

System Software Support for Reducing Memory Latency on Distributed Shared Memory Multiprocessors

Dimitrios S. Nikolopoulos and Theodore S. Papatheodorou

High Performance Information Systems Laboratory
Department of Computer Engineering and Informatics
University of Patras
Rion 26 500, Patras, Greece
{dsn,tsp}@hpclab.ceid.upatras.gr, <http://www.hpclab.ceid.upatras.gr>

Abstract

This paper overviews results from our recent work on building customized system software support for Distributed Shared Memory Multiprocessors. The mechanisms and policies outlined in this paper are connected with a single conceptual thread: they all attempt to reduce the memory latency of parallel programs by optimizing critical system services, while hiding the complex architectural details of Distributed Shared Memory from the programmer. We present four techniques that exhibit solid performance improvements: Efficient memory management for lightweight multithreading, highly scalable hybrid synchronization primitives, a virtual memory management scheme for DSM systems and transparent operating system services for adapting parallel programs to multiprogrammed environments.

1. Introduction

Distributed Shared Memory (DSM) multiprocessors are nowadays an attractive and viable platform for both parallel and mainstream computing. DSM systems are realized either as tightly integrated cache-coherent non-uniform memory access (ccNUMA) systems, or as loosely integrated networks of commodity workstations and symmetric multiprocessors (SMPs), equipped with a software DSM layer on top of a message-passing infrastructure [1]. Although DSM systems present strong advantages in terms of scalability, programmability and cost effectiveness, obtaining high performance out of these systems turns out to be much more cumbersome than expected. The primary hurdle towards high performance in modern DSM systems is the ever-increasing gap between processor speeds and memory access times [1, 2].

The structural organization of DSM systems presents the programmer with a globally shared memory address space, although the actual memory modules are physically distributed among processing nodes. This simplifies the task of building parallel programs as natural extensions of the corresponding sequential programs, by using loads and stores in shared memory as the communication medium. However, the very same DSM organization hides from the programmer the details of the underlying memory hierarchy and particularly the diversity (non-uniformity) of memory access latencies. Despite the rapid advances of interconnection network technology, accessing a remote memory module on a state-of-the-art tightly integrated DSM system (such as the SGI Origin2000 [5] or the Sequent NUMA-Q [6]) costs 3 to 8 times as much as accessing local memory. This cost is severely magnified in the case of networks of workstations and SMPs, where accessing remote memory may cost 100 or even 1000 times as much as accessing local memory. The side-effect of non-uniform memory access latencies

is that programmers have to carefully tune their applications by hand in a manner that minimizes remote memory accesses and exploits data locality. Although intensive research is being conducted in parallel programming languages, parallelizing compilers, runtime systems and operating systems, it appears that tuning the performance of parallel programs for moderate and large-scale DSM systems remains an ad-hoc procedure that puts a lot of a burden to the programmer and effectively negates the programmability of DSM systems [3]. Put simply, the programmer must be aware of all the details of the underlying DSM architecture and spend most of his/her development time on daunting tasks such as data alignment and padding, false-sharing elimination, minimization of synchronization, extensive low-level analyses of hardware performance etc. This problem is intensified, by the fact that modern DSM systems are also used as multiprogrammed computational servers, where parallel compute-intensive jobs execute concurrently with sequential I/O or network-intensive jobs. The programmer can no longer assume that parallel programs will run on a dedicated system and has to take into account that the threads of his/her program may be arbitrarily preempted and migrate between processors during the course of execution.

In this paper, we present a summary of results from our recent research work on building core system software for DSM systems. The thesis along which this research was conducted, is that most of the burden for tuning the memory performance of parallel programs on DSM systems should be moved to system software and more specifically to the runtime system and the operating system layers. These layers should export rich functionality and provide adequate flexibility for enhancing the performance of parallel applications on DSM systems, without significant programmer intervention. This approach is consistent with the initial design goals of DSM systems, namely scalability and programmability. Towards this direction, we developed a broad set of mechanisms and policies including:

- Memory management mechanisms for lightweight thread contexts with fine computational granularity.
- Scalable synchronization algorithms with minimal hardware support.
- Virtual memory management schemes for reducing remote memory accesses on DSM systems.
- Kernel mechanisms for multiprogramming scalability of parallel programs.

Most of the work presented in this paper was conducted as part of the NANOS project [7]. NANOS was initiated in October 1996, as an ESPRIT long-term research project, with the purpose of investigating the potentials of fine-grain multilevel parallelization and its integration with multiprogramming on modern shared memory multiprocessors. During the course of the project, it became more than evident that achieving high performance on DSM systems hits against the memory latency wall. This observation motivated intensive research work on providing sophisticated runtime and operating system support for improving the memory performance of parallel applications on DSM systems. The reader is referred to related publications of the authors [9, 10, 11, 12, 13, 14, 15] for more elaborate discussions and analyses of the issues outlined in this paper. Sections 2 through 5 provide a brief overview of this work and Section 6 reports on lessons learned and some planned future work.

2. Memory Management for Lightweight Multithreading

Multithreading a shared memory address space is a technique proposed to express parallelism with reduced system overhead, and hide latency by overlapping costly operations with useful

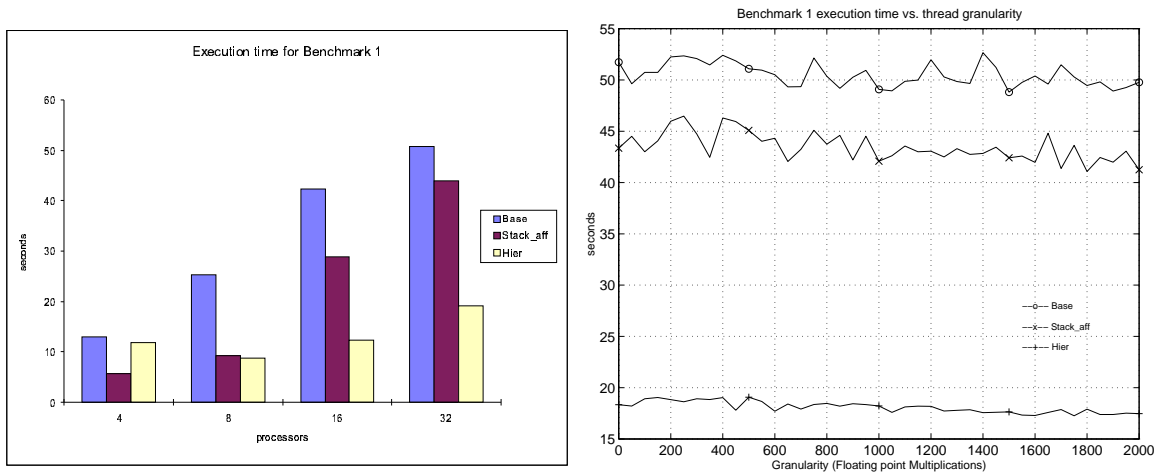


Figure 1: Performance of thread management alternatives on a 32-processor SGI Origin2000.

computation. The advances in parallelizing compilers and runtime systems technology enabled the extraction of fine-grain parallelism — in the order of tens or hundreds of processor instructions— from sequential applications. Multithreading was proposed as a viable solution for exploiting fine-grain parallelism, due to the low runtime cost of threads. Unfortunately, our early studies with a highly tuned multithreading runtime system on the SGI Origin2000 indicated that the cost of managing threads was so high, that the critical task size (i.e. the size beyond which parallelizing the task with multithreading makes sense) was at least a few thousand processor instructions. This trend clearly counteracts the ability of the compiler to discover fine-grain parallelism.

After analyzing the thread management mechanisms of several multithreading runtime systems, we detected that reducing the cost of managing threads translated naturally into a problem of poor memory performance of the actual thread management mechanisms. Operations such as thread creation, stack allocation and context switching incur frequent cache misses and remote memory accesses that exacerbate the cost of a thread’s life cycle (creation/execution/termination). Driven from this observation we designed and implemented two simple algorithms for managing the memory allocated for thread contexts, using distributed LIFO memory pools and a context recycle policy that enables frequent memory recycling from the processor caches. We matched memory pools with thread run queues in order to couple the threads scheduler with the memory allocator and we extended the basic policy to arbitrary memory hierarchies, by building a system of hierarchical memory pools and run queues for thread management. In this way, threads memory can be managed in closed Memory Locality Domains (MLDs), without incurring expensive remote memory accesses. Details on these mechanisms can be found in [9].

We implemented the mechanisms in the NanoThreads runtime library [8]. Figure 1 illustrates a sample of the results from a synthetic microbenchmark which evaluates the overhead of threads management in the NanoThreads library. The experiment shows that the devised mechanisms reduce the cost of thread management by 52% on average and their performance does not degrade as the granularity of threads becomes finer.

3. Scalable Synchronization Algorithms

Synchronization overhead is an intrusive source of bottlenecks in parallel programs for shared memory architectures. Several studies have shown that parallel programs may spend as much as half of their execution time in locks and barriers on moderate and large-scale DSM systems.

Algorithm	Processors						
	1	2	4	8	16	32	64
test-and-set coherent	1.00	1.00	1.00	1.00	1.00	1.00	1.00
test-and-set at-memory	1.04	1.23	1.12	1.02	1.01	2.17	2.85
test-and-set hybrid	1.04	1.23	1.11	1.00	0.98	2.12	1.81
queue lock coherent	1.03	1.21	1.09	1.00	1.00	2.04	1.95
queue lock hybrid	1.05	1.23	1.12	1.02	1.01	2.13	3.44
ticket lock coherent	1.03	1.21	1.12	1.01	1.00	1.14	0.76
ticket lock at-memory	0.97	0.80	0.88	0.97	1.00	0.47	0.50
ticket lock hybrid	1.03	1.25	1.12	1.01	1.00	2.17	4.00

Table 1: Raytrace performance with different lock primitives.

Although synchronization was extensively explored in the past, the sufficiency of scalable synchronization algorithms, as well as the ability of existent algorithms to actually improve the performance of parallel applications on modern DSM systems is an issue of considerable debate [4].

A recent study conducted by the authors [11] indicated that several highly sophisticated synchronization algorithms, such as concurrent queues and tree barriers, fail to scale on a 64-processor tightly integrated DSM system. Interestingly, the reason behind this behavior lies on the poor scalability of the elementary synchronization primitives employed by the algorithms (such as test&set and compare&swap) and not on algorithmic deficiencies. Elementary synchronization operations which are visible to the cache-coherence protocol (i.e. implemented on the microprocessor’s cache controllers) incur undue amounts of network contention and non-overlapped traffic due to coherence transactions performed on remote memory modules. The same study has shown that the scalability problems of elementary synchronization primitives can be resolved by implementing these primitives directly at the processing node’s DRAM memories with a single round-trip network transaction, thus making them invisible to the cache coherence protocol. This implementation requires minimal hardware support which is already available in commodity systems such as the SGI Origin2000 and the Cray T3E. In this case however, careful tuning of the implementations of synchronization algorithms is needed, in order to avoid dilating synchronization periods by creating a hot spot at the memory node which holds the synchronization flags.

We constructed a set of hybrid synchronization algorithms [14] that leverage hardware support to implement elementary synchronization operations at-memory and algorithmic support for exploiting the caches during the synchronization acquire and release phases. Our experimental results show that hybrid synchronization algorithms exhibit a significant potential for performance improvements on real applications with high synchronization overhead. Table 1 illustrates the speedups obtained by modifying the lock algorithms used in a parallel ray tracing program from the SPLASH-2 benchmark suite [20]. The experiments were conducted on a 64-processor SGI Origin2000. Hybrid locks provided up to 4-fold improvements on execution time compared to traditional lock implementations.

4. Virtual Memory Management for DSM Systems

Virtual memory management in complex memory hierarchies is a complicated task which requires a coordinated effort from the hardware and the operating system in order to meet the performance requirements of applications without underutilizing memory resources. In the case of DSM systems, managing virtual memory subsumes that the operating system will provide additional functionality for data alignment and topological collocation of computations with the

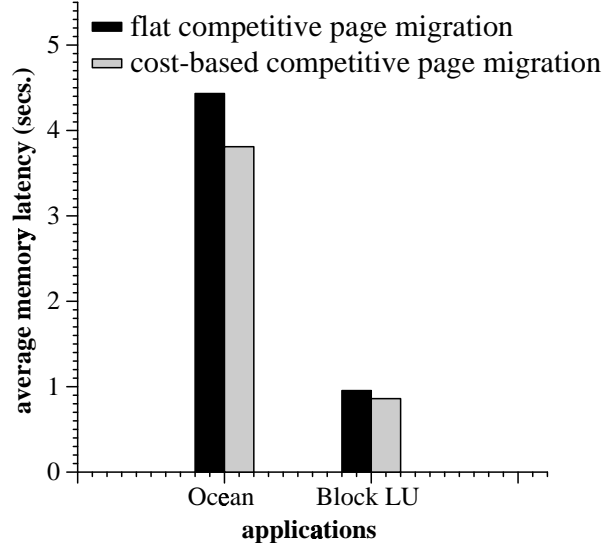


Figure 2: Average memory latency with two page migration algorithms (flat competitive and cost-based) for the SPLASH-2 OCEAN and LU benchmarks.

data that these computations use. In other words, processes should execute topologically close to their working sets and vice versa, i.e. the working set should be stored topologically close to the processes they belong to. The DSM property calls for memory allocation schemes that exploit Memory Locality Domains, i.e. clusters of neighboring memory and processor modules, where computations execute in isolation from other computations running on different modules [19].

Memory isolation schemes are not always sufficient for parallel programs that utilize large numbers of possibly distanced processor and memory modules, programs with dynamic and unpredictable memory access patterns, or even programs that migrate between processors due to the operating system scheduling strategy. In these cases, the hardware and the operating system should make special provisions for reducing the number of remote memory accesses incurred by the programs. Page-level coherence through dynamic page migration and replication [18] is a technique proposed to improve locality on DSM systems by dynamically moving virtual memory pages closer to the processors that actively use the pages more frequently. Previously proposed algorithms for page migration used flat competitive schemes, based on reference counters attached to the memory modules. Although such algorithms are already applied in real implementations (e.g. in Cellular IRIX), they rarely succeed in improving the memory performance of parallel applications, primarily due to the cost of page migration which outweighs the potential benefits. We propose a cost-based competitive algorithm for page migration that moves pages only if the potential gain from reducing remote memory accesses is expected to exceed the cost of migrating the page [15]. Figure 2 illustrates preliminary results extracted from an accurate execution-driven simulator of a 32-processor DSM architecture, running two benchmarks from the SPLASH-2 suite, a program that simulates the movement of ocean currents and a parallel blocked LU decomposition. Cost-based page migration reduces the average memory latency experienced by the processors by 14% on average.

5. Operating System Services for Scalable Multiprogramming

Parallel programs for shared memory multiprocessors suffer from poor scalability when executed in multiprogrammed environments. The primary reason is that the runtime systems on top of which parallel programs execute are oblivious of multiprogramming, while the operating system kernel is oblivious of fine-grain interactions in parallel programs such as synchronization or interdependencies between tasks. This lack of coordination has two undesirable effects:

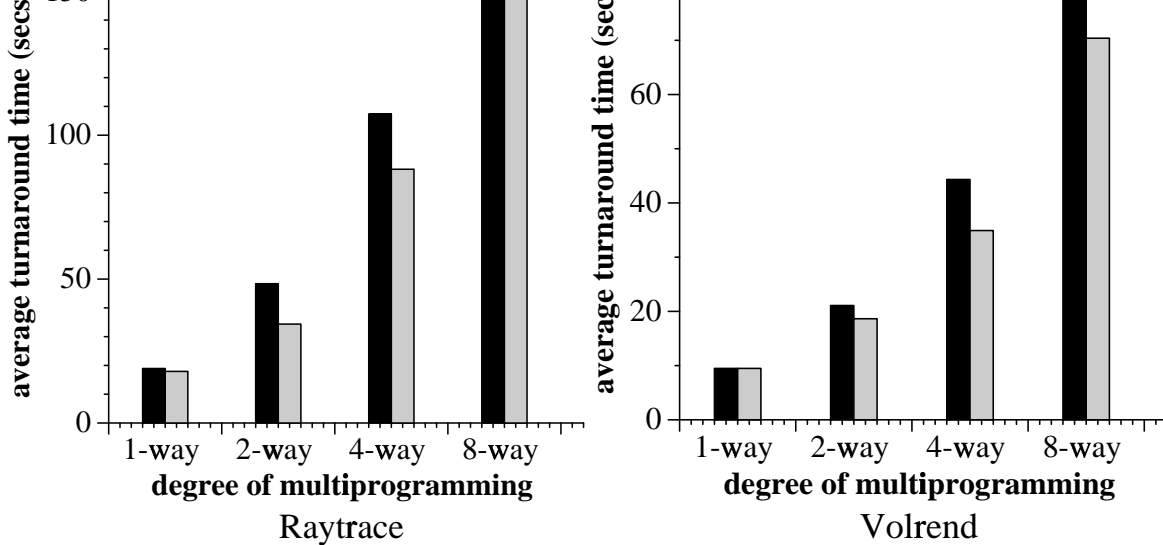


Figure 3: Performance of the SPLASH-2 Raytrace and Volrend benchmarks with the native Linux kernel and the Linux kernel enhanced with our nanothreading interface, under various degrees of multiprogramming.

malicious preemptions of threads from the operating system, which hurt the performance of synchronizing parallel programs and lead to processor underutilization; and unfortunate thread migrations which degrade memory performance by forcing excessive remote memory accesses from migrating threads.

We designed and implemented a lightweight kernel interface (the *nanothreading interface*), which exports to parallel programs all necessary functionality, in order to let them adapt themselves to multiprogrammed environments. Although the idea of providing such operating system services is not new, our implementation differentiated from previous related work in several aspects. We minimized the latency of the interface by exploiting shared memory as the communication medium and structuring the kernel-user communication data structures in a way that exploits data locality within both the kernel and the application spaces. The latter feature is particularly important for DSM architectures. We provided fast cloning paths for kernel threads through upcalls, as well as a robust and highly efficient mechanism for resuming maliciously preempted kernel threads, in order to ensure that parallel applications will always make progress along their critical path, even in the presence of a heavy multiprogramming load. The implemented mechanisms for multiprogramming adaptivity do not compromise memory performance, as they are coupled with scheduling policies that exploit the cache footprints of kernel threads during the processor allocation phase.

We initially implemented a simulator of the nanothreading interface in the Cellular IRIX operating system [16, 17] and proceeded to a complete implementation from scratch in the Linux kernel [13] and an implementation based on scheduler activations and the process control mechanism of the Solaris kernel [12]. All three implementations exhibited significant performance improvements compared to the native operating system kernels. Figure 3 for example, illustrates the performance improvements achieved from our nanothreading Linux kernel interface compared to the native Linux SMP kernel, when two benchmarks from the SPLASH-2 suite (ray tracing and volume rendering) are executed under various degrees of multiprogramming.

6. Conclusions

Coping with memory latency in modern DSM systems is a challenging problem that motivates in-depth investigations of both hardware and software mechanisms that transparently improve application performance without sacrificing the programmability of shared memory systems.

There are three lessons learned from our recent work on this topic. First, there is an enorm potential for performance improvements at all levels of system software for DSM systems. Many of these improvements may come at low implementation costs, by carefully tuning the memory performance of critical system services such as memory allocation and thread management. Second, proper performance tuning on state-of-the-art DSM systems can only be realized by understanding the synergy between hardware and software components. Unfortunately, system software vendors tend to underestimate the importance of the hardware/software interface and expect a lot from the programmer, thus making parallel programming difficult at best. Third, system software should exploit, whenever possible, additional hardware support such as cache bypassing, prefetching etc. for implementing critical system services.

The current trends in parallel computer architectures necessitate further and deeper investigation of the problem of memory latency. Parallel computing is moving to the desktop and high performance clusters are now built at extremely low-costs by leveraging off-the-shelf components. However, at the same time, memory hierarchies are becoming deeper and more complex, while the performance of applications on low-cost platforms lags far behind the performance of the same applications on tightly integrated supercomputers. Memory latency is once again the critical bottleneck and system software is expected to play a catalytic role in overcoming it.

Acknowledgements

This research was funded by NANOS (ESPRIT Framework IV, LTR Project No. 21907). We wish to thank Constantine Polychronopoulos for his help and useful advice throughout the course of this work.

References

- [1] D. Culler, J. P. Singh and A. Gupta. *Parallel Computer Architecture, a Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.
- [2] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. 2nd edition, Morgan Kaufmann Publishers, 1996.
- [3] D. Jiang and J. P. Singh. *Scaling Application Performance on Cache-Coherent Multiprocessors*. Proc. of the 26th Annual International Symposium on Computer Architecture, pp. 305–316, Atlanta (USA), 1999.
- [4] S. Kumar, D. Jiang, R. Chandra and J. P. Singh. *Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance*. Proc. of the 1999 ACM SIGMETRICS Conference, pp. 23–34, Atlanta(USA), 1999.
- [5] J. Laudon and D. Lenoski. *The SGI Origin2000: A ccNUMA Highly Scalable Server*. Proc. of the 24th Annual International Symposium on Computer Architecture, pp. 241–251, Denver (USA), 1997.
- [6] T. Lovett and R. Clapp. *STiNG: A CC-NUMA Computer System for the Commercial Marketplace*. Proc. of the 23rd Annual International Symposium on Computer Architecture, Philadelphia (USA), 1996.
- [7] The NANOS Project Consortium. *NANOS: Effective Integration of Fine-Grain Parallelism Exploitation and Multiprogramming*. ESPRIT IV Framework Project No. 21907. <http://www.ac.upc.es/NANOS>.
- [8] The NANOS Project Consortium. *NANOS User-level Threads Library Implementation*. Deliverable M2.D2, 1998.

- [9] D. Nikolopoulos, E. Polychronopoulos and T. Papatheodorou. *Efficient Runtime Thread Management for the Nano-Threads Programming Model*. Proc. of the 2nd IEEE IPPS/SPDP Workshop on Runtime Systems for Parallel Programming, pp. 183–194, Orlando (USA), 1998.
- [10] D. Nikolopoulos, E. Polychronopoulos and T. Papatheodorou. *Enhancing the Performance of Autoscheduling on Distributed Shared Memory Multiprocessors*. Proc. of the 4th International EuroPar Conference, pp. 491–501, Southampton (England), 1998.
- [11] D. Nikolopoulos and T. Papatheodorou. *A Quantitative Architectural Evaluation of Synchronization Algorithms and Disciplines on ccNUMA Systems: The Case of the SGI Origin2000*. Proc. of the 13th ACM International Conference on Supercomputing, pp. 319–328, 1999.
- [12] D. Nikolopoulos, E. Polychronopoulos and T. Papatheodorou. *Fine-Grain and Multiprogramming-Conscious Nanothreading with the Solaris Operating System*. Proc. of the 5th International Conference on Parallel and Distributed Processing Techniques and Applications, Vol. IV, pp. 1797–1803, Las Vegas (USA), 1999.
- [13] D. Nikolopoulos, C. Antonopoulos, I. Venetis, P. Hadjidoukas, E. Polychronopoulos and T. Papatheodorou. *Achieving Multiprogramming Scalability of Parallel Programs on Intel SMP Platforms: Nanothreading in the Linux Kernel*. Proc. of the Parallel Computing'99 (ParCo'99) Conference, Delft (The Netherlands), 1999.
- [14] D. Nikolopoulos and T. Papatheodorou. *Scalable Synchronization on Large-Scale Shared Memory Multiprocessors Using Hybrid Primitives*. Technical Report 010799, High Performance Information Systems Laboratory, University of Patras, 1999.
- [15] D. Nikolopoulos and T. Papatheodorou. *Cost and Distance-Based Page Migration Algorithms*. Technical Report 020799, High Performance Information Systems Laboratory, University of Patras, 1999.
- [16] E. Polychronopoulos, X. Martorell, D. Nikolopoulos, J. Labarta, T. Papatheodorou and N. Navarro. *Kernel-Level Scheduling for the Nano-Threads Programming Model*. Proc. of the 12th ACM International Conference on Supercomputing, pp. 337–344, Melbourne (Australia), 1998.
- [17] E. Polychronopoulos, D. Nikolopoulos, T. Papatheodorou, N. Navarro and X. Martorell. *An Efficient Kernel-Level Scheduling Methodology for Multiprogrammed Shared Memory Multiprocessors*. Proc. of the 12th International Conference on Parallel and Distributed Computing Systems, Fort Lauderdale (USA), 1999.
- [18] B. Verghese, S. Devine, A. Gupta and M. Rosenblum. *Operating System Support for Improving Data Locality on ccNUMA Compute Servers*. Proc of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 279–289, Cambridge (USA), 1996.
- [19] B. Verghese, A. Gupta and M. Rosenblum. *Performance Isolation: Sharing and Isolation in Shared Memory Multiprocessors*. Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 181–192, San Jose (USA), 1998.
- [20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta, *The SPLASH-2 Programs: Characterization and Methodological Considerations*. Proc. of the 22nd Annual International Symposium on Computer Architecture, pp. 24–36, Santa Margherita Ligure (Italy), 1995.