Multi-Core Processor Lab Report 1 + 2

Maximilian Kamps (jjb140)

11.02.2025

1 Introduction

This document serves as a report for the Lab project in the Multi-Core Processor Systems course at the University of Amsterdam. It documents the behavior of various levels of cache complexity simulated using SystemC.

2 Assignment 1: Single processor cache

Assignment 1 involves simulating a single 32kiB, 8-way set-associative L1 D-cache with a 32-Byte line size. Consequently, the cache has 128 8-way sets. The cache employs a least-recently-used write-back replacement strategy and an allocate-on-write policy. Cache access is modeled with a 1-cycle delay, while main memory access incurs a 100-cycle delay. Section 3.1 details the abstraction of the cache for simulation, and Section 2.2 evaluates its behavior across various memory access traces.

2.1 Design

Figure 1 illustrates the design of an L1 cache for a single processor. To implement cache behavior, all wires from the CPU to memory are rerouted through the cache, effectively placing it between the two. An additional wire connects the newly added *Status* and *CacheStatus* ports, allowing the cache to inform the CPU whether a requested read or write resulted in a hit or miss. Access to main memory during a miss is modeled implicitly; whenever the cache needs to read or write to main memory, it waits for 100 cycles.

When the cache receives a new address from the CPU, it first computes the block address by dividing the address by the line size (32). The block address is then used to find the set where the address should be inserted by taking its modulo with the cache associativity (8). Finally, the address is either inserted into a new cache line (in case of a miss), which may cause eviction of a different cache line, or, in case of a hit, the existing cache line gets updated. A cache line consists of a dirty bit (set when the cache line has been written to), a valid bit (indicating if the cache line is valid), a timestamp using the simulation time (for LRU eviction), and a tag corresponding to the block address.

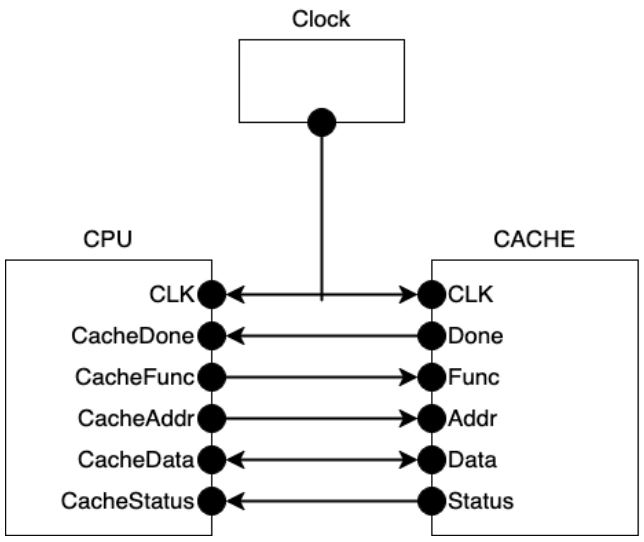

Figure 1: Design of a single processor cache

2.1.1 CLI commands

The command line structure of the assignment has been slightly modified. After the trace file, the user can specify the verbosity (0 for no logging and 1 for logging).

2.2 Simulation

This section presents the results of the cache simulation, evaluating its performance in various scenarios. It compares cache access times, hit rates, and memory delay reductions under different conditions, such as optimization levels and cache sizes. The experiments focus on understanding the impact of cache memory on system performance and highlight the key metrics that define its efficiency.

2.2.1 Cache vs direct memory access

This section explores the reduction in memory delay, when introducing a cache between the main memory and the CPU. Figure 2 compares the memory delay reduction (%) when introducing a cache for different trace files, with the *-O2* compiler optimization enabled and disabled. Using a cache significantly reduces memory delay, achieving reductions of approximately 98.0%-95.3% when *-O2* is disabled compared to direct main memory access without a cache. With *-O2* enabled, the reduction ranges from approximately 98.0%-82.0%. Consequently, utilizing a cache greatly reduced memory delay.

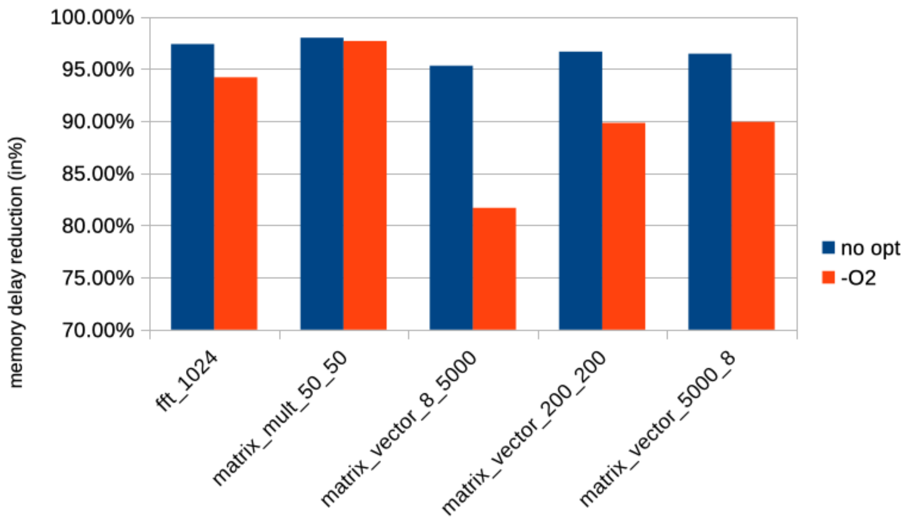

Figure 2: Memory delay reduction by using a Cache (in %) for different traces and optimization levels

2.2.2 Cache hit rate

Figure 3 shows the combined read/write cache hit rate (%) for different trace files, with and without the *-O2* compiler optimization. The chart indicates that enabling *-O2* reduces the hit rate across all traces by approximately 3.2% to 13.6%. Section 2.2.4 investigates the reduction in cache hits caused by the optimization.

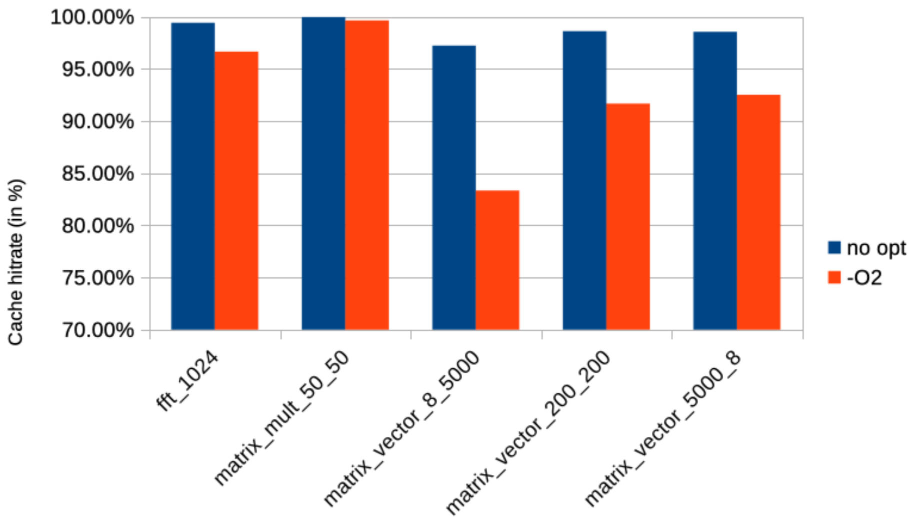

Figure 3: Cache hit rate (in %) for different traces and optimization levels

2.2.3 Absolute memory delay of the Cache

Figure 4 illustrates the total memory delay (in ns). Enabling *-O2* optimizations significantly reduces memory delay across all traces, with reductions ranging from 35% to 84%. The most substantial reduction is observed in the matrix multiplication trace (*matrix_mult_50_50*). Section 2.2.4 investigates the reduction in memory delay caused by the optimization.

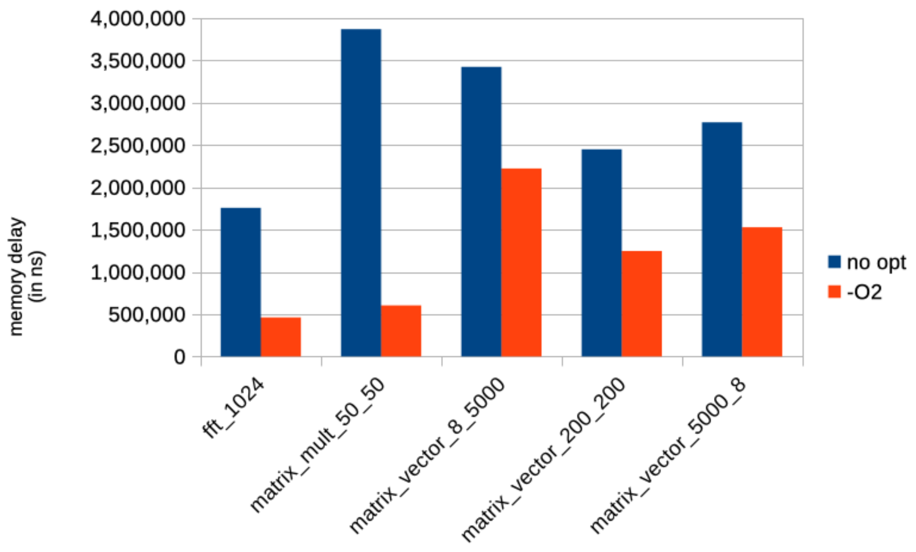

Figure 4: Memory delay (in ns) for different traces and optimization levels

2.2.4 Consequences of optimizing with -O2

To investigate the improved memory delay when enabling *-O2*, Figure 5 shows the number of reads and writes in the trace files with and without *-O2* enabled.

Trace files without optimization exhibit significantly more reads than writes. Comparing across optimizations, the *-O2* files contain substantially fewer reads and writes. The most notable difference is observed in the trace (*matrix_mult_50_50*), where the optimized file contains only 15% of the reads and 1% of the writes compared to its non-optimized counterpart. Consequently, the gap in memory delay shown in Figure 4 is explained by simulations using *-O2* traces, which perform significantly fewer reads and writes while maintaining a similar cache hit rate, as shown in Figure 3. In conclusion, *-O2* reduces the number of memory operations required while trying to preserve caching accuracy.

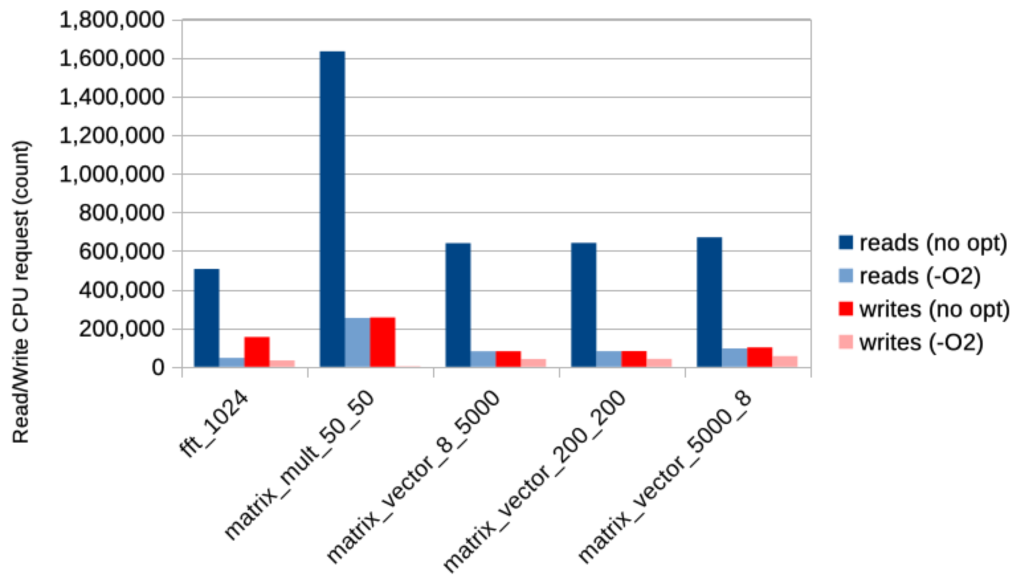

Figure 5: Number of reads and writes in different traces with different optimization levels

2.2.5 Cache size vs hit rate

Figure 6 plots the effect of different cache sizes (in B) on the cache hit rate (in %) for the fast furrier transformation and matrix multiplication traces, with the `-O2` optimization enabled and disabled. It is expected that increasing the cache size should also increase

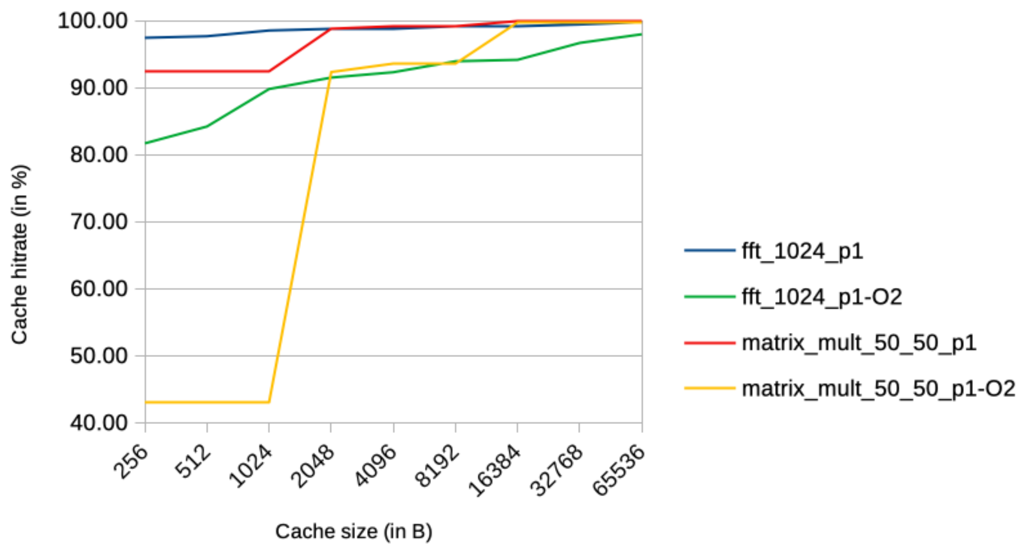

Figure 6: Cache hits (in %) for different traces and cache sizes (in B)

the hit rate. This is because the cache evicts lines less frequently, leading to fewer costly memory swaps between main memory and the cache. The plot supports this hypothesis. For the matrix multiplication transformation without optimizations, increasing the cache size causes a gradual rise in cache hits from around 81.66% to 97.94%. When considering the same algorithm with optimizations enabled, the increase is far more dramatic. Here, the cache registers only 43.05% hits before increasing the cache from 1024B to 2048B, which causes a spike to 92.3%. Further increasing the cache size gradually raises the accuracy to 99.63%. The difference caused by optimization levels is likely due to memory access, with `-O2` being far less predictable, leading to a significantly higher number of evictions compared to its unoptimized counterpart. Consequently, cache hit rates remain relatively stable for the unoptimized trace, even for small caches. In conclusion, when running an algorithm that requires substantial memory on a very small cache, it can be beneficial to disable optimizations.

3 Assignment 2: Implement a multiprocessor cache system

This section covers Assignment 2, where the task was to implement a multi-core processor system with L1 caches per CPU, connected via a split transaction bus. First, the design of the implementation is presented, followed by an evaluation of the simulation results, which are then compared to the single cache system from Assignment 1 in Section 2.

3.1 Design

Figure 7 illustrates a multi-core processor system, where multiple CPUs, each with their own L1 cache, are connected via a shared bus. The connections between each CPU and its cache remain unchanged from the design in Assignment 1, except that the wire used to indicate cache hits or misses has been removed. Instead, cache hits and misses are now logged directly within the cache module. Memory access is again modeled implicitly.

Bus access follows a round-robin scheme, managed by a `Lock`, ensuring that each response triggered by a bus request is processed by all caches before the next request is handled. Initially, CPU 0 gains bus access, issuing memory read requests (on cache misses) and writes via function calls on the bus. The bus then broadcasts responses to **all** caches using four dedicated wires:

- `Func`: Specifies whether the request is a read or a write.
- `Addr`: The memory address being accessed.
- `Cache_ID`: The ID of the requesting cache.
- `Trans_ID`: A unique identifier for each bus request.

All caches register the response. Caches that did not initiate the request "snoop" the `Addr` to detect and invalidate local cache blocks if the request corresponds to a memory write. Once processing is complete, the current cache releases the lock, passing bus access to the next cache in sequence.

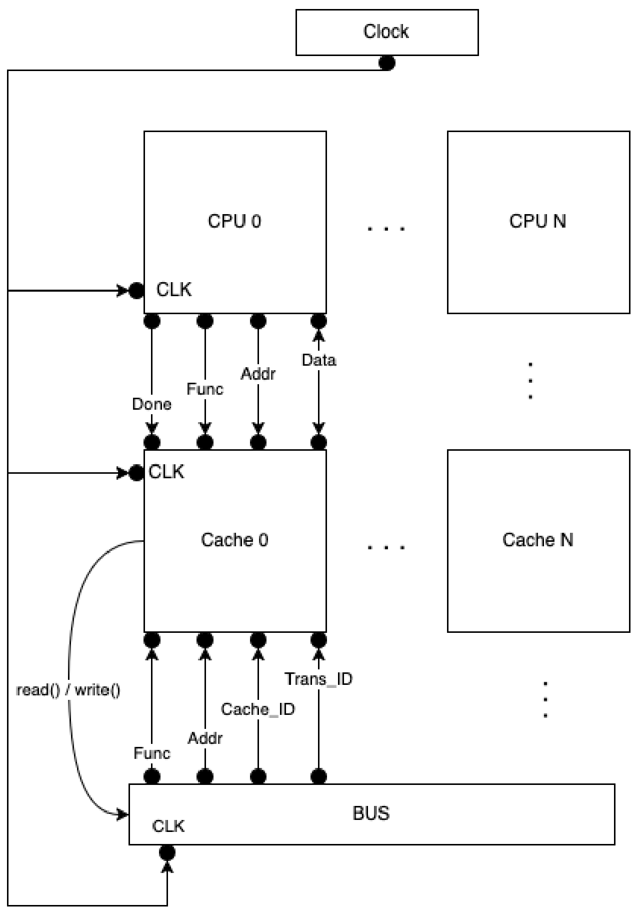

Figure 7: Design of a multiprocessor cache system with a bus

3.1.1 Design details

This section covers some implementation details of the bus.

NOP instructions

To ensure that every processor registers every bus request, the CPU additionally passes all NOP requests to its Cache. When a Cache receives a NOP request from the CPU and it acquires the Bus `Lock`, it simply does nothing. Passing the NOP request is needed, in order for the Cache to register all request on the Bus for local invalidation, even though it does not currently have any read or write operations to execute. Essentially, each cache must complete its operation before all CPUs proceed to fetch their next instruction.

Simulating a split transaction bus

To simulate the behavior of a split transaction bus, bus acquisition wait times are logged as follows. While a cache does not hold the lock, it tracks the number of bus acquisitions by other processors by counting the number of snooped addresses. The bus acquisition time for the cache is then recorded as the number of preceding bus acquisitions when it eventually acquires the lock and issues a memory request to the bus.

Write through cache

When a cache writes to a local address, it immediately writes the updated data to the bus, allowing all other caches to snoop the address and invalidate any potential occurrences of that data in their cache blocks.

3.1.2 CLI commands

The command line structure of the assignment has been slightly modified. After the trace file, the user can specify the verbosity (0 for no logging and 1 for logging).

Note: The implementation is relatively slow, and the simulation may take up to 30 seconds to terminate for some trace files.

3.2 Simulation

This section explores the effect of a bus on caching behavior when scaling to multiple processors. All examples were tested with an 8-way set-associative 32KiB cache, using 32B cache lines, and the Fast Fourier Transformation traces with the `-O2` optimization both enabled and disabled.

3.2.1 Acquisition time and average acquisition time

Figure 8 illustrates the total acquisition time (in ns) for 1, 2, 4, and 8 processors. The acquisition time refers to the simulation time required by a cache to obtain the bus after receiving a new instruction from the CPU and requiring a memory request. As the number of processors increases, the total acquisition time also increases for both traces. The increase is much more pronounced for the trace with `-O2` disabled. Specifically, for 8 processors, the total acquisition time for the `fft_1024` trace is 470.743 ns, while for the `fft_1024-O2` trace, it is 182.550 ns.

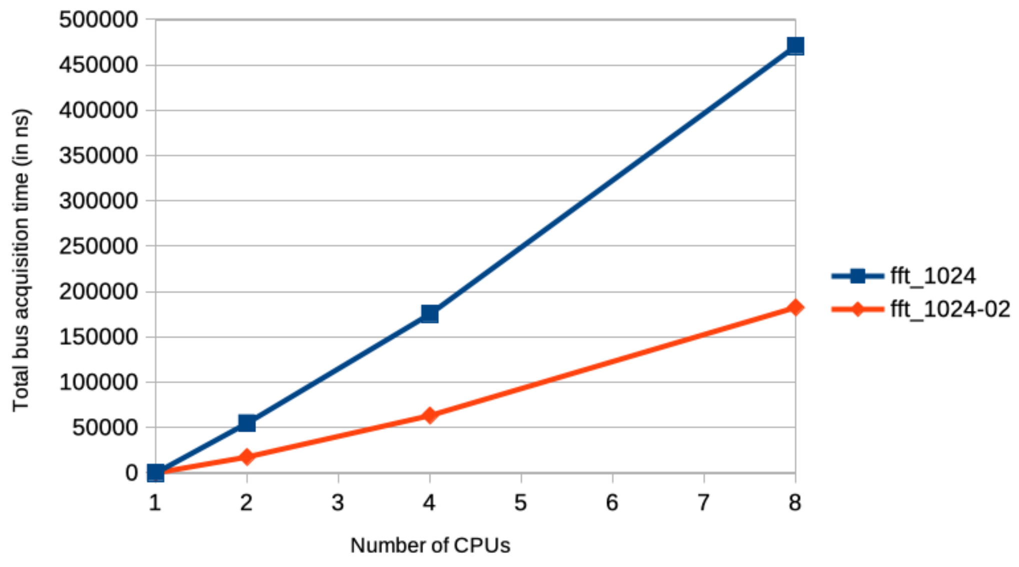

Figure 8: Total acquisition time (in ns) seconds for different number of processors

When analyzing the average acquisition time in Figure 9, there is a noticeable increase in time for both traces. The increase is more pronounced when the `-O2` optimization is disabled, with the average time being approximately 1.3 ns higher for 8 CPUs in the unoptimized trace. To explain this behavior, the next section examines the number of reads and writes to the bus.

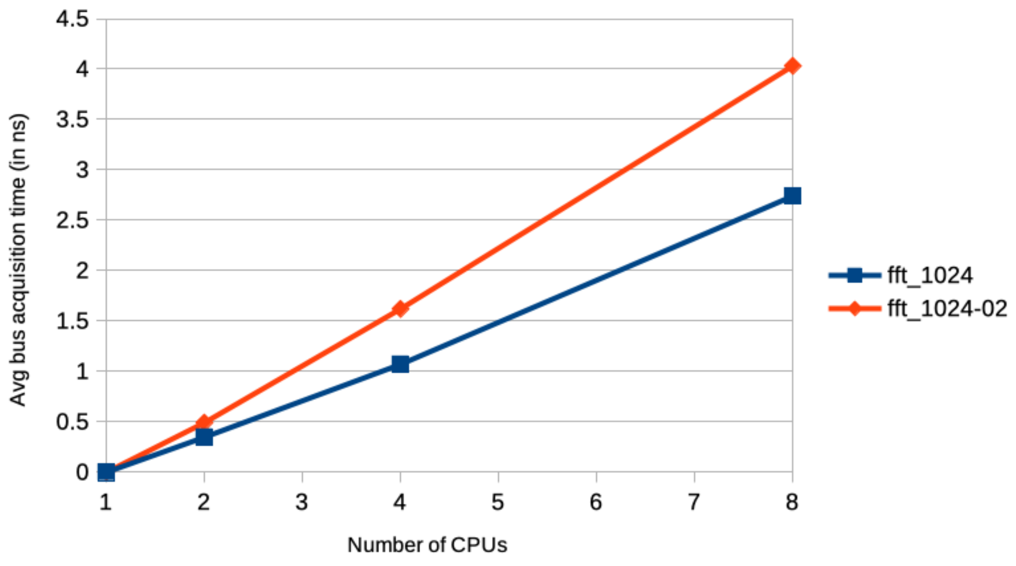

Figure 9: Average acquisition time (in ns) seconds for different number of processors

3.2.2 Number of reads/writes to the bus

Figures 10 and 11 illustrate the number of reads and writes, respectively, issued to the bus for different processor counts. While the number of reads is quite similar across optimization levels, the number of writes shows a significant difference. This outcome aligns with the findings from Section 2.2.4, which concluded that `-O2` reduces the number of memory operations

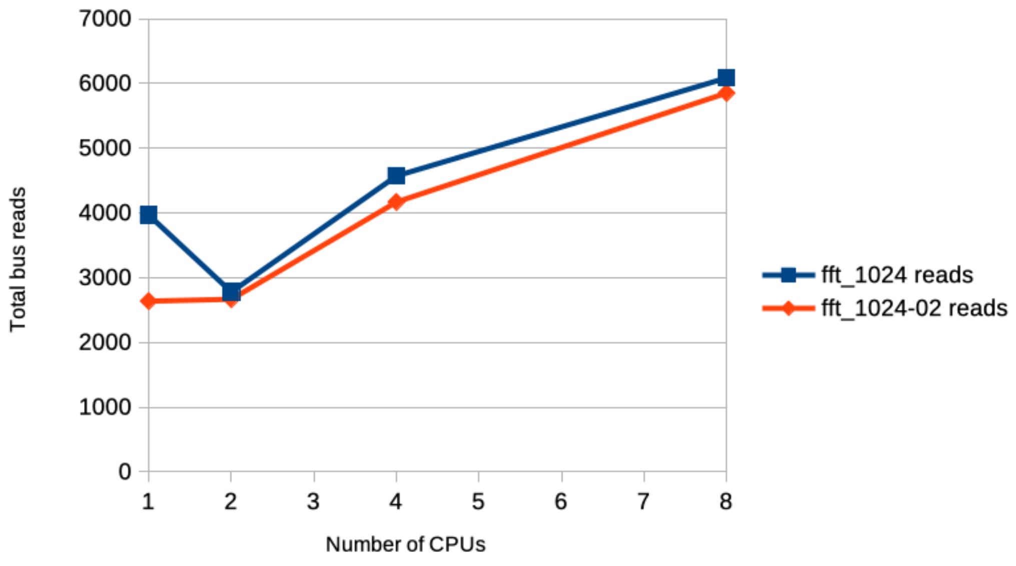

Figure 10: Number of bus reads for different number of processors

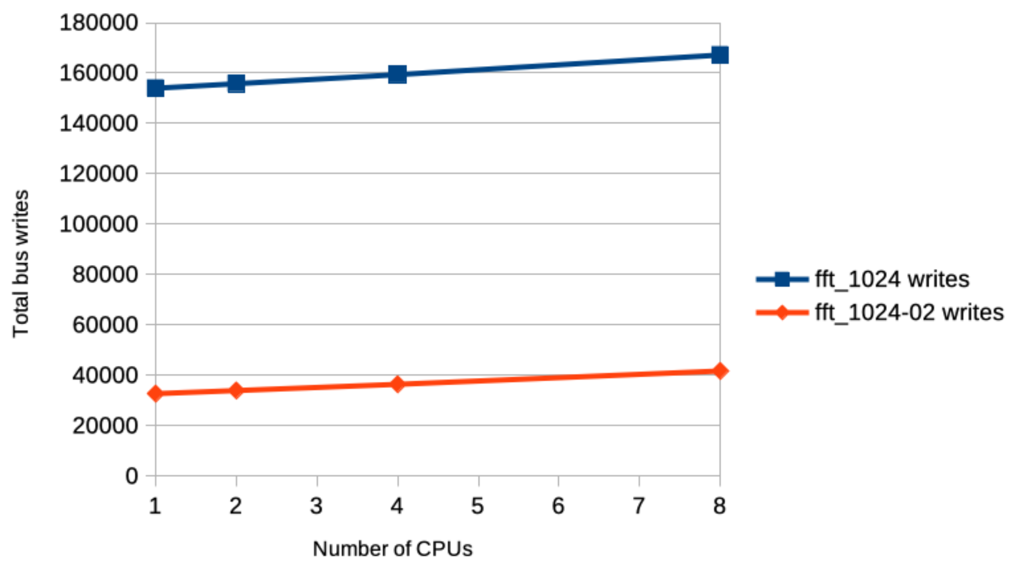

Figure 11: Number of bus writes for different number of processors

The acquisition time results from Section 3.2.1 can now be explained as follows: The optimized trace makes more requests to the bus than the unoptimized trace, leading to a higher total acquisition time. However, it experiences less bus congestion, probably because the bus requests conflict less frequently within the same cycle, leading to a smaller average.

3.2.3 Simulation time

Figure 12 illustrates the total simulation time for the two traces with different numbers of processors. When comparing the simulation time for one processor with a write-back cache (as discussed in Section 2.2.3) and the simulation time for one processor with a bus, it is clear that the simulation time is significantly higher for the bus system—approximately 1.7 million ns for the single processor and around 17 million ns for the multicore bus system with one CPU using the unoptimized trace. This difference arises because the bus system triggers a write-back to memory for every write request to the cache (making it a **write-through** system), while in the single cache system, writes to memory only occur on a cache miss, when insertion requires eviction (making it a **write-back** system).

Increasing the number of processors only slightly increases the simulation time. This effect correlates with the number of invalidation each cache needs to perform, when snooping a write from the bus. An invalidation leads to a cache having to re-fetch an address from memory in later operations to the same address. A larger number of processor for the same trace leads to more invalidation, because more processors share the same address.

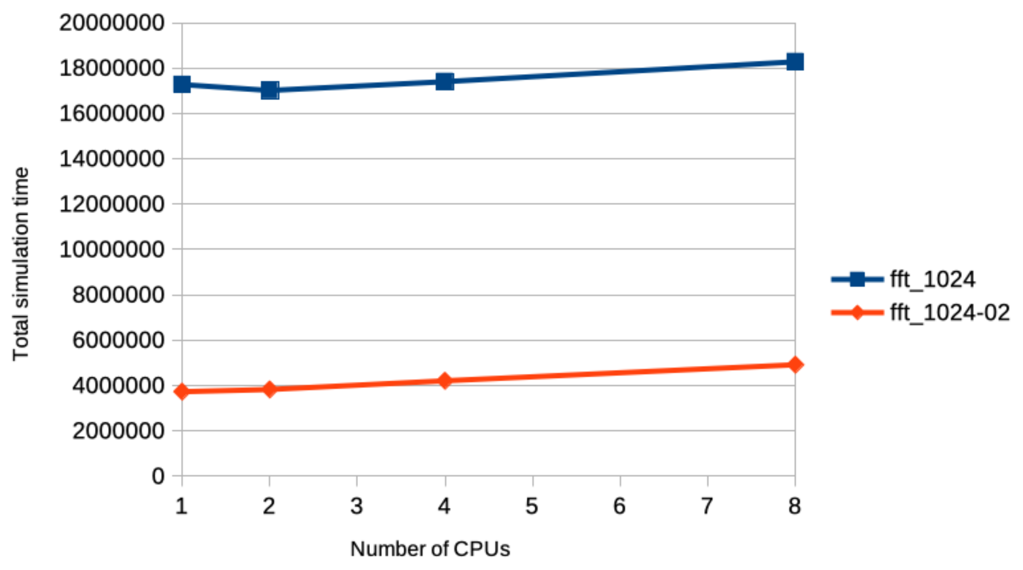

Figure 12: Total simulation (in ns) for different number of processors

3.2.4 Cache hit rate

Figure 13 illustrates the average, minimum, and maximum cache hit rate across different processor counts for the fft_1024 trace, with `-O2` enabled and disabled.

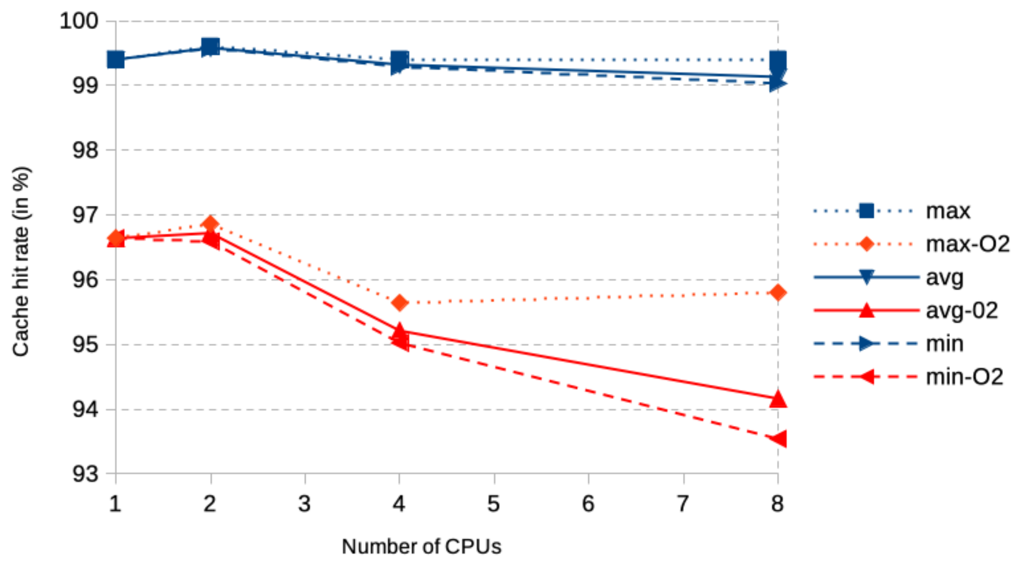

Figure 13: Avg, max and min cache hits (in %) across processors for different number of processors and the fft_1024 trace

For one processor, the cache hit rate is identical to that of the single-core processor discussed in Section 2.2.2. This is because the systems behave in the same way, performing identical evictions, and the bus system does not register any invalidations that would require additional memory fetches.

Scaling to 2 processors slightly improves the cache hit rate, likely due to the increased memory available from having two caches, which results in fewer evictions. This is supported by Figure 10 from Section 3.2.2, which shows the read requests to the bus. For the unoptimized trace, the bus receives around 1,100 fewer reads, suggesting that addresses are re-fetched less often.

However, as the processor count increases further, the hit rate starts to decline. This is likely due to an increase in invalidations as more addresses are shared between processors, requiring more memory fetches.

For the unoptimized trace, the cache hit rates across processors remain relatively close even as the number of processors increases. In contrast, for the optimized trace, the gap between the processor with the highest and lowest hit rate widens significantly as the number of processors grows. Since the optimized trace performs fewer memory operations, there is likely more variance in the need to re-fetch addresses between processors after an invalidation.

3.2.5 Disabling snooping

Disabling snooping for the bus system leads to all caches being in an inconsistent state after the simulation (if the number of caches is greater than 1). This occurs because when one cache writes to memory, other caches holding the same address are not informed of the change, causing those CPUs to work with stale data. Nevertheless, running the simulation with snooping disabled results in an increased cache hit rate, as no invalidations are performed and subsequent accesses to stale addresses never result in a cache miss.

3.2.6 Comparing Single-Core Cache and Multi-Core Bus Cache Architectures

In conclusion, the comparison between the single cache and the multi-core bus cache systems reveals significant differences in performance. While the single processor cache efficiently reduces memory delays, the multi-core bus system introduces complexities such as increased acquisition times and cache invalidation's as the number of processors grows. Although the multi-core system improves cache hit rates for small processor counts, this advantage diminishes as more processors are added, largely due to the overhead of maintaining consistency across caches. Additionally, optimizations like the *-O2* compiler level reduce memory operations, improving performance, but also contribute to higher bus contention. Overall, while the multi-core bus system offers scalability, it comes at the cost of increased simulation time and bus-related delays.