thread-Aware Dynamic Re-Reference Interval Prediction (TADR-RIP) [7][8]. An in-depth explanation of the algorithms is in the next section.
2 Replacement algorithms

In this section the classic replacement algorithms (LRU, NRU) along with new proposals (LRR, NRR, SRRIP...) are explained.

2.1 Least Recently Used

One of the most known algorithms in cache line replacement is the LRU [16]. This algorithm tries to exploit temporal locality in recently accessed lines. For this, LRU keeps the lines that has been accessed most recently since it supposedly will be accessed in the near future.

LRU works like a stack. For each set, the least recently used (LRU) line is at the bottom of the stack, while the most recently used (MRU) line is at the top. Whether we have a miss or a hit, the referenced line will be inserted at the MRU position. When there is a replacement the line at the LRU position of the stack will be evicted. This algorithm is described in Figure 4.

```
miss or hit {
    Put line at the MRU position
}
replacement {
    Evict line at the LRU position
}
```

**Figure 4:** LRU algorithm

In Figure 5 we can see a brief sequence of accesses. Each cache set is divided in 8 slots, each one representing a line of the cache. Each line is identified by its tag. The striped cache line represents the replacement candidate, in this case the line in the LRU position.

When there is a hit (0x0000), the referenced line goes to the MRU position and the other lines move forward in the stack. In the miss (0x00A4) the LRU line (0x00B1) leaves the cache and the new line is inserted in the MRU position. The other lines also move forward.
LRU is not usually implemented in the Last Level Cache (LLC), since the cost of the implementation increases with the associativity [13] [14].

2.2 Non Recently Used

Another well-known replacement algorithm is the NRU [17]. The purpose of NRU is the same as LRU: to keep in cache the most recent lines in order to exploit temporal locality. However, it uses less information than LRU. Instead of tracking the exact order in which the lines are referenced, it only excludes the last referenced line against eviction. By doing this it will have less precision when evicting a line, and probably less hit rate. A specific implementation of NRU groups the lines in two categories: recently used and non recently used.

In Figure 6 we see the NRU algorithm. In a hit or a miss it will mark the referenced line as RU (Recently Used), and if all the other lines in the same set are also marked to RU, it will mark them to NRU (Non Recently Used). When there is a replacement, a random line of the NRU group is evicted.
miss or hit {
  Set RU line
  if (all lines RU) {
    Set NRU all lines
    Set RU line
  }
}
replacement {
  Evict random line between the NRU
}

**Figure 6:** NRU algorithm

Figure 7 is an example of the behavior of the NRU algorithm. Each cache set is divided in 8 slots, each one representing a line of the cache. Each set now has two different line groups: the blank ones representing the RU lines and the striped ones representing the NRU lines.

If there is a hit on a NRU line (0x0000) it is updated to RU. In a miss (0x00A4) a random line of the NRU group (0x00B4) is evicted. Then the new line is inserted in the RU group.
A simple NRU implementation would use one bit per cache line and a pointer to select amongst the NRU lines.

2.3 Least Recently Reused

Various algorithms have been studied in order to improve hit rates in the Last Level Cache (LLC). One of them is the LRR. This algorithm is based on the assumption that not all blocks referenced in the LLC will have reuse in the LLC, since the reuse is exploited in the lower cache levels. Therefore, LRR will predict reuse on a line referenced at least twice (the initial miss and a hit).

LRR groups the cache lines of each set in three different groups, as seen in Figure 8:

- Used
- Non-used reused

Figure 7: NRU example. Sequence of cache accesses. The vertical groups of boxes represent a cache set, being each box a line. Each line is identified by its tag in hexadecimal. The striped lines are the eviction candidates.
• Non-used non-reused

The most priority group is the used one, since it means that this line is currently in lower cache levels. The non-used reused group is where lines that have shown reuse but are not in any private cache are. Finally, the least priority group is the non-used non-reused group, composed by cache lines that have not been reused and are not in any private cache. Each of this groups is ordered by Recently Used, just like the LRU algorithm.

![Diagram of LRR groups]

**Figure 8: LRR groups**

In Figure 9 we see the LRR algorithm. When there is a miss or a hit the referenced line is moved to the MRU position of the used group. Additionally, on a hit the line is marked as reused. On a replacement the LRU line of the less priority group that exists is evicted. When a line is evicted from the last private cache, then the line leaves the used group and goes to the non-used group that matches with its reused bit.
miss {
    Put line at the MRU position of the used group
}

hit {
    Put line at the MRU position of the used group and mark reused
}

replacement {
    if (non-used non-reused group not empty)
        Evict line at the LRU position of the non-used non-reused group
    else if (non-used reused group empty)
        Evict line at the LRU position of the non-used reused group
    else
        Evict line at the LRU position
}

last private cache eviction {
    if (reused) {
        Put line at the MRU position of the non-used reused group
    }
    else {
        Put line at the MRU position of the non-used non-reused group
    }
}

Figure 9: LRR algorithm

In Figure 10 there is an example of how LRR works. The set is ordered by groups, and inside each group by use temporality. The reuse bit is shown next to each line tag. In a hit (0x0012) the line is moved to the MRU position of the used group, and is marked as reused. When there is a miss (0x0A4), the LRU line of the non-used non-reused group (0x0000) is evicted. Finally, the new line is inserted in the used group with the reuse bit set to 0.
When implementing LRR, its complexity increases with the set associativity like LRU.

### 2.4 Non Recently Reused

NRR is an algorithm that simplifies the LRR implementation but keep its objectives, analogously to NRU with LRU. The idea is to keep the same groups that LRR had but without maintaining a temporal order inside the groups.

Figure 11 explains the NRR algorithm. When there is a miss or a hit the line is put in the used group. On a hit, it is also marked as RR (Recently Reused), and if the non-used non-reused group is empty, all the lines of the non-used reused group are moved to the non-used non-reused group. On a replacement the eviction order is the same as in LRR, but inside each group the line is randomly selected. When a line is evicted from the last private cache it is moved to the non-used group that matches its reuse bit.
miss {
    Put line in the used group
}
hit {
    Put line in the used group
    Mark reused line
    if (non-used non-reused group is empty) {
        Move all non-used reused lines to non-used non-reused
    }
}
replacement {
    if (non-used non-reused group not empty)
        Evict random line from the non-used non-reused group
    if (non-used reused group not empty)
        Evict random line from the non-used reused group
    else
        Evict random line
}
last private cache eviction {
    if (reused) {
        Put line in the non-used reused group
    }
    else {
        Put line in the non-used non-reused group
    }
}

Figure 11: NRR algorithm

In Figure 12 we can see an example of a hit and a miss. The set is divided in three groups, maintaining no particular order inside each group. The reuse bit is next to the tag of each line. When there is a hit the line on the non-used reused group is moved to the used group. On a miss the replacement evicts a random line of the non-used non-reused group, and then inserts the new line in the used group without the reuse bit set.
A simple NRR implementation will only need an extra reuse bit per cache line to track the reuse and a pointer, since it uses the presence vector to know which lines are in use.

2.5 Static Re-Reference Interval Prediction

Static Re-Reference Interval Prediction (SRRIP) [7] is based on the same idea as NRR and LRR: to keep reused lines in the cache more time than non-reused lines. However, its behavior is different from NRR and LRR. SRRIP gives each cache line a value called Re-Reference Interval Prediction (RRIP). Lines with a high RRIP value will be evicted before lines with lower RRIP value.

In Figure 13 we see the SRRIP algorithm. Each cache line has M bits to represent the RRIP value. The highest RRIP value \(2^M-1\) lines are the eviction candidates. When a line misses, it gets a \(2^M-2\) RRIP value and when it hits, it gets a 0 RRIP value. In a replacement, if there are no more RRIP value of \(2^M-1\), all RRIP values are incremented. This way the eviction order will be temporally by reuse order.
miss {
  Put line and set RRIP to \( (2^M)-2 \)}

hit {
  Set RRIP to 0
}

replacement {
  while (not exists one line with RRIP \( (2^M)-1 \)) {
    Increment all RRIP by 1
  }
  Evict random line with RRIP \( (2^M)-1 \)}

**Figure 13:** SRRIP algorithm

In Figure 14 we can see an example of a hit and a miss. Each line has a RRIP value between 0 and 3 (\( M = 2 \)).

When there is a hit (0x0012) the line updates its RRIP value to 0. On a miss (0x00A4) the replacement evicts a random line between the lines with RRIP value 3 (\( 2^M-1 \)) and then inserts the new line with a RRIP value of 2 (\( 2^M-2 \)).

**Figure 14:** SRRIP example. Sequence of cache accesses. The vertical groups of boxes represent a cache set, being each box a line. Each line is identified by its tag in hexadecimal. The number next to it represents the RRIP value. The stripped lines are the eviction candidates.
2.6 Set Dueling

A problem with replacement algorithms in a chip multiprocessor is that committing to a single replacement algorithm can be harmful for some of the programs. A solution to this problem is to use a dynamic insertion policy, which selects between two different algorithms using Set Dueling [15].

A straightforward implementation of Set Dueling would be to use an extra tag directory for each cache line, keeping track of both algorithms and using the one that misses the least. However, this is an expensive implementation and it is not used.

A cost-efficient way of implementing set dueling is via set samples. As seen in Figure 15 some sets implement one algorithm (1, 4) while others (2, 7) implement a different one. These sets are the sample sets, and using a saturated counter we increase it when one algorithm misses on its sample sets and decrease it when the other misses on its sample sets. A sampling time is defined, and when that time passes, the counter is reset to 0 and the winning algorithm is used in the rest of sets (0, 3, 5, 6) until the next sampling time is up.

![Figure 15: Set Dueling](image)

2.7 Dynamic Re-Reference Interval Prediction

When the re-reference interval of the lines mapped to a set is bigger than the number of ways of the cache, SRRIP does not reuse any cache lines. To avoid this scenario, Bimodal Re-Reference Interval Prediction (BRRIP) is proposed. BRRIP inserts most...
cache lines with a distant re-reference interval prediction (value of $2^M-1$) and some lines with a long re-reference interval prediction (value of $2^M-2$). However, in cases where this does not happen, using BRRIP would decrease performance. Set dueling is used to select which algorithm is better for each execution. Dynamic Re-Reference Interval Prediction (DRRIP) is the algorithm that uses Set dueling to choose between SRRIP and BRRIP.

2.8 Dynamic Re-Reference Interval Prediction with Protection

In DRRIP a block present in lower level caches can be evicted if its RRIP value is $2^M-1$. This is not the desired behavior under the hypothesis that lines present in lower caches will be eventually reused.

Dynamic Re-Reference Interval Prediction with Protection (DRRIP+) protects the lines present in the lower cache levels when there is an eviction. When a block present in the lower cache levels is about to be evicted, its RRIP value is decremented by one and a new candidate is selected. Only when all the remaining candidate lines are present in the lower level cache one of them will be evicted.

2.9 Thread-Aware Dynamic Re-Reference Interval Prediction

Since a multicore processor can have multiple programs running at the same time, there could be a program that would benefit from BRRIP and another that would harm from it. TADRRIP [4] addresses this problem. Instead of having a single policy selection counter and a group of sample sets, it has as many counters and groups of sample sets as processors. This way each program can have the algorithm that suits it best, without being influenced by other programs.
3 Simulation

Simulations will be run in order to compare the different replacement algorithms and its influence in the performance of the system. First we explain the Simulator tools used. Then we explain the simulated system and specifically the memory hierarchy.

3.1 Simulator

In order to simulate the system we need to use 3 different simulation tools, as shown in Figure 16.

3.1.1 SIMICS

SIMICS is a functional simulator that can run unchanged programs for specific architectures (for example x86-64, ARM, MIPS, SPARC...). The operating system simulated in SIMICS is Solaris.

3.1.2 GEMS: Ruby

General Execution-driven Multiprocessor Simulation (GEMS) is a suite of diverse modules that implement extra functionalities to SIMICS. In this simulation we use Ruby, a memory hierarchy simulator. Ruby simulates the whole cache hierarchy, including the L1i, L1d, L2 and L3, its internal networks and the replacement algorithms. It also simulates the coherence protocol specified in the Specification Language Including Cache Coherence (SLIC) domain specific language.

3.1.3 DRAMSim2

DRAMSim2 is a module that simulates the DRAM. Amongst its functionalities it can simulate different page policies, such as open page and close page, different mapping schemes and also take energy measurements.
3.2 Simulated system

The simulated system is formed by 8 SPARC IV processors, 8 L1 caches, 8 L2 caches, a shared L3 cache divided in 4 banks and 4 memory controllers each one connected to an L3 bank. The memory lines have a size of 64B. In Figure 17 we see a schematic representation of the simulated system.

3.2.1 Caches and DRAM specification

The cache and DRAM characteristics are shown in Table 1.
### Table 1: Cache and DRAM characteristics

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private L1i / L1d</td>
<td>32KB, 4-way, 64B line size, LRU replacement, 1-cycle access latency</td>
</tr>
<tr>
<td>Private L2</td>
<td>256KB, 8-way, 64B line size, LRU replacement, 7-cycle access latency</td>
</tr>
<tr>
<td>Shared L3</td>
<td>8MB inclusive (4 banks of 2MB each), 64B line size. Each bank: 16-way, 10-cycle access latency</td>
</tr>
<tr>
<td>DRAM</td>
<td>32GB divided in 4 cache controllers, 8 ranks, 64K rows, 1K columns, DDR3 1333 MHz</td>
</tr>
</tbody>
</table>

In the simulations we will vary the replacement algorithm used in the L3 cache in order to analyze their behavior. The replacement algorithms user are: LRU, NRU, LRR, NRR, DRRIP and DRRIP+. The DRRIP and DRRIP+ implementations will use RRIP values of 3 bits.

The L2 and L3 cache address mapping is seen in Figure 18.

#### 3.2.2 DRAM mapping and page policy

We will simulate two different DRAM mappings, shown in Figure 19:

- **Mapping 1**
  
  In the mapping 1, dirty line evictions from the L3 cache can or can not be mapped to the same bank and rank. The access latency will be lower if they are mapped to different bank and rank than the same.

- **Mapping 2**
  
  In the mapping 2, when there is a miss in the L3 cache that generates a dirty line eviction, both the read and the write will map to the same memory bank and rank, and
different row, since the bits that map the set in the L3 cache are the same. This means that in a dirty line eviction the latency will always be equal or higher than mapping 1.

![Figure 19: DRAM mapping](image)

Another simulation parameter will be the page policy. We will simulate a DRAM with open and close page:

- **Open page**
  After each access it holds the row for a certain time or until it hits 4 times in the sense amplifiers. If the next access goes to the same row, it will save the time needed for a precharge-activate, but if it fails it will have to issue a precharge-activate, making the latency for this access increase. The DRAM behavior is explained in depth in the Annex B.

- **Close page**
  Each access has to issue an activate and after the read/write a precharge.

### 3.3 Benchmarks

In the simulation we use 50 different benchmarks. Each benchmark has 8 applications executing (one per processor). The applications have been randomly selected amongst the SPEC 2000 applications. The entire benchmark list is shown in Table 2. For each benchmark we will simulate 500M cycles for filling the caches. This cycles we will not track statistics because of all the misses it would have in all the replacement algorithms. Afterwards, we simulate 1000M cycles generating statistics.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Applications Running</th>
</tr>
</thead>
<tbody>
<tr>
<td>GemsFDTD</td>
<td>tonto, soplex, soplex, leslie3d, xalancbmk, zeusmp, bwaves</td>
</tr>
<tr>
<td>milc</td>
<td>milc, gromacs, zeusmp, gromacs, gromacs, calculix, mcf</td>
</tr>
<tr>
<td>gcc</td>
<td>mcf, povray, leslie3d, h264ref, lbm, nand, gmc</td>
</tr>
<tr>
<td>libq.</td>
<td>h264ref, leslie3d, sjeng, zeusmp, wrf, omnetpp, tonto</td>
</tr>
<tr>
<td>sphinx3</td>
<td>mcf, povray, libq, lbm, leslie3d, bwaves, hmmer</td>
</tr>
<tr>
<td>libq.</td>
<td>perlbench, gobmk, dealII, dealII, soplex, leslie3d, mcf</td>
</tr>
<tr>
<td>omnetpp</td>
<td>astar, milc, perlbench, leslie3d, libq, milc, zeusmp</td>
</tr>
<tr>
<td>namd</td>
<td>h264ref, gobmk, gromacs, GemsFDTD, h264ref, mcf, GemsFDTD</td>
</tr>
<tr>
<td>h264ref</td>
<td>bzrp2, soplex, sjeng, perlbench, gobmk, zeusmp, gcc</td>
</tr>
<tr>
<td>gameess</td>
<td>zeusmp, dealII, hmmer, astar, GemsFDTD, gcc, sjeng</td>
</tr>
<tr>
<td>sjeng</td>
<td>perlbench, leslie3d, sjeng, sphinx3, wrf, calculix, gameess</td>
</tr>
<tr>
<td>sphinx3</td>
<td>mcf, soplex, bzrp2, tonto, sjeng, mcf, gameess</td>
</tr>
<tr>
<td>soplex</td>
<td>h264ref, bwaves, gromacs, soplex, milc, astar, libq</td>
</tr>
<tr>
<td>sphinx3</td>
<td>sjeng, GemsFDTD, zeusmp, soplex, namd, dealII, bzrp2</td>
</tr>
<tr>
<td>omnetpp</td>
<td>bwaves, gobmk, wrf, soplex, astar, milc, dealII</td>
</tr>
<tr>
<td>Leslie3d</td>
<td>gromacs, leslie3d, gobmk, omnetpp, xalancbmk, gcc, GemsFDTD</td>
</tr>
<tr>
<td>soplex</td>
<td>leslie3d, bzrp2, cactusADM, gromacs, leslie3d, calculix, leslie3d</td>
</tr>
<tr>
<td>Zeusmp</td>
<td>zeusmp, xalancbmk, leslie3d, xalancbmk, bwaves, sjeng, povray</td>
</tr>
<tr>
<td>libq.</td>
<td>soplex, astar, calculix, bzrp2, GemsFDTD, hmmer, milc</td>
</tr>
<tr>
<td>libq.</td>
<td>omnetpp, zeusmp, bwaves, zeusmp, calculix, gobmk, gcc</td>
</tr>
<tr>
<td>gobmk</td>
<td>povray, calculix, cactusADM, tonto, GemsFDTD, milc, cactusADM</td>
</tr>
<tr>
<td>wrf</td>
<td>mcf, gobmk, gromacs, calculix, tonto, gameess, bzrp2</td>
</tr>
<tr>
<td>soplex</td>
<td>dealII, sphinx3, gobmk, soplex, perlbench, mcf, gromacs</td>
</tr>
<tr>
<td>milc</td>
<td>tonto, namd, bwaves, povray, bwaves, gameess, hmmer</td>
</tr>
<tr>
<td>gcc</td>
<td>h264ref, gobmk, bzrp2, GemsFDTD, sphinx3, h264ref, omnetpp</td>
</tr>
<tr>
<td>sphinx3</td>
<td>zeusmp, dealII, h264ref, leslie3d, xalancbmk, namd, perlbench</td>
</tr>
<tr>
<td>gameess</td>
<td>namd, perlbench, hmmer, xalancbmk, wrf, xalancbmk, milc</td>
</tr>
<tr>
<td>bzrp2</td>
<td>gcc, wrf, perlbench, gcc, GemsFDTD, sphinx3, povray</td>
</tr>
<tr>
<td>sphinx3</td>
<td>bzrp2, cactusADM, gameess, sjeng, dealII, milc, hmmer</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>tonto, mcf, calculix, zeusmp, calculix, gromacs, mcf</td>
</tr>
<tr>
<td>dealII</td>
<td>gcc, xalancbmk, tonto, mcf, calculix, gameess, gameess</td>
</tr>
<tr>
<td>leslie3d</td>
<td>libq, gcc, sphinx3, milc, bwaves, dealII, wrf</td>
</tr>
<tr>
<td>namd</td>
<td>povray, xalancbmk, soplex, cactusADM, gameess, soplex, hmmer</td>
</tr>
<tr>
<td>calculix</td>
<td>gameess, perlbench, sphinx3, GemsFDTD, omnetpp, omnetpp, namd</td>
</tr>
<tr>
<td>gameess</td>
<td>sphinx3, astar, bzrp2, hmmer, soplex, perlbench, calculix</td>
</tr>
<tr>
<td>sjeng</td>
<td>gobmk, milc, astar, GemsFDTD, zeusmp, tonto, wrf</td>
</tr>
<tr>
<td>leslie3d</td>
<td>calculix, gcc, omnetpp, sphinx3, wrf, bwaves, libq</td>
</tr>
<tr>
<td>leslie3d</td>
<td>mcf, astar, gcc, gcc, cactusADM, bwaves, leslie3d</td>
</tr>
<tr>
<td>hmmer</td>
<td>soplex, wrf, dealII, bwaves, povray, perlbench, sjeng</td>
</tr>
<tr>
<td>bwaves</td>
<td>dealII, gobmk, wrf, zeusmp, gromacs, soplex, leslie3d</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>wrf, sphinx3, GemsFDTD, tonto, povray, mcf, bwaves</td>
</tr>
<tr>
<td>gameess</td>
<td>tonto, GemsFDTD, bzrp2, zeusmp, gcc, soplex, cactusADM</td>
</tr>
<tr>
<td>dealII</td>
<td>leslie3d, namd, gobmk, tonto, tonto, gcc, h264ref, calculix</td>
</tr>
<tr>
<td>tonto</td>
<td>bwaves, zeusmp, wrf, sphinx3, zeusmp, leslie3d, cactusADM</td>
</tr>
<tr>
<td>gromacs</td>
<td>sphinx3, soplex, namd, wrf, sjeng, sphinx3, sjeng</td>
</tr>
<tr>
<td>gobmk</td>
<td>gcc, dealII, soplex, hmmer, omnetpp, cactusADM, povray</td>
</tr>
<tr>
<td>lbm</td>
<td>sphinx3, lbm, milc, lbm, GemsFDTD, gromacs, povray</td>
</tr>
<tr>
<td>tonto</td>
<td>lbm, sphinx3, leslie3d, sjeng, gameess, h264ref, tonto</td>
</tr>
<tr>
<td>calculix</td>
<td>sphinx3, gromacs, zeusmp, xalancbmk, hmmer, perlbench, namd</td>
</tr>
</tbody>
</table>

Table 2: Benchmarks. Each benchmark has eight applications running, one in each processor.
4 Results

In this section the results of the simulations are presented. First, we explain the replacement algorithms performance, the eviction distribution and the energy consumption. Then we compare the open page and the close page policies. Finally, we analyze the mapping schemes.

4.1 Replacement algorithms

We will analyze the performance, evicts distribution per L3 bank and memory energy consumption for each replacement algorithm.

4.1.1 Performance

In Figure 20 we can see the speedup (CPI) over LRU of the different algorithms. The benchmarks are sorted by the DRRIP+ speedup, since it is the replacement algorithm that achieves better results overall. LRR and NRR are near DRRIP+ in performance, this is because the three algorithms protect the cache lines present in lower cache levels. DRRIP performs better in some benchmarks and worse in others, meaning that sometimes it is beneficial not to protect the lines present in the L2 cache. Finally, NRU is the worse algorithm, showing almost no improvement over LRU.
Figure 20: Performance of the different replacement algorithms over LRU. The benchmarks are ordered by DRRIP+ speedup.

In Figure 21 we see that the relation between the speedup and the misses of the DRRIP. It is clearly shown that the more misses, the higher the CPI is. The other algorithms show the same relation.

Figure 21: Relation between the performance and the misses of DRRIP. The benchmarks are ordered by DRRIP+ speedup.

4.1.2 Miss distribution

Ideally, in the L3 cache the distribution of misses should be equal between the different banks. However, in a lot of the benchmarks this distribution is more irregular than it should be. In Figure 22 we see an example of a distribution where an L3 bank
can notably have more misses than the others.

Figure 22: Miss distribution by L3 bank. We can see the distribution of the L3 misses between the L3 banks for each replacement algorithm in the benchmark 33.

4.1.3 Energy consumption

In Figure 23 we can see the DRAM energy reduction over LRU for each replacement algorithm in all the benchmarks. The benchmarks have been ordered by DRRIP+ energy reduction. DRRIP+ is the one that reduces more energy, together with LRR and NRR. As it happened with the performance, DRRIP has more energy reduction in some cases and less in others. This is due the fact that the less the L3 cache misses, the less accesses to the DRAM, making the memory consume less energy.
Figure 23: DRAM energy reduction over LRU. The benchmarks are ordered by DRRIP+ energy reduction.

4.2 Page policy

Both the open page (Figure 24) and close page (Figure 25) policies perform almost exactly in all the benchmarks. The benefit of open page over close page is based on the assumption that there will be page hits in the memory. However, there are less than 1% of hits in memory accesses, making it behave similar to close page. A reason for this low amount of hit percentage is that the DRAM closes the page after a certain time without a hit or after 4 hits. A possible improvement for increasing the hit percentage would be to increase the time for the page to remain open or increase the number of hits before the page closes.
4.3 Mapping scheme

Between the two mapping schemes used (described in Section 3.2.2 DRAM mapping and page policy) there is no notable difference in all the benchmarks (Figure 26 and Figure 27). In a dirty line eviction in mapping 2, both the read and the write map
to the same bank and rank, but different lines. In mapping 1, it can map the same line but it is highly improbable.

Figure 26: CPI and MPKI with Mapping 1. We can see the CPI and the MPKI for each replacement algorithm and program of benchmark 11.

Figure 27: CPI and MPKI with Mapping 2. We can see the CPI and the MPKI for each replacement algorithm and program of benchmark 11.
5 Project description and planning

5.1 Project description

The main goal of the project is to compare how different replacement algorithms behave in the cache. We will also analyze the behavior of memory mapping schemes and memory page policies. In order to achieve that goal, we will use simulation software. Part of the software is provided by companies (SIMICS), but we will also work with modified code by researchers (GEMS). The modified code includes the implementations of the replacement algorithms.

The project will have three parts that will be repeated several times:

- Simulation set up
  All the simulation scripts have to be programmed, the simulation code has to be compiled and installed and the different algorithms have to be understood.

- Simulation run
  The different simulations will be ran. In this part of the project more software will have to be modified.

- Result analysis
  Once we have the simulation results, they will be interpreted and we will do the process again with different parameters.

5.1.1 Objectives

The main goal of the project is to analyze the behavior of the replacement algorithm inside a Chip Multiprocessor (CMP).

In order to achieve the goal, we will analyze:

- How the different content management policies used in the Last Level Cache (LLC) impact the performance of the overall system.

- The distribution of misses in the LLC.

5.1.2 Risks

Although other unknown risks may happen, the most important risks are the following:
• Already modified software
The software that we will use has already been modified in order to imple-
ment the different protocols. This means that the existing software has to be
understood and reviewed in case there are mistakes.

• Project planning
We must also take into consideration the time that we have to make the project.
The project planning in Section 5.2 explains in depth this problem.

5.1.3 Methodology
Since we want to test the behavior of a hardware component, there are two approaches
to the methodology: hardware simulation and on chip testing.

• Hardware simulation
Using simulation software reproduces the behavior of a real chip. The bench-
marks run slow since the system is being simulated. On the plus side it is
cheaper since we only have to pay the software licenses and hardware on where
it runs.

• On chip testing
It consists in producing different chips with the desired algorithms and running
the benchmarks. The benchmarks executions are fast because we are executing
on actual hardware, but the cost of designing and manufacturing a new chip is
really high.

For logistic and economic reasons, we will simulate the hardware using software sim-
ulation tools.

5.1.4 Tools
The work tools are classified in two: hardware resources and software tools.

• Hardware resources: We will use two hardware tools:
  – Computer cluster: a 153 shared node computer cluster will be used to
    execute the simulations. The cluster has Intel Xeon processors and 64GB
    of RAM memory and 1TB of data storage per processor.
Desktop computer: a desktop computer will be used as a client in order to launch the simulations and program.

- Software tools: The different software tools we will use are the following:
  - SIMICS software [9]: a functional simulator of computer system. It will simulate a SPARC processor and run Solaris.
  - GEMS memory emulation software [10]: a module that simulates the memory hierarchy system and is attached to the SIMICS simulator.
  - DRAMSim2 emulation software: a module that simulates the DRAM.
  - SQL database: we will set up a database in order to store and consult the simulation results.
  - GNUplot: this software helps to plot different statistics.
  - Bash scripts: used in order to automatize the simulation execution and data processing.

5.2 Temporal planning

5.2.1 Estimated project duration

The estimated project duration is approximately 4 months: it starts on February 17th and finishes by June 27th. The effective work days once removed holidays and weekends are 82, which divides the number of total hours of the final project (450-500) meaning that the effective time invested in the project is about 5 hours per day.

5.2.2 Project tasks

The different tasks of the project are explained, as well as the dependencies between them and some other considerations.

- Control meetings

  These meetings with the directors of the project and a university professor (Angel Olive) will be held once or twice a week, with a duration of about 2 hours each meeting. In these meetings the author of the project will explain his progress and current situation.
• GEP course
The GEP course is a course on project management that must be completed in the first weeks of the final project. It includes several partial deliveries covering different areas of project management. At the end of the project a final delivery containing all the other deliveries and a presentation about the project will be done.

• Simulation software installation
The functional simulation (SIMICS) and simulator (GEMS) software must be compiled and installed on the cluster that we will use for the simulation. This task includes modifying the Makefiles in order to include different paths, environmental variables and libraries amongst others.

• Database setup
In order to access all the simulation data easily it is convenient to set up a database. The setup includes creating the tables, programming the inserts and the queries.

• Simulation scripts programming
In order to run hundreds of simulation’s the process must be automatized. For programming these scripts we need to understand the cluster’s queueing system.

• DRAM module study
The DRAM module that we will add to the existing GEMS software will be studied and the different options for adding the module to the software will be evaluated.

• Replacement algorithm study
In order to test the influence in the performance of the different replacement algorithms we must understand them. This involves reading different papers and understanding the implementations of these algorithms.

• Simulation software modifications
The GEMS memory system simulation software must be modified. These modifications include the addition of the DRAM module and the obtaining of specific statistics about the simulation. We will use Doxygen in order to understand the code that we have to modify.
• Simulation execution, results treatment and analysis
  This three tasks generate a cycle: first we run a simulation, then we parse the results, add them to the database, plot and analyze them. We will do this process several times since we will want to obtain different statistics.

• Graphic scripts programming
  Various graphics will need to be generated from the multiple results in order to help analyzing the simulations. This task will consist in testing different plot organizations to conclude the optimal for the result analysis.

• Mapping schemes study
  In order to test the influence in the performance of the different mapping functions we must understand them.

• Final document writing
  The final document with all the conclusions must be written. This process will be done in parallel with the development of the project.

• Lecture preparation
  Final lecture preparation. The slides of the presentation will also be done.

• Final lecture
  Final lecture exposition and the afterwards questions of the tribunal.
5.2.3 Task estimated time and dependencies

In Table 3 we see the estimated time of each task and its dependencies. In Figure 28 we see the Gantt chart of the project.

<table>
<thead>
<tr>
<th>ID</th>
<th>Task</th>
<th>Time (h)</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Control meetings</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>GEP course</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Simulation software installation</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Database setup</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Simulation scripts programming</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>DRAM module study</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Replacement algorithm study</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Simulation software modifications</td>
<td>40</td>
<td>6, 7, 13</td>
</tr>
<tr>
<td>9</td>
<td>Simulation execution</td>
<td>*</td>
<td>3, 5, 8</td>
</tr>
<tr>
<td>10</td>
<td>Simulation results treatment</td>
<td>50</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>Result analysis</td>
<td>80</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>Graphic scripts programming</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Mapping schemes study</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Final document writing</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Lecture preparations</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Final lecture</td>
<td>3</td>
<td>14, 15</td>
</tr>
</tbody>
</table>

Table 3: Task estimated time and dependencies

* Simulation execution time is not accounted for here because it is not personal work time.
### 5.2.4 Gantt chart

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Control meetings</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEP course</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation software install</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Database set up</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation scripts programming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM module study</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replacement algorithm study</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation software modification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graphic scripts programming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation results treatment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping functions study</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation software modification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation results treatment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final document writing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lecture preparation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final lecture</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 28:** Gantt chart
5.2.5 Action plan

In order to ensure that everything goes as planned or to discuss different solutions in case there are any problems, there will be one or two meetings every week with the project directors.
Additionally, two weeks of the project have not been assigned to any particular task in case extra time is needed.

5.2.6 Validation method

In order to ensure the correctness of the data, the resulting measures will be calculated using different methods. Additionally, on the results analysis any abnormality will be studied in order to ensure that is its real behavior and not an error.

5.3 Project budget

The resources are divided into 3 groups: human resources, software resources and hardware resources.

5.3.1 Budget justification

The results of the project will have an impact in the performance of a multiprocessor chip. Since companies that manufacture chips need to improve the performance in each generation in order to keep competing, and these companies have high revenue (order of thousands of millions), the cost of this project is affordable. Furthermore the results of these project will be applicable to all kinds of processors: from mobile phones to desktop computers or supercomputers; because all chips nowadays are multiprocessor.

5.3.2 Human resources

There are three different tasks in the project: developer, analyst and project manager. Based on the Gantt diagram used for the time schedule, we can conclude that 40 hours (project meetings) will be of project managing, 268 hours (algorithm study and paper writing) will be of analyzing and 164 hours (programming different applications) will be of developing.
In Table 4 we can see the price per hour, as well as the total price after adding the 31% social security tax. The salaries have been consulted from a salary study [11].
5.3.3 Software resources

The memory system simulator GEMS is an open source simulator from the University of Wisconsin, and is free of cost. The SIMICS simulator is a software tool from the company Wind River, and costs 7.500€ per year, representing a cost of 2.500€ since we will use it for 4 months. The DRAMSim2 module is open source from the University of Maryland.

5.3.4 Hardware resources

There are two different hardware costs: computer depreciation and cluster rental. The computer costs 1.200€, and its depreciation cost will be 200€, since we will use it 4 out of the 24 amortization months.

The cluster rental price is 0.40€ [12] per hour per core, this price includes the electricity cost and any necessary maintenance. The execution time is the following:

\[
\frac{6 \text{ hours}}{\text{simulation}} \times 50 \text{ simulations} \times 6 \text{ algor} \times 2 \text{ page policies} \times 2 \text{ mapping schemes} = 7.200 \text{ hours}
\]

In Table 5 the total price with tax (21%) is calculated:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Price (€/h)</th>
<th>Time (h)</th>
<th>Subtotal (€)</th>
<th>Total (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servers</td>
<td>0.4</td>
<td>7.200</td>
<td>2.880</td>
<td>3.484</td>
</tr>
<tr>
<td>Computer</td>
<td>200</td>
<td></td>
<td>242</td>
<td></td>
</tr>
</tbody>
</table>

\[
3.080 \quad 3.726
\]

Table 5: Hardware resources costs
5.3.5 Total budget

The final budget for the project is shown in Table 6.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Total cost (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human capital</td>
<td>14.908</td>
</tr>
<tr>
<td>Software resources</td>
<td>2.500</td>
</tr>
<tr>
<td>Hardware resources</td>
<td>3.726</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>21.134</strong></td>
</tr>
</tbody>
</table>

Table 6: Total costs

5.4 Sustainability

5.4.1 Social impact

Since the results of the project will improve a multiprocessor’s chip performance, new and more powerful applications will be possible. It will also be possible to run the same applications faster, which will represent cost and power savings.

5.4.2 Environmental impact

In the simulations we will also evaluate the power consumption of each alternative. This means that instead of only comparing the performance, we will also compare the power consumption and the performance per power consumption.
A Cache coherence protocol

A.1 High level description

A multiprocessor system can have one or more multiprocessor chips, each one with two private cache levels and a shared cache level. The L1 cache level has separate caches for data and instructions, while the L2 cache level stores both data and instructions. The shared cache L3 is divided in banks, with a different controller each bank. There is an interconnection network that connects L2 caches, L3 banks and the memory directory. We can see a representation in Figure 29:

![Memory hierarchy description](image)

**Figure 29:** Memory hierarchy description

A.1.1 Protocol

The L3 shared cache is inclusive and knows about all the copies in the L2s. This information is stored using one bit per cache line in a presence vector. The L2 caches have three stable states:

- M: the cache line has been modified and its value is not updated in the L3 cache.

- S: the cache line has not been modified, it can be present in more than one L2 cache.

- I: the cache line is invalid, needs to fetch it from L3 cache.

The L3 stable states are the following:
• M: the cache line has been modified and the memory value is not updated.

• O: the cache line is owned by this L3, and it is responsible for serving it to other L3 caches.

• S: the cache line has not been modified.

• I: the cache line is invalid, needs to fetch it from memory.

• SS: the cache line has copies in at least one of the private L2 caches.

• SO: the cache line has copies in at least one of the private L2 caches and it has been modified, the memory value is not updated.

• MT: a L2 cache has the block in exclusivity and the L3 copy is not updated.

A.1.2 Events

• L2 miss: GETS

The cache L3 controller immediately replies to the request if it is on one of these states: M, S, O, SS or SO.

If the state is MT, the L3 cache controller requests the L2 cache that has the block that releases the exclusivity. The L2 cache line changes to the S state and sends a copy to the L3 cache. Finally, the L3 cache sends a copy to the L2 cache that requested the line.

If the block is in I state, the L3 makes a request to the directory. The directory requests another chip or the memory for the data. If it asks another chip, the same chip is the one that sends the block. If not the memory fetches and sends the line.

• L2 miss: GETX

The L3 controller replies right away if the line is in M state.

If the line is in MT state, the L3 controller sends an invalidation request to the L2 that has the line exclusivity. This cache invalidates the line and sends a copy of it to the L3 cache. Finally, the L3 cache sends the copy to the L2 cache that requested it.

If the line is in O, S or I state, the L3 controller makes a request to the directory. Then the L3 receives the number of invalidations (and the line in I state) before answering the L2 that made the request.

Finally, if the line is in SS or SO state, the L3 sends invalidations to all the L2 that
have the line. After the acknowledgements the L3 sends the line and exclusivity to the L2 requestor.

A.1.3 Other considerations

- An L3 cache replies to all the L2 replacements.
- An L2 cache replies to all the invalidation petitions from L3.
- The network between chips is point to point and keeps the order of the messages.
- The network between the L3 and L2s is point to point and keeps the order of the messages.

A.2 Protocol implementation

In this section the implementation of a simple protocol is explained. First, we announce the hypothesis for this implementation. Then we explain the different L2 and L3 requests and the transient states. Finally, we explain the state diagrams and transitions. However, in the protocol used in the simulations the L3 will be able to miss (there will be a memory) and there will be concurrency of L2 petitions.

A.2.1 Hypothesis

The hypothesis for this simplification of the protocol are:

- There is only one chip multiprocessor.
- There are no misses in the L3 cache. We suppose that the L3 acts as memory and directory.
- There is only one L2 cache petition at the same time.

In Figure 30 we can see a scheme of the memory system hierarchy proposed.
A.2.2 Requests

L2 uses a MSI protocol. Table 7 shows the different L2 requests.

<table>
<thead>
<tr>
<th>L2 Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETS</td>
<td>Line request for a data read</td>
</tr>
<tr>
<td>GET_INSTR</td>
<td>Line request for an instruction read</td>
</tr>
<tr>
<td>GETX</td>
<td>Line request for a write</td>
</tr>
<tr>
<td>UPGRADE</td>
<td>Line exclusivity request for a write</td>
</tr>
<tr>
<td>PUTS</td>
<td>Line eviction request for an unmodified line</td>
</tr>
<tr>
<td>PUTX</td>
<td>Line eviction request for a modified line</td>
</tr>
</tbody>
</table>

**Table 7:** L2 to L3 requests

All L2 requests are answered by the L3 as seen in Table 8.
In order to treat an L2 request, the directory (L3) can request:

- Invalidations to L2 caches that have copy of a line.
- An exclusive copy of the line in L2 to be sent to L3.

In the first case, the directory waits for the invalidation acknowledgements before answering the requestor. In the second, the directory wait the cache to send the line before replying the requestor.

The different directory requests to L2 are seen in Table 9 and its replies in Table 10.

<table>
<thead>
<tr>
<th>Directory request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DG</td>
<td>Send the cache line and change to S state</td>
</tr>
<tr>
<td>INV</td>
<td>Send the cache line and change to I state</td>
</tr>
<tr>
<td>INV,S</td>
<td>Send the cache line</td>
</tr>
</tbody>
</table>

Table 9: Directory requests to L2
A.2.3 States

The stable states of the L3 cache are shown in Table 11.

<table>
<thead>
<tr>
<th>Stable state</th>
<th>Description</th>
<th>Possible L2 states</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>No L2 caches have a copy of the line</td>
<td>I</td>
</tr>
<tr>
<td>SS</td>
<td>At least one L2 cache has a copy of the line</td>
<td>At least one S, others I</td>
</tr>
<tr>
<td>MT</td>
<td>An L2 cache has the line in exclusivity</td>
<td>One in M, others I</td>
</tr>
</tbody>
</table>

Table 11: L3 stable states

In Table 12 the transient states of the L3 cache are explained:

<table>
<thead>
<tr>
<th>Transient state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MO</td>
<td>Waiting for an L2 cache to provide the line for a read</td>
</tr>
<tr>
<td>MV</td>
<td>Waiting for L2 caches with copy to invalidate</td>
</tr>
<tr>
<td>MIT</td>
<td>Waiting for an L2 cache to provide the line for exclusivity</td>
</tr>
</tbody>
</table>

Table 12: L3 transient states

A.2.4 State diagrams and transitions

In Figure 31 we see the full state diagram for the simple coherence algorithm proposed.
In order to explain in detail the different transitions, we will analyse separately each L2 request.

- **GETS and GET_INSTR**

**S and SS**

If the line is in the S or SS state, the line is directly sent to the L2 requestor, and the L2 that requested for the line is added to the presence vector. The following state after this request is the SS state.

**MT**

If the line is in the MT state it means that one L2 cache has it in exclusivity. The L3 controller asks for the line to the L2 cache that has it, and requests it to downgrade the exclusivity to the block. When it gets the response it sends the line to the requestor. When waiting for the response the line is in the MO state, and the line finishes in the SS state. Finally the L2 requestor is added to the presence vector.

---

**Figure 31**: L3 states and transitions
- **GETX and UPGRADE**

**S**

If there are no L2 copies of the line the line is sent to the requestor. The line state is MT and the requestor is added to the presence vector.

**SS**

If the request is an UPGRADE and the only L2 that has the line is the requestor, it can upgrade directly without waiting.

If is an UPGRADE but other L2 caches also have the line, it has to send invalidation messages to all the other L2s. It waits in the MV state until it receives all the acknowledgement answers, and then it allows the requestor to upgrade and ends in the MT state. It removes the previous L2 sharers from the presence vector.

If the request is a GETX it must invalidate all the L2 existing copies before sending the data. It also waits in the MV state until all the L2s have replied, and then sends the data and goes to the MT state. It removes the L2 previous sharers from the presence vector and adds the L2 requestor.

**MT**

The L3 requests for the data to the L2 cache that has exclusivity on the line. It waits on the MIT state until the L2 replies with the data, and then it is sent to the L3 cache and goes back to the MT state. It updates the presence vector removing the old L2 owner and adding the L2 requestor.
**PUTS and PUTX**

**SS**

If the L2 requestor is the only one that has the cache line, the presence vector is emptied and the line goes to the S state.

If there are more L2 sharers of the line, the L2 requestor is removed from the presence vector.

**MT**

The evicted line is sent to the L3 cache and the L2 requestor is removed from the presence vector. Since no L2 caches have the line it goes to the S state.

**Figure 33:** GETX and UPGRADE transitions

**Figure 34:** PUTS and PUTX transitions
B  DRAM behavior

In this section we explain how a Dynamic Random-Access Memory (DRAM) works. First, we introduce the DRAM internal organization, then we explain the different DRAM states. Afterwards, we talk about the temporal constraints between different accesses and finally we speak about the energy consumption of DRAM.

B.1 Memory system

As we can see in Figure 35, each DRAM is composed of a Memory controller and a number of Dual In-line Memory Module (DIMM)s, which are the physical boards attached to the system. Each of these DIMMs has a number of chips, and both sides of the DIMM can have chips.

![Figure 35: DRAM DIMMs. Two DIMMs connected to the Memory controller via two logical buses.](image)

The memory controller is connected to the DIMMs via two buses, one for commands and the other for data. The command bus is unidirectional while the data bus is bidirectional. All devices are connected to the same clock, which is exclusive for the DRAM and different from the processor one.

Some of the DIMM chips can be for error-correcting and we will not take those into consideration. A rank is a group of chips that are accessed in parallel in order to
serve a memory access (usually 8 chips). Inside each chip of the rank the accessed position is the same. A DIMM can have one rank per side of the memory module or just a rank in one side.

In each chip there are storage groups called banks, as seen in Figure 36. The access to each bank is independent and a group of banks can be accessed at the same time. However since they share the same I/O interface the concurrency is limited.

![Figure 36: DRAM banks. Eight DRAM banks inside a chip share the same I/O interface.](image)

In an access, each chip supplies a number of bits each cycle (usually 64 bits), and this number sets the total of bytes read per access. This number of bits are usually sent in a burst during consecutive clock cycles. In order to decrease the time of the burst, in both the rising and falling edge of the memory clock a transfer is made.

In order to supply a number of bits in parallel, each bank is divided in arrays as shown in Figure 37. Each array accesses the same line and row at the same time, supplying one bit each array.
Figure 37: DRAM arrays. Each bank is divided in eight arrays, in order to access them concurrently.

In a read a row is selected using the row decoder, and it is sent to the sense amplifiers. Then the column decoder selects the desired bit. In a write the process is the same but with the difference that it does not go through the sense amplifiers.

In Figure 38 we see a memory cell. Each cell stores a bit in a transistor that acts as a capacitor. When a cell is accessed by the word line, the bit is sent through the bit line.
Figure 38: DRAM cell. Each bit is stored in a transistor that works as a capacitor and addressed through the bit and word line.

B.2 DRAM states

The commands used to access the DRAM are explained in Table 13.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVATE</td>
<td>Send a row from the array to the sense amplifiers</td>
</tr>
<tr>
<td>READ</td>
<td>Read from the line at the sense amplifiers</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write to the memory</td>
</tr>
<tr>
<td>PRECHARGE</td>
<td>Get the open line back to the DRAM</td>
</tr>
<tr>
<td>REFRESH</td>
<td>Refresh the DRAM</td>
</tr>
</tbody>
</table>

Table 13: DRAM commands

In Table 14 we see a simplification of the different states and their description.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Ready to activate</td>
</tr>
<tr>
<td>RowActivate</td>
<td>A row in the sense amplifiers</td>
</tr>
<tr>
<td>Precharging</td>
<td>Waiting for the PRECHARGE to finish</td>
</tr>
<tr>
<td>Refreshing</td>
<td>Refreshing a row</td>
</tr>
<tr>
<td>PowerDown</td>
<td>Memory not active</td>
</tr>
</tbody>
</table>

Table 14: DRAM states
Figure 39 shows the state diagram for the DRAM. When in the Idle state the DRAM receives an ACTIVATE, it goes to the RowActivate state, and it remains there while there are READs and WRITEs. From RowActivate it goes to Precharging when it receives a PRECHARGE command. It remains in the Precharging state a specific amount of time, and then goes back to Idle. When in the Precharging state, no commands are processed.

Each certain time, all the banks of a chip make a REFRESH, that is recover the electric charge they hold. Then they wait in the Refreshing state until it finishes and go back to Idle. In order to initiate a REFRESH all DRAM banks of a chip must be in the idle state.

A DRAM can go into low power under certain circumstances. We suppose that it can happen when all the banks of a chip are in Idle, then with a memory controller signal they go to the PowerDown state until the controller wakes them up again.

Figure 39: DRAM states and transitions

### B.3 Temporal constraints

In this section we explain the different restriction times between two consecutive commands, although sometimes a sequence of commands has to be taken into consideration. The accesses can be to the same bank, different banks but same rank or
different ranks. In Table 15 we see a list of the abbreviations used.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL</td>
<td>Read Latency</td>
</tr>
<tr>
<td>BL</td>
<td>Burst Length</td>
</tr>
<tr>
<td>WL</td>
<td>Write Latency</td>
</tr>
<tr>
<td>tRCD</td>
<td>Row to Column Delay</td>
</tr>
<tr>
<td>tRAS</td>
<td>Row Access Strobe</td>
</tr>
<tr>
<td>tRP</td>
<td>Row Precharge</td>
</tr>
<tr>
<td>tRC</td>
<td>Row Cycle: tRAS + tRP</td>
</tr>
<tr>
<td>tWR</td>
<td>Write Recovery</td>
</tr>
<tr>
<td>tCCD</td>
<td>Column to Column Delay</td>
</tr>
<tr>
<td>tRTP</td>
<td>Read to Precharge</td>
</tr>
<tr>
<td>tWTR</td>
<td>Write to Read</td>
</tr>
<tr>
<td>tRRD</td>
<td>Row activation to Row activation Delay</td>
</tr>
<tr>
<td>tRTRS</td>
<td>Rank to Rank Switch</td>
</tr>
</tbody>
</table>

Table 15: DRAM latencies. Times indicated with a preceding t (tRAS, tRCD...) are expressed in DRAM cycles.

- Same bank

Before a READ or WRITE command, the row must ACTIVATE. tRCD is the time that has to pass between these commands. In order to open another row from the same bank, first there must be a PRECHARGE and then an ACTIVATE. The time between ACTIVATEs is tRC.

The burst length of the data is BL and since each cycle there are two transmissions, for the whole transfer we must wait BL/2.
### Table 16: DRAM same bank accesses delays. The columns are the previous commands and the rows are the following commands. Times indicated with a preceding t (tRAS, tRCD...) are expressed in DRAM cycles.

- **Different bank same rank**

  When the consecutive accesses are to different banks, the ACTIVATE command can be sent to one bank while accessing the other one, thus increasing productivity.

<table>
<thead>
<tr>
<th>ACT</th>
<th>READ</th>
<th>WRITE</th>
<th>PRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>tRCD</td>
<td>max(tCCD,BL/2)</td>
<td>WL+BL/2+tWTR</td>
</tr>
<tr>
<td>WRITE</td>
<td>RL+BL/2-WL</td>
<td>max(tCCD,BL/2)</td>
<td></td>
</tr>
<tr>
<td>PRE</td>
<td>tRAS</td>
<td>tRTP</td>
<td>WL+BL/2+tWR</td>
</tr>
<tr>
<td>ACT</td>
<td>tRC</td>
<td></td>
<td>tRP</td>
</tr>
</tbody>
</table>

### Table 17: DRAM different bank same rank accesses delays. The columns are the previous commands and the rows are the following commands. Times indicated with a preceding t (tRAS, tRCD...) are expressed in DRAM cycles.

- **Different rank**

  If the consecutive accesses are to different ranks, the delays are the same as in the same rank and different bank, since they are also different banks, but adding the delay to switch to the other rank: tRTRS. There is also no delay between ACTIVATEs because they can be issued in parallel to both ranks.

<table>
<thead>
<tr>
<th>ACT</th>
<th>READ</th>
<th>WRITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>max(tCCD,BL/2)</td>
<td>WL+BL/2-RL</td>
</tr>
<tr>
<td>WRITE</td>
<td>RL+BL/2-WL</td>
<td>max(tCCD, BL/2)</td>
</tr>
<tr>
<td>ACT</td>
<td>tRRD</td>
<td></td>
</tr>
</tbody>
</table>

### Table 18: DRAM different rank accesses delays. The columns are the previous commands and the rows are the following commands. Times indicated with a preceding t (tRAS, tRCD...) are expressed in DRAM cycles.
B.4 Energy consumption

The energy and power are computed with the ACTIVATE-PRECHARGE pair, the READ, WRITE and REFRESH commands.

B.4.1 ACTIVATE-PRECHARGE

During the ACTIVATE time there is a current consumption and thus power consumption as seen in Figure 40. When the line is closing in the PRECHARGE there is also energy consumption. The energy consumption during the ACTIVATE-PRECHARGE period will be called $E_{ACT}$.

![Diagram of ACTIVATE-PRECHARGE current through time](image)

**Figure 40:** ACTIVATE-PRECHARGE current through time

The average current consumption between an ACTIVATE and a PRECHARGE will be called $I_{DD0}$. There is also background energy consumption. The average current consumption during the tRAS period is called $I_{DD3N}$, and the one during the tRP period $I_{DD2N}$.

Thus, the average current coming only from the ACTIVATE and PRECHARGE commands is:

$$I_{ACT} = \frac{I_{DD0} \cdot tRC - I_{DD3N} \cdot tRAS - I_{DD2N} \cdot (tRC - tRAS)}{tRC}$$

Then the energy consumption is:

$$E_{ACT} = I_{ACT} \cdot V \cdot tRC$$
And the average power, being $ACTPRE_{num}$ the number of ACTIVATE-PRECHARGE commands:

$$P_{ACT} = \frac{E_{ACT} \ast ACTPRE_{num}}{Time}$$

**B.4.2 READ and WRITE**

In a READ command, the current in the BL/2 period is called $I_{DD4R}$. The average current of a READ, subtracting the background energy is:

$$I_{READ} = I_{DD4R} - I_{DD3N}$$

The energy consumption is:

$$E_{READ} = I_{READ} \ast V \ast BL/2 \ast tCK$$

And the average power, being $READ_{num}$ the number of READ commands:

$$P_{READ} = \frac{E_{READ} \ast READ_{num}}{Time}$$

The WRITE command is the same as the READ, except the use of $I_{DD4W}$ instead of $I_{DD4R}$ and the use of $WRITE_{num}$ instead of $READ_{num}$.

![Figure 41: READ current through time](image-url)
B.4.3 REFRESH

In a REFRESH command all the banks from a chip are refreshed. During the refresh period (tRFC) the current is $I_{DD5}$. In order to compute the REFRESH current we must subtract the background current:

$$I_{REFRESH} = I_{DD5} - I_{DD3N}$$

The energy consumption is:

$$E_{REFRESH} = I_{REFRESH} \cdot V \cdot tRFC$$

And the power, being $\text{REFRESH}_{num}$ the number of REFRESH commands:

$$P_{REFRESH} = \frac{E_{REFRESH} \cdot \text{REFRESH}_{num}}{Time}$$
Glossary

**benchmarks** Set of programs that serve as a point of reference against which different hardware can be compared. 37

**BRRIP** Bimodal Re-Reference Interval Prediction. 22

**cluster** Set of connected computers that work together so that in many aspects they can be viewed as a single system. 37, 39, 44

**CMP** Chip Multiprocessor. 36

**DIMM** Dual In-line Memory Module. 58

**DRAM** Dynamic Random-Access Memory. 58

**DRRIP** Dynamic Re-Reference Interval Prediction. 23

**DRRIP+** Dynamic Re-Reference Interval Prediction with Protection. 23

**GEMS** General Execution-driven Multiprocessor Simulation. 24

**LRR** Least Recently Reused. 11, 15

**LRU** Least Recently Used. 11, 12

**mapping functions** Algorithm used to decide where in the cache a new line is allocated. 40

**NRR** Non Recently Reused. 11, 18

**NRU** Non Recently Used. 11, 13

**replacement algorithm** Algorithm used when a cache set is full in order to select the evicted line. 39

**simulation** Process of mimicking the internal behavior of an existing target. The result of a simulation is that the simulation model will emulate the target. 39, 40

**SLIC** Specification Language Including Cache Coherence. 24
SLLC Shared Last-Level Cache. 11

SRRIP Static Re-Reference Interval Prediction. 20

TADRRIP Thread-Aware Dynamic Re-Reference Interval Prediction. 11, 23
References


