A Study of the Structure and Performance of MMU Handling Software

Yousef A. Khalidi
Vikram P. Joshi
Dock Williams

SMLI TR-94-28 June 1994

Abstract:

Modern operating systems provide a rich set of interfaces for mapping, sharing, and protecting memory. Different memory management unit (MMU) architectures provide different mechanisms for managing memory translations. Since the same OS usually runs on different MMU architectures, a software "hardware address translation" (hat) layer that abstracts the MMU architecture is normally implemented between MMU hardware and the virtual memory system of the OS. In this paper, we study the impact of the OS and the MMU on the structure and performance of the hat layer. In particular, we concentrate on the role of the hat layer on the scalability of system performance on symmetric multiprocessors with 2-12 CPUs. The results show that, unlike single-user applications, multi-user applications require very careful multi-threading of the hat layer to achieve system performance that scales with the number of CPUs. In addition, multi-threading the hat can result in better performance in lesser amounts of physical memory.

Sun Microsystems Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
yousef.khalidi@eng.sun.com
vikram.joshi@eng.sun.com
dock.williams@eng.sun.com

A Study of the Structure and Performance of MMU Handling Software

Yousef A. Khalidi

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

Vikram P. Joshi *Dock Williams*

SunSoft
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

Modern operating systems, such as Chorus® [1], MACH® [9], NT™ [4], Solaris®/UNIX® SVR4 [5], and Spring [7] provide a rich set of interfaces for mapping, sharing, and protecting memory. Typically, these systems allow portions of files (or more generally, memory objects) to be mapped simultaneously in different address spaces at different virtual addresses, with perhaps different protection attributes. These systems also provide other Virtual Memory (VM) mechanisms such as copy-on-write, and zero-fill on-demand that are heavily used in program start-up and execution. All these features lead to rather complex VM system implementations.

Most modern operating systems are portable to different hardware architectures. Portability requires careful separation of the machine-independent code from machine-dependent code, and keeping the OS as machine-independent as possible. These requirements necessitate a layer of software to abstract the memory management unit (MMU) architecture and to isolate the bulk of the VM system from the particulars of the hardware. A software "hardware address translation" (hat) layer is normally implemented between the VM system and the MMU hardware.

This paper is concerned with the structure and performance of the so-called hat layer. Specifically, we are interested in studying the following two aspects of this layer:

- The functionality needed in the hat layer as dictated by, on one hand, the requirements of the OS, and on the other hand, the diversity of MMU architectures.
- The impact of the hat layer on the scalability of the OS on medium scale symmetric multiprocessors (2-12 CPUs) using multi-user benchmarks.

The following approach was taken in this study: We surveyed several modern portable operating systems. The operating systems included micro-kernel as well as more traditional monolithic systems, and ranged from research prototypes to “industrial-strength” systems. We then used the common machine-independent VM features of these systems to define the interfaces and required functionality of the hat layer. Subsequently, we chose a typical hat layer on a commercial multi-threaded OS and examined its performance with a set of multi-user and compute-intensive benchmarks on medium scale multiprocessor systems (2-12 CPUs). We conducted several experiments and used the results of the experiments to guide our changes of the implementation.

The results show that careful multi-threading of the hat layer is required to achieve system performance that scales with the number of CPUs for multi-user applications. Also, preliminary results show that multi-threading the hat can allow the system to perform better when less physical memory is available, due to parallel handling of minor page faults and more aggressive page table reuse.[1]

This paper is structured as follows: Section 2 surveys some work related to the subject of the paper. Section 3 derives a set of requirements that affect the structure of the hat layer. Section 4 then presents the structure of a typical hat layer. In Section 5, the performance of a particular hat implementation is studied through a series of experiments with various benchmarks on medium scale multiprocessor systems. Lessons learned from this study and future work are presented in section 6.

2 Related Work

Most modern operating systems have a clean separation between the machine-dependent and independent portions of the system. The literature describes several implementations of machine-dependent MMU hat layers[2] [1, 4, 5, 10]. Although hat implementations normally do support multiprocessor systems, to our knowledge there is no published material that describes the performance aspects of the hat layer and the scalability of the implementations on multiprocessor systems.

Other related material includes a large body of literature on MMU hardware structures (see [6] for a good survey), and software page table issues (e.g., [3, 8]). In general, previous work concentrated on the structure of the MMU as it affects performance on uniprocessor systems. With the notable exception of [6], previous works did not consider multi-user benchmarks to any great extent.

3 Requirements

The structure and the interface of the hat layer are dictated by a number of requirements. Such requirements fall into two broad categories: requirements imposed by the operating system (in particular, the VM system); and requirements imposed by the hardware (in particular, by MMU/TLB/cache architecture).

3.1 OS requirements

We studied several modern operating systems, specifically Chorus, MACH, NT, Solaris/UNIX SVR4, and Spring. All of these systems share the following features:

1. Each system runs on several machine architectures, including uniprocessor and multiprocessor systems. Each of the systems has been ported to at least three MMU architectures.

1. Minor faults occur when the required page is in memory, but the page table does not contain the required translation to the page. Copy-on-write faults are also included in this category.

2. Other names used for the software module that handles the MMU include Chorus’s GMI, MACH’s pmap, and NT’s HAL. Solaris, UNIX SVR4, and Spring all use the term “hat,” and actually share very similar implementations of this module.

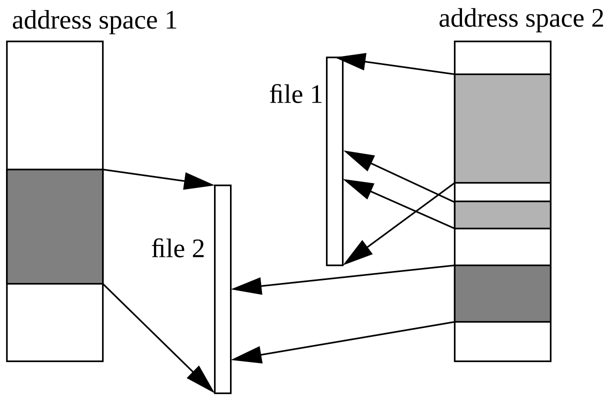

Figure 1. Example of possible memory mappings

2. There is a clear separation between the machine-independent and machine-dependent parts of the system. In particular, the VM system is by and large written without assuming a particular MMU or machine architecture.
3. There is a rich set of interfaces for mapping, sharing, and protecting memory. A given memory object (e.g., a file) may be mapped simultaneously in different address spaces at different virtual addresses, with perhaps different protection attributes. The fundamental VM operations can be abstracted as follows:

   ```
   map(address_space, memory_obj,
       address, length, mobj_offset,
       protection_attributes)
   ```

   ```
   unmap(address_space, address,
       length)
   ```

   ```
   change_protections(address_space,
       address, length,
       new_protections)
   ```

   Figure 1 shows some memory mappings that are possible using these VM operations.
4. There is a need to keep track of which pages are modified (dirtied), and which are referenced. The VM normally requires some support from the machine-dependent portion of the system for maintaining the modified and referenced bits.

3.2 MMU requirements

A wide variety of MMU architectures are used by modern operating systems. Despite this wide variety of architectures, some characteristics can be observed:

1. Some MMUs impose a specific page table structure that is accessed by the hardware when handling translation lookside buffer (TLB) misses. In general, this hardware-imposed page table is only a cache of all translations, and intervention of system software is necessary if the hardware fails to locate a desired translation in the page table.
2. Some MMUs require the software to handle all TLB misses. Such MMUs do not impose a specific page table structure. However, for performance reasons, they do require fast software TLB miss handling. Fast software TLB handling usually implies the need for (software) page tables that are fast to access.
3. Some MMU and machine architectures require other cache and TLB handling from the software. For example, in multiprocessor systems, TLB shoot-down might be required when unmapping pages. In virtually-indexed caches, appropriate cache flushing might be needed.
4. Some MMUs require the software to manage a limited number of address space contexts or segment registers.
5. Finally, some machine architectures require managing separate I/O or graphics MMUs that are used to set up translations to and from device space. Depending on the machine architecture, the I/O translations maybe restricted to the kernel address space, or they may map the device space to any user address.

4 HAT Structure

The requirements of Section 3.1 result in a set of operations that the hat layer should provide. Section 4.1 describes such a hat interface. In turn, the interface of the hat layer coupled with the MMU requirements of Section 3.2 dictate a set of data structures that the hat layer should implement. The data structures can be divided into two parts:

machine-independent structures that are needed to support the interface of the hat layer (Section 4.2), and data structures that depend on the particular MMU/machine architecture (Section 4.3).

4.1 Interface

The interface to the hat layer is usually simple, and can be abstracted as follows:

1. Load translations from an address space to a set of pages.
2. Unload translations to a set of pages from all address spaces (and CPUs).
3. Unload translations of a set of virtual addresses from one address space (and all CPUs).
4. Change the protection on a set of virtual addresses from one address space (and all CPUs).
5. Set/Clear reference and dirty bits.

4.2 Machine-independent data structures

Most hat operations involve address spaces and/or individual pages, and there is a hat-specific data structure to represent each address space and page in the system. Since a given page maybe mapped at more than one virtual address simultaneously, each page structure includes a list of all active mappings to the page. Moreover, since an address space may have translations active in more than one MMU simultaneously, a list of MMU specific information (one per MMU) is needed per address space. Figure 2 graphically shows the relationships among the hat data structures. Much of the complexity of the hat layer lies in efficiently arranging and manipulating these data structures.

Note that although hat layers from different operating systems may have a somewhat different structure than the general structure shown in Figure 2, the information maintained by a different hat implementation is basically the same. For example, some implementations may not have a list of MMU descriptors, but instead may allocate a fixed array of descriptors. Figure 2 closely resembles the hat layer implementations used in Solaris, UNIX SVR4, and Spring.

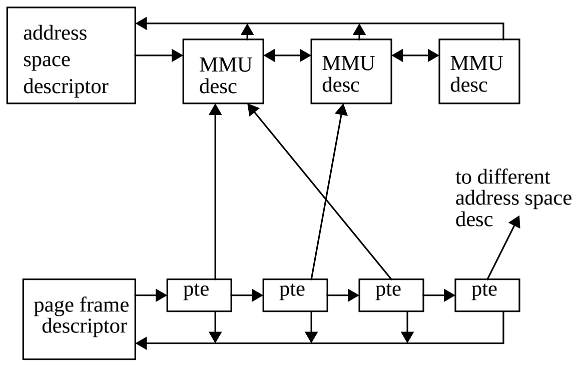

Figure 2. Basic hat data structures

Each address space descriptor has a list of MMU descriptors, where each MMU descriptor includes an MMU specific page table for the address space when using this particular MMU. The size of this list is ≤ the number of MMUs in the system. Each page frame descriptor has a list of all translations to the page (indicated as page table entries, pte's, in the figure). Each translation belongs to one address space in a particular MMU.

4.3 Machine-dependent data structures

In addition to the data structures described in the previous section, the hat layer maintains the MMU page tables (if any are required). For example, some MMUs such as the one used in SuperSPARC^{TM} [2] require a page table per address space organized as a three-level tree. The "MMU descriptor" in Figure 2 would in this case contain a pointer to such a three-level table.

The hat layer may implement a page table that acts as an extended cache for the TLB even if the MMU does not require such a page table. For example, the MMU in the R4000® [9], has a software-controlled TLB and the hardware does not specify a "page table" structure. However, fast lookup by the software is very important for efficient TLB miss handling and, therefore, the hat layer may implement a page table of translations anyway. This page table is then searched by a fast trap handler on TLB misses. If the translation is not found by the handler, a higher-level fault is declared, and the VM system takes over the task of locating the translation.

For machines that require support for I/O or graphics MMUs, the "MMU descriptors" of Figure 2 may also contain a pointer to the I/O device-specific page table (which may have an organization different from the MMU page table format).

Finally, the hat may also maintain some other machine-dependent data structures, for example, free memory pools of page table structures.

4.4 Synchronization

The simplest implementation of a hat layer can synchronize several simultaneous requests by using one global hat lock. Each hat call acquires this lock, executes the operation, then releases the lock. Multiple threads are serialized using the global lock and no parallelism of hat operations is possible. On uniprocessor systems or very small multiprocessors, this is an acceptable solution.

More sophisticated implementations may use several locks to allow more than one hat operation to execute in parallel. In the remainder of this section we look into some of the issues related to multi-threading the hat layer. We investigate the impact of the various synchronization schemes on performance in the next section.

Most hat operations fall into one of two classes:

1. Operations that modify one or more page table entry (pte's) on a page descriptor mapping list (e.g., unloading the translations to a page, or updating the reference and modified bits from the TLB on some MMUs). These operations usually change some of the state flags in the page frame descriptor as well.
2. Operations that modify one or more mappings in an address space (e.g., loading or unloading a virtual address to a page translation).

Note that some hat operations end up falling into both categories. As can be seen from Figure 2, there are several data structures that need to be protected:

- Each page frame descriptor and its pte mapping list.

  Since operations that fall into category (1) above usually access at least one pte and the page frame descriptor itself, it is sufficient to provide one mutual exclusion lock per page frame descriptor that also protects the mapping list.
- Each address space descriptor.

  Since the number of MMU descriptors for a given address space does not change very often (and is bounded by the number of MMUs in the system), using a reader/writer lock to coordinate accesses to the list using the following semantics is appropriate: a reader lock is held to traverse the list, while a writer lock is needed to add or remove from the list.
- Each MMU descriptor.

  Operations of category (2) above traverse the MMU descriptor list of a given address space (after acquiring a reader's lock on the address space descriptor), then modify the state of an MMU descriptor (which contains among other things any hardware-dependent page tables). To allow parallel operations on the same address space, a mutual exclusion lock is provided per MMU descriptor.
- Free resource lists.

  The hat layer usually maintains some free resource lists, and accessing these lists needs to be protected by appropriate mutual exclusion locks (such free resource lists may include memory blocks that represent machine-dependent page table components).

Due to memory considerations, allocating a mutual exclusion local variable (mutex) per descriptor may not be feasible. (For example, on a 64MB machine with 4KB page size, reserving 4 bytes for a mutex per page frame descriptor consumes 64KB of memory just for the space taken by the mutex variables.) To conserve space, the per-frame descriptor and per-MMU descriptor locks can be logical locks instead of actual mutexes. A monitor-like implementation then would use:

- a per descriptor *busy* and *wanted* flags, and
- a global descriptor mutex for page frame descriptors, and another for MMU descriptors.

The implementation of the lock is straightforward: after acquiring the appropriate global mutex, the per-descriptor busy flag is checked. If the flag is set, then the wanted flag is set and the thread sleeps on a condition variable.[3] When the busy flag is clear, the thread sets it (to indicate that it has acquired the lock) and releases the global mutex.

An alternative implementation is to use an array of global mutexes, where each mutex in this array controls a set of descriptors. A simple mapping function is then used to choose a mutex given a descriptor (e.g., by hashing the address of the descriptor).

5 Performance Evaluation

5.1 Scalability on MP systems

In evaluating the performance of the hat layer, we wanted to answer the basic question of how the structure of the hat layer affects the scalability of multiprocessor systems?

To answer this question, we started with an actual implementation of a multi-threaded operating system and its hat layer. Specifically, we used the Solaris operating system, a multi-threaded operating system that runs on a variety of machines and MMUs, including uniprocessor and multiprocessor systems. The particular release of the system we used (Solaris 2.1) had a fully multi-threaded kernel, but used a single lock around all hat operations. Although considerable effort went into multi-threading the VM system, the file system and other traditional parts of the OS, a single lock around the hat layer was initially thought to be an acceptable solution to protect the hat data structures.

The next section describes the benchmarks used to evaluate the scalability of the system. We conducted several experiments (Section 5.3) and used the results of the experiments to guide our changes of the implementation.

5.2 Benchmarks

We chose four benchmarks that exercised different portions of the system. Table 1 summarizes the benchmarks. Kenbus1 and TPC-B are macrobenchmarks that characterize the capacity of a system in a multi-user environment, while SPECrate_int92 is mostly a CPU benchmark. Kenbus1 is based on the SPEC SDM (System Development Multi-Tasking) suite [12]. It measures the throughput of the system by simulating different users executing around eighteen UNIX commands, including the C compiler and other commands. TPC-B is a standard database benchmark.

SPECrate_int92 measures CPU capacity (throughput) of a system as opposed to CPU speed. The benchmark consists of running multiple copies of the SPEC92 suite to determine the capacity of the system [12].

As its name implies, Microbench is a micro benchmark that measures various OS operations such as forking a process, and page fault handling. It attempts to measure individual operations under contention by increasing the load as the number of CPUs increase. The version we used is based on the BSD benchmark suite. Microbench is useful in understanding the effects of various changes on the behavior of individual

3. A per-descriptor condition variable is usually needed anyway, so this scheme adds only two flags per-descriptor.

operations and in guiding the effort of tuning the operating system.

Benchmark	Type	Description
Kenbus1	macro	Represents UNIX/C usage in an R&D environment.
TPC-B	macro	A database benchmark
SPECrate_int92	CPU	Measures CPU throughput of an MP system as opposed to CPU speed.
Microbench	micro	Measures individual system operations under contention.

Table 1. Benchmarks used

5.3 Experiments

We conducted most of our experiments on a SPARCcenter^{TM} 2000, a shared bus symmetric multiprocessor [11]. The machine we used had 12 50Mhz SuperSPARC CPUs each with a 1MB second-level cache. We varied the amount of physical memory from 128MB to 512MB depending on the experiment. We also used a 4 CPU SPARCserver^{TM} 600 series machine for some Microbench experiments as described below. Several runs were used for each data point.

Starting with the single global mutex, we ran a profiled version of the OS that calculated the amount of lock contention when running TPC-B benchmark on an 8 CPU system with 512MB of memory. The results showed that the hat layer lock was found busy (held) between 16-20% of the total times it was acquired. Calculating the total wait time on the single hat mutex showed that this lock had by far the longest total wait time of all locks in the OS. This clearly showed that the hat lock was a bottleneck in at least this database benchmark.

We proceeded with multi-threading the implementation of the hat layer as described in Section 4.4: we used a lock per page frame descriptor, a lock per MMU descriptor, and a global lock for the MMU free resource list. (The address space descriptor already had a readers-writers lock as part of the VM system, so we used this lock to control traversals of the MMU descriptor list.) The per-page frame and per-MMU descriptor locks were logical locks implemented as described before in Section 4.4, using per-descriptor busy and wanted flags and two global mutexes (page_mutex and mmu_mutex). We used a third global mutex for the machine-dependent resource pool of the hat (hat_resource_mutex).

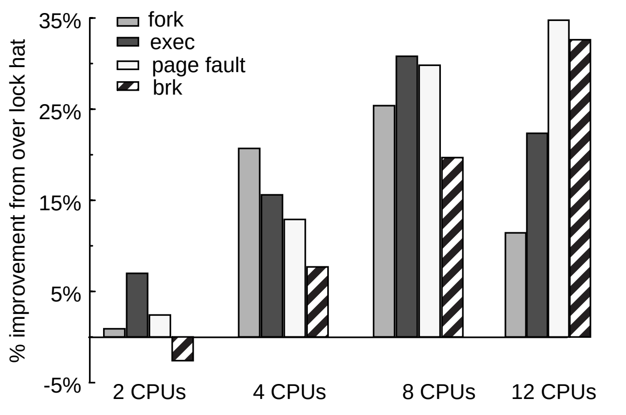

Figure 3. Microbench using 512MB of memory

Four categories were measured: "fork" includes `fork` and `vfork` with sizes ranging from 0 to 1MB, "exec" includes `exec` and `vexec` of a null process with sizes also ranging from 0 to 1MB. "Page fault" includes minor (non-disk related) faults, plus copy-on-write faults. "Brk" results in allocating more zero-filled memory to the requesting process.

Measuring lock contention on the multi-threaded hat using the same TPC-B benchmark in an 8 CPU configuration showed contention values of 6-9% for page_mutex, 2-3% for mmu_mutex, and <1% for hat_resource_mutex. Although the total number of mutex lock operations increased by 38%, the total wait time on hat-related locks went down by 49%.

Next, we evaluated the micro effects of multi-threading the hat. Specifically, we ran Microbench using the single-lock hat and the multi-threaded hat on several CPU configurations. As described in Section 5.2, Microbench attempts to calculate the cost of individual operations under contention. We concentrated on four operation categories that involved the hat: forking, execing, handling minor page faults, and increasing the memory allocated to a process ("brk" operation). For each category we averaged several operations. For example, "fork" is actually the average of fork and vfork operations, where each operation is executed for sizes 0B, 10KB, and 1MB. Figure 3 shows the results as a percentage decrease of the operation cost relative to the operation cost in the single lock implementation. With the exception of the "brk" operation for the 2 CPU case, the multi-threaded implementation is faster for all configuration and the performance difference increases as the number of CPUs increase up to 8 CPUs. Beyond 8 CPUs, other kernel locks start to reduce the effectiveness of the multi-threaded hat for the fork and exec calls.

To evaluate the macro effect of multi-threading the hat, we ran the TPC-B benchmark on an 8-CPU machine with 512M of memory. The results are shown in Figure 4. Multi-threading the hat makes a very pronounced difference on the scalability of this database benchmark

Figure 5 shows the results after running the Kenbus1, another macro benchmark. Although this benchmark does not scale with the number of CPUs as well as TPC-B, the positive effects of the multi-threaded hat implementation are clearly seen nonetheless. Examining the lock contention data for this experiment showed that the dip in performance when using 12 CPUs is not due to the hat layer. Multi-threading the hat layer allowed the system to scale to a larger number of CPUs and exposed other kernel lock bottlenecks.

Figure 6 shows the results of running the SPECrate_int92. As we expected, our changes to the hat had little impact on this mostly CPU intensive benchmark.

Next, we were interested in seeing the effect of multi-threading the hat while varying the amount of physical memory. We conducted a series of

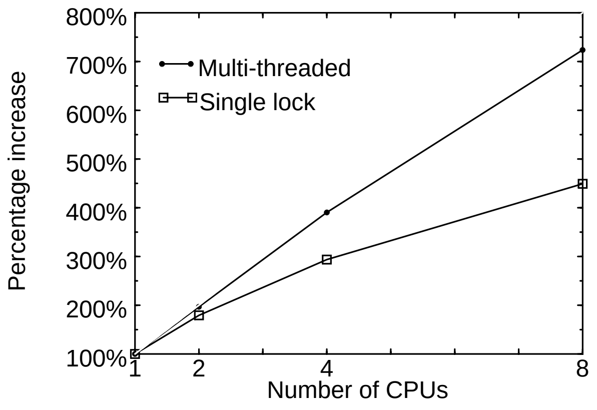

Figure 4. TPC-B using 512MB of memory

Comparison of TPC-B benchmark performance using 1-8 CPUs. The multi-threaded implementation scales almost linearly, while the single-thread implementation does not scale well beyond 2 CPUs.

experiments using Microbench on a 1-4 CPU SPARCServer 600 system, while varying the amount of physical memory from as low as 16MB to 128MB. The results are shown in figures 7 and 8. There was no significant paging activity during any of the Microbench runs.

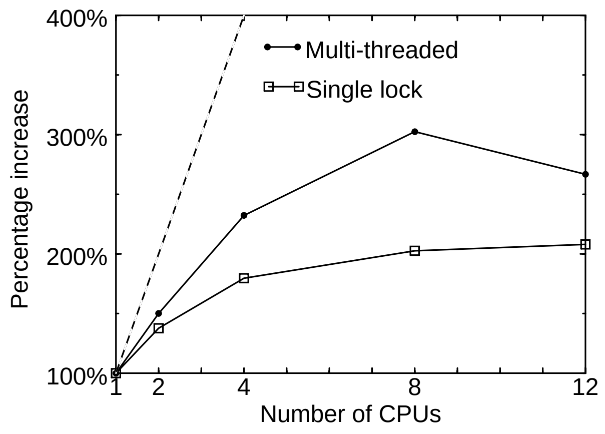

Figure 5. Kenbus1 using 512MB of memory

Comparison of Kenbus1 performance using 1-12 CPUs. Note that the drop between 8 and 12 CPU is due to the multi-threaded hat layer exposing other kernel lock bottlenecks.

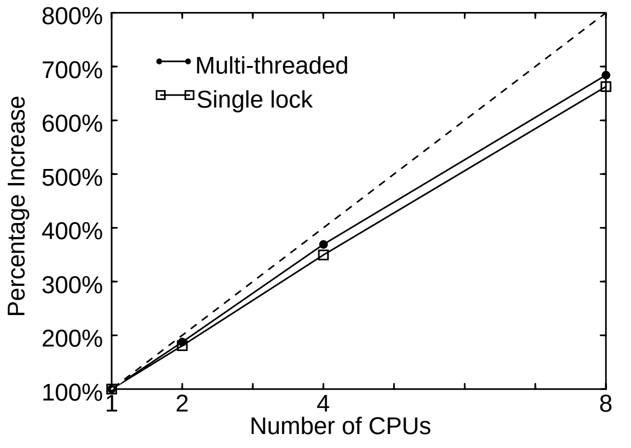

Figure 6. SPECrate_int92 using 512MB of memory

SPECrate_int92 shows little improvement using the multi-threaded hat implementation.

It is interesting to note the comparison of performance between the 128MB single lock vs. 16MB multi-threaded implementations (left-most column in figure 8). For this micro benchmark, multi-threading the hat enables better performance at 16MB than a single-lock implementation that uses 128MB of memory. We believe that this is partly due to the multi-threaded implementation aggressively returning page table memory to the free resource pool when unloading translations. A load operation running in parallel can then reuse the memory and complete the operation while the unload operation is still running. In contrast, the single lock implementation delays freeing the memory until its task is complete (which may involve a lengthy TLB shoot down operation).

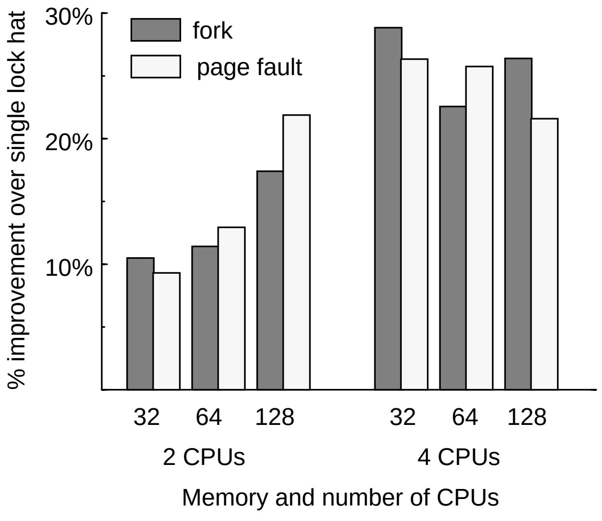

Figure 7. Microbench w/different memory and number of CPUs

Microbench runs for different memory sizes and number of CPUs. All memory sizes are in megabytes.

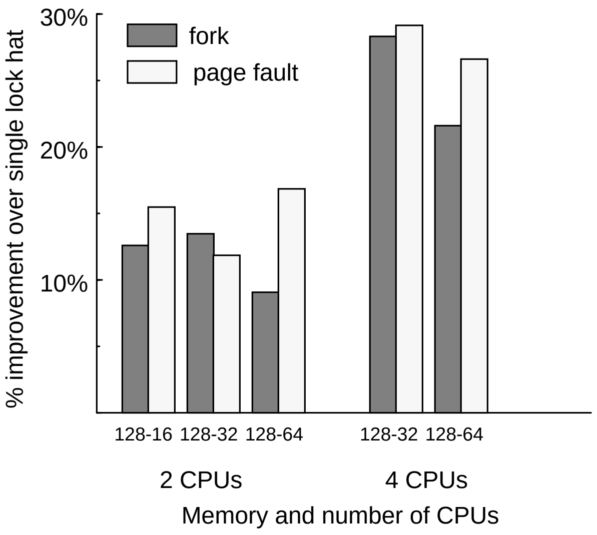

Figure 8. Microbench comparing 128MB single-lock implementation with different multi-threaded memory sizes

Comparison of a 128MB single-lock hat system with multi-threaded hat systems of various memory sizes. For example, “128-16” designates a comparison of the performance of 128MB single lock hat system with a 16MB multi-threaded hat system. Using this benchmark, a multi-threaded system using only 16MB of memory has better performance than a single-lock hat system with 128MB of memory.

We did not experiment with varying the size of physical memory when running multi-user benchmarks. The interaction of I/O, physical memory, and the role of the hat layer is an area we plan to investigate as part of our future work (see Section 6.2).

6 Conclusions and Future Work

6.1 Conclusions

Multi-threading the hat layer is required for applications that use the OS heavily. For a class of multi-user applications, a single lock around the hat data structures does not allow the system to scale to more than a few CPUs. The OS, after all, is a server that is exercised heavily by multi-user, general purpose applications. The TPC-B and Kenbus1 benchmarks benefited considerably from the extra parallelism.

Increasing the parallelism in the hat layer allows more page faults to proceed in parallel. In particular, more minor faults can be handled in parallel, which has implications on the throughput of a multiprocessor:

- More CPUs can be handling minor faults in parallel.
- Less amount of physical memory may need to be set aside for (hardware or software) page tables.

However, the results of the interaction of memory size and the hat are very preliminary.

Finally, multi-threading the hat layer alone is not enough. Our work was motivated by previous multi-threading efforts that were done in early versions of Solaris, particularly to the VM and file systems. The overhead of the hat lock was observed only after the VM and file systems were considerably tuned. In turn, our work exposed newer bottlenecks in the system. Anticipating the time when the hat becomes a bottleneck again, we identify a number of further improvements in Section 6.2.

6.2 Future Work

We identify below more possible improvements to the hat layer implementation. These improvements may become necessary in the future after other OS locking bottlenecks are removed:[4]

1. Hash the global page_mutex as described in Section 4.4.

   The global page_mutex shows moderate (6-9%) contention in some benchmarks. This mutex can be replaced by an array of mutexes. A particular mutex is then chosen based on hashing the address of the page structure pointer. The hashing would prevent unnecessary blocking of unrelated accesses to pages that fall into different hash buckets. A similar scheme can also be applied to the mmu_mutex if the need arises.

2. Handle sparse address spaces efficiently.

   We see a trend towards sparse address spaces, with the average size of each contiguously mapped area getting smaller. This can be seen today in UNIX systems that use a large number of small dynamically-linked libraries. Most current hat implementations are not well-suited for handling sparse address spaces.

3. Use multiple locks for hat free resource lists.

   There is a single hat_resource_mutex that protects all of the hat free lists which can be broken down into a lock per list.

4. Simplify LRU context list.

   It may help to do away with maintenance of an LRU context list on a machine with lots of contexts until a tunable percentage of them are allocated.

We are now seeing more and more multi-threaded applications that run in the same address space. To allow maximum thread parallelism, there is a need to multi-thread hat operations on the same address space. Our locking strategy (Section 4.4) allows many hat operations to proceed in parallel on the same address space (e.g., two load translation operations on different MMUs). It would be interesting to quantify the effect of this potential parallelism on real applications.

4. Most of the listed improvements were made to more recent versions of the Solaris operating system after our study was concluded.

An area which we plan to address is the interaction of I/O, and the amount of physical memory as they relate to the hat layer. A fixed percentage of physical memory is normally dedicated to the hat translation tables. This percentage is calculated at boot time and is a function of the total amount of physical memory (plus perhaps a function of the number of CPUs). The translation tables are used as a cache of total translations in the OS. If the cache is too small, the system may experience a large number of minor faults. If the cache is too big, memory that otherwise could be used for applications may be wasted. We would like to experiment with different sizes for the hat page table pool, and with the classes of applications that would benefit from bigger page tables.

Acknowledgments

We would like to thank Gyan Bhal for obtaining much of the performance data. We would like to also thank Nhan Chu and Todd Clayton for their help.

References

[1] Abrosimov, Vadim, Marc Rozier, and Marc Shapiro. “Generic Memory Management for Operating System Kernels.” *Proceedings of the 12th Symposium on Operating Systems Principles (SOSP '89)* (1989): 123–136.

[2] Blanck, G. and S. Krueger. “The SuperSPARC Microprocessor.” *COMPCON* (February 1992): 136–141.

[3] Cheriton, D., A. Gupta, P. Boyle, and H. Goosen. “The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation.” *Proceedings of the 15th International Symposium on Computer Architectures* (May 1988).

[4] Custer, Helen. *Inside Windows NT.* Microsoft Press. 1993.

[5] Gingell, Robert A., Joseph P. Moran, and William A. Shannon. “Virtual Memory Architecture in SunOS.” *Proceedings of 1987 Summer USENIX Conference* (June 1987).

[6] Huck, Jerry and Jim Hays. “Architectural Support for Translation Table Management in Large Address Space Machines.” *Proceedings of the 20th International Symposium on Computer Architectures* (May 1993).

[7] Khalidi, Yousef A. and Michael N. Nelson. “A Flexible External Paging Interface.” *Proceedings of the 2nd Workshop on Microkernels and Other Kernel Architectures (*September 1993). Also *Sun Microsystems Laboratories, Inc. Technical Report SMLI-TR-93-20* (October 1993).

[8] Nagle, David et al. “Design Tradeoffs for Software-Managed TLBs.” *Proceedings of the 20th International Symposium on Computer Architectures* (May 1993).

[9] MIPS Computer Systems. *MIPS R4000 Microprocessor User’s Manual* (1991).

[10] Rashid, R., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. “Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.” *IEEE Transactions on Computers* 37 (8) (August 1988): 896–908.

[11] Sun Microsystems, Inc. *SPARCcenter 2000 Architecture and Implementation*. Technical white paper (November 1992).

[12] *SPEC Newsletter* 3 (2) (spring 1991).

About the authors

Yousef A. Khalidi is a Senior Staff Engineer at Sun Microsystems Laboratories, Inc. His interests include operating systems, distributed object-oriented software, high speed networking, and computer architecture. He is one of the principal designers of the Spring operating system. He has a Ph.D. in Information and Computer Science from Georgia Institute of Technology.

Vikram Joshi is a Staff Engineer at SunSoft, Inc., and is currently working on the virtual memory system of the Spring operating system. He has been with Sun Microsystems, Inc. for six years, and amongst other things, has ported Solaris to Sun's desktops/high end servers. Prior to Spring, he was involved in the area of software management for MMU/TLB/Cache architectures and performance evaluation for Sun's 2-20 processor machines. Mr. Joshi's research interests are in distributed systems and multiprocessor architectures. He received his M.S. in Physics and B.S. in Chemical Engineering from BITS, Pilanti, India in 1985.

Dock Williams received an S.B. in Electrical Engineering and Computer Science from M.I.T. in 1980. Prior to joining Sun, he worked at American Information Systems, ONYX Systems, TriComp Systems, and Hughes Aircraft Radar Systems.