International Letters of Social and Humanistic Sciences
ISSN: 2300-2697, Vol. 13, pp 75-87
doi:10.18052/www.scipress.com/ILSHS.13.75
CC BY 4.0. Published by SciPress Ltd, Switzerland, 2014

Online: 2013-10-25

A SystemC cache simulator for a multiprocessor shared memory system

Alfred Mutanga

Institutional Planning and Quality Assurance, University of Venda, University Road, Thohoyandou, Limpopo Province, 0950, South Africa

E-mail address: alfred.mutanga@univen.ac.za

RETRACTED

ABSTRACT

In this research we built a SystemC Level-1 data cache system in a distributed shared memory architectural environment, with each processor having its own local cache. Using a set of Fast-Fourier Transform and Random trace files we evaluated the cache performance, based on the number of cache hits/misses, of the caches using snooping and directory-based cache coherency protocols. A series of experiments were carried out, with the results of the experiments showing that the directory-based MOESI cache coherency protocol has a performance edge over the snooping Valid-Invalid cache coherency protocol.

Keywords: Cache Coherency; Cache Simulator; Multiprocessor Architectures

1. INTRODUCTION

Architecturally, computing systems have their memory organized hierarchically and this memory nomenclature is scientifically termed the memory hierarchy (Hennessey and Patterson, 2007; Stalling, 2012). The nearer the memory module is to the processor, the smaller and faster are the components resulting in an inverse relationship between the size and speed of the memory module. However, according to Hennessey and Patterson (2007) fast memory comes with cost implications as these modules are relatively expensive per byte. Altogether the memory modules in a computer system collectively allow the data and instructions to flow through the system. The central processing's unit registers are the most vital as these store the operands and results of all computations capitalizing on the principle of locality (Hennessey and Patterson, 2007).

The computer program and data are typically stored on non-volatile storage such as disk drives and tapes before execution but these are first loaded into main memory, which is much faster, but still significantly slower than the registers (Hennessey and Patterson, 2007, p. 288-299). As an intermediate step in the memory hierarchy, caches were invented to avoid the penalties of memory access by keeping the most recently used data and delivery is much faster to the processor. Cache memories are therefore the conceptual foundation for this research.

This paper is an open access paper published under the terms and conditions of the Creative Commons Attribution license (CC BY)
(https://creativecommons.org/licenses/by/4.0)

2. PROBLEM STATEMENT

As has been observed through various computer architecture research the problems facing the multicore processor systems at large are that, processor speeds are “rising dramatically at approximately 75 % per year”, according to McKee (2004). The memory clock speeds at the same time are increasing steadily at a paltry 7 % per annum (Hennessey and Patterson, 2007). The research by NASA and scientists at the University of Virginia confirm this dilemma, that, there is a divergence in the operating speeds of memory architectures and processor systems, referred to as the Memory Wall (McKee, 2004). The challenge facing computer scientists and engineers today is therefore to design a memory architecture that operates at the same clock speeds as the processor architecture.

The computing industry facing the dilemma of the memory wall resolved that to increase performance on computing systems should be as a result of building latency tolerance prefetching non-blocking cache memory systems (McKee, 2004).

This resulted in the computing industry building processor architectures consisting of larger cache memory systems and more latency tolerance on chip. Memory architectures are organized hierarchically, with the memory components nearer to the processor being smaller and faster (Hennessey and Patterson, 2007). Cache memory systems are there to prevent the penalties of memory access by keeping the most recently or frequently used data and deliver it as fast as is possible to the processor. The memory wall results in memory being considered as the bottleneck for processor performance, and modern computer architectural designs feature different cache memory levels (Hennessey and Patterson, 2007).

Caches exploit the benefits of temporal and spatial locality of the data in the computer’s main memory by having regular access patterns. Typically each memory request goes through the cache memory and subsequently channelled to the main or a higher level cache memory if the requested data or instruction is not found in that cache.

Complications arise when multiple processors with each having a local cache have a shared main memory system. If the various caches keep private copies of shared data while being unaware of what is the state of these copies in the other caches, undefined cache performance behaviour may arise.

Cache coherency protocols are required to maintain the cache consistency of all the data stored in the different local caches (Leiserson and Mirmam, 2008). The cache coherency protocols consist of cache line state transitions that can be captured by cache simulators. However it is not easy to get the actual behaviour of these caches and also to prove the correctness of such cache behaviour. Despite their benefits, multiprocessors can only scale so far and bottlenecks can occur when several CPUs on a board share a single memory system and a bus (Hennessey and Patterson, 2007).

In this research we evaluated the performance of Level 1 data cache memory systems in a multiprocessor environment by looking at the influence of the bus traffic, and cache coherence protocols, number of processors and cache associativity. We addressed the following research questions:

1. To what extend do the number of processors in multiprocessor architectures affect the performance of Level 1 (L1) data cache memory systems?
2. How do cache coherency protocols influence the Level1 data cache memory performances of multiprocessor architectures?

RETRACTED

RETRACTED

3. THEORETICAL FRAMEWORK

The problems that have been identified for uniprocessors have been addressed by the development of multi-core architectures. The real world is parallel, and the reason why single processors have faced problems is that they have been executing instructions sequentially in short bursts of time. The real explanation why chip companies shift to multi-cores is prosaic in the sense that it includes several reasons that are not within the context of this research. There is an inherent concept that multi-cores increase the speeds of execution of multiple tasks, but achieving parallelism is not a trivial task (Nussbaum and Smith, 2002). What are the challenges or problems which multi-core designers face? Let us look into these problems briefly.

3. 1. Programmability

Historically parallel processing computer architectures and multi-cores have presented computer architecture designers and system software developers programming challenges. The programming challenges include intellectual programming skills needed to develop programs for such systems, and the need for specialised software tools to program them. The daunting task for programmers is on the "parallelisation of sequential programs" (Szydlowski, 2005). The multi-core programming model should be based on standard programming tools and programming languages. There are no real standards in the programming landscape of multi-cores (Duller and Towner, 2003; Towner et al., 2004; Jourbet, 2008). Echoing the same sentiments about programming multi-cores (Leiserson and Mirman, 2008) wrote that "multi-core processors are parallel computers and parallel computers are notoriously difficult to program". Chris Jesshope identified 3 different models of machine/programming models which are sequential; ad-hoc parallel and fully parallel models (Jesshope, 2008). Even though these programming models exist there is need to address the issue of standards and automation of multi-core programming tasks (Blyler, 2009).

3. 2. Scalability

Multi-cores reduce system latency but one of the challenges that multi-core systems developers face is developing systems that are scalable. Multi-cores produce tangible benefits but making the processes parallel brings with it programming challenges as mentioned before. Increasing more processor cores on chip might entail that the whole system has to be rewritten (Blyler, 2009; picoChip, 2007). Rewriting code for more cores has a direct implication on production costs, longer marketing times and consumers end up paying for these shortfalls. In the event of increasing more processor cores the programmer has to rethink about the routines to use and in partitioning the processing operations between the individual processors added.

3. 3. Communications

Multi-cores present problems in the communication channels used by the processing elements to communicate between or to each other. PicoChip identified the "saturation of the communications links between processing elements" (Panesar et al., 2005, 2006; picoChip, 2007) as a major drawback especially to multi-cores with more than 10 processors. Race conditions are also "pernicious bugs" (Leiserson and Mirmam, 2008) that are difficult to detect. There is always need to have a reliable and efficient way to eliminate race conditions. Designing the interconnection channels between the various processing elements is crucial in order to achieve higher performance gains. The data or instructional dependencies may cause some of the processors to be idle hence loosing performance gains. The width of the communication channel is an important factor to consider. There is a concern that power dissipation can increase with multiple processing elements operating concurrently.

3. 4. Managing a heterogeneous architecture

Multi-core systems are in most cases constituted by different types of processors and technically the architecture is referred to as a heterogeneous architectures. The heterogeneous architecture is not as easy to program as the homogeneous architecture that consist of similar processing elements. Homogenous architectures are easy to implement on silicon (picoChip, 2007, Hobson et al., 2006). Heterogeneous architectures provide greater yields in execution speeds because they include dedicated processing elements for specific application tasks, some elements are designed to speed up code.

3. 5. Cache Memory Systems

As mentioned earlier processor speeds have been scaling up faster than memory speeds resulting in the memory wall. Computer engineers have seen that both processor and memory clock cycles have been decreasing overs time (processor by about 60 % per year, Moore's Law and the memory by about 7 % per year, Less' law) (Jesshope 2008, Hobson et al., 2006). There have been of course attempts to increase memory bandwidth by introducing concurrency in memory accesses through pipelining (Jesshope 2008, Hobson et al. 2006), but, this requires regular memory access patterns and random access to the main memory bringing with it degradation in memory performance (Chevance, 2006, Jesshope 2008). The memory hierarchy brings conflicting requirements in the memory system as computing systems require a large and fast memory to scale up performances.

A memory hierarchy attempts to make a large slow memory appear fast by buffering data in smaller faster memories close to the processor (Hennessey and Patterson, 2007). Electronic systems slow down as they increase in size, for example the speed of light is approximately 1ns for 30cms and 1ns is 3 clock cycles in a state of the art processor (Jesshope, 2008). Memory performance is therefore a compromise between power and performance, as is the processor performance today (Chevance, 2006, Hennessey and Patterson, 2007). The key indicators of memory performance are the memory bandwidth and latency (Hennessey and Patterson, 2007). Memory latency is the delay required to obtain a specific item of data (measured in seconds), and, this is larger in dynamic random access memory (DRAM) than in static random access memory (SRAM) (Hennessey and Patterson, 2007). SRAM can access any bit each cycle DRAM is restricted to bits in the same row, cell address space (CAS) cycles. Memory Bandwidth is the rate at which data can be accessed (e.g. bits per second), Bandwidth is normally 1/cycle time and this rate can be improved by concurrent access (Hennessey and Patterson, 2007).

The most common solution to the memory wall is to cache data and caching requires locality of access or memory reuse, which may be achieved by compiler optimisations that can help to localise data (Jesshope, 2008). Computing scientists also designed banked memory systems to provide high bandwidth to random memory locations (Hennessey and Patterson, 2007; Jesshope, 2008), but, some access patterns still break the memory (Jesshope, 2008). Processors that tolerate high-latency memory accesses have been designed but this requires concurrency in instruction execution (Hennessey and Patterson, 2007; Jesshope, 2008). Caches are largely transparent to the programmer, but, programmers must be aware of the cache while designing code to ensure regular access patterns (Hennessey and Patterson, 2007; Jesshope, 2008, 2009, 2011). Caching the right data is the most critical aspect of caching to improve

RETRACTED

maximum system performances. More catch misses end up reducing performance instead of improving and this might end up consuming more memory and at the same time suffering from more cache misses lead to system deadlocks, where the data is not actually getting served from cache but is re-fetched from the original source. The development of a cache simulator requires a deeper understanding of how the memory hierarchy operates (Schintke, Simon, and Reinfield, 2012).

4. DESIGN AND IMPLEMENTATION

This research study is based on a simulating a 32KB 8-way set-associative Level1 Data Cache. In this research study we have concentrated on the Shared Memory Architecture. The reason for choosing shared memory architecture is that we wanted to scale up the cache simulator, from having one processor to a maximum of eight processors using different trace files. We have to modify the architecture to make sure that each processor node has access to a local cache (reads and writes). The architectural implementation for this research implies that each processor node can write to a memory location, and its local cache stores the memory contents locally, consequently a read of the same memory location by another processor node can be of a different value from its cache. The modified shared memory architecture used in this research is not unique as Jesshope (2011), suggested such memory architecture for scaling up processor frequencies. Associativity of caches (Hill and Smith, 1991) is an important metric that determine cache performance.

The implementation environment based on SystemC (Black and Donovan, 2004; OSCI, 2005; Bhasker, 2009; Ma, 2011) resulted in us simulating a 32KB Level 1 data cache within the Arch Linux environment. We developed SystemC C++ code for the implementation of the CPU, Memory, Cache, Bus and used Jesshope's (2011) Trace Files used to drive the simulator. For our simulation we used the UNIX platform Arch Linux 3.8 (http://www.archlinux.org) with GNU C++ compiler versions gcc 4.8. It is one of the lightweight GNU/Linux based operating system. The installation of Arch Linux takes place as if you will be building your own operating system, and it is heavily command driven. The three main issues that one should take care of when installing Arch Linux is the graphics, network especially wireless networks and UEFI. We chose to install the KDE desktop environment for our Arch Linux environment because of having used it in another Linux ambience which is the Linux Mint environment. We followed the instructions in the INSTALL document that comes with the SystemC-2.2.0 package to compile it in Arch Linux. The SystemC installation is a nasty experience and it took us some days to compile it and run in Arch Linux. Jesshope (2011) provided the theoretical and programming foundations of the trace files used for this research, and we use his trace files version 3, and his philosophy behind these trace files to drive our simulator.

The approach to implement the SystemC Level 1 Data-cache simulator followed the conventional programming norms of increasing the programming complexity as the demands of the system increases. We started by implementing a bus snooping cache coherence protocol, the **Valid-Invalid protocol**. The term 'snooping' allows for each cache node in the system to monitor the activities on the bus to which each of the cache nodes can write exclusively. In the event of a write enquiry if a cache node realizes that another processor belonging to another cache node has written to an address which it has a copy, the cache line containing a stale copy of the associated memory segment is immediately invalidated. The programming logic behind this protocol is that it does not allow for two cache lines to be valid in different cache nodes, in the event that they are mapped into the same set and even share the same address tag. The

RETRACTED

implementation of this protocol served as the basis for diagnosing anticipated programming problems and we used the debugging traces to eliminate errors until we were satisfied with the program executions.

We then implemented the MOESI Cache coherence protocol which is theoretically and programmatically built as an extension to the MESI protocol. The MESI protocol is the most common cache protocol that supports the write-back replacement strategy. The acronym MESI indicates that the protocol supports four cache line state transitions and these are Modified, Exclusive, Shared and Invalid, which logically implies that it implements the same cache line invalidation scheme as the valid-invalid cache coherency protocol. The difference to the valid-invalid cache coherency protocol is that it monitors whether the cache line is shared or not. The caches are allowed to make the cache line dirty if the cache line is in a modified or exclusive state. The MOESI cache coherence protocol introduces a fifth cache line transition state 'owned' which means it has characteristics of exclusive modified and shared cache line state transitions. We have to point out that this cache coherency protocol allows for cache lines to be shared, and is not supposedly written back to memory before the sharing [illegible]

As a starting point we build a single 32KB 8-way set associative cache with 32 Byte line size. We also built a CPU module connected to the cache that was asking for reading or writing some data from or to memory through the cache. In addition we made a memory module to help in checking the correctness of the data. The connection between the memory and the cache has been made from an 8-bit wire, therefore to fill the 32 Byte cache line, the cache has to read the memory 32 times. This was also useful to simulate the memory latency. We only used the random trace file for one processor to test the correctness of our simulator. The result of the simulation can be seen in the Table 1.

Table 1. Results of simulating a Uniprocessor.

Property	Value
Execution Time	55329 ns
CPU Read	6140 times
CPU Write	6081 times
Read Hit	5113 (83.3%)
Read Miss	1027 (16.7%)
Write Hit	5017 (82.5%)
Write Miss	1064 (17.5%)

RETRACTED

The results in the show that the CPU made 12221 requests composed as 6140 read requests and 6081 write requests. The results further show that more than 80% of the requests hit the cache, with an execution time of 55329 ns.

4. 1. Comparative Results Using Graphs

We plotted graphs to make a fair comparison of the trace files used and also the snooping and directory based cache coherency protocols. We made a comparative analysis of the protocols considering that there is no bus snooping, no barrier synchronization and with barrier synchronisation for each protocol. We started by comparing the Average Cache hit Rate and the two graphs represented by Figure 1 and Figure 2 indicate that there is no major significant difference between the Valid-Invalid and MOESI cache coherence protocols in terms of the cache hit rates, when random trace files are used. The different configurations made to the simulator did not show distinguishable cache performance indicators between the two sets of traces. The MOESI protocol theoretically outperforms the Valid-Invalid protocol.

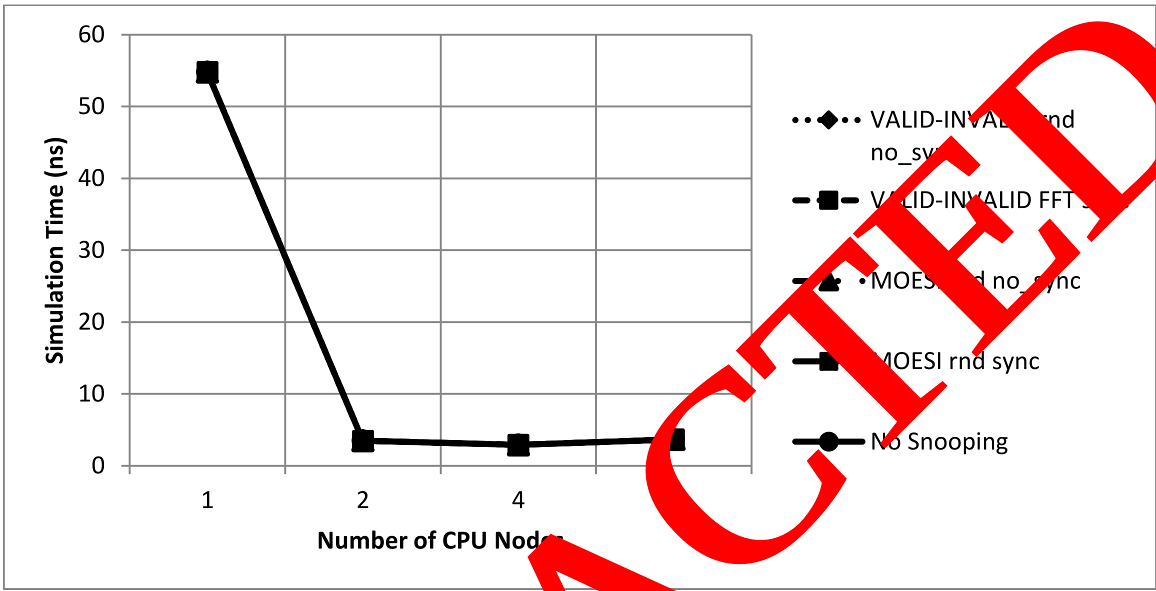
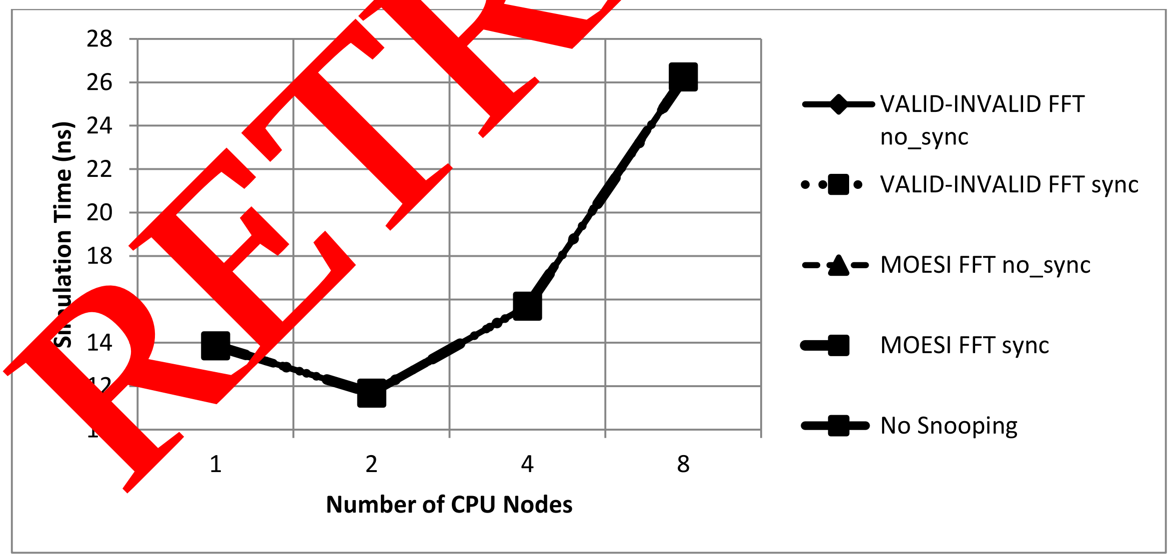


Figure 1. Average Hit Rate Using Random Traces.



RETRACTED

Figure 2. Average Hit Rate Using Fast-Fourier Transform Traces.

The other result that was very important to the SystemC cache Simulator experiment was to investigate the contention of the bus interconnection network. This was achieved by taking a count of the time stamps (delta cycles) in which the bus had more than one request to handle. This was handled by a member function in the Bus module which was designed to indicate the number of requests in the queue. The bus contention when using the two sets of traces is shown by Figure 3 and Figure 4.

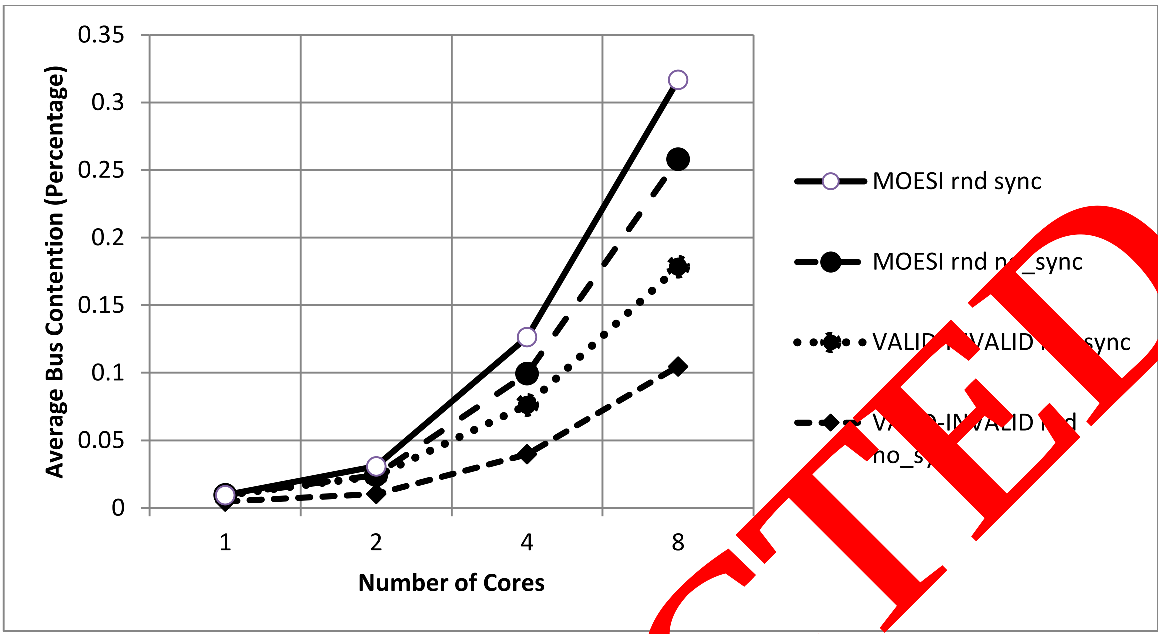


Figure 3. Average Bus Contention Using Fast-Fourier Transform Traces.

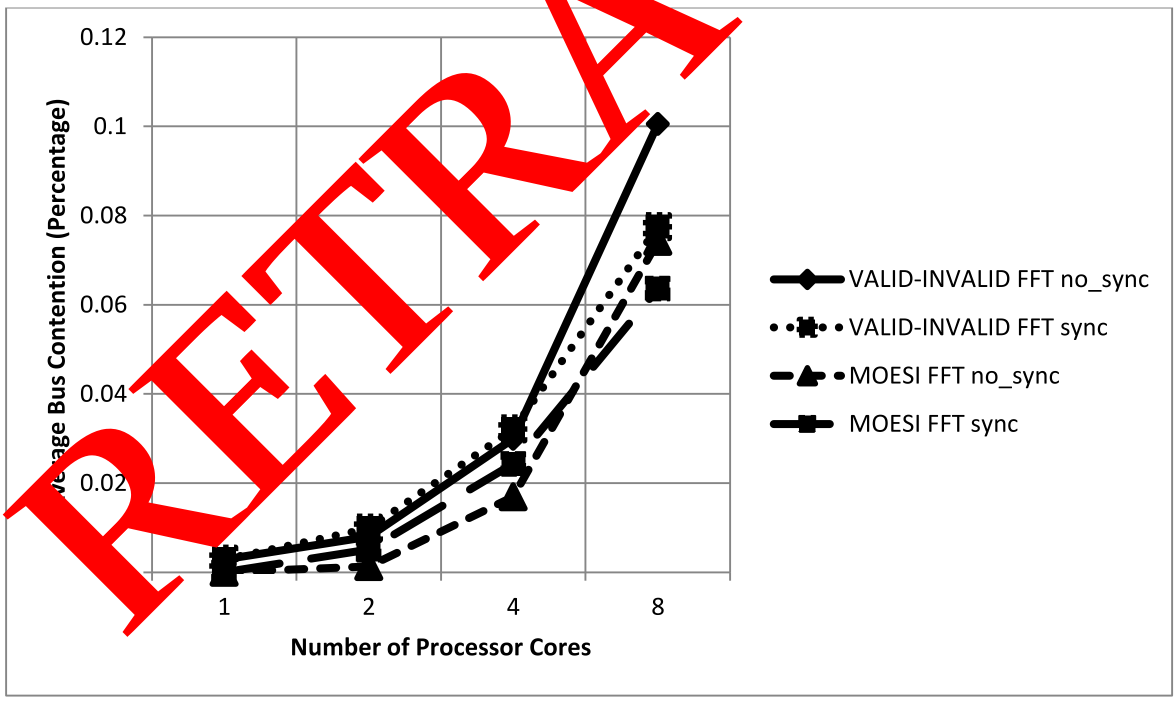


Figure 4. Average Bus Contention Using Fast-Fourier Transform Traces.

RETRACTED

The synchronised cache simulator runs show a reduction in the bus contention. The synchronisation event relieves the interconnection network as it oblige the processor nodes to wait until the barrier threshold instead of putting them in a race condition towards the end of each trace. The MOESI cache coherence protocol exhibit a smaller footprint on the interconnection network (bus), due to the deferred writes, but consequently uses more memory resources.

5. CONCLUSIONS

The SystemC cache simulator we have developed initially showed some feeble plugs, maybe because, of the fact that the trace files we have used in the simulation were designed to pick up read and write addresses for (hits/misses), instead of showing how the data is moved around in the system. In that way we would actually have testified that the processors constituted in the system have actually performed the reads and writes of the data they were supposed to. We also noticed that even if the trace files provided for checking whether the processors read/write the data they are supposed to, there is no assurance that the cache simulator is correct. We introduced a component of non-determinism in the event that the different cache nodes attempted simultaneously to access the bus.

The introduction of a memory latency of a century of cycles did not generally assume that a read issued just a few cycles after a write onto the same memory address, would harvest the correct data response. If the memory was responding to a read request within the memory cycle latency limit, a write request issued to the same memory address was not permissible, and the stale data value was not send back to the bus. The cache coherency protocols resolved such a situation by implementing two further cache coherency organizations, and these are the write-invalidate or the write-update.

As Jesshope (2011) argued that write-invalidate need the management of dynamic requests and the logic to rearrange requests is needed. The write-update requires an extra hardware in the form of a buffer that will contain the addresses of the requests, and the associated data elements, forcing the main memory to behave the same as the cache. We implemented the write-invalidate scheme as it is conservative and compatible with our chosen cache coherency protocols. It further pruned away the existence of duplicate read requests by allowing for a small degree of performance optimizations. We studied the graphs and come to the conclusion that cache coherence protocols are comparable, even when we use different traces and different number of processors. We therefore use the experimental data and graphs to answer our research questions.

Answering the Research Questions

The first research question refers to an investigation of the performance of the cache when we increase the number of processors. Based on this postulate we then give our response to the following first research question entitled:

To what extend do the number of processors in multiprocessor architectures affect the performance of Level-1 (L1) data cache memory systems?

We have noted that the runs of all the cache simulator experiments we have made did not end up in an inconsistent state. The execution time (simulation time) of the cache simulator increases as we have more processor cores. The average hit rate did not increase significantly with the increase of the processor cores. We have also noted that other factors such as snooping have a direct effect on the performance of the cache. From the results of the simulations we could see that increasing the number of cores does not imply an increase in cache performance as there are coherency issues to be taken care of. The deactivation of the snooping on the interconnection network subsequently increased the average hit rate even when using different trace files.

RETRACTED

Without snooping on the bus, there is now invalidation in case of probe write hits, meaning that the cache writes to a shared cache line and the status of the cache line remains the same. In such an instance the cache gets a higher hit rate. As performance is determined by the hit rate we would argue that the cache performs much better without snooping. However when we deactivated bus snooping we could not guarantee and assure the integrity of the cache line when we repeatedly run the cache simulator. The other factor that comes into play when we increased the processor nodes is synchronisation of the caches and taking care of the cache misses. One way of taking care of this aspect is to optimize the compiler, by code rearrangement including data rearrangement. Loop interchange and cache blocking could also optimize the cache by improving temporal locality. We can conclude that increasing the number of processors on the multiprocessor architecture implies more cache programming complexity and cache coherency is a major concern in the performance of the caches of a multiprocessor system.

Rightfully we can say that given optimizations in the compiler and having synchronised multibanked caches in the multiprocessor system, we can increase the cache performance. As mentioned earlier increasing processor nodes with their local caches mean that there is a lot of programming issues to consider. In our case we pipelined the cache access so that we would increase the cache bandwidth. We have mentioned earlier that cache coherency is an important aspect to consider in a multiprocessor environment. We therefore investigated how our chosen cache coherency protocols affected the performance of our cache simulator. The research question to answer is the folloing:

How do cache coherency protocols influence the level-1 (L1) data cache memory performances of multiprocessor architectures?

We have used trace caches to reduce the hit time in our system henceforth improve the cache hit rate. Each implementation of our SystemC cache simulator had to run a set of Random and Fast-Fourier Transform trace files on 1, 2, 4, and 8 processor environments. The comparison graphs showed that the directory-based cache coherence protocol (MOESI) has a slight performance edge over the snooping cache coherence protocol (Valid-Invalid). Though the difference can be regarded as statistically insignificant, MOESI protocol outperforms Valid-Invalid protocol because it can transfer data from one cache to another cache. In such cases the cache miss doesn't always mean the cache has to read/write from/to memory. Lesser memory access reads leads to faster execution time because the need to wait for memory access latency can be reduced. The hit ratio of the MOESI protocol is better than the hit ratio in Valid-Invalid protocol meaning that consecutive writes will always contribute to a cache miss. In the MOESI protocol if a write miss occurs, the cache line will be updated (read) and the consecutive write will be marked as write hit. Another contributing factor to the better performance of the MOESI protocol is that it has a lower contention rate of the bus usage. One of the reasons for this could be that the memory access rate in Valid-Invalid protocol is more than in the MOESI protocol. Since the bus will be used when the cache modules want to have memory access, higher memory access will imply a higher request to use the bus. Following the memory hierarchy principles, accessing the bulk shared memory will take more time compared to accessing another cache. The Valid-Invalid have to wait longer to access the memory than in MOESI protocol.

Unexpectedly in some instances the MOESI cache coherence protocol used more memory writes which might be as a result of a bug in our SystemC cache Simulator. We have actually managed to preserve the coherency of the caches in all our experiments and all simulations. We still need to conduct a proof of the program correctness of our simulator using acceptable,

RETRACTED

scientific, standard proof-of-program correctness methodologies. All the simulations never ended up in an inconsistent state, which is a significant leap towards the optimization of the cache simulator. We therefore have the following recommendations for the improvements of the cache simulator.

RECOMMENDATIONS

The performance graphs showed that there is no significant performance difference between the snooping protocol and the directory-based protocols we have chosen. Theoretically this is wrong and one of the reasons is that there might be a programming error (a bug) in the bookkeeping of the memory writes through the traces used or in the cache simulator itself. We therefore recommend a program proof-of-correctness procedure to be carried out and also to revise the configurations of the trace files. The Valid-Invalid protocol outperformed the MOESI protocol when random trace files were used which is a point of concern. The caches cannot expect randomness as they are based on programming attributes and the coherency attribute is a result of programming efforts. We therefore recommend a revisit on the trace files and an increase in the range including the types of trace files to be used by the simulator.

We have not taken into consideration issues of increasing the cache bandwidth. As a future area of research and improving the cache performance we have to consider various cache optimizations schemes and also record the data for the memory accesses. The implementation of various cache optimizations will bring an increase in program complexity of the cache simulator. Concurrency has been a major programming issue during the execution of the simulator. When we implemented the SystemC simulator we had Error (115), which did not allow us to start the simulator with two or more drivers. We have actually resolved this error by implementing SC_SIGNAL_WRITE_CHECK= "DISABLED" at the start of each simulation involving more than one processor but we recommend that we have to create an environment variable that allows for explicit parallelism to occur during the simulation.

We also recommend the use of a wide range of cache coherency protocols rather than choosing only one type of each category. As SystemC can be implemented in the multi-platform environment and the simulator exhibits the characteristics of the hardware being simulated, we will in the try the simulator in different multiprocessor environments. However this has been a learning curve for us and this research is useful in multiprocessor design.

RETRACTED

References

[1] Bartolini S., Giorgi R., *Journal of Embedded Computing* 2 (2003) 137-139.

[2] Bhasker J. (2009). *The SystemC* ^{TM} *Primer.* Allentown, Star Galaxy Publishers.

[3] Black C., D., Donovan J (2004). *SystemC from the Ground Up,* Boston, Kluwer Publishers.

[4] Byler J. (2009). *Software Programmers face Multicore Challenges, in Embedded Intel.* [Online]. Intel. Available: http://www.embeddedintel.com/from_intel.php?article=1050 [Accessed 11 March 2013 2013].

[5] Chevance J. R. (2006). *Servers Architectures: Multiple Processors, Clusters, Parallel Systems, Web Servers, Storage Solutions,* London, Elsevier.

[6] Duller A. P. G., Towner D. (2003). *Parallel Processing - the picoChip way! . In:* Broenik, J., F And Hilderink, G. (ed.) *Communicating Process Architectures.* London: PicoChip.

[7] Hennessy L. J., Patterson A. D (2007). *Computer Architecture. A Quantitative Approach,* San Francisco, Morgan Kaufmann.

[8] Hill D., Smith A. J., *IEEE Transactions on Computers* 40 (1991) 371-390.

[9] Hill D., Smith A. J., *IEEE Transactions on Computers* 38 (1991) 1612-1630.

[10] Hobson R., Cheung K. L., Ressi B.,Signal Processing with Teams of Embedded Workhorse Processors. *EURASIP Journal on Embedded Systems* (2006) 16.

[11] Jesshope C. R. (2008). A model for the design and programming of multicores. *In:* GRANDINETTI, L. (ed.) *Advances in Parallel Computing, 16, High performance Computing and Grids in Action* London: IOS Press.

[12] Jesshope C. R (2009). *Multiprocessor Memory Systems*, Amsterdam, Universiteit van Amsterdam

[13] Jesshope C. R .(2011). *A SystemC Tutorial*. Amsterdam, Universiteit van Amsterdam

[14] Joubert G. R. (2008). "*Parallel computing current and future issues of high end computing*", Forschungszentrum, John von Neumann Institute for Computing Jülich

[15] Leiserson C. E, Mirman I. B. (2008). *How to Survive the Survive Multicore Software Revolution* [or at Least Survive the Hype]. . *In:* CLICKARTS, I. (ed.). New York: Click Arts Inc.

[16] Ma N., "*Modelling and evaluation of multi-[illegible] multithreaded processor architectures in SystemC*", Proquest, (2011) 1109.

[17] Mckee A. S. (2004). *Reflections on the Memory Wall. In:* ACM, ed. Proceedings of the 1st conference on Computing frontiers (CF'04). 2004 New York. New York: ACM, 1-6.

[18] Nussbaum S., Smith J. E. (2002). *Statistical Simulation of Symmetric Multiprocessor Systems. In:* IEEE, ed. In Proceedings of the 35th Annual Simulation Symposium (SS '02) , 2002. Washington, DC, USA, 89. IEEE Computer Society, 89-97.

[19] OSCI. (200[illegible]). *An Introduction to System Level Modelling in SystemC 2.0.* Available: www.es.ele.tue.nl/~heco/courses/EmbSystems/WhitePaper20.pdf [Accessed 15 March 2013].

[20] Panesar G. T. D., Duller A., Gray A., Robins W. (2005). Deterministic Parallel Processing. *In:* BROENIK, J., F AND HILDERINK, G. (ed.) *Microgrid workshop-2005.* London: IOS Press.

[21] Panesar G. T. D., Duller A., Gray A., Robins W., *International Journal of Parallel Programming* 34(4) (2006) 323-341.

[22] PICOCHIP. (2007). *Technical White Paper*: Practical, Programmable Multi-Core DSP. Available: http://www.picochip.com/downloads/4eac6c97aa70840ad7f4d12aec82ebf1/Multicore_June_2007.pdf [Accessed 13 March 2013].

[23] Schintke F., Simon J., Reinfield A. (2012). *A Cache Simulator for Shared Memory Systems* (unpublished paper)

[24] Stalling W. (2012) *Computer Organization and Architecture* London, Prentice Hall.

RETRACTED

[25] Sutter H. 2005. *The free lunch is over: A fundamental turn toward concurrency in software.* [Online]. New York. Available: http://www.gotw.ca/publications/concurrency-ddj.htm [Accessed 13 March 2013 2013].

[26] Szydlowski C. (2005). *Multithreaded Technology & Multicore Processors* [Online]. New York: Dr. Dobb's. Available: http://www.drdobbs.com/multithreaded-technology-multicore.../18440607 [Accessed 13 March 2013 2013].

[27] Towner D. P. G., Duller A., Gray A., Robins W. (2004). *Debugging and Verification of Parallel Systems - the picoChip way*! *In:* BROENIK, J., F AND HILDERINK, G. (ed.) *Communicating Process Architectures* London: IOS Press.

(Received 04 October 2013; accepted 21 October 2013)

RETRACTED