Comparison of memory write policies for NoC based Multicore Cache Coherent Systems

Pierre Guironnet de Massas, Frédéric Pétrot
System-Level Synthesis Group
TIMA Laboratory
46, Av Félix Viallet, 38031 Grenoble, France

Abstract

The following study shows a direct comparison of memory write policies in Shared Memory Multicore Systems. Although there are much work and many studies about this issue, our work takes into account the difficulties related to on chip communication using network-like interconnects. Our study is based on Cycle Approximate Bit Accurate simulations (CABA) of platforms with up to 64 processors, modelling accurately all the aspects of multi-threaded program execution and memory accesses. Our main results show that write-through caches perform well compared to write-back ones, with a slightly simpler implementation and comparable traffic.

1. Introduction

In the past few years, the search for processors with higher-frequencies and evermore complex pipelines seems to have stopped. Instead, as integration capabilities still increase according to Moore's law, architects are putting together more and more processors, memories and devices on a single die. Such chips with dozen of processors achieve high computational throughput by exploiting parallelism at task level in applications. Since buses do not scale well with more than few processors – say ten – Network on Chip (NoC) interconnect [6] seems to be one of the best solutions for the next decades. In NoCs, the available bandwidth is not limited by the number of nodes because each one comes with its own set of wires. Moreover, all communications are on chip, hence NoC's achieve higher throughput bandwidth and much lower latency than off-chip large-scale Symmetric MultiProcessors (SMP).

A high number of programmable components significantly increases software design and implementation complexity. Having a hardware that provides a simple programming model will be a major architectural argument in the future. Some architectures do not include any more caches since they are targeting low-power devices with streaming application models. But as shown in [10] hardware coherent caches offer similar performances in streaming applications and better overall performances when they do not perform *write-allocate* protocols. Hence, caches with hardware coherency are a good choice to provide a simple programming model and reduce accesses latencies in a large scale NoC.

For the sake of simplicity, we will use sequential consistency in our platform simulation. Nevertheless our comparison remains valid with a weaker model as the one used in commercial designs.

Cache Area is an important trade off, since in actual high-end processors it represents 50% of the area, and it will still increase in future implementations. Nevertheless, integrating several processors in the same die implies less area allowed to caches and memories. Consequently, the cache implementation should be as simple as possible with uniform access and in-order request issues.

Due to the presence of a NoC, we use a directory-based coherency scheme inspired from one proposed by Censier and Feautrier [5]. Its area overhead does not scale well with a high number of processors. Our work can be adapted to more efficient solutions as one reviewed in [16, 2].

Therefore, our work focuses on on-chip shared memory multicore systems with NoC and caches. The memory hierarchy implements sequential consistency and hardware coherence with a directory scheme.

Hardware protocols can be divided into two categories to maintain coherency, *write-update* and *write-invalidate* [15]. We implement the later solution since it is the most commonly used and surely the best one in our context. As a result, we focus our study on memory write policies. In actual designs, the *write-back* is the most widely used policy. In contrast, *write-through* memory updates are well known in the literature to give poor performances and are almost unused in actual designs.

The high bandwidth available in NoC architectures, and the high latencies to access foreign on chip nodes, makes it necessary to re-evaluate the *write-through* memory update policy which is currently considered inefficient.

The main contribution of our work is a head to head comparison between directory-based generic *write-back* MESI like and *write-through* invalidate protocols implemented on top of a NoC. The comparison is done simulating at CABA level a full application software with an OS. Therefore, all the instructions and memory accesses are fully modelled.

To give a self contained study, we present hereafter the necessary background definitions. Secondly, we show the related work on the subject, followed in the third section by a description of the compared protocols and their implementations. In the fourth section we describe the experiments, and finally we present our results and concluding comments.

2. Background and Definitions

Our study focuses on directory-based solutions and *write-invalidate* protocols. A memory location may have several readers but only one writer. Hence, when a processor executes a store instruction, all the copies of the targeted block must be invalidated before the new value is written. This ensure the coherency and sequential consistency of the memory. Two memory update policies named *write-through* and *write-back* are available with the *write-invalidate* protocol.

Write-through policy: A write request is sent to the main memory for each store issued by a processor. Thus, main memory is always up-to-date but severe contention may appear on a bus based architecture.

Write-back policy: This is the most widely used policy. When a processor executes a store instruction, it modifies the local copy but no request is sent to the main memory. The modified block is said to be "dirty", it will be *written-back* when it is evicted. This policy is commonly implemented as *write-allocates*: on a write miss the block is retrieved from memory and modified locally. The main advantage of this solution is that only the necessary memory accesses are done (on a block basis however).

3. Related works

Cache coherency has been a problem for architects from the earlier shared memory multiprocessors systems. The first proposed solutions to maintain coherency are the Write-once, Illinois and Berkeley protocols, followed few years later by the Dragon and Firefly protocols. A review of these protocols was done by Archibal and Baer[4]. Other well-known reviews and studies about existing protocols [16, 2, 18] show that *write-through* invalidate is the less efficient protocol in a bus-like interconnect.

Although previous cited work may seem a bit old, there is an up-to-date work comparing existing snooping protocols with cycle accurate simulations [11]. This work shows that Write-through Invalidate protocol is less efficient than *write-back* MESI. As we have stated before, *write-back* implementations take advantage of bus snooping facilities and leverage the bus locally updating a block and writing back on eviction.

With the use of non-bus interconnects, like crossbars or later NoC's solutions, directories schemes have been (and still are) a good solution. Different solutions are reviewed in [16, 2].

With the growing number of caches and components in the same chips, interconnect latencies to access directory and foreign data is a major problem. There are several solutions like [12, 7] relying on specific network communication and coherency message schemes, but both of them still implement MESI like protocols. Protocol optimizations like [8, 9], try to hide or reduce access latencies to foreign dirty blocks. Nevertheless, all proposed solutions (to our knowledge) allow a block to be dirty in a cache and thus falls in the *write-back* category.

4. Coherency protocols and implementation

4.1 Compared protocols:

Write-through Invalidate (WTI): This protocol is the simplest to implement as attested by its finite state machine diagram on figure 1. A *write-through* policy is combined with a *write-invalidate* protocol. When a write request is issued, the main memory, through its directory, will invalidate all the cached copies to ensure the coherency. Traffic overhead is the major drawback of this protocol. The main advantage is that the main memory always contains clean copies.

Write-back MESI (WB-MESI): For comparison purposes we implemented a generic MESI like protocol (Illinois [13]). When a processor executes a store instruction, the cache must get the exclusivity of the targeted block. This request will invalidate all other copies, retrieve dirty blocks from foreign caches and allocate the clean block if needed. These actions can take a long time on a high latency NoC, and this is the major drawback of this protocol. Its finite state transition diagram can be seen in figure 1.

4.2 Protocols implementation

Implementing a WTI or WB-MESI protocol (at CABA or RTL level) can lead to completely different solutions. Hereafter we present the main distinctive behaviours of both protocols. We call a *hop* the delay needed to cross the NoC from one node to another, and use this unit to characterize the protocols actions. In our context a node is either a cache, its associated processor and its controller or the main memory, its controller and its directory. Transferring messages

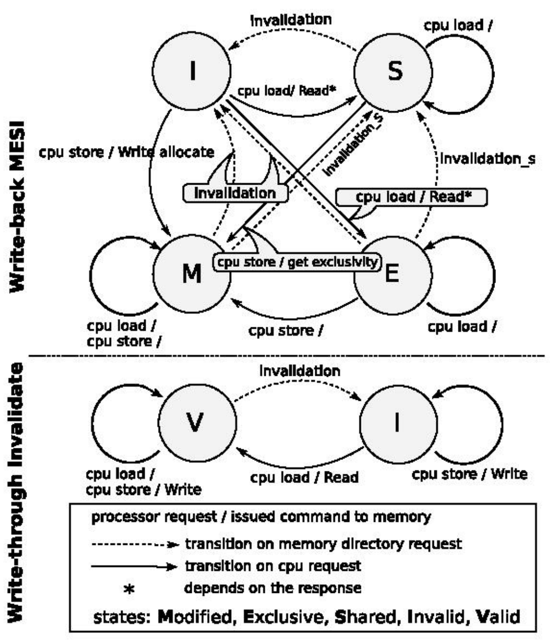
Write-back MESI
Invalidation
cpu load /
I
S
cpu load/ Read*
cpu store / Write allocate
Invalidation
invalidation_S
cpu load / Read*
invalidation_s
M
E
cpu store / get exclusivity
cpu load / cpu store /
cpu store /
cpu load /
Write-through Invalidate
Invalidation
V
I
cpu load / cpu store / Write
cpu load / Read
cpu store / Write
processor request / issued command to memory
transition on memory directory request
transition on cpu request
* depends on the response
states: Modified, Exclusive, Shared, Invalid, Valid

Figure 1. Finite State Diagram of the WTI and WB-MESI protocols

from node to node contributes for almost all the data latencies. Hence, counting *hops* is a good comparison basis for the protocol's implementation efficiency.

Read requests: In both implementations, a read miss leads to at least a two *hops* request (read a clean copy from the memory). Nevertheless, in WB-MESI the block may be in **M**odified state in a foreign cache. In our implementation, the corresponding action is decomposed as follows: 1. a read request is sent by the cache to the main memory node. 2. the main memory node sends a request to the owner of the clean copy. 3. The foreign cache response contains a clean copy and puts his in **S**hared state. 4. The main memory node responds to the requesting cache with a clean copy of the block.

Write requests: In WTI, misses and hits are handled identically. A write command containing the modified word is sent to the main memory through its write buffer. The main memory node can respond immediately if there is no shared copies (2 *hops* action), or send invalidate commands to all the caches which contains a copy and waits for the acknowledgements before sending a response to the requesting node (4 *hops*). These actions are non-blocking for the cache controller until the buffer is full.

In WB-MESI, a write misses when the block is either in **S**hared or **I**nvalid state. In the first case, an exclusivity will be granted and all copies will be invalidated. This action is blocking for the data cache and the processor and costs 2 or 4 hops. In the second case, the block is allocated and can lead to an up to 6 *hops* action as shown in figure 2, which is blocking (*stalls* the processor) until step 4.

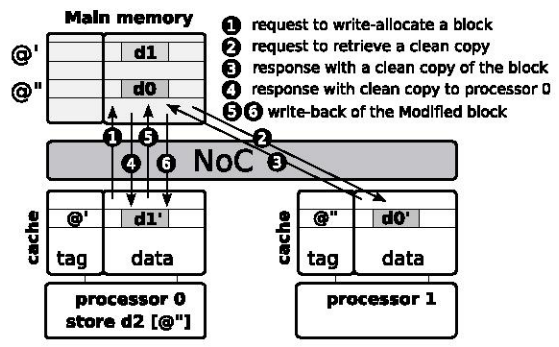
Main memory
@'
@"
d1
d0
❶ request to write-allocate a block
❷ request to retrieve a clean copy
❸ response with a clean copy of the block
❹ response with clean copy to processor 0
❺❻ write-back of the Modified block
NoC
cache
@'
d1'
tag
data
processor 0
store d2 [@"]
cache
@"
d0'
tag
data
processor 1

Figure 2. a 6 hops example

processor action	WTI	WB-MESI
read hit	0	0
read miss	2 b. *hops*	2 or 4 *hops* b., (+2 n.b.)
write miss	2,4 *hops* n.b.	2, 4 blocking, (+2 n.b.)
write hit S	2,4 *hops* n.b.	2, 4 *hops* blocking
write hit E	-	0
write hit M	-	0

Table 1. Cost in hops of each request in both protocols, b. and n.b. stands for blocking and non-blocking requests for the processor.

In the table 1, we summarize the cost of each action for both protocols. Our implementations can be optimized by allowing cache to cache transfers or any other architectural enhancement. For example, invalidation acknowledges can be sent directly to the requesting node (cache) leveraging the memory node and saving one *hop* transfer. Nevertheless, our implementations were done with identical behaviours in actions steps, leading to a as fair as possible comparison. Moreover, typical protocol optimizations can often be applied on both protocols.

5. Experiments

5.1 Simulation environment and Hardware architecture

The simulation environment: Our simulation platforms are built using the Cycle Approximate, Bit Accurate components of the SoCLib [1] library which uses the VCI [3] protocol to communicate. This library allows to design and simulate in an easy way platforms with dozen of processors running cross-compiled software.

Although most components (processors, caches, memories and many devices) are simulated with cycle-accurate precision, some are instead cycle-approximate. The main cycle-approximate component is the Generic Micro Network (GMN). This component does not truly represent a set of routers, but a configurable crossbar like interconnect with internal delay fifos. Nevertheless, setting the minimum transfer delay and fifo's depth allows us to model a NoC with 2D mesh latencies and contention characteristics. This cycle-approximate component has no major impact in our results since it is used for all the configurations and gives us fair comparisons between the different studied protocols.

Modeled architecture description: In figure 3 we present our modelled architectures. The Sparc processors are connected to an instruction and data cache. The data cache and the memory implement one of the protocols (WTI, WB-MESI). The instruction and data cache use the same interconnect port in order to minimize the NoC area. As a result, high data workload can interfere with instruction misses requests, increasing the average instruction access latency. We implemented two architectures to evaluate the impact of the contention at memory banks in the protocols. As we can see, one has few memory banks which will receive requests from up to 64 processors. The other one spreads the memory accesses among several memory banks (one per processor and three shared one). The table 2 summarizes the main hardware characteristics.

number of processors	$n = \{4, 16, 32, 64\}$
number of memory banks	$m = \{2, n+3\}$
processor model	SPARC-V8 with FPU
data cache size	$4Kb$
instruction cache size	$4Kb$
data/instruction block size	32 bytes
cache associativity	Direct-mapped
write-buffer size	8 words (32 bytes)
NoC topology	Mesh
NoC Latency	$3.\frac{2}{3}\sqrt{n} + m + 3$

Table 2. Simulated platforms characteristics

5.2 Software architecture

Application: On top of the CABA simulation platform, we executed the Ocean and Water SPLASH-2 benchmark tests. In [17] can be found a detailed description of each

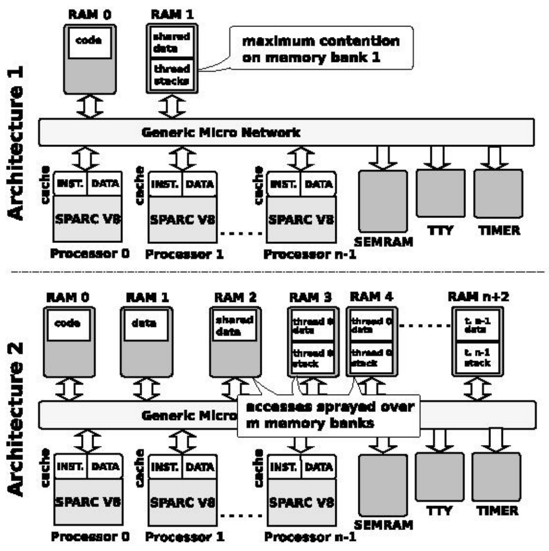

Figure 3. Modeled architectures to compare coherency protocols with different levels of memory contention

test bench with their main characteristics. This test bench suite was designed to evaluate Shared Memory Multiprocessors architectures. It is widely used and accepted by the research community. In our simulations, we execute a comparable workload per processor in each platform configuration (4, 16, 32 and 64 processors). That way, the number of load/store instructions executed by each processor will not decrease while increasing the processors number. The configuration characteristics are presented briefly in figure 4. These test benches need some services, typically granted by a POSIX operating system. Hereafter we shortly describe the used one and its possible configurations.

Operating System configurations: We used a lightweight operating system [14], which implements POSIX pthreads allowing to execute parallel tasks on multicore systems. It can be configured in two ways:

1. Symmetric Scheduling (SMP). In this configuration the OS distributes the workload on the available processors on a first come, first served basis. Hence, a task can migrate from processor to processor in an unpredictable way. This configuration is not suitable for large scale platforms since the centralized scheduler access becomes a bottle-neck.

2. Decentralized Scheduling (DS). This configuration greatly limits contention as each processor has its own scheduler. Tasks can be pin pointed on a desired processor in order to avoid migration.

Memory layout: On the architecture 1, we use the SMP kernel. All the shared and local data and thread stacks are in the same memory bank. This configuration leads to a maximum memory workload and contention increasing accesses latencies.

On the architecture 2, we use the DS version. Each thread is pin pointed to a dedicated processor. Its stack and local data are stored in a dedicated memory bank. Shared dynamic and static data are contained in different memory banks. The purpose of this layout is to spread as fairly as possible the accesses to all memory banks. Consequently there are only few contention points and coherency protocol workload is slightly lower than in architecture 1.

6. Results and comments

We present hereafter the obtained results on two SPLASH-2 benchmarks: execution time, NoC traffic and data latencies.

6.1 Execution time

We run the applications until completion and report the execution time for both architectures on figure 4:

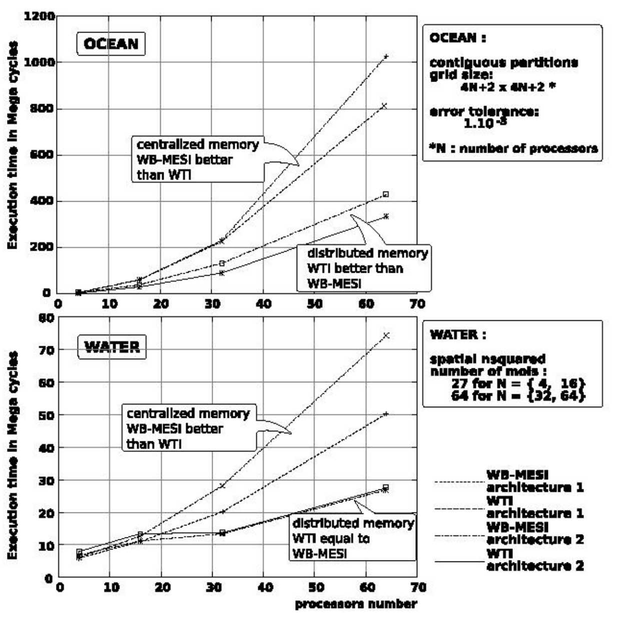

Figure 4. Execution time in Mega cycles for each simulation platform and both protocols

SMP: We see that WTI and WB-MESI have almost the same execution times. Nevertheless, increasing the processors number above 32 gives an advantage to WB-MESI protocol. This behavior was expected due to the centralized memory. In fact, sever contention appears on a specific memory bank as one that can be shown on a bus like interconnect with too many processors.

DS: The executions took slightly less time. This is due to less contention on scheduler structures by the processors, and to a better repartition of the communication workload since accesses are sprayed among all the memory banks. Moreover, pinpointing tasks on processors avoids migration overhead.

Consequently, the Ocean execution is up to 30% faster and the difference is increasing with the number of processors. In Water execution, both protocols give the same performances without a clear advantage for one of them.

These results show that, in our context, WTI protocols performs well compared to WB-MESI ones. Moreover, future MPSoCs will embed many memory banks distributed across the die. As a result, WTI protocols are a viable solution in terms of performances.

6.2 Traffic load over the NoC

As we can see in figure 5, the traffic is of the same magnitude order in both protocols. Differences in kernels, applications and architectures gives different behaviors. Therefore, there is not a clear advantage for none of the studied protocols.

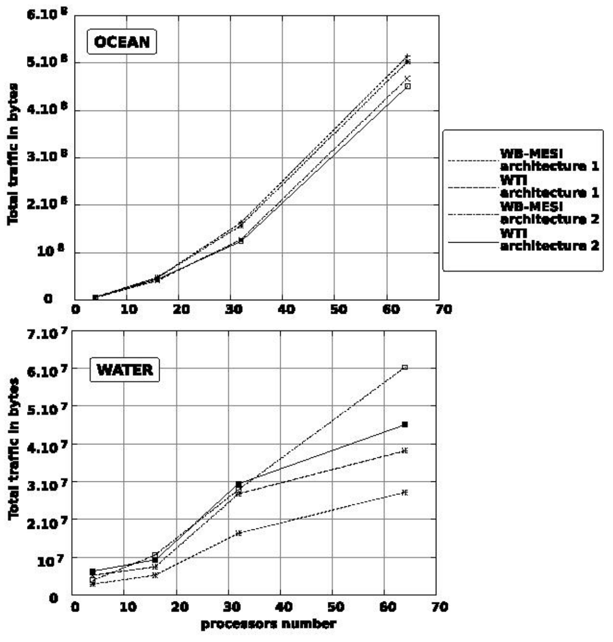

Figure 5. Total traffic on the NoC in bytes on a complete run execution

6.3 Data cache stall cycles

Instead of comparing directly data latencies, we compared the average stall time due to data cache accesses. The reason is that data latencies rely on loads instructions only. Therefore, it does not show stall cycles due to full write buffers and *write-allocate* actions. In the figure 6 we see that both protocols have almost identical results on both architectures. As expected, on architecture 1 there is more contention than on architecture 2. With more than 32 processors, the time being stalled by the data caches reaches almost 70% of the execution time.

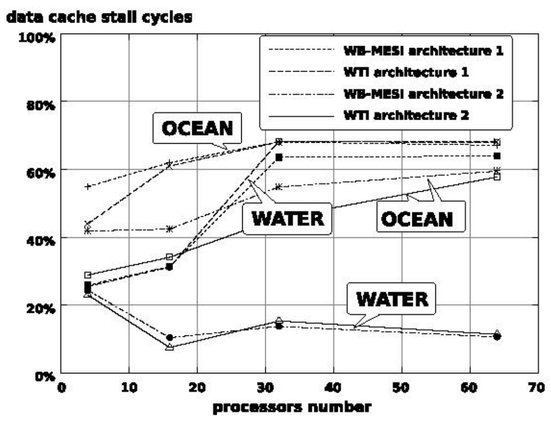

Figure 6. Percentage of data cache stall cycles

7. Conclusion

In this paper we compared the memory update policies in NoC based shared memory multicore systems implementing hardware coherence. Although there has been many works and studies around cache coherency, none of them (to our knowledge) has shown the usability of the very simple *write-through-invalidate* protocol. Moreover, all the new proposed protocols and optimizations make use of *write-back* policies. The obtained results have shown that in our context *write-through-invalidate* protocols are a possible and simple solution to maintain coherency. This protocol performs very well compared to a classic *write-back*-MESI protocol in both execution time and generated traffic. The main limitation of this work is the lack of best-case / worst-case results wich will be done in future work.

References

[1] Soclib project. `http://www.soclib.lip6.fr/Home.html`.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *ISCA '88*.

[3] V. Alliance. Virtual component interface standard (ocb 2 2.0). 2000.

[4] J. Archibald and J.-L. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, 1986.

[5] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *ieee Transactions on Computers*, C-27:1112–1118, 1978.

[6] W. J. Dally and B. Towles. Route packets, not wires: on-chip inteconnection networks. In *DAC '01: Proceedings of the 38th conference on Design automation.*

[7] N. Eisley, L.-S. Peh, and L. Shang. In-network cache coherence. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[8] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: making use of incoherence. In *ASPLOS-XI'04.*

[9] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *ISCA '00.*

[10] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *ISCA '07.*

[11] M. Loghi, M. Poncino, and L. Benini. Cache coherence tradeoffs in shared-memory mpsocs. *Trans. on Embedded Computing Sys.*, 5(2), 2006.

[12] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: decoupling performance and correctness. In *ISCA '03.*

[13] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture.*

[14] F. Pétrot and P. Gomez. Lightweight implementation of the posix threads api for an on-chip mips multiprocessor with vci interconnect. In *DATE, 2003.*

[15] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, June 1990.

[16] M. Tomasevic and V. Milutinovic. Hardware approaches coherence in shared-memory multiprocessors, part 1. *IEEE Micro*, 14(5):52–59, 1994.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture.*

[18] Q. Yang, L. N. Bhuyan, and B.-C. Liu. Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor. *IEEE Trans. Comput.*, 38(8):1143–1153, 1989.