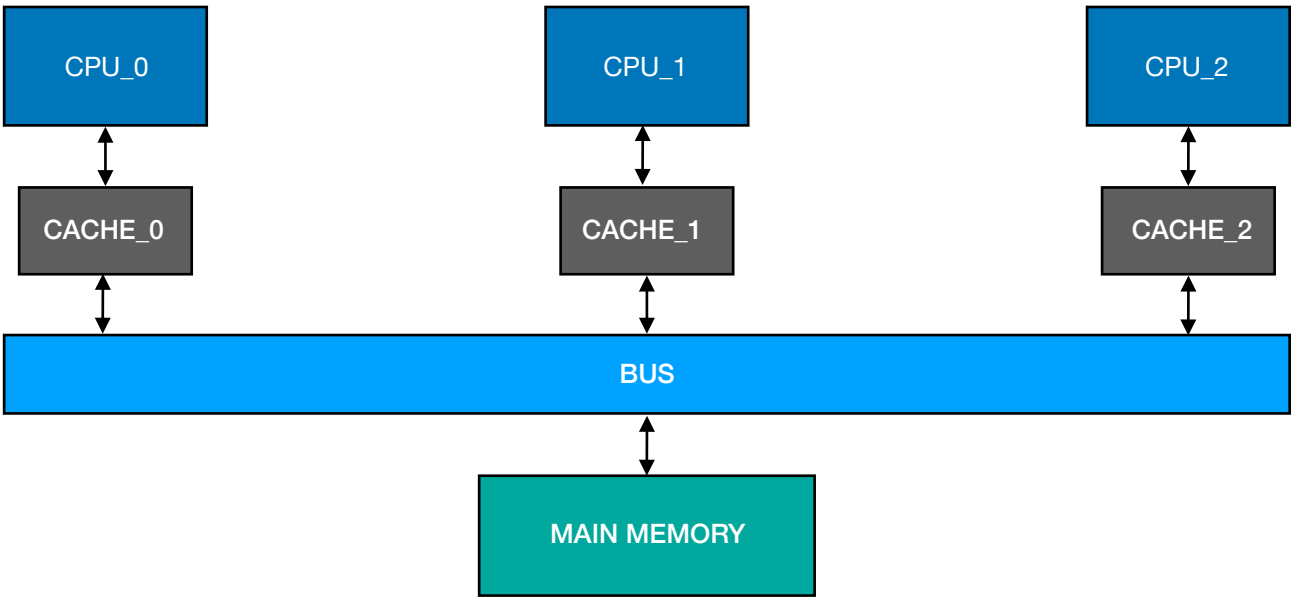
Parallel System Architectures
Lab 2: Implement a multiprocessor cache system
Theodoros Zois [UvA: 12491659 - VU: 2652652]

1. Explanation

High level overview

In this lab we have to build a multiprocessor cache coherent system. Each of the processors is associated with a local cache and all cache modules are connected to a single bus along with the main memory of the system. The caches follow the write-through and write-allocate policies. With the write-through, when a cache modifies data locally it needs also to update them synchronously in the main memory. This incident takes place in case of a “write-hit” or of a “write-miss” where the cache after fetching the data from the main memory, it modifies them. In order to provide the required coherency among caches every cache needs to broadcast the occurring event on the bus. The only event that there is not any reason to broadcast is the “read-hit”, which doesn’t affect in any way the integrity of the system. All the caches are also implementing a bus-snooping protocol. Each cache receives every request placed on the bus and takes certain actions. The protocol that defines the actions that a cache takes is determined by the VALID-INVALID protocol. In practice when another processor modifies (“write-hit” or “write-miss") a block all the other caches that own that block locally need to mark the line as invalid. So, the next time that they will search for that specific block a “cache miss” will occur since the local data are marked as invalid.



Screenshot 1: This is an example of the system. There are 3 CPUs along with their caches. Every cache is connected with its own CPU and with the BUS. The main memory is also connected with the BUS.

Components

1. MAIN MEMORY: The main memory is a pipelined memory that can handle one request per cycle and has a latency of 100 cycles. For the memory I did not create a separate module in the code, but I tried to mimic the behaviour.

2. CPU: The CPU implementation derives from the example provided with some minor modifications in order to support multiple processors. Those are:

1) Each CPU is associated with a unique CPU_ID.
2) Each CPU gets its own actions from the trace file based on that CPU_ID
3) In order to call sc_stop() the implementation makes sure that all CPUs have finished with their execution.

3. CACHE: The cache model that I am using for each processor, is similar to lab 1. Each cache is 32KB 8-way set associative L1 D-Cache with a 32-Byte line size that reads addresses of 32bits from trace files. The difference with the previous lab is that now the cache is write-through write-allocate **instead of** write-back write-allocate that was in the previous lab. That means that when a “write cache hit” or a “write cache miss” occurs, the data must be updated immediately in the main memory, before the cache proceeds to serve a new request from the CPU. When a cache hit occurs, the latency to serve the corresponding CPU is 1 cycle. The replacement strategy that the cache follows is still based on LRU (least-recently-used) algorithm with some changes that are reported later on.

8-Way Set Associative cache representation

	WAY 0	WAY 1	...	WAY 7
SET 0	LRU V TAG DATA	LRU V TAG DATA	...	LRU V TAG DATA
SET 1	LRU V TAG DATA	LRU V TAG DATA	...	LRU V TAG DATA
...	...	...	...	...
SET 127	LRU V TAG DATA	LRU V TAG DATA	...	LRU V TAG DATA

Cache size: 32KB = 32 * 1024 = 32768 Bytes
Set Size: 8 ways (lines per set)
Block size: 32Bytes = 2^5 => 5 bits block offset
Cache lines: Cache size / Block size = 32768 / 32 = 1024 Lines
Number of sets: Cache lines / Set size = 1024 / 8 = 128 sets => 2^7 => 7 bits for set index
Since address is 32 bits: 32 - 7 (set index) - 5 (block offset) = 20 bits for tag

Our address looks like the following:

31 12	11 5	4 0
TAG (20)	SET INDEX (7)	BLOCK OFFSET (5)

Each cache way (line) in a set contains the LRU counter, a valid bit and the tag. Normally it would also contain the data but we are not interested in storing any, so they are not included in the implementation (but they are in the representation). At start, all the valid bits are initialised to 0 and all the tags to -1.

LRU Algorithm: The main idea of the LRU algorithm is that for each way in a set, we keep a counter that takes values from 0 to set_size - 1 (7) without any of them in the set being duplicate. The LRU way in a set, is either the one with its counter being zero when all valid bits are 1 or the one (if multiple - the first one) with its valid bit being 0. Since we have a multiprocessor system and data can be invalidated in the cache from snooping the various bus events, we can first select to discard that data instead of the valid ones. When accessing the LRU block, its counter becomes the maximum (set_size - 1) and all the other counters are getting decreased by 1. In case the MRU (Most Recently Used) block is accessed, there is no change in the LRU counters. If the block accessed is neither LRU nor MRU, then all counters that are above the counter of the accessed block are decreased by 1 and the counter of the accessed block becomes the MRU (set_size - 1), the ones below the value of the counter of the accessed block remains untouched.

Implementation: A cache is implemented using structs. There are 3 structs that are used to build the whole cache structure. The cache_way, the cache_set and the l1_cache which is actually an array of cache_set structs. To have a better feeling related to the implementation of the data structure, have a look at the screenshot below.

```
// Cache way
typedef struct {
    unsigned int lru;
    unsigned int valid:1;
    unsigned int tag:20;
} cache_way;

// Cache set (consists of many ways)
typedef struct {
    cache_way ways[NUM_OF_WAYS];
} cache_set;

// Cache (consists of many sets)
typedef struct {
    cache_set sets[NUM_OF_SETS];
} l1_cache;

l1_cache *cache;
```

Screenshot 2: The structure of a cache. Three different structs have been used. The first one represents the cache way, the second is the cache set which consists of many ways and the last one is the actual cache which consists of many cache sets.

Each cache is associated with a unique CACHE_ID which matches the corresponding CPU_ID, as well as a bool that specifies if bus-snooping is on or off for that particular cache. This is very useful for the experiments later on where we want to disable snooping for some caches (minimal snooping) or for all. In order to be able to make a cache to serve the CPU and at the same time snoop the bus, each cache runs two separate threads, the “execute” and the “bus_handler”.

The “execute” thread is related with all the cache events that need to be done in order to serve the CPU. When a CPU provides an address the cache firsts needs to decode it, to find the set_index and the tag. After that, it determines if the CPU sent a “read” or a “write” request for that address and then different actions are taken, depending if there was a “read-hit”, “read-miss”, “write-hit” and “write-miss”. In the last three cases in order to keep the system coherent, the caches need to broadcast the event on the bus so other caches are aware. For the “read-hit” however, nothing needs to be broadcasted on the bus because it does not affect the integrity of the system at all. The bus exposes an interface with various functions that a cache can call in order to place one of the predefined requests. I am not going to dive into more details for the bus (I will do it in a later section) except from the types of requests, but it is necessary to explain them because they are inseparable with the cache-events.

- **PROBE_READ**: Needs to be placed on the bus in case of a “read-miss”, in order to fetch the block of a particular address from the main memory.
- **PROBE_WRITE**: Needs to placed on the bus in case of a “write-hit”, in order to denote that the cache needs to update the value of an address in the main memory.
- **PROBE_READX**: Needs to be placed on the bus in case of a “write-miss”. This is a combination of PROBE_READ and PROBE_WRITE since the cache has to perform two actions. Firstly, it has to fetch the block of an address from the main memory and after it modifies it locally it has to update the address in the main memory since we have a write-through cache.

How a cache determines the related cache-event? In case the CPU sends a “read” request, the cache first searches all the ways of the set_index that has been specified during the address-decode phase. If the valid bit is one, which means that our data are valid, and the tag matches with the tag from the one found from decoding the address, then we have a “read-hit”. In that case we update the LRU values and we can serve the CPU with the local copy. In case it has not been found we have a “read-miss”. In that case the cache places a PROBE_READ request on the bus. After the block is fetched, the cache stores it locally (write-allocate policy), the valid bit is set to one, the LRU values are updated and the CPU gets served. In case the CPU sends a “write” request for an address, the cache again checks if the block is stored locally but now the value of the valid bit is not checked, only the tag. If the tag matches, we have a “write-hit” and the LRU values are updated, the cache sets the valid bit to one, serves the CPU and places a PROBE_WRITE request on the bus in order to update the address in the main memory. If the tag is not found, we have a “write-miss”. The cache places a PROBE_READX request

on the bus. After the block is fetched from the main memory, the cache modifies it and stores it locally. The valid bit is set to one, the LRU values are updated, the CPU is getting served and the cache broadcasts the modification back to the main memory.

The “bus_handler” thread is related with snooping the bus. That thread is simply listening to all the requests that are placed on the bus and takes certain actions on certain cases. In the VALID-INVALID protocol, the only action that a cache has to take is to invalidate its cache entry if owns the particular address (tag) that is broadcasted on the bus. This is happening when another cache has a “write-hit” or a “write-miss”, which means that it’s going to update (or eventually-update) the value of that address in the main memory. Practically, the cache has to invalidate its copy when a request of type PROBE_WRITE or a PROBE_READX is placed on the bus.

4. BUS: The bus is implemented in the fashion of a split-transaction bus that can be occupied only by one module (cache or main memory) at a certain time. All the occurring events in the bus are split in the form of request-response. The requests on the bus that are related with fetching data from the main memory are non-blocking (“read-miss” or “write-miss”). Since the bus is not occupied by the cache that is waiting for a response to come back, in the meantime other caches can place their own requests. The responses from the main memory can come back out-of-order. The bus keeps track of when a response is ready to be broadcasted back to the requestor, and gives them priority over the unplaced (pending) requests.

All the requests placed on the bus, are being tracked into a table (requests_table) that the bus manages. The need to track the requests comes from the fact that we want to avoid conflicting requests (those that are referring to the same block). For instance, if a CACHE_0 wants issue a request in order to fetch a block from the main memory with tag 128 and immediately CACHE_1 wants issue a request in order to write that block, there is a chance that CACHE_0 will read outdated data. In order to prevent such, or similar cases, the bus allows only one of them to be issued and the other one waits until the first finishes. The only requests that are not considered as conflicting (for a block), are the ones related with “read-miss” where the caches are just fetching the data from the main memory without making any modification. The screenshot below shows the form of an entry in the requests_table.

```
typedef struct
{
    unsigned int tag;
    Request request;
} bus_request;
```

Screenshot 3: The form of a request in the requests_table. The bus stores in the table the tag and the associated request type: PROBE_READ, PROBE_WRITE or PROBE_READX.

So, how the bus grants access to caches? First of all, the bus checks for pending responses from the main memory, since they have priority. If there are any, the cache is being blocked until they are broadcasted. If there are not any, conflicting requests are being checked. If the cache wants to place a request that has a conflict with another, it is getting blocked until the already placed one finishes. In case that there are not any (or not anymore) conflicting requests the bus is checking if there is any cache that has occupied it. Again, if there is already a cache that has occupied the bus, the one that wants to get access is getting blocked until the bus is free. When it becomes available, the bus grants access to the cache.

As it is already mentioned in (3), the bus exposes an interface that the caches can call when they want to place a certain request. In the following lines I will provide some details for each of one the available function calls.

- **read():** This function is called by a cache that wants to place a PROBE_READ request on the bus. Firstly, the cache tries to acquire lock. After is successfully granted access to the bus the request is placed into the requests_table and the address, the cache_id and the request are broadcasted. The cache is waiting for 1 cycle for everyone to snoop the bus and then releases the lock but remains blocked for 100 ns since it waits for a response from the main memory. After 100 ns, the bus records that there is a pending main memory response ready. The “main memory module” locks the bus and returns the “response” releasing the lock. In the last part, where the main memory returns the response, the bus also removes the entry from the requests_table.
- **write()**: This function is called by a cache that wants to place a PROBE_WRITE request on the bus. Firstly, the cache tries to acquire lock. After is successfully granted access to the bus the request is placed into the requests_table and the address, the cache_id and the request are broadcasted. The cache is waiting for 1 cycle for everyone to snoop the bus and then another 100 ns in order to mimic the behaviour of updating the contents of the main memory. The entry is then removed from the requests_table and the cache releases the lock on the bus.
- **readX_fetch && readX_write:** Both of these calls are related to a PROBE_READX request. Normally, they should be one function but the reason I implemented it in two, is because I didn’t want the cache to occupy the bus for the time that the cache needs in order to store locally the fetched block from the main memory. The readX_fetch() is exactly the same as read() but instead a PROBE_READX is broadcasted. In the readX_write() the cache tries to get a lock on the bus. After is successfully granted access, it waits for 100 ns (mimic the access in the main memory) and the cache releases the lock while the bus removes the entry from the requests_table that has been placed when readX_fetch() has been called.

Part of the bus is an inspiration from the approach of Silicon Graphics Inc. (SGI). In their approach 1) the requests_table lies in every cache and 2) when a response is ready it is matched with the related request. In the current implementation, the requests_table for simplicity is part of the bus and is managed from it. I wanted to avoid, for the purposes of

the simulation, the fact that each cache had to keep track of the requests that are taking place and remove them from their local requests_table when the requests are getting served.

Some general details for the implementation:

- In the code there are two variables that aim to either truncate or expand the number of messages in stdout. One of them is related to NOP messages produced by the CPUs and the other with the locking/unlocking mechanism of the BUS. By default the output is truncated for the NOP messages because I observed that stdout is overloaded, plus NOP messages are not so important for the execution. On the other hand, the lock/unlock messages by default are not truncated. I decided to keep them as a part of the logging trace because it is clearer for one to observe the events related with the exclusive usage of the bus. However, you can modify them as you wish on top of the file *lab2.cpp*.
- In sc_main() all the CPUs and caches are getting a unique ID and a unique name in the form of (CPU_ID / CACHE_ID). The ID is the same for each pair of CPU-cache.
- For the bus I have used 3 ports. One to broadcast the address, one for the cache ID that places the request and one for the type of the request. Also, especially for the last one, instead of a sc_signal I have used an sc_buffer because we want to trigger the value_changed_event() in the snooping thread even if the same value (with the one before) has been sent.

2. Experiments

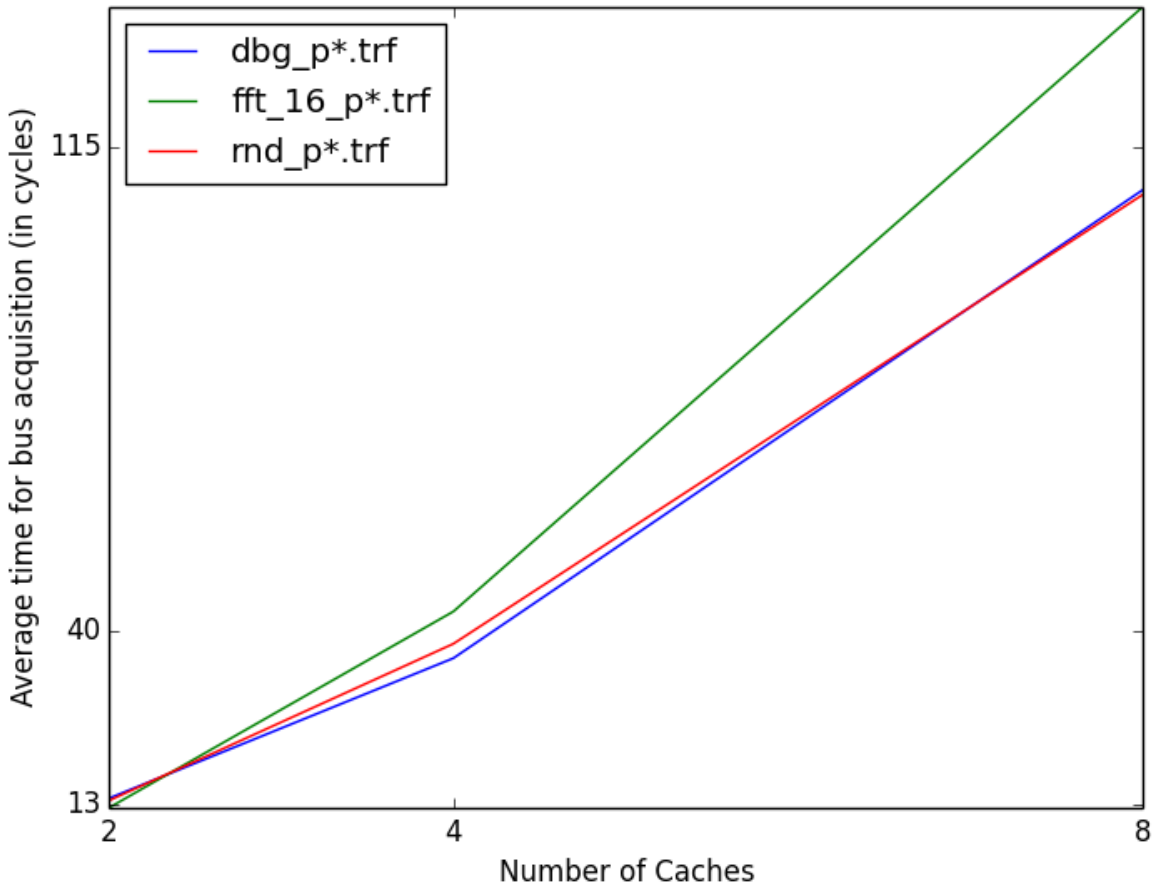
Below there are the experiments that I performed using various configurations. I completely excluded from the runs the files that are using one processor because it does not make any sense for the nature of this lab. However, I crosschecked the results with the previous lab and the hit rate remains the same. Except from the main experiment where all the caches having the snooping protocol active, I have performed some others with the caches performing minimal snooping (in some caches its active) or no-snooping at all. Snooping protocol helps for the integrity of the system. Is snooping is disabled the caches are not coherent anymore.

Snooping Active

For the main experiment “Snooping Active” except from the hit rate I have also captured some other metrics for which I am going to explain their meaning and how I calculated them.

- **Requests on BUS**: The name is already explanatory enough, the metric shows the number of requests placed on the bus (PROBE_READ, PROBE_WRITE, PROBE_READX). As I have already explained in a previous paragraph, PROBE_READX is an action that takes place in 2 parts, the fetch (read from memory) and the write (replace data in the memory) and that is why I decided to capture the requests separately (even though in practice its one). However, as you will see in the screenshots below ReadsX_Fetch and ReadsX_Write must always have the same value. The number of requests on the bus can also be calculated from the table which shows the stats of each cache.
- **Bus contention**: This metric is related to the number of cycles that a cache has to wait in order to get access granted to the bus. There are multiple reasons that a cache has to wait. One is due to ready responses from the main memory (they have priority), the other is because the bus is already occupied by another cache and lastly because there are conflicting requests for the same block. In order to calculate the average time for bus acquisition I summed all the waiting cycles (regardless the reason) and I divided the result with the total number of requests on the bus.
- **Main memory access rates**: In order to find the result for this metric, I first calculated the total number of requests that were placed on the bus. Then I calculated the total number of cycles that the experiment needed in order to finish and I divided those numbers with the latter being the denominator.
- **Total main memory responses**: The main memory practically responds in two cases. When a PROBE_READ request is placed on the bus, or the fetching phase of a PROBE_READX request. Thus, I summed the total number of those types of requests. The total number of the main memory responses can also be calculated from the table which shoes the stats of each cache.
- **Total time**: For that metric I have calculated the “sys” and the “real” time. The “sys” time is practically the number of cycles needed for the experiment, while the “real” one is the actual duration of the experiment.

What we can observe from the results, regardless the trace file, is that as the number of processors-caches increases, so does the "Total waits due to priority for MEM responses" and the "Total waits for BUS locks"; thus, the average time for BUS acquisition. The graph below shows the relationship between the total number of caches (x-axis) with the average time for BUS acquisition (y-axis). Regardless the trace file, as the number of caches doubles, the average time for the BUS acquisition tends to triple.



From the next page and on you will find the results for each one of the various trace files.

dbg_p2.trf

```
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       30      0       30      25      1       24      1.818182
1       32      0       32      35      1       34      1.492537

** Requests on BUS **
- Reads: 62
- ReadsX_Fetch: 58
Total (Reads): 120
- Writes: 2
- ReadsX_Write: 58
Total (Writes): 60
Total (All): 180

** BUS contention (for caches-only) **
Total waits due to priority for MEM responses: 0 cycles
Total waits for BUS locks: 2516 cycles
Total waits for CONFLICTING requests (same tag): 0 cycles
Total waits: 2516 cycles
Average time for BUS acquisition: 13.9778 cycles


** Main memory access rates **
- Reads: 0.0097166
- Writes: 0.0048583
Total rate: 0.0145749

Total main memory responses: 120


** Total time **
Total simulation time (sys) 12350 ns
Total simulation time (real) 0.0102146 sec
```

(continues to the next page)

dbg_p4.trf

```
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       8       0       8       8       1       7       6.250000
1       27      0       27      32      0       32      0.000000
2       43      1       42      38      2       36      3.703704
3       45      0       45      42      0       42      0.000000

** Requests on BUS **
- Reads: 122
- ReadsX_Fetch: 117
Total (Reads): 239
- Writes: 3
- ReadsX_Write: 117
Total (Writes): 120
Total (All): 359

** BUS contention (for caches-only) **
Total waits due to priority for MEM responses: 1393 cycles
Total waits for BUS locks: 11426 cycles
Total waits for CONFLICTING requests (same tag): 0 cycles
Total waits: 12819 cycles
Average time for BUS acquisition: 35.7075 cycles


** Main memory access rates **
- Reads: 0.0128061
- Writes: 0.00642983
Total rate: 0.0192359

Total main memory responses: 239


** Total time **
Total simulation time (sys) 18663 ns
Total simulation time (real) 0.0199482 sec
```

(continues to the next page)

dbg_p8.trf

```
ALL RIGHTS RESERVED
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       6       0       6       4       0       4       0.000000
1       34      0       34      22      0       22      0.000000
2       35      0       35      43      0       43      0.000000
3       39      2       37      46      2       44      4.705882
4       36      0       36      55      0       55      0.000000
5       52      0       52      47      0       47      0.000000
6       48      3       45      51      2       49      5.050505
7       42      1       41      55      5       50      6.185567

** Requests on BUS **
- Reads: 286
- ReadsX_Fetch: 314
Total (Reads): 600
- Writes: 9
- ReadsX_Write: 314
Total (Writes): 323
Total (All): 923

** BUS contention (for caches-only) **
Total waits due to priority for MEM responses: 21810 cycles
Total waits for BUS locks: 74530 cycles
Total waits for CONFLICTING requests (same tag): 3611 cycles
Total waits: 99951 cycles
Average time for BUS acquisition: 108.289 cycles


** Main memory access rates **
- Reads: 0.016638
- Writes: 0.0089568
Total rate: 0.0255948

Total main memory responses: 600


** Total time **
Total simulation time (sys) 36062 ns
Total simulation time (real) 0.066283 sec
```

(continues to the next page)

fft_16_p2.trf

```
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       29313   3423    25890   15417   2013    13404   12.152918
1       29262   3176    26086   14801   1897    12904   11.513061

** Requests on BUS **
- Reads: 51976
- ReadsX_Fetch: 26308
Total (Reads): 78284
- Writes: 3910
- ReadsX_Write: 26308
Total (Writes): 30218
Total (All): 108502

** BUS contention (for caches-only) **
Total waits due to priority for MEM responses: 0 cycles
Total waits for BUS locks: 1356840 cycles
Total waits for CONFLICTING requests (same tag): 0 cycles
Total waits: 1356840 cycles
Average time for BUS acquisition: 12.5052 cycles


** Main memory access rates **
- Reads: 0.0113981
- Writes: 0.00439972
Total rate: 0.0157978

Total main memory responses: 78284


** Total time **
Total simulation time (sys) 6868171 ns
Total simulation time (real) 6.93317 sec
```

(continues to the next page)

fft16_p4.trf

```
ALL RIGHTS RESERVED
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       11274   1752    9522    6553    948     5605    15.145566
1       9417    1464    7953    6083    759     5324    14.341935
2       9834    1527    8307    5523    613     4910    13.935013
3       9881    1510    8371    5972    703     5269    13.959503

** Requests on BUS **
- Reads: 34153
- ReadsX_Fetch: 21108
Total (Reads): 55261
- Writes: 3023
- ReadsX_Write: 21108
Total (Writes): 24131
Total (All): 79392

** BUS contention (for caches-only) **
Total waits due to priority for MEM responses: 476616 cycles
Total waits for BUS locks: 2932204 cycles
Total waits for CONFLICTING requests (same tag): 995 cycles
Total waits: 3409815 cycles
Average time for BUS acquisition: 42.9491 cycles


** Main memory access rates **
- Reads: 0.0154813
- Writes: 0.00676029
Total rate: 0.0222416

Total main memory responses: 55261


** Total time **
Total simulation time (sys) 3569524 ns
Total simulation time (real) 6.00243 sec
```

(continues to the next page)

fft16_p8.trf

```
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       4956    1108    3848    3154    746     2408    22.860666
1       3983    824     3159    2890    546     2344    19.933071
2       3997    868     3129    2703    502     2201    20.447761
3       4043    859     3184    2695    501     2194    20.184031
4       4055    852     3203    2662    475     2187    19.755843
5       4055    848     3207    2733    513     2220    20.050088
6       4074    828     3246    2710    489     2221    19.413325
7       4124    832     3292    2705    478     2227    19.182896

** Requests on BUS **
- Reads: 26268
- ReadsX_Fetch: 18002
Total (Reads): 44270
- Writes: 4250
- ReadsX_Write: 18002
Total (Writes): 22252
Total (All): 66522

** BUS contention (for caches-only) **
Total waits due to priority for MEM responses: 1833868 cycles
Total waits for BUS locks: 7122757 cycles
Total waits for CONFLICTING requests (same tag): 0 cycles
Total waits: 8956625 cycles
Average time for BUS acquisition: 134.642 cycles


** Main memory access rates **
- Reads: 0.0178051
- Writes: 0.0089496
Total rate: 0.0267547

Total main memory responses: 44270


** Total time **
Total simulation time (sys) 2486367 ns
Total simulation time (real) 5.48076 sec
```

(continues to the next page)

rnd_p2.trf

```
ALL RIGHTS RESERVED
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       16496   385     16111   16402   393     16009   2.364885
1       24556   865     23691   24804   877     23927   3.529173

** Requests on BUS **
- Reads: 39802
- ReadsX_Fetch: 39936
Total (Reads): 79738
- Writes: 1270
- ReadsX_Write: 39936
Total (Writes): 41206
Total (All): 120944

** BUS contention (for caches-only) **
Total waits due to priority for MEM responses: 0 cycles
Total waits for BUS locks: 1646748 cycles
Total waits for CONFLICTING requests (same tag): 0 cycles
Total waits: 1646748 cycles
Average time for BUS acquisition: 13.6158 cycles


** Main memory access rates **
- Reads: 0.0090674
- Writes: 0.00468574
Total rate: 0.0137531

Total main memory responses: 79738


** Total time **
Total simulation time (sys) 8793917 ns
Total simulation time (real) 6.55331 sec
```

(continues to the next page)

rnd_p4.trf

```
ALL RIGHTS RESERVED
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       8039    85      7954    8251    89      8162    1.068140
1       20450   0       20450   20221   0       20221   0.000000
2       26553   982     25571   26498   1056    25442   3.841586
3       29600   0       29600   29695   3       29692   0.005059

** Requests on BUS **
- Reads: 83575
- ReadsX_Fetch: 83517
Total (Reads): 167092
- Writes: 1148
- ReadsX_Write: 83517
Total (Writes): 84665
Total (All): 251757

** BUS contention (for caches-only) **
Total waits due to priority for MEM responses: 1097467 cycles
Total waits for BUS locks: 8455248 cycles
Total waits for CONFLICTING requests (same tag): 99 cycles
Total waits: 9552814 cycles
Average time for BUS acquisition: 37.9446 cycles


** Main memory access rates **
- Reads: 0.0126363
- Writes: 0.00640279
Total rate: 0.0190391

Total main memory responses: 167092


** Total time **
Total simulation time (sys) 13223136 ns
Total simulation time (real) 15.3988 sec
```

(continues to the next page)

rnd_p8.trf

```
CPU     Reads   RHit    RMiss   Writes  WHit    WMiss   Hitrate
0       4113    22      4091    4030    35      3995    0.699988
1       18318   0       18318   18440   0       18440   0.000000
2       25834   21      25813   25470   30      25440   0.099407
3       29549   1275    28274   28801   1180    27621   4.207369
4       31366   2       31364   30548   1       30547   0.004845
5       32015   11      32004   31782   26      31756   0.057996
6       32327   1625    30702   32285   1653    30632   5.073361
7       32536   1687    30849   32584   1771    30813   5.310197

** Requests on BUS **
- Reads: 201415
- ReadsX_Fetch: 199244
Total (Reads): 400659
- Writes: 4696
- ReadsX_Write: 199244
Total (Writes): 203940
Total (All): 604599

** BUS contention (for caches-only) **
Total waits due to priority for MEM responses: 12254848 cycles
Total waits for BUS locks: 52457503 cycles
Total waits for CONFLICTING requests (same tag): 314567 cycles
Total waits: 65026918 cycles
Average time for BUS acquisition: 107.554 cycles


** Main memory access rates **
- Reads: 0.0174149
- Writes: 0.00886437
Total rate: 0.0262792

Total main memory responses: 400659


** Total time **
Total simulation time (sys) 23006723 ns
Total simulation time (real) 40.5983 sec
```

(continues to the next page)

Snooping Inactive

In this experiment I deactivated snooping for all the caches in the system. Below, I am reporting only the results for the files *fft_16_p4.trf, fft_16_p8.trf* and *rnd_p8.trf*. In all the other files the hit rate does not change at all compared to the "Snooping Active" experiment. For each one of the results of the current experiment there is a small screenshot on the right side, with the hit rate of the same trace file and snooping being active. We can observe that the hit rate for some caches either does not change, or it does not change significantly. In some of the cases the hit rate for each cache in this experiment is higher than the "Snooping Active" experiment while in other cases its lower or the same. The deviation however, is pretty small and you can barely tell the difference. My expectations were that the hit rate would be really high since there is not any invalidation mechanism between the caches and more cache lines would have valid data even though they wouldn't be coherent. The reason that we might not observe this result, is probably because the CPUs do not reuse the same blocks of memory and the cache lines are getting replaced very often.

fft_16_p4.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	11274	1752	9522	6553	945	5608	15.128737
1	9417	1463	7954	6083	758	5325	14.329032
2	9834	1527	8307	5523	613	4910	13.935013
3	9881	1510	8371	5972	704	5268	13.965811

Hitrate
15.145566
14.341935
13.935013
13.959503

fft_16_p8.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4956	1107	3849	3154	742	2412	22.799014
1	3983	822	3161	2890	544	2346	19.874873
2	3997	868	3129	2703	502	2201	20.447761
3	4043	859	3184	2695	501	2194	20.184031
4	4055	852	3203	2662	476	2186	19.770731
5	4055	847	3208	2733	513	2220	20.035357
6	4074	828	3246	2710	490	2220	19.428066
7	4124	833	3291	2705	478	2227	19.197540

Hitrate
22.860666
19.933071
20.447761
20.184031
19.755843
20.050088
19.413325
19.182896

rnd_p8.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4113	22	4091	4030	35	3995	0.699988
1	18318	0	18318	18440	0	18440	0.000000
2	25834	21	25813	25470	30	25440	0.099407
3	29549	1275	28274	28801	1180	27621	4.207369
4	31366	2	31364	30548	1	30547	0.004845
5	32015	12	32003	31782	26	31756	0.059564
6	32327	1626	30701	32285	1653	30632	5.074909
7	32536	1687	30849	32584	1771	30813	5.310197

Hitrate
0.699988
0.000000
0.099407
4.207369
0.004845
0.057996
5.073361
5.310197

Minimal Snooping

Regarding the minimal snooping, I kept snooping active only to the half of the caches. I have performed four different experiments. In the first experiment, “Snooping half odd”, snooping is active for the caches that their ID is an odd number (IDs start from zero and its incremented by one). In the second experiment, “Snooping half even”, I kept snooping active for the caches with an even number of ID. In the third experiment, “Snooping first half”, I kept snooping active for the first half of the caches. Finally, in the last experiment “Snooping last half”, I kept snooping active for the last half of the caches. Similar to “Snooping inactive”, the only files that there is a difference in the hit rate are: *fft_16_p4.trf, fft_16_p8.trf and rnd_p8.trf*. In all the other files, the hit rate does not change at all compared always with the “Snooping active”. For each one of the results of the following experiments there is a small screenshot on the right side, with the hit rate of the same trace file and snooping being active. As you will observe from all the results, the hit rate follows the trend of “Snooping inactive” experiment. In some cases it is higher, in other the same or lower. The difference is pretty small to tell. This is happening for all the four experiments and thus, I am not discussing them separately. In case we compare the minimal snooping with no-snooping at all, we can still see that the results do not have significant difference and remain more or less the same or exactly the same (e.g: rnd_p8). The same can be observed if we cross-compare the results of all the other experiments (odd, even, first half, second half).

Snooping half odd

fft_16_p4.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	11274	1752	9522	6553	948	5605	15.145566
1	9417	1463	7954	6083	758	5325	14.329032
2	9834	1527	8307	5523	613	4910	13.935013
3	9881	1510	8371	5972	704	5268	13.965811

Hitrate
15.145566
14.341935
13.935013
13.959503

fft_16_p8.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4956	1108	3848	3154	746	2408	22.860666
1	3983	822	3161	2890	544	2346	19.874873
2	3997	868	3129	2703	502	2201	20.447761
3	4043	859	3184	2695	501	2194	20.184031
4	4055	852	3203	2662	476	2186	19.770731
5	4055	847	3208	2733	513	2220	20.035357
6	4074	828	3246	2710	489	2221	19.413325
7	4124	833	3291	2705	478	2227	19.197540

Hitrate
22.860666
19.933071
20.447761
20.184031
19.755843
20.050088
19.413325
19.182896

rnd_p8.trf

For this trace file the results were exactly the same as in the experiment “Snooping inactive”.

Snooping half even

fft_16_p4.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	11274	1752	9522	6553	945	5608	15.128737
1	9417	1463	7954	6083	758	5325	14.329032
2	9834	1527	8307	5523	613	4910	13.935013
3	9881	1510	8371	5972	704	5268	13.965811

Hitrate
15.145566
14.341935
13.935013
13.959503

fft_16_p8.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4956	1107	3849	3154	742	2412	22.799014
1	3983	822	3161	2890	544	2346	19.874873
2	3997	868	3129	2703	502	2201	20.447761
3	4043	859	3184	2695	501	2194	20.184031
4	4055	852	3203	2662	476	2186	19.770731
5	4055	847	3208	2733	513	2220	20.035357
6	4074	828	3246	2710	490	2220	19.428066
7	4124	833	3291	2705	478	2227	19.197540

Hitrate
22.860666
19.933071
20.447761
20.184031
19.755843
20.050088
19.413325
19.182896

rnd_p8.trf

For this trace file the results were exactly the same as in the experiment “Snooping inactive”.

Snooping first half

fft_16_p4.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	11274	1753	9521	6553	948	5605	15.151175
1	9417	1464	7953	6083	759	5324	14.341935
2	9834	1527	8307	5523	613	4910	13.935013
3	9881	1510	8371	5972	704	5268	13.965811

Hitrate
15.145566
14.341935
13.935013
13.959503

fft_16_p8.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4956	1108	3848	3154	746	2408	22.860666
1	3983	824	3159	2890	546	2344	19.933071
2	3997	868	3129	2703	502	2201	20.447761
3	4043	859	3184	2695	501	2194	20.184031
4	4055	852	3203	2662	476	2186	19.770731
5	4055	847	3208	2733	513	2220	20.035357
6	4074	828	3246	2710	490	2220	19.428066
7	4124	833	3291	2705	478	2227	19.197540

Hitrate
22.860666
19.933071
20.447761
20.184031
19.755843
20.050088
19.413325
19.182896

rnd_p8.trf

For this trace file the results were exactly the same as in the experiment “Snooping inactive”.

Snooping last half

fft_16_p4.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	11274	1752	9522	6553	945	5608	15.128737
1	9417	1463	7954	6083	758	5325	14.329032
2	9834	1527	8307	5523	613	4910	13.935013
3	9881	1510	8371	5972	705	5267	13.972119

Hitrate
15.145566
14.341935
13.935013
13.959503

fft_16_p8.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4956	1107	3849	3154	742	2412	22.799014
1	3983	822	3161	2890	544	2346	19.874873
2	3997	868	3129	2703	502	2201	20.447761
3	4043	859	3184	2695	501	2194	20.184031
4	4055	852	3203	2662	475	2187	19.755843
5	4055	848	3207	2733	513	2220	20.050088
6	4074	828	3246	2710	489	2221	19.413325
7	4124	832	3292	2705	478	2227	19.182896

Hitrate
22.860666
19.933071
20.447761
20.184031
19.755843
20.050088
19.413325
19.182896

rnd_p8.trf

CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
0	4113	22	4091	4030	35	3995	0.699988
1	18318	0	18318	18440	0	18440	0.000000
2	25834	21	25813	25470	30	25440	0.099407
3	29549	1275	28274	28801	1180	27621	4.207369
4	31366	2	31364	30548	1	30547	0.004845
5	32015	11	32004	31782	26	31756	0.057996
6	32327	1624	30703	32285	1653	30632	5.071813
7	32536	1687	30849	32584	1771	30813	5.310197

Hitrate
0.699988
0.000000
0.099407
4.207369
0.004845
0.057996
5.073361
5.310197