A methodology to characterize critical section bottlenecks in DSM multiprocessors *

Benjamín Sahelices[1], Pablo Ibáñez[2], Víctor Viñals[2], and J.M. Llabería[3]

[1] Depto. de Informática. Univ. de Valladolid. benja@infor.uva.es
[2] Depto. de Informática e Ing. de Sistemas, I3A and HiPEAC. Univ. de Zaragoza. {imarin, victor}@unizar.es
[3] Depto. de Arquitectura de Computadores. Univ. Polit. de Cataluña. llaberia@infor.uva.es

Abstract. Understanding and optimizing the synchronization operations of parallel programs in distributed shared memory multiprocessors (DSM), is one of the most important factors leading to significant reductions in execution time.
This paper introduces a new methodology for tuning performance of parallel programs. We focus on the critical sections used to assure exclusive access to critical resources and data structures, proposing a specific dynamic characterization of every critical section in order to a) measure the lock contention, b) measure the degree of data sharing in consecutive executions, and c) break down the execution time, reflecting the different overheads that can appear. All the required measurements are taken using a multiprocessor simulator with a detailed timing model of the processor and memory system.
We propose also a static classification of critical sections that takes into account how locks are associated with their protected data. The dynamic characterization and the static classification are correlated to identify key critical sections and infer code optimization opportunities (e.g. data layout), which when applied can lead to significant reductions in execution time (up to 33 % in the SPLASH-2 scientific benchmark suite). By using the simulator we can also evaluate whether the performance of the applied code optimizations is sensitive to common hardware optimizations or not.

1 Introduction

Shared memory multiprocessors use the memory as a means to synchronize and communicate processors. But, if several processors try to execute the same critical section concurrently by enforcing the required serialization, the scalability of parallel programs is often limited. When this happens, optimizing the lock ownership transfer among processors is essential for achieving high performance.

* This work was supported in part by Diputación General de Aragón grant "gaZ: Grupo Consolidado de Investigación", Spanish Ministry of Education and Science grants TIN2007-66423, TIN2007-60625, Consolider CSD2007-00050, and the european HiPEAC-2 NoE.

To understand the opportunities for tuning critical sections, fine grain metrics such as lock time and time distribution inside the critical sections are required. Also, in order to identify and quantify bottlenecks in parallel applications, a precise control over the environment where the measures are taken is required. For these reasons, we use a multiprocessor simulator (RSIM) that models out-of-order processors, memory hierarchy and interconnection network in detail [1,2]. Within our simulation environment it is possible to take fine grain statistics, break down the execution time of critical sections accurately, or turn on/off architectural enhancements, which is difficult to accomplish in real systems. Unlike other simulation environments which generate a trace using a trace driven simulator and later need to define an interleave of memory reference streams, our simulator already interleaves the memory reference streams [3].

In this work we suggest a two-step approach. In the first step we compute the execution time of all the critical sections, separating lock management and shared data access. We also characterize critical sections *dynamically* by quantifying two other important features: a) lock contention and, b) degree of data sharing. As a measure of lock contention we use the number of processors trying to access a lock variable when the lock is released, and to identify the degree of data sharing we measure the number of different shared data lines accessed in consecutive executions of the critical section. Finally, we further classify critical sections *statically* in two types, depending on how the data structure is protected by the lock variables. As we will see, we will combine the dynamic characterization and the static classification in order to identify the key critical sections and infer code optimization opportunities, such as data layout optimizations [4,5,6]. In this step the dynamic statistics are gathered in a Baseline system. The Baseline system has no architectural enhancements that might introduce noise in the measures, such as prefetching [7] or directory caching [8,9].

In the second step, we turn on the architectural enhancements and evaluate again the performance of the applied optimizations. From this second run we realize that the applied optimizations increase performance on the enhanced system as well.

The structure of the paper is as follows: Section 2 details the proposed characterization and classification procedure. Section 3 presents the simulation environment and the used benchmarks. In Section 4 we show the static classification of all SPLASH-2 critical sections. Then, in Section 5 we gather all the dynamic statistics, select and optimize the key critical sections, and compare the performance gains in both the Baseline and the enhanced systems. Lastly, the related work and conclusions Sections follow.

2 Static classification and dynamic characterization of Critical Sections

This Section introduces first the static classification and then the execution time breakdown. Next we describe the twofold characterization of critical sections according to: a) lock contention and b) degree of data sharing. Finally, it is

explained how the methodology helps to detect and optimize code and data layout inefficiencies.

Static classification. By looking at the source code we classify critical sections into two types: in type A a single lock is used to protect access to a shared data structure, and in type B a lock array is used to protect fine-grained access to structured shared data. Each type is further split into two ad-hoc subtypes, 0 and 1. Inside the critical sections of type A.0 only a single shared variable is usually updated. Critical sections of type A.1 protect the access to several variables, for instance, when managing linked lists of shared-memory chunks. Type B.0 critical sections are used to protect access to task queues; here each processor has assigned a single task queue and only when this is empty the processor searches the task queues of other processors. Type B.1 identifies critical sections used to gain access to big data structures, where one entry of the lock array is used to protect the access to a data structure subset.

Time breakdown of critical section execution. Locks are used to assure exclusive access to shared data when executing a code section [10,11,12]. A processor requests the lock through a lock acquire operation (A) and frees it with a release operation (R), see Figure 1. In this paper we use a Test-and-Test&Set algorithm to acquire a lock. We call *lock time* the sum of acquire and release times, and critical section *Latency* the time between the end of the acquire operation and the beginning of the release operation. We can further split the acquire time into the waiting time (processor is spinning) and the hand-off time (it is being decided which processor enters next). Throughout the simulation we measure Lock time and Latency for each execution instance of every critical section.

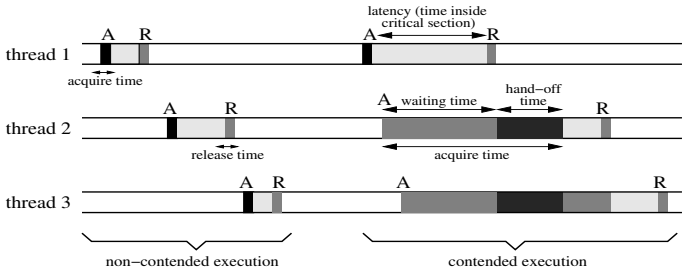

Fig. 1. Time breakdown of contended and non-contended execution of a critical section

When there is contention the Lock time is proportional to the Latency times the number of contending processors. So, small Latency reductions can have a large impact on the Lock time.

Lock contention. As a measure of lock contention we use the number of processors trying to access a lock variable when the critical section is released. The lock time of a critical section is directly related to its lock contention.

Data sharing patterns. To measure the degree of data sharing we compare the data cache lines accessed in consecutive executions of a critical section. We

distinguish three sharing patterns, namely *full-sharing*, *no-sharing*, and *some-sharing*. Full-sharing arises when a critical section protects the same line(s) in two consecutive executions. No-sharing arises when none of the lines accessed in a critical section has been used in the two previous executions. Finally, some-sharing collects the remaining execution instances that show an intermediate sharing pattern.

2.1 Detecting inefficiencies and inferring optimizations

In order to choose the critical sections key to the overall performance we add, for all instances of each critical section, the Lock time and the Latency, and select those that represent a relative high fraction of the parallel execution time.

We use the Latency in each execution of a key critical section in order to infer the possibility of a data layout problem and then inspect the code. For instance, if the Latency is not biased towards a few values there could be false sharing. This is particularly true in the type A.0. Also, by analyzing the critical section Latency, bad programming habits can be detected. We will see some examples later.

When a contended lock shows low degrees of data sharing, the contention is not caused by the simultaneous access to the shared data, but by the simultaneous access to the lock variable. In other words, the same contended lock is used to protect the access to different shared data in consecutive executions. For instance, a programming technique to protect the access to complex data structures is using a lock array and a hash function to distribute the concurrently accessed elements of the data structure uniformly among the elements of the lock array. If some elements of the lock array have high contention and low data sharing, performance may be improved by using a better hash function, by increasing the lock array size or both.

Section 5 details all the detected inefficiencies and the feasible solutions. Anyway, the most often employed optimizations to tackle the Latency problem consist in reorganizing the data layout to either remove false sharing [4] or/and enforce collocation of the lock and the shared variables [5]. For type B.1 the action taken has been to increase the number of entries in the lock array (finer granularity protection).

3 Baseline system and benchmarks

The runtime statistics are obtained with RSIM [1,2]. RSIM is an execution-driven simulator performing a detailed cycle-by-cycle simulation of an out-of-order processor, a memory hierarchy and an interconnection network. The simulator models all data movements including the cache port contention at all levels. The processor implements a sequential consistency model using speculative load execution [7]. The Baseline system is a distributed shared-memory multiprocessor with a MESI cache coherence protocol [13]. The interconnection is a wormhole-routed, two-dimensional mesh network. The contention of ports, switches and links is also modeled accurately. Table 1 lists the processor, cache and memory system parameters as well as the tested SPLASH-2 benchmarks [14].

Processor	32 processors, 1 Ghz	**L2/Memory Bus**	Split.32 B,3 cycle+1 arbit.
ROB	64 entry, 32 entry LS queue	**Memory**	4 way interl., 100 cycle DRAM
Issue	out−of−order issue/commit 4 ops/cyc.	**Directory**	SGI Origin−2000 based MESI
Branch	512 entry branch predictor buffer	Cycle	16 cycle (without memory)
Cache		Interleaving	4 controllers per node
L1 inst.	Perfect	**Network**	Pipelined point−to−point
L1 data	32 Kbyte, 4 way assoc., write−back	Network width	8 bytes/flit
	2 ports, 1 cycle, 16 outstanding misses	Switch buffer size	64 flits
L2	1 Mbyte, 4 way associative, write−back	Switch latency	4 cycles/flit + 4 arbit.
	10 cycle access, 16 outstanding misses		
L1/L2 bus	Runs at processor clock		
Line size	64 bytes		

Code	Ocean	Barnes	Volrend	Water−Nsq	Water−Spt	Radiosity	FMM	Raytrace
Input	258x258	16Kpartic.	head	512 molec.	512 molec.	test	16K partic.	Car 256

Table 1. Baseline system parameters and benchmarks.

3.1 Execution time breakdown

We use the mean of the parallel phase execution time across all processors as the main metric to analyze performance improvements. The execution time of each application is split into four categories according to the system components causing each cycle loss. Instructions are assigned to the *Lock* (acquiring and releasing critical sections) and *Barrier* categories by using software marks. In the remaining code (not enclosed between the software marks), attributing stall cycles to specific components is complex in multiprocessor systems with out-of-order processors, because many events happen simultaneously. The algorithm used to categorize the remaining execution time is taken from the work of Pai et al. in [2]. The employed categories are *Compute* and *Memory*. Of course, the *Memory* category includes shared variable accesses both inside and outside critical sections.

3.2 Benchmarks

Table 1 shows the SPLASH-2 benchmarks [14] we tested. The applications have been compiled with a Test-and-Test&Set based synchronization library implemented with a Read-Modify-Write (*ldstub* in sparc) type operation. Every lock variable does not share its cache line with another lock, even in array locks. This has been achieved employing padding. Barriers are implemented with a binary tree. Radiosity and Volrend can be considered non-deterministic because different execution paths of the tasks can produce different number of iterations on an active waiting loop and, as a consequence, the execution time experiences minor variations.

We simulate completely the parallel phase of each benchmark. In Figure 2 we show the execution time breakdown of the *base* experiment that will be used

later to compare with the optimized options. As can be seen the Lock time is noticeably high, ranging from 1 % in FMM to 50 % in Radiosity.

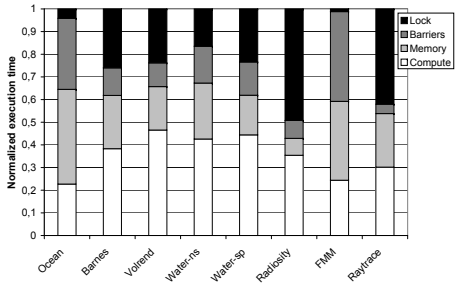

Fig. 2. Parallel execution time of the SPLASH-2 benchmarks

4 Critical section classification in SPLASH-2

In this section we analyze the code of each critical section of the SPLASH-2 benchmarks and we classify them in the types stated in Section 2.

Benchmark	**type A.0 Locks** SHVar += value	**type A.1 Locks** Several variables (e.g. linked lists)	**type B.0 Locks** Task queues	**type B.1 Locks** Complex data structures
Ocean	Id,Psiai,Psibi,Error			
Barnes	Count	Count		Cell[0:2047]
Volrend	Index,Count,QLock[32]		QLock[0:31]	
Water-ns	Index, IntrafVir, InterfVir, KinetiSum	PotengSum		Mol[0:511]
Water-sp	Index, IntrafVir, InterfVir, KinetiSum	PotengSum		
Radiosity	Index, Pbar, TaskCounter	SHLocks(0:700), free_patch, free_interaction, free_edge, free_element, free_elemvertex, avg_radiosity	tq[0:31].flock tq[0:31].qlock	
FMM	Count	Mal		LockArray[0:1235]
Raytrace	Pid	Mem	WP[0:31]	

Table 2. Critical section classification of the SPLASH-2 benchmarks. *Count* in Volrend protects two different codes in different execution phases, one is A.0 and the other A.1.

As can be seen in Table 2, type A accumulates the highest number of critical sections (21) while Type B has only 7 members. The simplest A.0 class is the most populated and covers all the applications, followed in importance by A.1. Regarding type B, it is worth noting that subtypes B.0 and B.1 only appear each in three out of the eight applications.

5 Experimental results

In this section we first characterize each application according to lock contention and degree of data sharing. Second, we identify the key critical sections according to their influence in the overall execution time. Third, for every key critical section we analyze contention, sharing patterns and Latency. Fourth, we focus on false sharing removing and opportunities for collocation and other code optimizations. Finally, we show a sensitivity analysis of the optimized code performance with respect to hardware enhancements.

Critical section characterization. Figure 3 (a) plots data sharing (y-axis) and contention (x-axis) for each application, placing each critical section in only one of the four regions. A critical section is placed in the *No Data Sharing* half if its dominant sharing pattern is no-sharing. On the other hand, the critical section is placed on the *Contention* half if more than 5 % of its instances execute with 2 or more processors trying to enter the critical section. Then for each application we group all the critical sections of a given region in a ball, weighting its area according to the aggregated number of execution instances[4]. The Lock time of all the critical sections included in a ball, with respect to the parallel execution time, is plotted close to each ball.

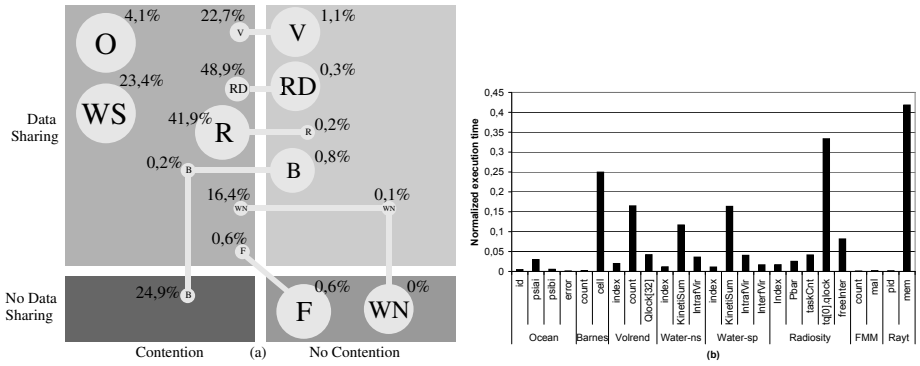

Fig. 3. (a) SPLASH-2 characterization based on contention and data sharing of their critical sections. (O:Ocean, WS:Water-sp, WN:Water-ns, B:Barnes, V:Volrend, RD:Radiosity, R:Raytrace, F:FMM). (b) Lock time of key critical sections

Lock time. To identify the key critical sections we analyze the contribution of each critical section to the parallel execution time, selecting the most relevant. Figure 3 (b) plots the Lock time of relevant critical sections normalized to the parallel execution time. In several applications one single critical section has a

[4] The ball area is proportional to the sum of the number of executions of all the critical sections grouped in the ball, relative to the number of executions of all the critical sections of the application.

significant impact on the execution time due to its high contention (eg. tq[0].qlock in Radiosity and Mem in Raytrace).

Contention. Figure 4 (a) shows the average number of waiting processors for the key critical sections. We see, on average, between 7 and 10 processors waiting on each execution. A critical section can show a highly contended behavior whereas its contribution to the parallel execution time may be low. For example, three critical sections in Ocean have high contention (14 processors on average) but their impact on the parallel execution time is low because they execute only a few times, see Figure 3 (b).

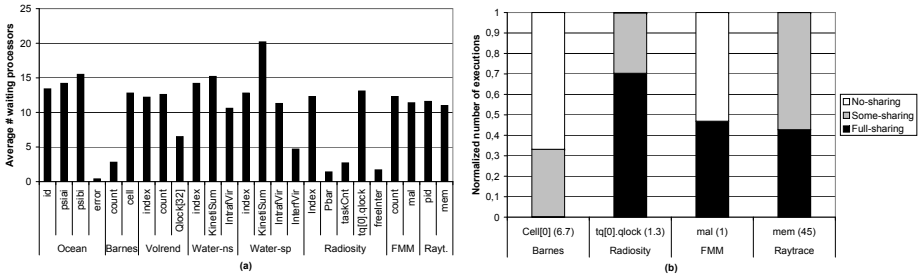

Fig. 4. (a) Average number of processors waiting on each key critical section execution. (b) Degree of data sharing of key critical sections not belonging to A.0 type; the average number of referenced cache lines appears in brackets.

Sharing patterns. In Figure 4 (b) we show the normalized number of occurrences of each sharing pattern for the key critical sections that are not A.0 type. The average number of referenced cache lines on each critical section execution appears in brackets. Of course, critical sections of A.0 type show only full-sharing. The locks tq[0].qlock (Radiosity) and Mem (Raytrace) show high degree of data sharing. The lock Cell in Barnes has a very low degree of data sharing and, finally, Mal in FMM shows equal amounts of full-sharing and no-sharing.

Latency. Finally, in Figure 5 (a) we show the average Latency, in execution cycles, of the key critical sections. In each bar we show compute and memory time. We see that in all the critical sections except two, the Latency almost equals the memory component. Note that the numbers are quite high, for instance, roughly half the critical sections spend 1000 cycles or more accessing memory. Therefore, reducing the memory component clearly appears as the main target of any optimization we are able to apply. Remember that the Lock time is proportional to the Latency times the number of contending processors, and so small Latency reductions can have a large impact on the Lock time.

5.1 Data layout and code optimizations

First we deal with removing inefficiencies, detecting false sharing and applying other code optimizations. Next, we apply collocation, verifying that it achieves a positive effect.

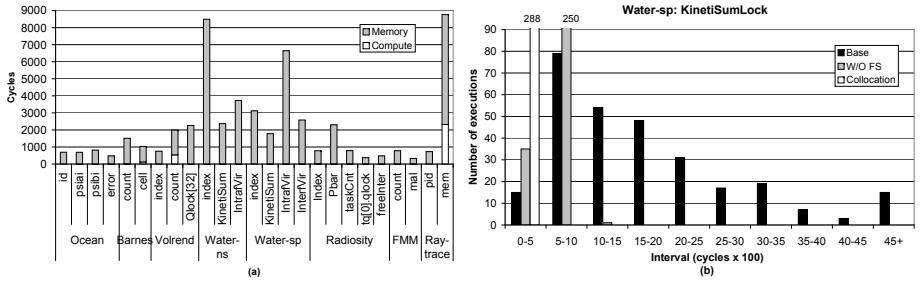

Fig. 5. (a) Average Latency of key critical sections. (b) Number of executions of the critical section *KinetiSum* (Water-sp) in each Latency interval for the Base experiment (*Base*), after removing false sharing (*W/O FS*) and after applying Collocation (*Collocation*).

When applying these code optimizations some changes in the Barrier time measured in Figure 2 can be observed. Due to optimization, the Barrier time can increase if some of the processors, but not the last one, arrive earlier to the barrier, but it can decrease if all the processors arrive earlier to the barrier. In Radiosity and Volrend there is an active waiting loop just before the barrier, therefore when the code is optimized the number of iterations will change and so will all the Barrier, Compute and Memory times.

False sharing. Comparing Figure 5 (a) with the classification in Table 2 we find some critical sections with an unusual behavior. For example in Water-sp the critical section KinetiSum has an average Latency of 1781 cycles but it is of A.0 type and does not have any conditional statement. Note that it is a too high Latency for a critical section that only reads and writes a single variable once (typical in type A.0 critical sections). Actually, if we add the raw latencies of whatever two memory accesses (276 and 377 cycles in average) and the negligible compute time, the resulting time is clearly lower than the measured Latency. In Figure 5 (b) we perform a more detailed analysis of this critical section by plotting the number of critical section executions (bar height) whose Latency is in a given interval (x-axis category). In the base experiment of this figure we see that there are significant variation in the measured Latency.

Analyzing the code of KinetiSum we observe the existence of false sharing [4,6]. Figure 5 (b) also shows the KinetiSum Latency once the false sharing has been removed through data padding (experiment W/O FS). We can see now that the Latency of almost all the executions is below 1000 cycles. Using this kind of Latency analysis the false sharing behavior is also detected in Ocean, Raytrace and Water-ns. The execution time without false sharing is shown in Figure 6 (a) (experiment W/O FS).

Optimizations in the code. In Barnes there is one specific lock belonging to the lock array Cell (type B.1) with high Lock time (see Figure 3 (b)), high Latency ($\simeq$ 1000 cycles, see Figure 5 (a)) and very low data sharing (see Figure

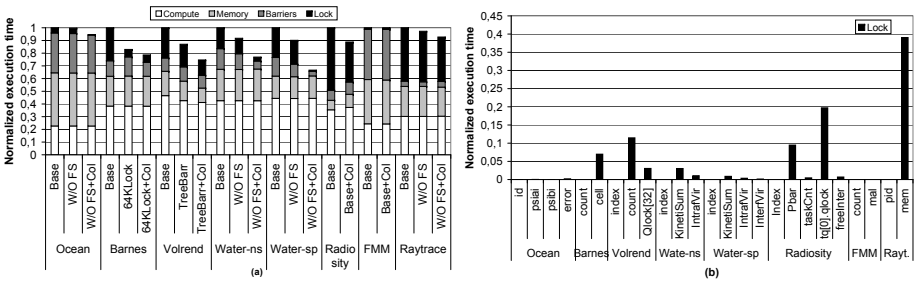

Fig. 6. (a) Normalized execution time for the base experiment (*Base*) after removing false sharing (*W/O FS*), increasing the size of the lock array in Barnes (*64KLock*), replacing a lock-based barrier with a tree barrier in Volrend (*TreeBarr*), and applying Collocation (*Col*). (b) Lock time of key critical sections after applying all optimizations

4 (b)). A B.1 type contended lock showing a low degree of data sharing indicates a too much coarse-grained protection of the shared data. Therefore, we have increased the lock array Cell from 2048 to 65536 locks, reducing 77 % Barnes lock time (Figure 6 (a), experiment *64KLock*).

In the Volrend benchmark the lock Count has high Lock time and Latency (see Figures 3 (b) and 5 (a)), but it does not suffer from false sharing. Analyzing the code we see that Count is of type A.0 but it implements a barrier by means of a shared counter. We replaced it with a simple binary tree barrier and results are shown in Figure 6 (a), experiment *TreeBarr*. All components in the execution time are reduced except the barrier component because the time of the new tree barrier is accumulated to the barrier category. The other categories decrease because one lock, the corresponding critical section and its control loop are eliminated.

Collocation. When several variables stored in the same line are used by different processors performance degrades (false sharing) but, on the other hand, if these variables are used by the same processor performance can improve. The software technique named Collocation aims at decreasing the latencies associated with data accesses by placing the lock and the variables together in the same line [5]. Bar *Collocation* in Figure 5 (b) shows the KinetiSum Latency distribution when Collocation is applied. We can see that all the KinetiSum executions have a Latency below 500 cycles (with an average Latency of 25 cycles). This is because the read and write accesses to the protected variable now almost always hit in the cache.

The bars *+Col* in Figure 6 (a) show the execution time breakdown when Collocation is applied on top of the previous optimizations. As can be seen, collocation decreases execution time further on all applications, between 5.2 % in Ocean and 23.0 % in Water-sp, except in FMM. In order to complete the picture, Figure 6 (b) shows the Lock time of the key critical sections. As we

can see by comparing it with Figure 3 (b) the Lock time has been eliminated or significantly reduced in most key critical sections.

Summarizing, after applying all the optimizations suggested in our methodology, the Baseline system execution time is reduced between 5.5 % in Ocean and 33.4 % in Water-sp. However, there is no reduction in FMM, which has no Lock time and therefore is insensitive to these kinds of optimizations. A significant lock time remains in Radiosity and Raytrace mostly, which would require to restructure two key critical sections, namely *tq[0].qlock* in Radiosity and *Mem* in Raytrace.

Sensitivity of code optimizations. In this section we evaluate the sensitivity of code optimizations to architectural enhancements. First we simulate processors that prefetch on the memory operations blocked by the memory consistency constraints, and that perform store buffering [7]. Later we simulate a directory cache in the coherence controller (four-entry fully associative) whose entries hold both the data line and its directory information, and are looked up before in the directory [15,8,9]. A line is placed in the directory cache only when it is requested by a processor and it is in Exclusive state in another processor node.

Figure 7 shows execution time of the enhanced processors normalized to the base experiment, running both the original and the optimized codes, including the base experiment for comparison. As can be seen, the enhanced processors run faster, except with the directory cache in Raytrace. In Raytrace, within the critical section *Mem*, the number of accessed and updated lines is high which causes a lot of capacity misses in the directory cache. Anyway, and more importantly, the optimized code on the enhanced processors always executes in less time, excluding FMM which is insensitive to optimization.

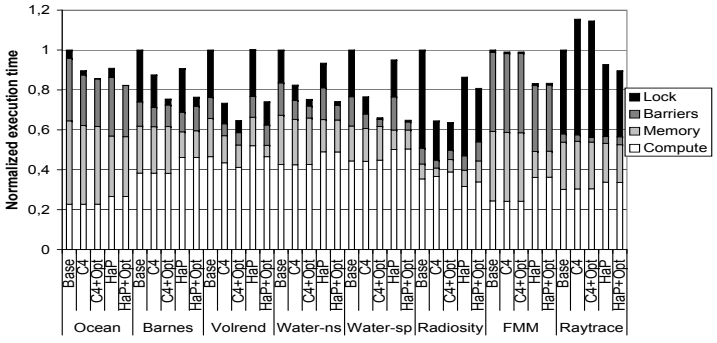

Fig. 7. Execution time normalized to the base experiment of several architectural enhancements with and without optimizations: four-entry cache in the coherence controller (*C4*, and *C4+Opt*), prefetch and store buffering (*HaP* and *HaP+Opt*).

6 Related work

Contemporary processors provide hardware counters that return a count of either cycles or performance-related events. Performance analysis tools insert library calls in the program to query the hardware counters during program execution. These tools provide graphical interface and correlate performance metrics with the program source code [16,17]. Instead, we use a simulator allowing to measure the overheads at the instruction level and to eliminate the nondeterminism introduced by real hardware. Therefore, we can take fine grain statistics, analyze data address streams or break down the execution time of critical sections accurately, which is difficult to accomplish in real systems. Also, we are able to connect and disconnect architectural optimizations that might introduce noise in the measures used in a first-step analysis.

Other performance analysis tools instrument the source program in order to obtain statistics or address traces but, as a consequence, the data address space can be disturbed [3,18]. The disruption may affect the absolute position of the data elements, which in turn may affect the cache behavior. Moreover, tools that use trace driven simulation require to implement a given interleave of the memory reference streams [3]. Although we also use hooks to instrument the code, the simulator allows us to eliminate all bookkeeping operations introduced by instrumentation. Besides, the execution-driven simulation already interleaves the memory reference streams in a straightforward way.

7 Concluding remarks

Critical sections are used in parallel applications to ensure serialized access to shared data structures, leading to a potential performance bottleneck. In this paper we propose a methodology to characterize and classify critical sections in order to guide optimizations in DSM multiprocessors. To characterize a critical section we use three parameters measured with an execution driven simulator that allows to take fine grain statistics: a) lock contention, b) degree of data sharing in consecutive executions of the same critical section, and c) Latency. The fine grain statistics taken in the simulation framework are correlated with a static classification. The static classification takes into account how locks are associated with their protected data.

Correlating the static classification of each critical section with its dynamic characterization, inefficiencies may be detected, isolated and understood. Characterization is used to guide optimization opportunities and classification allows us to identify the best suited optimization for each application. Data layout optimizations in critical sections have been introduced to reduce the execution time. We show that, for a highly contended short critical section with high degree of data sharing, data layout optimizations not only reduce the critical section Latency but also, and to a large extent, the Lock time, significantly reducing the execution time. Moreover, we show that the proposed optimizations hold their effectiveness even when considering architectural enhancements in the processor and the coherence controller.

In this work we have optimized the original version of SPLASH-2 for a DSM environment by carefully tuning the lock-based synchronization performance. In a future work we plan to test the effectiveness of the proposed optimizations in a design with very different architectural tradeoffs, such as for example, a chip multiprocessor.

References

1. Fernández, R., García, J.: RSIM X86:a cost-effective performance simulator. In: Proc. 19th European Conference on Modelling and Simulation ECMS. (2005)
2. Pai, V., Ranganathan, P., Adve, S.: RSIM reference manual version 1.0. Technical report 9705, Dept. Electrical and Computer Eng., Rice University (1997)
3. Marathe, J., Mueller, F.: Source-code-correlated cache coherence characterization of OpenMP benchmarks. Parallel and Distributed Systems, IEEE Transactions on **18**(6) (2007) 818–834
4. Eggers, S., Jeremiassen, T.: Eliminating false sharing. In: Proc. Int. Conf. Parallel Processing. Vol I. (1991) 377–381
5. Kagi, A., Burger, D., Goodman, J.: Efficient synchronization: let them eat QOLB. In: Proc. 24th ISCA. (1997) 170–180
6. Torrellas, J., Lam, M., Hennessy, J.: False sharing and spatial locality in multiprocessor caches. IEEE Trans. Computers **43**(6) (1994) 651–663
7. Gharachorloo, K., Gupta, A., Hennessy, J.: Two techniques to enhance the performance of memory consistency models. In: Proc. ICPP. (1991) 355–364
8. Michael, M., Nanda, A.: Design and performance of directory caches for scalable shared memory multiprocessors. In: Proc. 5th HPCA. (1999)
9. Woodacre, M., Robb, D., Roe, D., Feind, K.: The SGI altix 3000 global shared-memory architecture. White paper silicon graphics inc., SGI (2003)
10. Anderson, T.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel and Distrib. Systems **1**(1) (1990) 6–16
11. Graunke, G., Thakkar, S.: Synchronization algorithms for shared memory multiprocessors. IEEE Computer **23**(6) (1990) 60–69
12. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared memory multiprocessors. ACM Trans. Computer Systems **9**(1) (1991) 21–65
13. Laudon, J., Lenoski, D.: The SGI Origin: A CC-NUMA highly scalable server. In: Proc. 24th ISCA. (1997)
14. Woo, S., et al.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proc. 22th ISCA. (1995) 24–36
15. Acacio, M., González, J., García, J., Duato, J.: Owner prediction for accelerating cache-to-cache transfer misses in a CC-NUMA architecture. In: Proc. 16th Int. Conf. on Supercomputing. (2002)
16. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. ACM/IEEE Supercomputing Conference (2000) 42–42
17. De Rose, L., Reed, D.: Svpablo: A multi-language architecture-independent performance analysis system. In: Int.Conf. Parallel Processing. (1999) 311–318
18. Mellor-Crummey, J., Fowler, R., Whalley, D.: Tools for application-oriented performance tuning. In: Proc. 15th Int. Conf. Supercomput. (2001) 154–165