Opportunities for Cache Friendly Process Scheduling

Pranay Koka, Mikko H Lipasti
University of Wisconsin - Madison
Madison, WI – 53706
pkoka@cae.wisc.edu, mikko@engr.wisc.edu

ABSTRACT
Operating system process scheduling has been an active area of research for many years. Process scheduling decisions can have a dramatic impact on capacity and conflict misses in on-chip caches, since processes that do not share memory will compete for entries in the cache, while processes that do share memory can effectively pre-fetch blocks or warm the cache in a symbiotic fashion. In this paper we provide a detailed characterization of context switch misses and quantify its impact. We also investigate the potential of intelligent process scheduling that minimizes cache misses across context-switch boundaries. We have developed several greedy heuristics that enable us to perform a trace driven limit study on the benefits of cache friendly process scheduling. We show that up to 37% improvements in cache miss rates are achievable in some transactional workloads. We also propose some mechanisms to leverage this potential.

1 Introduction

Cache configurations and access methods have become a critical component in the design of fast processors. Giga-Hertz processors suffer from high off-chip memory access latencies of hundreds of cycles. Even the aggressive out-of-order processors fail to extract ILP to hide memory latencies. Such speed discrepancies cause high stall counts hence reducing the effective IPC.

This problem has spurred wide research in areas like, design of pre-fetching, cache structure and configuration, access and replacement policies. Another problem associated with caches that has not been this widely investigated is the effects of operating system process management on on-chip caches. Typical operating systems manage multiple processes by assigning a time quantum to each process. Each process is allowed to run till it exhausts its time quantum, at which point it yields the processor. A ready queue of processes is scanned at each schedule point to pick a process with an unexpired time quantum. During its quantum, a process warms the cache with its working set before entering steady phase. On a process switch, the next process loads the cache with its working set. Small caches cannot accommodate both the working sets, and hence evict the first. On the consecutive quantum, a process again incurs an initial warm-up phase which has a high cache miss rate. The negative impact of context switches depends on how long the warm-up phase is compared to the steady phase. In modern commercial systems, due to the I/O intensive nature of the applications, processes seldom use the entire time quantum in a single execution slice. Short execution slices make the warm-up phase more dominant. One way to measure the negative impact is to accurately identify the misses due to context switches and determine the fraction of total misses due to context switches. In this paper, we make detailed characterization of the dynamics of cache misses across context switches.

An ideal but impractical solution to this problem would be to have a cache large enough to hold the working sets of all active processes. Other practical solutions can be broadly classified into three categories.

1) Novel cache designs and replacement policies that accommodate context switch overheads
2) Pre-fetching techniques that load the working set of the next runnable process.
3) Cache aware process scheduling techniques that exploit data sharing nature of related processes.

The first two techniques involve redesign of the processor. We believe that an operating system based technique is more viable. In this paper we chose to investigate the opportunities that exist in cache aware scheduling techniques. We show that some up to 39% improvements in cache miss rates are achievable in some benchmarks.

Commercial applications like databases and web servers are designed to have multiple threads to cater requests. For example a web server has multiple stages of processing for each user request, such as request parsing, cache access, response framing. Every user request is assigned to a worker thread which works through all the processing stages. At some point there will exist few threads in some of the processing stages as shown in Figure1. Each processing stage has associated with it a set of instructions and data. Hence, more than one thread in a stage could access same data, in other words, have similar working sets (blue boxes in Figure1). For example more than one request could fetch the same html document. Such data sharing patterns can be exploited using smart scheduling techniques. We propose such a technique in this paper. In section 2 we present prior research work in this area and the contributions of this paper. Section 3 describes characterization methodology and results. We describe our cache-aware scheduling technique in section 4 and conclude in section 5.

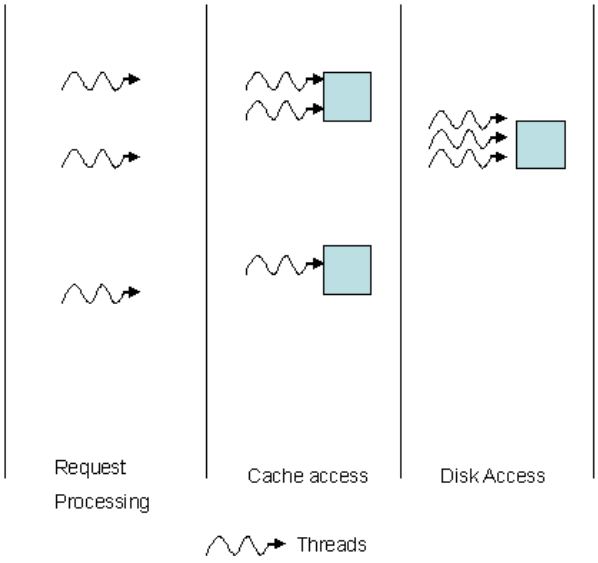

Figure 1. Application stages

2. Previous Work:

Prior research focused on characterizing the cache performance on multiprogrammed workloads. Initial work by [1] used single process traces, interleaved to form a multi-programmed workload. Thiebaut and stone [8] developed a theoretical model to estimate the cache miss rates in multi-programmed environment. Later research used accurate multiprogramming traces to evaluate various cache sizes and configurations [3] [4] [6]. Mogul et.al [10] performed a trace driven study on the impact context switches on cache miss rates. They used the variation of CPI after a context switch to estimate the impact. Their characterization accounts only for user-mode accesses but modern commercial applications spend a significant fraction of the execution time in the kernel. Also, none of the previous studies used traces from multi-tier commercial workloads. Another drawback of these studies is that they perform a high level characterization such as increase in miss-rate, and reduction in IPC due to process switches. They fail to provide enough architectural insights on the cache dynamics across context switches. We address these drawbacks in this paper.

Different variants of co-scheduling techniques have been proposed to improve cache performance of multithreaded workloads. Tullsen et al [2] describe a symbiotic scheduling approach for multithreaded processors. It aims at exploiting a shared cache in multithreaded processors. They employ a sampling phase where different combinations of processes are run on the processor and the performance is measured. The best schedule is then selected for the steady phase. The main drawback of this method is that it is a trial-error scheme. It does not rationalize on the sharing nature of the workload. Larus et al [9] propose an application level approach to exploit the data sharing nature of threads in a workload. They decompose an application into various stages. The stages correspond to different processing stages as illustrated in Figure1. The stages are implemented using special class libraries. Stages have the autonomy to control the execution of threads. Hence a stage can block threads till a threshold number of threads accumulate in that stage. The stage then batch schedules these threads to increase the cache re-use. Implementing cohort scheduling needs an extensive re-structuring of the application, rendering this technique impractical. Cache affinity scheduling in multiprocessors [11] [12] [14] aims to re-use residues of a process' working set in its consecutive slices. A process is given higher priority if its previous execution was on the same processor as that being allocated. An approach by [7] uses working set information of threads and analyzes static processor allocation schemes on various matrix and sorting algorithms. Static allocation schemes always do not work well with dynamically changing working sets of commercial workloads. In this paper

we provide a systematic approach of analyzing the opportunities of cache aware scheduling and also provide an application independent, kernel process scheduling technique that adapts to the dynamic working set of processes. To summarize, the contributions of this paper are three fold:

1) A detailed characterization of context switch misses. The results of the characterization help in
    a) Estimating the magnitude of context switch misses.
    b) Obtaining better architectural insights on cache misses across process switches
2) A theoretical approach to estimate the performance gains that can be leveraged using cache aware process scheduling. A trace driven analysis is performed using best case heuristics.
3) We propose a cache aware process scheduling scheme based on process phases of execution.

3. Characterization Methodology

All analysis performed are trace driven. Traces were collected from a full system simulator, Pharmsim [13] that simulates the PowerPC architecture. The traces contain data memory references annotated with context switch points and the current thread ids. In the PowerPC architecture process switches can be identified during the thread-id register writes. All traces were generated by running the workloads in a four processor configuration in the simulator. A separate trace for each processor was generated and analyzed. The final results are the harmonic mean of all four processors. We used TPC benchmarks [16], TPC-W, TPC-B, TPC-D and one from the Spec suite [15], SPEC-Web. We believe that these benchmarks are a good representation of the modern commercial server environments.

3.1. Architectural Characterization

Increase in miss rates due to context switches can be calculated by counting the number of actual such misses. In this section we describe our methodology for accurately separating context switch misses. We define the term 'cross-interval miss' as the first miss to a cache block in that interval for a process. A cross interval miss could be due to any of the three reasons illustrated in Figure2 Consider a cache block with address 'A', referenced in interval Tm and Tn with no intervening references. Tm and Tn are non-consecutive intervals. The reference in Tn could result in a miss for the following reasons:

1) 'A' was evicted from the cache in Tm itself – Self-Kill (SK)
2) 'A' was evicted in Tn before it was referenced – pre-matured kill (PK)
3) 'A' was evicted in an intervening interval. We call this context switch miss (CM).

Figure 2

The third type of cross interval misses are the misses due to context switches. We characterize the cross interval misses into the above components by maintaining local and global history of references, which contain state information about references and evictions for each cache line. In order to estimate the cache-performance cost of context switches we split the total misses for each workload into five components.

1) Cold misses
2) Self kills
3) Pre kills
4) Context switch misses
5) Others – Comprises of conflict and capacity misses within an interval.

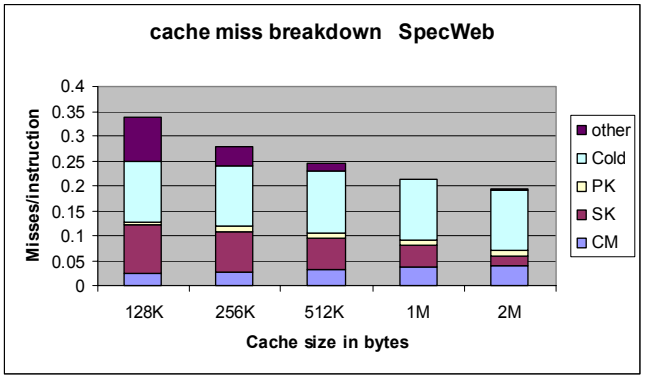

Figure 4. Cache Miss Breakdown – SPECweb

We study the variation of these components for various cache sizes. Figure 4 shows the cache miss breakdown for SPECweb. Cold misses constitute about 35% in a 128KB cache to 63% in a 2MB cache. We find that SKs are more dominant than CMs. Cache size increase as expected reduces the SKs to just 10% of the misses. A noteworthy observation is that the CMs increase with cache size. The reason for this can be explained from figure2.

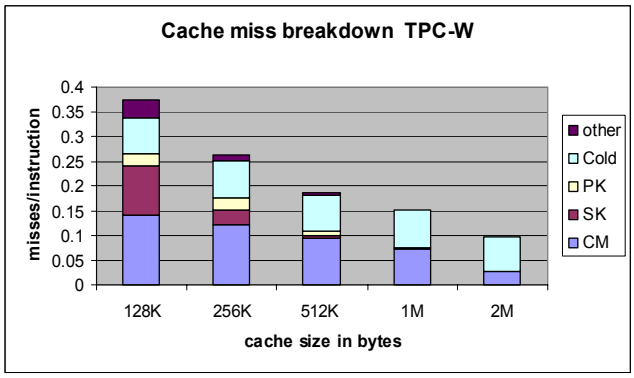

Figure 5. Cache Miss Breakdown – TPC-W

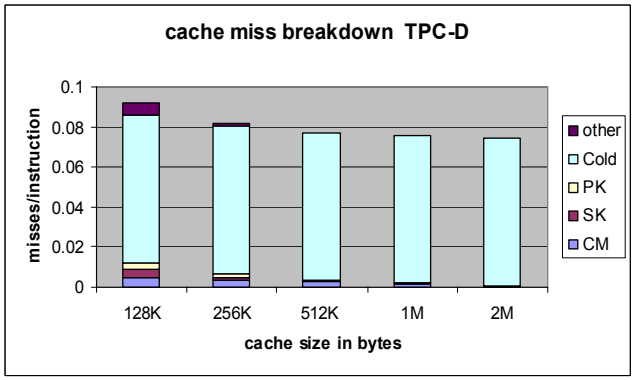

Figure 6. Cache Miss Breakdown – TPC-D

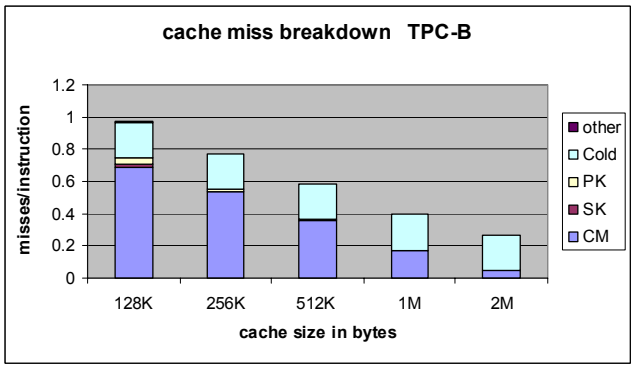

Figure 7. Cache Miss Breakdown – TPC-B

A large cache will preserve majority of the lines that would have otherwise been self-killed. Hence these lines are now exposed to the intervening intervals where they are evicted. Hence a fraction of the SKs get converted to CMs. We found that number of CMs as such reduces with large caches, but the spill of SKs to CMs is more than the reduction in CM. This behavior is seen only in SPECweb. In other workloads like TPC-W and TPC-B CMs reduce with increase in cache size. This is because both CMs and SKs get converted to hits. Large caches seem to be more effective for these two workloads. But resorting to a brute force approach of large caches is not always justified. CMs are quite dominant in TPC-W and TPC-B constituting 41% and 43% of the misses respectively even for a 1MB cache. TPC-D stands apart from the other benchmarks, in that cold misses form more than 90% of the total misses, leaving no opportunity for cache aware scheduling.

3.2. Opportunity Characterization

The results in previous section quantify the cache performance problem due to context switches in commercial workloads. We would like to estimate the best case improvements that could be achievable through better process scheduling techniques. In order to perform this limit study we formulated the scheduling problem as a graph problem. As an example assume there are 3 processes A, B, and C in the system. Intervals when A executed on the CPU are A1 A2 A3…., and similarly for B and C. In Figure 3 we show a graph with each node representing an interval of a process. An edge from one node to another represents a context switch in the direction of the edge. We associate a weight for each edge which represents the cost of the context switch. Cost of the switch is solely a cache-performance cost. We do not account for the other context switch overheads in this study. The graph in Figure3 shows some possible transitions between intervals. Traversing the graph in some order, visiting each node exactly once produces a ‘schedule’ and an associated total cost for the schedule which is the sum of costs of all the edges traversed.

Our objective is to obtain a schedule with minimum cost. The problem hence becomes an instance of the classical traveling salesman problem (TSP). We also specify certain constraints for the graph traversal to make the study more realistic.

1) Two intervals of the same process cannot be scheduled consecutively. For instance, A1 and A2 cannot be scheduled consecutively.

2) Time order between intervals of a process needs to be respected. For instance, A2 cannot be scheduled until A1 is scheduled.

We relax other constraints such as:

1) Synchronization ordering between processes.

2) Non-deterministic I/O blocking time.

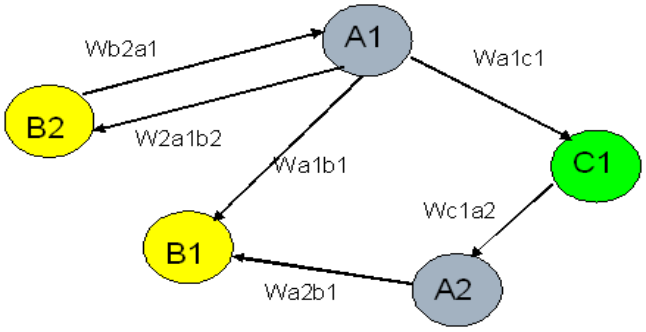

Figure 3. TSP analogy

Since the traveling salesman problem is a NP-Complete problem, finding an optimal solution is impractical. We have developed greedy heuristics to estimate the performance improvements achievable. Greedy heuristics require three pieces of information to compute a near optimal schedule.

1) Amount of future information. In this case the working set information of 'n' future intervals

2) Amount of past information. In this case the working set information of 'm' previous intervals.

3) Cost metric.

For our heuristics we have used the working set information of one interval ahead and one before. The two heuristics differ in the cost metric. Consider two intervals A1 and B1 with working sets a1 and b1. The "diff" heuristic uses the number of cache blocks that need to be fetched due to the switch from A1 to B1. In order words it is the number of blocks in B1 that are not in A1 ($b1 \cap a1^c$). The heuristic aims at minimizing this cost. The "union" heuristic uses the sharing between A1 and B1 ($a1 \cap b1$). The heuristic in this case aims at maximizing the benefit.

Our heuristic computes a cost matrix and then selects the first interval of a process called the initial interval. It picks the next interval traversing the edge with min/max weight in the cost matrix. This is repeated with different initial intervals pertaining to different processes in the system. The schedule with the minimal total cost is chosen for trace driven cache simulation. The annotated trace is simulated in the order of the computed schedule. The cache miss-rates for the computed schedule are compared against that of the kernel schedule. Figures 8 and 9 show the maximum cache-performance improvements that are achievable through cache aware scheduling techniques. These results have a direct correlation with results in Figures 4 to 7.

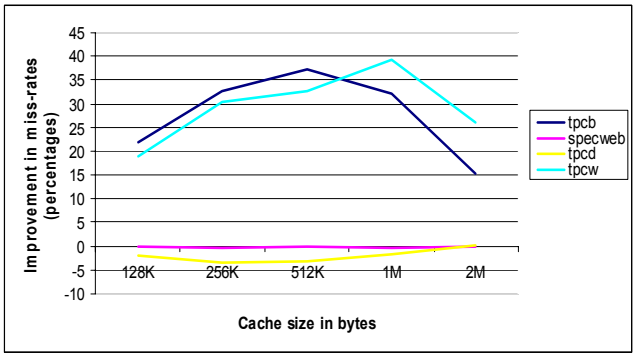

Figure 8. Miss-rate improvements using "diff" heuristic

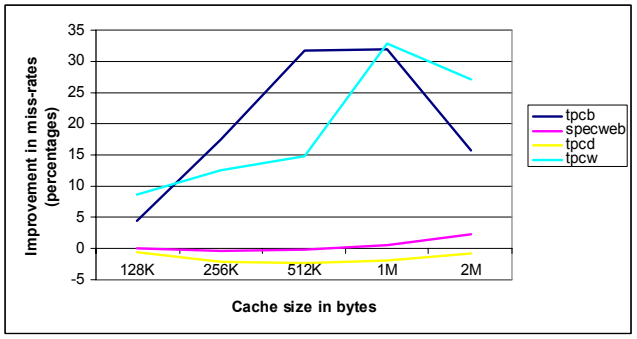

Figure 9. Miss-rate improvement using "union" heuristic

In TPC-W and TPC-B approximately 50% of the misses are CMs for a 1MB cache. Cache-aware scheduling techniques can produce up to 39% and 37% improvements in TPC-W and TPC-B respectively. For larger cache sizes the improvements decrease in correlation with the opportunities presented in the cache miss breakdown results. SPECweb and TPC-D show some performance degradation. This could be due to the non-optimal nature of the heuristics. On the whole the "diff" heuristics perform better than the "union" heuristic.

4. Cache Aware Process Scheduling

The scheduling heuristics discussed in section 3.1, use future information, and hence are not practical in real implementations. We intend to apply smart scheduling heuristics in the operating system kernel, transparent to the applications. We propose a kernel mechanism that requires no modification to the processor. The mechanism relies on the phase behavior of commercial workloads. Our scheduling algorithm essentially has two parts. The first part deals with identifying the working sets of processes and determining the process phase. The second part is the scheduling component that utilizes the computed phase information to make a scheduling decision.

Determining the working set of a process inside the kernel is a non-trivial problem. One solution is to modify the processor-OS interface. As in [5] the processor can compute the phase information using the memory reference traces and export it to the OS during process switch or system calls. We adopt a more practical solution that uses an unmodified processor. We infer the working set information from the process call stack. During a process switch, the kernel scheduler can unwind the call stack and produce a signature of the return instruction pointers, arguments, and local variables. The call trace of a process identifies the phase of the process and the arguments add information of the data involved in the phase. For example, in a web-server the existence of multiple threads in the "cache access" phase can be determined from the call trace. The arguments like the filename, offset give an indication of data sharing between some of the threads. The phase signature can be produced using simple hashing technique, as illustrated in Figure 10.

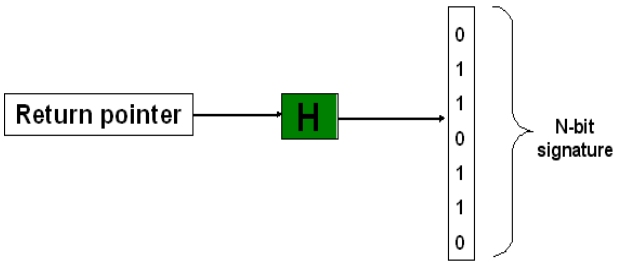

Figure 10. Phase Signature

The N-bit signatures can be stored as part of the process state. The scheduler, at the schedule points, compares the signature of the outgoing process and that of those in the ready queue. Similarity between signatures can be computed using hamming distance [5]. The process with the highest similarity should be scheduled next. We have adapted this algorithm to the Linux scheduler. A simple modification to boost the goodness value of the process proportional to the signature similarity with the outgoing process will achieve the same effect, while preserving the starvation guarantees of the original scheduler.

5. Conclusions and Future Work:

A detailed breakdown of the cache misses into the five components provides a better understanding about context switch misses. Apart from the conventional cold, capacity, conflict and coherence misses, we find that context switch misses too are a significant concern in modern commercial environments. The characterization reveals that opportunities exist in workloads like TPC-W and TPC-B. We would like to perform similar characterization for the instruction references and also extend this work for other workloads.

We have also shown the importance of cache aware scheduling techniques as a solution to this problem. Our trace driven simulations have shown that up to 39% cache miss-rate improvements are obtainable in a best case scenario. As part of future work we plan to extend the heuristic framework to reflect more realistic and less optimistic cases. We can annotate the traces with I/O points and factor in I/O time estimates in the heuristic simulator. Such a study would give more accurately the cache miss-rate improvements possible with cache aware scheduling. We plan to implement and study the proposed cache aware scheduling mechanism in future work.

References

[1] Alan Jay Smith, Cache Memories. ACM *computer surveys 14(3):473-530, September, 1982.*

[2] Allan Snavely, Dean M Tullsen, "Symbiotic Job Scheduling for a Simultaneous Multithreading Processor", *Proceedings of ASPLOS IX, November, 2000.*

[3] Anant Agarwal, John Hennessy, Mark Horowitz, Cache Performance of Operating System and Multiprogramming Workloads, *ACM Transactions on Computer Systems, November, 1988.*

[4] Anant Agarwal, Mark Horowitz, John Hennessy, "An Analytical Cache Model", *ACM Transactions on Computer Systems, May, 1989*

[5] A. Dhodapkar and J. Smith, "Managing multi-configuration hardware via dynamic working set analysis", *29th Annual International Symposium on Computer Architecture, May 2002*.

[6] Craig B Stunkel, W Kent Fuchs, "TRAPEDS: Production Traces for Multicomputers Via Execution Driven Simulation", *Proceedings of ACM SIGMETRICS, Int. Conf. on Measurement and Modeling of Computer Systems, May, 1989.*

[7] Dipak Ghosal, Giuseppe Serazzi, Satish K Tripathi, "The Processor Working Set and its Use in scheduling multiprocessor systems", *IEEE Transactions on software engineering Vol 17, May 1991*

[8] Dominique Thiebaut, Harold S Stone, Footprints in the Cache. *ACM Transactions on Computer Systems, November, 1987*

[9] James R Larus, Micheal Parkes, "Using Cohort Scheduling to Enhance Server Performance", *Usenix Annual Technical Conference, June 2002*

[10] Jeffrey C Mogul, Anita Borg, "The Effect of Context Switches on Cache Performance", Proceedings of ASPLOS-IV, April 1991

[11] Josep Torrellas, Andrew Tucker, Anoop Gupta, "Benefits of Cache-Affinity scheduling in shared-memory multiprocessors: A summary", *ACM Sigmetrics, 1993*.

[12] Josep Torrellas, Andrew Tucker, Anoop Gupta, "Evaluating the performance of cache affinity scheduling in shared memory multiprocessors", *Journal of Parallel and Distributed Computing, Vol 24, February 1995.*

[13] Kevin M. Lepak, Harold W. Cain, and Mikko H. Lipasti, "Redeeming IPC as a Performance Metric for Multithreaded Programs", *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, September, 2003

[14] Raj Vaswani, John Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors", *Symposium on Operating Systems Principals, October 1991*

[15] System Performance Evaluation Cooperative. SPEC Benchmarks, www.spec.org

[16] Transaction Processing Performance Council. TPC Benchmarks, www.tpc.org