Cautious, Machine-Independent Performance Tuning for Shared-Memory Multiprocessors

Sarah A. M. Talbot, Andrew J. Bennett, Paul H. J. Kelly

Department of Computing
Imperial College of Science, Technology and Medicine
London SW7 2BZ

Abstract. Coherent-cache shared-memory architectures often give disappointing performance which can be alleviated by manual tuning. We describe a new trace analysis tool, CLARISSA, which helps diagnose problems and pinpoint their causes. Unusually, CLARISSA works by analysing potential contention, instead of measuring predicted contention by simulating a specific memory system design. This is important because, after tuning, the software will be executed on different inputs and different configurations. The goal is to produce a program with robustly good performance. This paper explains the principle behind cautious trace analysis, describes our implementation, and presents our experience of using the tool.

1 Introduction

There has been considerable recent interest in developing tools to support manual performance optimisation of applications running on coherent-cache shared-memory multiprocessors (e.g. [1, 3]). The purpose of a performance tuning tool is to direct the programmer's attention to where a program is spending its time and to give as much guidance as possible into how to reduce the performance bottlenecks.

Existing performance tools measure (using special monitoring circuitry) or predict (using a simulation of the shared memory architecture) the behaviour of the machine for which the program is being developed. Although this is very useful in understanding the factors influencing performance, there are two fundamental problems in using such tools for producing high-quality software:

1. In the field, the software will be run with many different inputs, leading to behaviour different from that seen during tuning, and
2. The software will be installed on hardware with different characteristics from that used during tuning.

In this paper we present an alternative approach, *cautious trace analysis*, aimed at addressing these problems. The key idea is to identify behaviour which might lead to lost performance on some reasonable architecture, or with different timing assumptions. If we can diagnose and eliminate, or at least minimise, these characteristics, the program should behave well in service.

2 Cache Line Contention in Shared-Memory Systems

A shared-memory multiprocessor consists of several CPUs with associated caches linked to memory units via an interconnection network. A cache coherency protocol is required to ensure that CPUs do not use stale cached data. In addition to the overheads of maintaining coherence, such architectures can suffer from three problems: contention for nodes, cache lines and communication links. These all conspire to increase memory access times, and hence slow down the execution time of tasks running on the processors. The challenge is to minimise the causes of contention, i.e. to keep data in the local cache whenever possible and to avoid using the network. We use the following definitions of cache line sharing:

Active sharing: a data item is accessed by more than one processor during the execution of a program.

False sharing: this is where processors share a cache line without sharing data items within the cache line. With invalidation, a write to an item in a shared cache line requires all copies of that cache line to be invalidated, even if the other processors never use the data item which was changed.

Passive sharing: this occurs where shared data still remains in a processor's cache even though no objects on the cache line will be accessed by that processor again [2]. Since a write by another processor to any item in that cache line will require all other copies of the cache line to be invalidated, it is desirable that the redundant cache line is ejected after its last use.

These characteristics interact, and are affected by the memory access characteristics of a particular program and the shared-memory architecture. Cache line size is particularly relevant: larger cache lines would allow many objects to be allocated on each cache line, which could be helpful if an application has locality of access. However, the larger line size can lead to contention for cache lines, especially if false sharing plays a significant role in the behaviour of an application.

3 Cautious Trace Analysis and CLARISSA

The analysis process operates as a sequence of phases, in order to reflect barrier synchronisations (which prevent events occurring on different sides of a barrier from overlapping), limit the amount of analysis time and space required, and prune overlaps which are unlikely because they appear at widely-differing times in the trace (overlap is possible, since no synchronisation prevents them, but are unlikely). Essentially, what needs to be considered is which events could possibly occur in the same phase. Whatever the hardware configuration, the events for a particular CPU will always occur in the same order, but the order in which events occur between different CPUs can vary. In the example, Fig. 1, it is possible that an event in CPU_0 may occur before or after any event in, say, CPU_1 from the start of the program up to the first barrier synchronisation. However, it is not

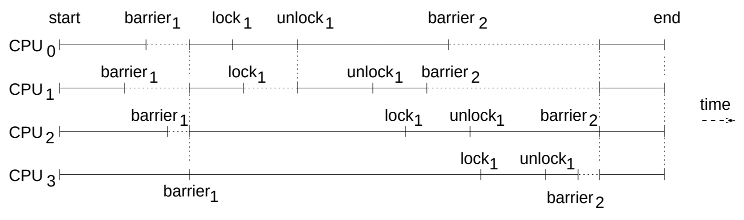


Fig. 1. An example execution path

possible for events occurring before the first barrier to overlap with events after that barrier.

Considering only barriers, Fig. 1 has three phases. Sharing can occur across barriers, but it is not currently part of our analysis. In programs where barriers are used regularly to ensure synchronisation, such as MP3D (Sect. 4), cautious trace analysis between barriers shows the sharing effects that may occur within the weak ordering programming model.

In applications where there are few or no barriers, the analysis becomes so broad that it is likely that CLARISSA will over-report the potential sharing, and the volume of data and phase end processing will be problematic. In such cases, we introduce fixed time-slots. The length of the time-slots depends on the overall length of the program. Too short a time-slot will result in some effects being missed and give too fine a level of summary information, whereas too long a time-slot will tend to over-report sharing and contention, and give a summary which is too coarse. In addition, edge effects have to be allowed for, i.e. the analysis must take into consideration sharing effects which cross a time-slot boundary. In Sect. 5 an example is given of using time-slot analysis (with overlaps) to tune the performance of an application which makes little use of barrier synchronisations.

Cautious analysis also has to allow for the use of locks. For example, in Fig. 1, if each CPU only reads and writes a particular data item when the processor has obtained $lock_1$ then, although there is still active sharing of the item between the barriers, the programmer has protected the data item from the possibility of simultaneous update by two or more processors.

3.1 Using CLARISSA

The CLARISSA tool is based on [5]. Input parameters include cache line size, class threshold (the N value in Table 1), phase type (barrier or time-slot), time-slot length and overlap. A classification system is needed for summarising the wealth of data. Table 1 gives the classification used in the SM-prof performance debugging tool, which reports cache line access for fixed time-slots in terms of read or write accesses and the number of CPUs involved [1]. In CLARISSA, an enhanced version of this categorisation is used, where the sharing categories (ending in E/F/M) are further split according to active or false sharing. The

Table 1. SM-prof classification of cache line accesses [1]

class	*degree of sharing*	*access mode*	*comments*
UNR	none	none	no processor referenced the cache line
ROE	exclusive	read only	one processor has done a read operation, but no write operation to the cache line
ROF	shared by few	read only	i processors have done read operations, but no write operations, to the cache line[a]
ROM	shared by many	read only	N or more processors have done read operations, but no write operations, to the cache line
RWE	exclusive	read/write	one processor has performed a read-modify-write sequence on the cache line
RWF	shared by few	read/write	i processors have performed read-write-modify sequences to the cache line.
RWM	shared by many	read/write	N or more processors have performed read-write-modify sequences to the cache line

[a] where $1 < i < N$

results are used to provide histograms of sharing activity for each phase during the execution of the program.

4 MP3D

MP3D is a particle-based wind tunnel simulation, from SPLASH [4]. It is used to study the shock waves created as an object flies at high speed through the upper atmosphere. It was run for 30000 molecules, using the supplied geometry file *test.geom*, for 10 time-steps. Two large arrays of structures account for more than 99% of the static data space used by MP3D; the first structure stores the state information for each *particle* and the second structure stores the properties of each *cell* in the active space.

The way MP3D uses barriers for synchronisation within each step meant that CLARISSA barrier analysis was the most appropriate, and the resulting phase level graphs are shown in Fig. 2. These graphs, in conjunction with the summary and detail level sharing information generated by CLARISSA, showed that the dominant sharing was active sharing of the *cells* array. For many data items within the *cells* array, more than one processor updates the same data item between barriers, and this generates a high number of coherency protocol invalidation messages.

To avoid the active sharing of *cells*, MP3D was modified so that the scheduling of work for the CPUs was driven by cells rather than particles. When the trace output for MP3D-NEW was analysed by CLARISSA, there was a substantial

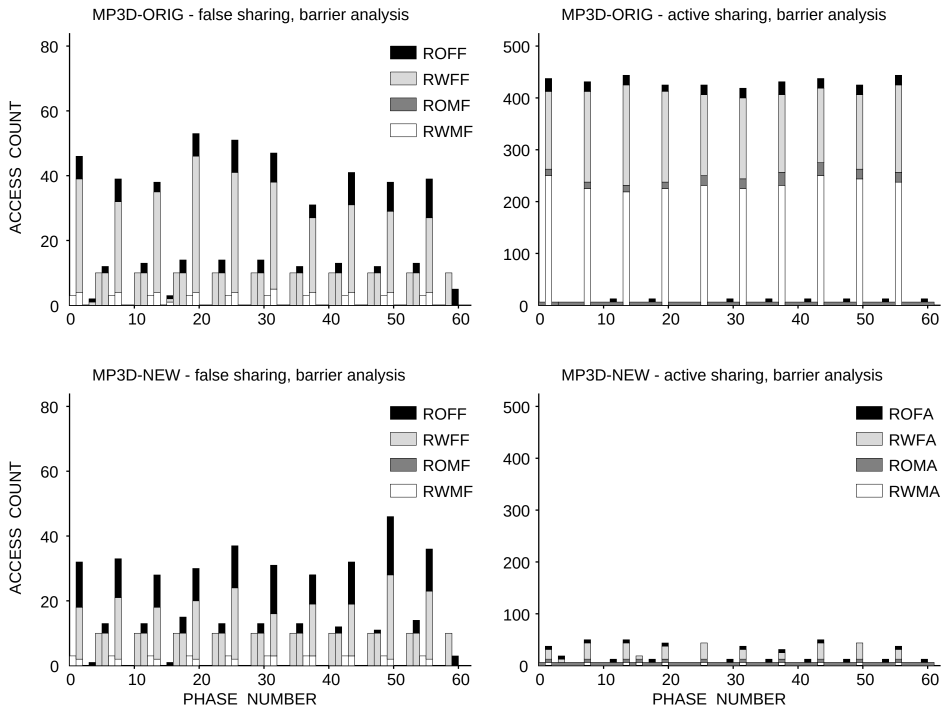
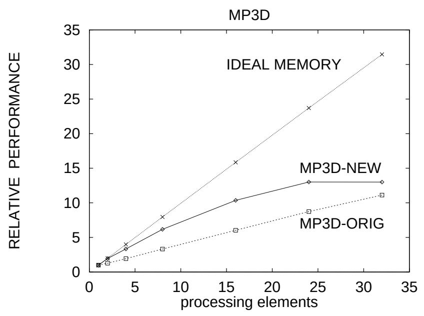


Fig. 2. Phase level graphs for MP3D-ORIG and MP3D-NEW



	miss rate (%)	
no. of pe's	MP3D-ORIG	MP3D-NEW
1	0.1682	0.2262
2	2.0008	0.3486
4	3.0215	0.4420
8	3.5685	0.5549
16	3.8490	0.7858
32	4.0972	1.1188

Fig. 3. Relative performance and cache miss rates for MP3D

reduction in the active sharing of the *cells* array, and false sharing had also been reduced[1]. This is illustrated by the active and false sharing phase level graphs shown in Fig. 2.

The two different versions, MP3D-ORIG and MP3D-NEW, were run on an execution-driven simulator to obtain CC-NUMA execution timings, cache miss

[1] Similar changes have been made by other researchers e.g. [1], but we made the change because it was specifically indicated by the active sharing information from CLARISSA.

rates and relative performance. The results are shown in Fig. 3. The best speedup is achieved by MP3D-NEW. Simulation statistics confirm that the miss rate is substantially lower for MP3D-NEW, so the strategy of reducing sharing has been successful. However, this is starting to lose its "edge" at 32 CPUs: the drop in performance is believed to be because of poor load balancing given the relatively small problem size (i.e. 30000 molecules) used for these tests.

5 Computational Fluid Dynamics

CFD is a major application area of high performance computing. The system modelled in our CFD application is a laminar flow in a square cavity with a lid which slides across the cavity introducing a zone of re-circulatory fluid. CFD uses barriers to ensure that CPU_1 has updated global variables before all the processors move on to the next stage, but there are long periods without a barrier [6]. Time-slot analysis was therefore appropriate, and time-slot length was chosen to give around 100 slots over the execution time, i.e. to give a reasonably detailed profile without being swamped by too much information.

The phase level graphs generated by CLARISSA are shown in Fig. 4. The graphs, in conjunction with the summary and detail level sharing information, indicated that the most significant sharing was false sharing within the *var* data structure. The 2-D arrays in *var* were originally distributed to the CPUs column-wise, and the program was modified to create CFD-NEW, which uses square block distribution. Without the help of CLARISSA, it would only have been possible to pinpoint the performance problem by gaining a thorough knowledge of the application program. When the trace output for CFD-NEW was analysed by CLARISSA, there was a substantial reduction in false sharing messages relating to the *var* data structure, reflected in the improved false sharing phase level graph shown in Fig. 4. Active sharing was increased by the change but, as shown by the performance results below, any coherence overhead incurred by this increase is more than compensated for by the reduction in false sharing. CFD-ORIG and CFD-NEW were run on the simulator to obtain timings, cache miss rates and relative performance, shown in Fig. 5. The best speedup running under real memory is achieved by CFD-NEW. In addition, the simulations showed that the cache miss rate was always lower for CFD-NEW in comparison with CFD-ORIG. The reduction in false sharing lead to a significant improvement in performance, even though active sharing increased slightly.

6 Related Shared-Memory Tools

MemSpy [3] assists in locating bottlenecks by providing detailed information that focuses the programmer's attention on the problem areas in the application. SM-prof [1], is similar to that presented here, but has the drawback that it does not distinguish between active and false sharing of cache lines. It also splits a program's execution up into time-slots, but does not allow for boundary effects

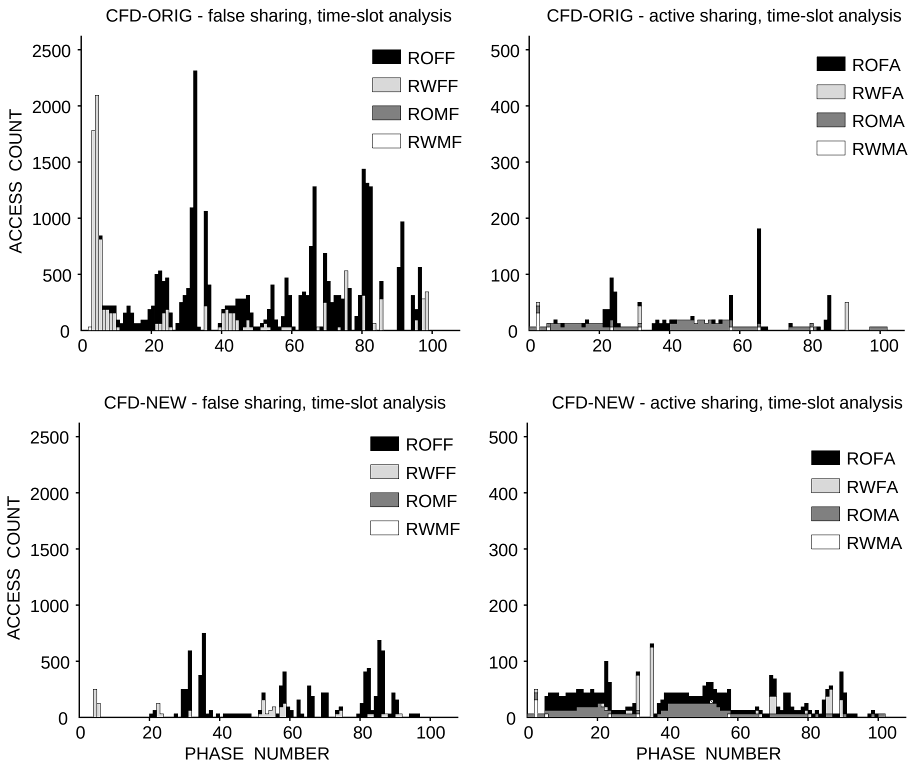
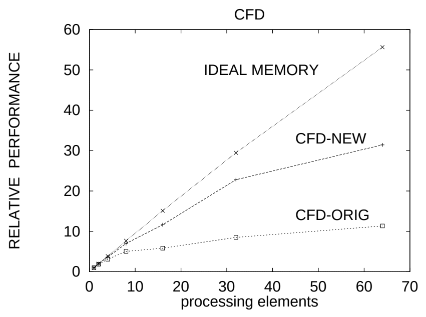


Fig. 4. Phase level graphs for CFD-ORIG and CFD-NEW



	miss rate (%)	
no. of pe's	CFD-ORIG	CFD-NEW
1	0.0007	0.0007
2	0.6117	0.0356
4	1.7589	0.6702
8	4.0980	0.7320
16	9.2645	1.8799
32	13.2287	2.0094
64	18.9424	4.2586

Fig. 5. Relative performance and cache miss rates for CFD

between adjacent slots; consequently analysis has to be performed multiple times with different time-slot lengths.

The information given by CLARISSA differs from that of existing performance analysis tools because it diagnoses *potential* contention rather than problems arising from a particular architecture. It also distinguishes between different types of sharing, i.e. active, passive and false sharing. This is important as the action to be taken depends on the type of sharing that needs to be eliminated. For example, in [1], false sharing is "suspected" in MP3D, but CLARISSA reports false sharing precisely. Similarly MemSpy can report that cache misses are high for a particular variable, but cannot say whether this is due to active, false or passive sharing. Finally, CLARISSA allows the analysis to be carried out on a barrier or time-slot basis, so that the timing of phases is appropriate for a particular application.

7 Conclusions

CLARISSA is a new tool which has been shown to be effective in analysing the cache-line sharing effects in shared-memory parallel applications. It uses a novel approach, *cautious trace analysis*, to locate potential cache line contention rather than measuring actual contention in a specific memory system design. We have shown how it was used to greatly improve the performance and cache behaviour of two scientific programs.

As further work, we plan to enhance CLARISSA to provide boundary analysis of cache lines, i.e. to cater for false sharing effects that may not show up for a particular problem size due to data structures happening to align with cache line boundaries. In addition, the performance of CLARISSA should be improved by closer integration with the simulator.

Acknowledgements. This work was funded by the U.K. Engineering and Physical Sciences Research Council through an Advanced Course Studentship and project GR/J 99117. Enormous thanks are due to Ashley Saulsbury for allowing us to use his simulator.

References

1. Mats Brorsson. SM-prof: A tool to visualise and find cache coherence performance bottlenecks in multiprocessor programs. In *Proceedings of the ACM SIGMETRICS and Performance '95*, pages 178–187, May 1995.
2. Susan J. Eggers and Randy H. Katz. A characterisation of sharing in parallel programs and its application to coherency protocol evaluation. *15th Annual International Symposium on Computer Architecture, Honolulu, May, in Computer Architecture News*, 16(2):373–382, May 1988.
3. M. Martonosi, A. Gupta, and T. Anderson. Tuning memory performance of sequential and parallel programs. *IEEE Computer*, 28(4):32–40, April 1995.
4. Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.
5. Sarah A. M. Talbot. Performance tuning of programs for shared-memory multiprocessors. Master's thesis, Department of Computing, Imperial College, London, U.K., 1995.
6. B. A. Tanyi. *Iterative Solution of the Incompressible Navier-Stokes Equations on a Distributed Memory Parallel Computer.* PhD thesis, UMIST, 1993.