Appears in *Proceedings of the Sixteenth Symposium on Operating Systems Principles (Work in Progress Session)*, October 1997.

# FOLLOW-ON SCHEDULING
## USING TLB INFORMATION TO REDUCE CACHE MISSES

FRANK BELLOSA
bellosa@informatik.uni-erlangen.de
*Department of Computer Science IV, University of Erlangen-Nürnberg, Martensstraße 1, 91058 Erlangen, Germany*

Processor caches are designed to store the most recently used subset of the main memory and to provide this subset with low latency. Most contemporary cache architectures use physically indexed caches to facilitate cache coherency and to avoid cache flushing during a context switch. Memory of multiple contexts can simultaneously be encached in physically indexed caches. If two threads with a different working set follow upon each other, the cache content is displaced. In this situation, the processor stalls due to cache misses in the instruction-fetch and load/store pipeline stage.

Kernel threads frequently share a lot of memory. Examples are code segments, shared memory segments, or shared address spaces in multithreaded applications. Switching between threads that share large parts of their working set results in high cache reuse, few cache misses, and thus in good system performance [1]. Our analysis of typical application- and information-servers shows that a large fraction of the physical memory pages is shared by multiple contexts. In a highly loaded server system, there are normally many runnable threads in each runqueue sharing a lot of memory. It is desirable for threads with the same working set to follow upon each other to reuse the content of the cache, but in contemporary operating systems the sequence of execution is independent of working-set aspects.

Our approach to improve system performance uses information derived from the translation lookaside buffer (TLB) to detect kernel threads which share a lot of memory pages. To determine the pages a thread is accessing, we need to analyze the content of the TLB. As we cannot easily look into the TLB, we have to modify the TLB- miss handler or to analyze data structures used to cache the page tables (e.g., translation storage buffer TSB in the Sparc V9 architecture [2]). If *related threads* reside in the same dispatch queue, the order of the dispatch queue is carefully changed so that related threads follow upon each other. This approach draws most profit from operating systems where threads are enqueued into a fixed-priority dispatch queue after they have returned from sleep (e.g, as in System V Release 4).

The determination of related threads is a compute-intensive procedure, because there exists no trivial mapping from protection domains (physical pages – contexts) to active entities (threads). We were therefore forced to implement a reverse lookup mechanism to coalesce threads and pages.

Measurements conducted on a prototype implementation using the Solaris operating system demonstrate the advantages of using TLB information in a multiprogrammed information- and application-server (SUN Enterprise X000 architecture).

## 1. Shared Pages Promote Cache Reuse

Exploiting locality of reference is the purpose of caches. Reusing the content of caches narrows the gap between the slow main memory and the fast processing unit. The efficiency of caches is determined by the degree cache entries can be reused.

While small virtual caches are common as primary caches because they permit fast access, physical caches dominate the area of second level caches. Physical caches ease cache consistency for I/O operations and multiprocessing and require no additional support to eliminate aliases or ambiguities. Physically indexed caches store a small subset of physical memory without any consideration concerning context, address space and memory mapping. Therefore, physically indexed caches can store the working sets of multiple threads, even if these threads belong to different processes.

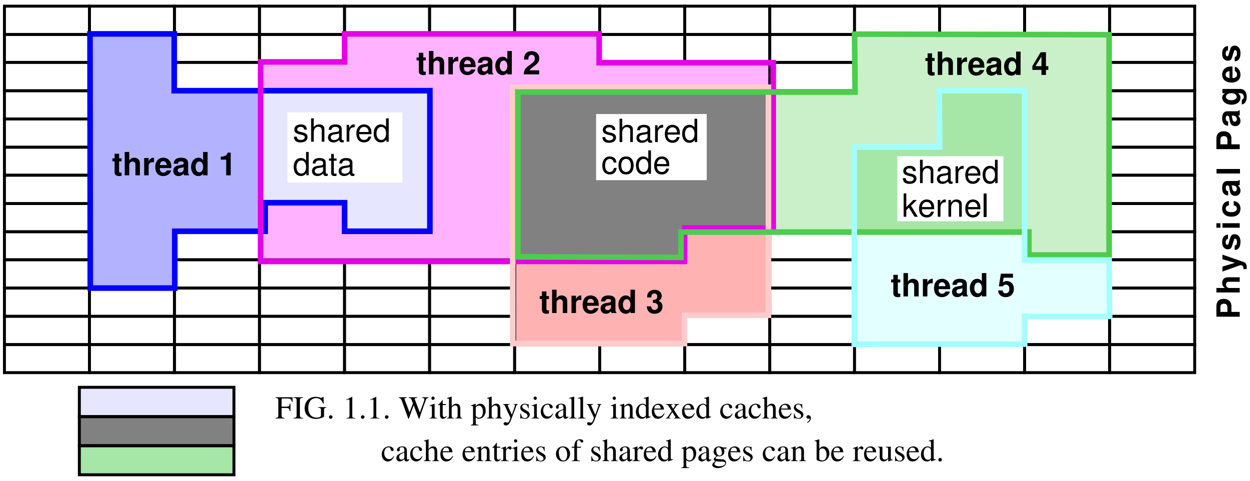


FIG. 1.1. With physically indexed caches, cache entries of shared pages can be reused.

If the working sets of threads differ, large portions of the second level cache have to be loaded after a switch because multiple divergent working sets do not fit into the cache. On the other hand cache entries of shared memory regions can be reused after a switch. Our approach to improve cache reuse is to exploit the property of shared pages exhibited by many applications.

To investigate the property of page sharing we have to analyze three aspects:

– the type of memory which is shared among threads
– the sets of threads sharing memory
– the degree of sharing between threads

Multithreaded applications and those using shared memory segments for inter-process communication (IPC) share physical pages filled with data. Shared code pages can be found if a shared library is mapped into multiple address spaces or if multiple instances of the same executable are running. Threads which trigger similar kernel activities (networking, filesystem, synchronization) also share pages related to the kernel (see FIG. 1.1.).

Those threads which share pages have to be identified. If two threads share a large portion of their working set they are declared as *relatives*. The scheduler should influence the dispatcher such that related threads follow upon each other. We introduce this technique called ***Follow-On Scheduling***. The benefit of Follow-On Scheduling lies in the reduction in the number of cache misses a thread experiences after the switch, if a related thread has run on the same CPU before. Especially on highly populated run-queues, the sequence of threads can be modified, such that cache misses due to changed working sets can be considerably reduced. A good example of a highly populated queue is the queue dedicated to threads returning from sleep in a Unix System V Release 4 (see FIG. 1.2.). On a highly loaded server we observed more than 50 runnable threads in this queue. The longer the queue, the more effectively the sequence of threads can be optimized.

## 2. TLB-Miss Guided Enqueueing

Scheduling decisions are based on information. *Follow-On Scheduling* extends the deciding factors of contemporary schedulers (timing, priority, event-type, swap-state of process, reason for blocking) by adding information concerning the relationship between threads. To gather this additional information we considered three data structures which store memory-access information.

- The page table provides the complete mapping from virtual to physical address space. The detection of heavily used pages is difficult due to the low scan rate of the clock algorithms used to provide information for paging decisions. The page table is not a CPU-local structure, so we cannot distinguish the page access of multiple processors. Traversing the page table is a complex task and the access bits of the page table entries do not provide information to distinguish the page access of multiple threads assigned to the same process. Finally, the page table does

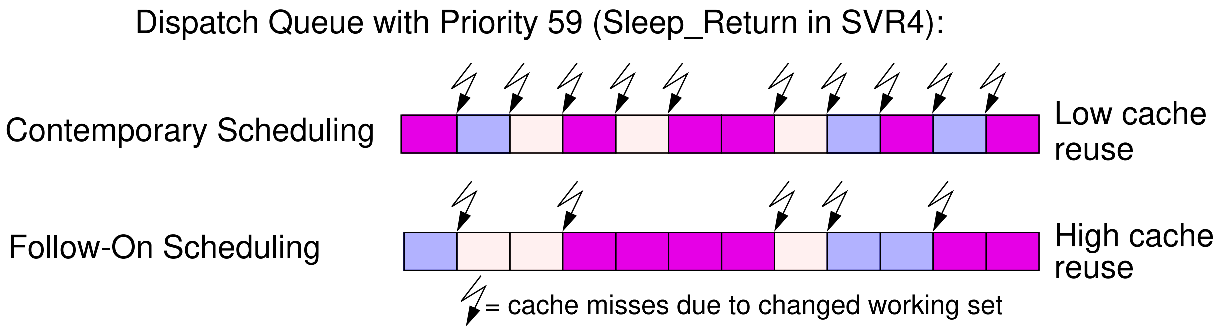


FIG. 1.2. Reordering of run-queues by Follow-On Scheduling decisions

not allow the detection of shared kernel resources.

- Small memory regions are frequently used to cache the page table. These page table caches narrow the gap in access speed between the tiny full-associative TLB and the complex page table. The translation storage buffer (TSB) of the SPARC V9 architecture is one example of such caches, which are managed by software. The design objective of the TSB is to store the hot spots of the page table in the CPU caches, speeding up the resolution of TLB misses. The TSB is a CPU-local structure which is easy to access. The typical size of the TSB allows the analyses of the relationship between processes as the number of TSB entries (usually 32768 pages) is quite high. It can therefore be used to identify shared pages. The TSB gives no hints to enable the detection of related threads because it contains context information but no thread-specific information.
- The TLB is a hardware structure which is hard to read. But the TLB-miss handler can be modified such that it makes a note of TLB misses in a trace buffer local to each CPU. Because there is no kernel virtual memory while executing the TLB-miss handler, the thread-id of the currently running thread has to be stored in a CPU-local memory region in physical memory. This task is done in the switch routine. Now we can trace the thread-id causing a TLB miss and the page frame number loaded into the TLB (see FIG. 2.1.). Because the trace is not influenced by the MMU context, the page access of a thread can be observed independent whether the thread is running in user- or in kernel-space.
By periodically analyzing the trace buffer, we know exactly which pages are accessed by a specific thread and the degree of relation between all threads of execution.

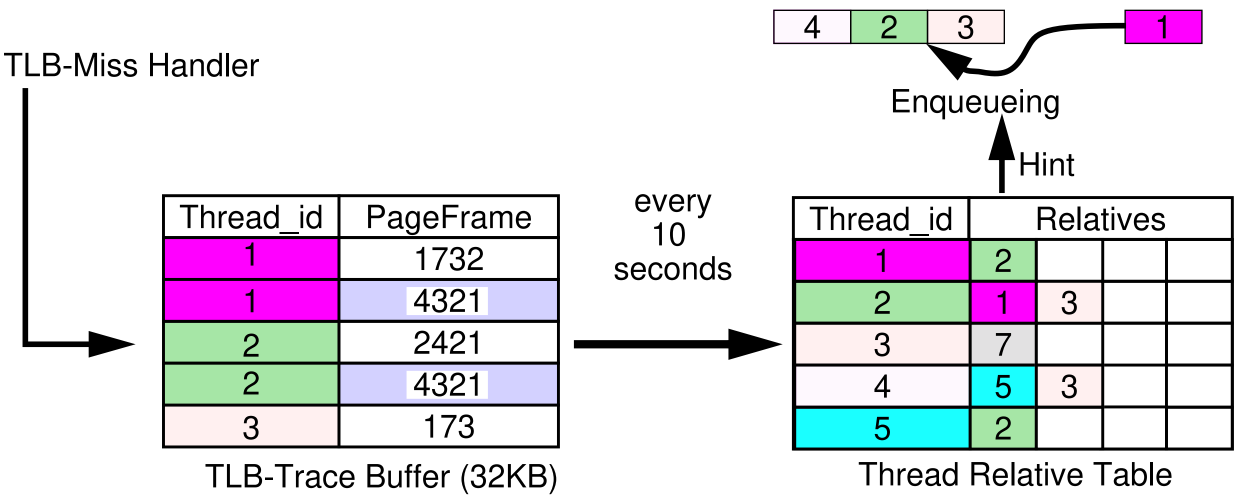


FIG. 2.1. TLB-Miss Guided Enqueuing

Despite the complex modification of trap handling, switching routines and CPU-specific data structures, we chose the third approach for our implementation of Follow-On Scheduling. A SUN Enterprise server running the Solaris 2.5.1 operating system is the platform of our prototype.

The analyses of the trace data are done by the scheduler, which updates a table representing the relation between threads surviving multiple scheduling cycles (short-living interrupt threads are filtered out).

The *Thread Relative Table* (see FIG. 2.1.) gives hints to the enqueueing functions. If a related thread is found near the location where a thread is designated to be enqueued (normally at the front or back of the queue), other threads can be bypassed. To prevent starvation, the scope of the lookup function should be limited to two or three queue positions. Furthermore a thread can only be bypassed a few times.

TABLE 2.1.Snapshot of runqueue 59

| Contemporary Scheduling | Follow-On Scheduling |
|---|---|
| elis | elis |
| elis | elis |
| httpd-1.2.1 | elis |
| httpd-1.2.1 | elis |
| elis | httpd-1.2.1 |
| elis | httpd-1.2.1 |
| httpd-1.2.1 | httpd-1.2.1 |
| httpd-1.2.1 | httpd-1.2.1 |
| elis | httpd-1.2.1 |
| httpd-1.2.1 | elis |
| elis | httpd-1.2.1 |
| httpd-1.2.1 | httpd-1.2.1 |
| httpd-1.2.1 | httpd-1.2.1 |
| elis | elis |

The results are very promising. On our server we could clearly identify classes of threads sharing a lot of pages. For example we ran a highly loaded Apache WWW-server and the **E**rlangen **Li**brary **S**ystem (ELiS) with 30 active users searching the libraries of the university. Both applications imply frequent blocking due to I/O events. Snapshots of the runqueues (see TAB. 2.1.) of a highly loaded server demonstrate the effects of Follow-On Scheduling. Without any application-specific knowledge, the scheduler is able to identify threads sharing physical pages and to enqueue them so that they follow upon each other.

## 3. State of Work

Follow-On Scheduling is running in a production environment on SUN Ultra-I workstations and on a SUN Enterprise 3000 server at our department.
With approximately 20,000 TLB misses per second the overhead of TLB tracing amounts to about 5000 second-level cache misses per second. This can be neglected on a server where half a million cache misses occur per second. The analysis of the trace buffer every 2-10 seconds adds 10,000 cache misses per second.
To determine the number of cache misses more precisely, we have established virtual event counters, which register events like cache misses or clock ticks. While handling traps we update virtual 64-bit counters located in the CPU- and thread-structures using the values of the physical 32-bit counters (see FIG. 3.1.) .

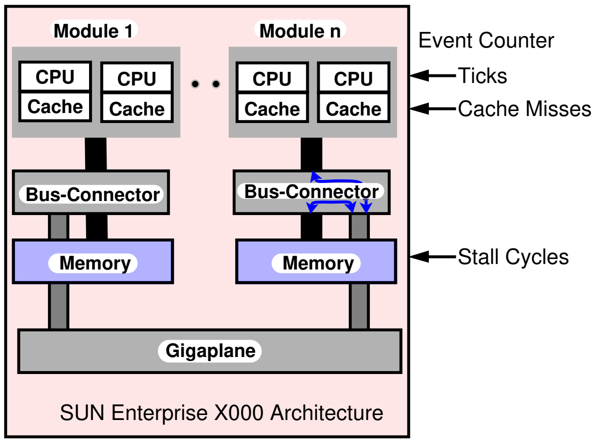


FIG. 3.1. Enhancing the thread- and CPU-context by level-2 cache misses and clock ticks

It was our intention to access the virtual counters from user-space without any system call because we do not want to influence the operating system by additional calls. Our approach is an enhancement of the /proc-filesystem, which provides kernel-virtual addresses of those regions where virtual counters are stored. After mapping those regions from /dev/kmem into the address-space of a profiling-

daemon, there is no need for system intervention to access sampling data.
An unmodified application can now be observed from a dedicated CPU without any influence on the application's execution.

Having obtained detailed information about the cache- and TLB-behavior of our system, we realized that the benefit of Follow-On Scheduling is equivalent to the overhead. We could therefore see no reproducible performance improvement on our production servers. The reasons are:

- The CPUs browse through large code and data regions inside the scheduler and dispatcher. This exceeds the capacity of the caches found in our machines (512 KB 2nd-level caches)
- The shared libraries used in our applications (libc, libsocket, etc.) do not use functionality provided by the hardware (Visual Instruction Set) to bypass the cache while copying memory. Polluting the cache with load/store operations of one-way data displaces reusable cache entries.
- The physical pages frequently have the wrong page color. Erratic page coloring implies a large number of conflict misses.

Therefore the second-level cache is frequently cleared of shared data and code. The few remaining cache lines available for reuse are not sufficient to demonstrate that the number of cache misses can be reduced and thus the overall performance improved.

## 4. Future Work

With direct mapped caches, a properly colored memory is a prerequisite for reproducible results on a high level. We hope to have access to systems which provide page recoloring in the near future.
To prove the concept of Follow-On Scheduling we plan to design a memory-conscious scheduler and dispatcher which is adapted to the special needs of high clocked RISC architectures with large caches.

## 5. References

[1]. F. Bellosa, "The Performance Implications of Locality Information Usage in Shared Memory Multiprocessors", *Journal of Parallel and Distributed Computing*, Special Issue on Multithreading for Multiprocessors, Vol. 37 No. 1, Aug. 96

[2]. "UltraSPARC Programmer Reference Manual", Revision 1.0, SUN Microsystems Inc., 1995