A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machines

Ramesh Balan, Kurt Gollhardt – UNIX System Laboratories

ABSTRACT

This paper describes the design and implementation of the UNIX® SVR4.2 Virtual Memory (VM) Hardware Address Translation (HAT) layer that can be used as a model for other multiprocessor (MP) platforms in terms of scalability and MP related interfaces between the HAT layer and the machine independent layer. SVR4.2 is a SVR4.1 ES based kernel that supports shared memory multiprocessors and light weight processes in a shared address space. By implementing a fine-grained locking mechanism, a lazy Translation Lookaside Buffer (TLB) shootdown evaluation policy and other improvements over the SVR4 design, the memory management feature is made scalable in terms of number of processors as well as size of memory supported. Providing a small set of interfaces between the machine dependent and independent layers for TLB consistency and a simple set of locking requirements between the two layers, SVR4.2 facilitates the portability of the memory management feature to other multiprocessor platforms.

Introduction

A scalable and portable HAT layer that supports multiprocessors and multiple threads in an address space is described in this paper. The scalability of the implementation is primarily due to three reasons:

- The TLB shootdown policy and algorithms.
- A fine-grained locking scheme that allows memory management as a whole to be scalable with respect to number of processors.
- Design to support large physical memory configurations.

A small set of well defined MP related HAT interfaces is introduced for use by other layers of the kernel. The purpose of these HAT functions is to maintain TLB consistency in a multiprocessor environment. SVR4.2 does not assume hardware support for TLB consistency[1] and the support is provided by the HAT layer.

The HAT layer is the Memory Mangement Unit (MMU) dependent part of the memory management facility in SVR4.0 UNIX implementations. Other UNIX Virtual Memory implementations also usually contain such a machine dependent layer. In SVR4.2 (a derivative of SVR4.1 that provides support for multiprocessors and light weight processes), all but a small portion of rest of the VM subsystem is machine independent.

Traditionally, most of the porting effort is spent on implementing the HAT layer when porting SVR4 memory management feature to various architectures. This effort is much more complex in a multiprocessor environment. Also, typically, scalability issues are not emphasized during porting efforts. By providing a well defined set of interfaces and a simple locking protocol, the porting effort will be routine without any loss in the performance of the system.

Related Work

Previous work done on providing a general interface for the hardware dependent layer of VM includes the MACH pmap layer [1] and the original SVR4 HAT layer interfaces that were derived from SunOS [2]. The TLB shootdown policy implemented in SVR4.2 is similar to the MACH policy [3], however kernel address space shootdowns are handled differently from the user address space. Several solutions to TLB consistency with and without assuming hardware cache consistency have been discussed in various papers [4]. An implementation of TLB synchronization that uses a particular TLB format (TLB ID entry) has been described in [5]. The SVR4.2 implementation does not expect the TLB to contain any fields such as TLB ID other than a subset of fields in the page table entry. However, there are two areas in which SVR4.2 implementation of TLB shootdowns is machine dependent: one is when clearing the page table entries of other processors which is dependent on the MMU structure and the other is in sending inter-processor interrupts for synchronization of the processors whose TLB is being shot down.

[1] However, cache coherence is assumed to be supported by the hardware.

Background

The SVR4.0 HAT data structures were retained for SVR4.2. The reason for this is that the data structures efficiently support large, sparse address spaces in terms of space and time. The principal factor behind this efficiency is the *mapping chunks* data structure. A *mapping chunk* is used to keep track of all virtual mappings to a physical page. Each page table entry has a corresponding mapping chunk entry and each physical page has a linked list of mapping chunk entries that denotes the virtual translations to the page (*mapping chain*). The size of a mapping chunk is much smaller than that of a page table[2]. Non-active translations does not have an entry in the mapping chunk. Due to this reason, sparsely populated page tables waste very little space for providing the *mapping chain*. When operating on a large address range[3], all the page table chunks that does not have a corresponding mapping chunk are skipped, and no time is spent looking at the non existent page table entries.

The Uniprocessor (UP) interfaces from the SVR4.0 HAT layer have also been retained since the interfaces have been found to be sufficient in supporting different architectures that SVR4 has been ported to so far (including Intel386, SPARC®, Motorola 88000, MIPS). The most frequently executed UP HAT functionalities in SVR4 were to load a translation to a given page (*hat_memload()*), to unload translations for a range of addresses (*hat_unload()*) and to unload all translations to a given physical page (*hat_pageunload()*).

The reference port for SVR4.2 is on an Intel386/486 architecture and thus the initial HAT implementation is targeted for the Intel386 MMU. The following is a list of its features that are of interest:

- The Intel386 MMU uses a two level page table structure to define an address space [6]. When references to the page table entries are denoted as level 1 entries or level 2 entries in the sections below, they are in regards to this structure.
- Level 1 is the page table directory consisting of 1024 entries, each of which points to a page table. This page table is referred to as the level 2 page table.
- Level 2 page table consists of 1024 entries, each of which point to a physical page.
- The physical page size is 4096 bytes.
- The modify and reference bits are in the page table entry and are updated by the hardware.
- The i386 also provides an interlocking facility when accessing the reference and modify bits; i.e. no other accesses to the page table entry are possible when the hardware is changing these bits for that entry.

The i386 architecture can support 4 Gigabytes of virtual address space. In many RISC architectures (such as MIPS®), the modify and reference bits are simulated in software and thus, unlike the Intel386 implementation, TLB shootdowns are not required when these bits are modified.

Multiprocessor Interfaces

Most MMUs implement a simple cache known as Translation Lookaside Buffer for caching virtual to physical translations to avoid real memory accesses. In a multiprocessor environment the same virtual address can reside in multiple TLBs and the coherence of these translations needs to be maintained between the TLBs. In SVR4.2, all the exported MP related HAT interfaces are used for maintaining the TLB consistency. The number of active CPUs in the system for the kernel address space and the number of CPUs a user address space (execution entity : a *process* (consisting of one or more *light weight processes*)) is associated with is recorded to do selective TLB flushes. This is referred to as TLB accounting. The HAT layer records the TLB accounting in a HAT data structure that is associated with each address space, including the kernel address space.

All online CPUs in the system can execute in the context of the kernel address space (*kas*). The kernel address space HAT accounting structure records the current set of online CPUs. Two HAT functions are provided for establishing this accounting when bringing CPUs online or offline. These functions are used in accounting which processors' TLBs will be flushed for the kernel address space.

- `hat_online()`: Called when onlining an engine (CPU) in the system. Sets *active cpu count* field and the processor's bit in the *kas* HAT structure. It also flushes the engine's TLB.
- `hat_offline()`: Called when taking an engine (CPU) offline in the system. Clears the processor's bit set in `hat_online()` and decrements the count of active *cpus* in *kas* HAT structure.

The processor accounting for user level address spaces for shooting down TLBs is done at context switching time. Since threads within an address space can be running at the same time on different CPUs, the CPUs that are executing in the context of the same address space must be known to perform selective TLB flushes. The following interfaces are used when scheduling a light weight process (LWP) on any CPU in the system.

- `hat_asload(as)`: Called when context switching to a new LWP. It adds the

---

[2] In the i386 implementation, the size of a mapping chunk is 1/32nd of page table size.

[3] Such as unloading an address range or changing protections.

processor to the active engine (processor) accounting in the HAT structure of this address space *as* and loads this address space into the MMU (just the level 1 page table entries on the i386 architecture).

- `hat_asunload(as, flag)`: Called when context switching out a LWP. It unloads the MMU mappings for this process (again, just the level 1 translations on the i386) and takes the engine out of the active engine accounting of the HAT structure. The flag parameter indicates whether the caller wants a TLB flush to be done by this function after unloading the mappings[4]. Except for the CPU accounting, the rest of the functionality needs to be done on a UP platform as well[5]. Note that there is no need to call *hat_asunload()* if the context switch is to select a LWP in the same process.

The following are the HAT interfaces for lazy shootdown of TLBs used only on the kernel address space by the kernel segment drivers. To implement lazy TLB shootdowns (details of which is explained later), an object opaque to all other layers of VM except the HAT layer called a *cookie*, is maintained. The *cookie* reflects the age of virtual translations with respect to the TLB. In the i386 HAT implementation, the *cookie* is a timestamp but it could be a counter of some sort in other implementations. The state of the TLB the HAT records is the timestamp of the last TLB flush. The state of a virtual address will be explained in section 6.1.1. The following are the interfaces:

- `hat_getshootcookie()`: Returns an opaque value that indicates the "age" of a TLB, which is used for lazy shootdown.
- `hat_shootdown(cookie_t cookie, u_int flag)`: TLB shootdown routine for kernel address space. If any of the active CPUs in the system has an older *cookie* than the passed-in *cookie*, then the TLBs of these CPUs will be flushed. The *flag* argument is used by clients which do not use lazy shootdown[6], so all the CPUs in the system are flushed regardless of the *cookie* passed in.

[4]The SVR4.2 i386 context switch implementation does not request *hat_asunload()* to flush the TLB. This is because it has to flush the TLB after copying the page table entries for the U area of the new process it is loading. Thus it forgoes the TLB flush after unloading the mappings of the old process.

[5]The TLB flush is not necessary on some architectures whose MMUs (such as SPARC and MIPS) provide the context number as part of every TLB entry and on those architectures where TLBs are flushed on each context switch.

[6]There is no such client in SVR4.2.

The TLB shootdown interfaces for user level address spaces are not seen outside the HAT layer – the shootdown is immediate and it is done during one of the following HAT operations: unloading a translation, changing the protection of a page (only in the case of restricting permissions), remapping a virtual address to a different physical address, and in the case of clearing a modify bit of a page table entry (architecture specific).

Scalability Solutions

This section will discuss some of the features that makes the SVR4.2 implementation scalable.

TLB Shootdowns

On platforms that does not support TLB consistency in hardware, a multiprocessor kernel needs to maintain the consistency for translations that are cached in several processors' TLBs. The TLB is a common feature of present day architectures since it avoids any memory accesses (two in the case of a i386 architecture) in translating a virtual address to physical address if the address is present in the TLB cache. The shootdown algorithms depend on the existence of a hardware facility to issue cross-processor interrupts. TLBs are fully flushed[7] as opposed to flushing single TLB lines [5]. Since, the shootdown algorithms are MMU architecture dependent, they are part of the HAT layer in SVR4.2.

A lazy shootdown policy for the kernel address space has been used whereas immediate shootdowns are employed for the user address space. Since the kernel virtual address usage is in the control of the kernel, a lazy evaluation of the inconsistent TLB states can be done. However, for a user address space, multiple TLBs need to be immediately brought to a consistent state since SVR4.2 supports multiple LWPs in an address space which can concurrently execute on multiple CPUs.

Lazy Shootdowns

A lazy evaluation policy is very important for the kernel address space. When a kernel virtual address translation is unloaded, all processors' TLBs in the system need to be brought to a consistent state. This is because all processors in the system share the kernel address space (in a symmetric multiprocessor architecture). Delaying shootdowns may avoid doing the shootdowns entirely since the TLBs might be flushed already when the evaluation is done (due to a context switch, for example).

The kernel segment drivers essentially determine the laziness of a shootdown in kernel address space. Two major users of this policy in SVR4.2 are the *segkmem* driver which manages the permanently resident kernel memory and the *segmap* driver which

[7]The Intel386 architecture does not support single line TLB flushes except through the use of an unsupported multi-instruction sequence.

manages transient file mappings used by file system read and write system calls.

When a kernel virtual address is freed by a kernel thread, then typically that address would need to be flushed from all the TLBs in the system. But the SVR4.2 *segkmem* driver delays this shootdown until this address is about to be reused by the kernel. The virtual space managed by the *segkmem* driver is represented as a bitmap and the bitmap itself is divided into zones (the size of the zone is a tuneable; the default value is 16 bytes). Each zone has associated with it a *cookie* (explained in the previous section), which is set when an address in the zone is freed. At the time of allocation, when it is found that a page is allocated from a freed zone whose addresses have still not been flushed from the TLBs, *hat_shootdown()* is called with the *cookie* associated with the zone as an argument. What *hat_shootdown()* does with this *cookie* will be explained shortly.

Similarly, *segmap* manages its virtual space in fixed-size *chunks* (configured as 8K as the default value) and each *chunk* has an associated *cookie*. Unlike *segkmem*, however, when a *segmap chunk* is freed (last reference is released), the *cookie* for the chunk is set through the *hat_getshootcookie()* interface but the translations are not unloaded. Instead, this chunk is linked on to a list; the *segmap* aging daemon periodically looks at this list and unloads the translations at this time but does not perform a shootdown of the unloaded addresses. When the chunk is then reused by *segmap*, it calls *hat_shootdown* with the associated *cookie*. The shootdown can be delayed after the unloading since no other context can access this file page in the mean time. This technique allows us to eliminate the shootdown entirely, if the chunk is reused with the same identity (same physical pages) before it is aged.

Lazy Shootdown Algorithm

Inside the HAT layer, a *cookie* is associated with each processor that denotes when the processor's TLB was flushed last. In a separate global variable, the *cookie* of the least recently flushed TLB is maintained. If the *cookie* passed in to *hat_shootdown()* is older than this value, then it immediately returns since it knows that all the TLBs in the system have been flushed since the *cookie* was acquired. If this is not the case, the following steps are executed by the initiator (the context that is initiating the shootdown):

1. Acquires a global spin lock. This spin lock disallows the active processor set of the system from changing underneath. It also serializes lazy shootdowns in order to set the *cookie* for each processor.
2. Scans the list of all processors that have been *hat_online()*'ed (see interface definition in the last section) and selects all the processors whose *cookie* is "older" than the passed in *cookie*. While selecting the processors to be interrupted, it recomputes the least recently flushed value and sets the *cookie* for each processor it selects (to *lbolt* in our implementation).
3. Sends cross processor interrupts to the processors.
4. Unlocks the global spin lock it acquired earlier once all the responders have begun processing the interrupt.

The responders (processors at the receiving end of these interrupts) then flush their own TLBs before again becoming active. The cross processor interrupt executes at the highest interrupt priority level (ipl) in the system because no interrupts can be allowed while servicing a shootdown. Otherwise, this could result in a deadlock if the interrupt level routine causes a shootdown itself. This interrupt level is even higher than the normal "block-all" interrupts level (splhi) to avoid latency problems; we are careful to avoid changing anything in the cross-processor interrupt service routines which could interfere with splhi-protected critical regions. Note that the responders do not wait for any synchronization with the initiator in this algorithm. All they have to do is a TLB flush since the translations have been modified earlier by the segment drivers. The initiator does not wait for all the responders to complete their operation.

Immediate Shootdowns

The interfaces for immediate shootdowns employed for the user address space are hidden in the HAT layer and are not exported to other layers in VM. This is because immediate shootdowns are caused only by operations within the HAT layer such as unloading a translation and changing protections for a translation.

Immediate Shootdown Algorithm

The algorithm for immediate shootdown is similar to the lazy shootdown algorithm. The following steps are executed by the initiator:

1. Grab the same global spin lock that we acquire in the lazy algorithm for the same reason (to keep anybody else from changing the active processor set or performing another shootdown).
2. Send cross-processor interrupts to all the processors that share this address space (the processor list that is updated by *hat_asload* and *hat_asunload*). Unlike the lazy algorithm, the responders spin waiting on synchronization with the initiator.
3. Modify the page table entries (level 2 entries) as appropriate for the operation (zero page table entries if unloading translations, change the protection bit or clear the modifying bit if *sync*ing the page table entry to the page

structure).

4. Increment the counter that the responders are spinning on. The responders perform a TLB flush and return from the interrupt.
5. Perform a TLB flush for the initiator's processor.
6. Unlock the global spin lock acquired earlier. The initiator again – as in the lazy case – does not wait for the responders to finish flushing their TLBs.

This algorithm has been optimized for the i386 architecture when the initiator has to modify a large range of page table entries (example: when unloading a large range of addresses). The initiator holds the HAT resource lock (a spin lock) that is associated with the address space being modified at the outset of the algorithm. After the responders are in a spinning state, instead of changing all the page table entries the initiator just unloads the level 1 entries for the affected page tables. Thus, the initiator spends less time when all other processors are spinning. The initiator then increments the counter that releases the responders from spinning on the barrier. The responders then flush their TLB before returning from the interrupt. If any of the LWPs running on the responders try to access the inconsistent page table entry, it will take a fault because of the non-existent level 1 entry. The trap code will then try to acquire the HAT resource lock and will block until the initiator releases the HAT lock. This reduces the time processors spin uselessly in the shootdown algorithm.

Pageout

The implementation of local working set aging for pageout in SVR4.2 also prevents shootdowns when compared to the global pageout policy in SVR4. The global pageout daemon scans all the physical pages in the system and clears the modify bit if the bit is set for a page (after calling VOP_PUTPAGE() on the page) or clears the reference bit if it is set. Both of these actions would require shootdowns (since these bits are in the page table entry). But with the working set aging, the process to be *aged* is seized; i.e. all the LWPs in the process except the current context are brought to a quiescent state. Thus, there is no need to shootdown when modifying the page table entries. The i386 context switch code flushes the TLBs when switching back in these LWPs.

Other Architectures

The above mentioned interfaces and algorithms provide flexibility in supporting various architectures, requiring minimal changes to them.

Architectures supporting single TLB entry flushes:

- The lazy shootdown algorithm need not change at all. Even though the algorithm flushes the whole TLB, most shootdowns are totally avoided by this policy (see section "Performance Data") and thus result in very little overhead when compared to flushing individual entries.
- The immediate shootdown interfaces (both the initiator and the responder) would change to take in the address range as an argument and flush just those entries. Since these interfaces are not exported to other VM layers, changing the interfaces is acceptable.
- There may be a point in such architectures where flushing a whole TLB is cheaper if the number of lines to be flushed in the TLB is too large. The algorithms should take this into account when deciding which is more efficient.

Architectures whose TLB entries contain a field for context number:

- It is unnecessary to flush the TLB on context switches.
- The lazy shootdown algorithm would not change. If no other local TLB flushes are done by the kernel[8], all the *cookie*s associated with the processors would be in the same state and only one *cookie* would be needed.
- For the user address space, a lazy shootdown algorithm may be possible as implemented in [5].
- No changes to interfaces are necessary.

Locking Design

The locking design implemented for the VM subsystem as a whole should scale well on parallel activities (intra-process and inter-process) that occur on the system. The primary motive in arriving at the current locking model was to keep things simple and not to have the locking requirements between the VM layers (the page layer, the segment layer and the HAT layer) too complex. As a result, porting of this HAT layer to other architectures should be almost as straightforward as a Uniprocessor HAT layer.

The principal locks in the VM layer are:

- Page Layer
  - A global spin lock in the page layer for protecting the page hash chains
  - A per page spin lock for mutexing the fields of the page structure
  - A read/write sleep lock which is acquired in reader mode to ensure that the page state, identity and data are valid and remain so and acquired in writer mode if modifying any of the above.
- Segment Layer (user segment driver)
  - A reader/writer lock per segment. This lock is acquired in writer mode when changing the attributes of a segment

[8]This is not the case in SVR4.2.

(such as protection) and in reader mode when the attributes of the segment are to remain valid for the duration of operation.
  - A per segment spin lock which guards the sleep lock.
- HAT Layer
  - There is only one spin lock associated with each address space for guarding the HAT resources.

Making the HAT lock finer grained by moving it to the page table level was considered but decided it wouldn't be much of a gain for the following reasons: most UNIX processes fit in one page table and there would be extra locking round trips for HAT functions that cross page tables. If found necessary, other ports can move this to a page table level (architectures where the page table size is small) without any need to change the locking requirements.

Analysis of the Locking design

Two widely occurring system events in UNIX systems, page faults and fork()/exit() operation, would be a good indicator of scalability in the VM layer.

- When generating concurrent page faults in different address spaces, the only lock contention will be for the global page layer spin lock that is guarding the page hash chains. The lock hold time for this lock is very low. There will be different instances of the HAT lock (due to different address spaces) and segment locks (faulting on different segments).
- For concurrent page faults generated within a process among its LWPs, there would be contention for the HAT resource lock but the lock hold time during loading of a translation will again be very small. Faulting on the same segment by various LWPs would cause contention for the per segment sleep lock. Some faults require the lock to be held only in reader mode and thus allows for parallelism between such faults at the segment layer.
- When concurrent fork()/exit() operations take place in different address spaces, the only contention at the VM level would be for locks at the *anon* layer (which manages anonymous pages) and at the *swap* layer for reserving anon pages and swap space for the child processes respectively. Again, the lock hold times during the reservation operation would be very small.
- Intra-process concurrent fork()/exit() operations could cause lock contention at the HAT layer and the segment layer but both the locks will be held only while each segment is being copied. Reducing the lock hold time on the HAT resource lock by dropping the HAT lock after copying each *mapping chunk* (32 page table entries) is being considered.

Examples of the Locking requirements for HAT interfaces

To get an idea of the locking requirements, some of the HAT interfaces are listed here. In all these operations, the HAT resource lock is acquired by the HAT layer.

`hat_memload()`: Load a virtual address translation. Called with the reader/writer lock for the physical page held. The caller can not hold any spin locks.

`hat_unload()`: Unload a range of virtual address translations. The caller need not hold any spin locks. This routine acquires the spin lock associated with the physical page structure in order to modify the mapping chain for the page.

`hat_pageunload()`: Unload all the virtual translations to a given physical page. The spin lock for the page is held by the caller.

Physical Memory Scalability

SVR4 had a limit on the physical memory it was able to support on the i386 platform. Changes were made in SVR4.2 to avoid this limit[9]. Several kernel functions in SVR4 relied on the fact that all of physical memory in a machine is mapped into the kernel virtual space. These functions generally need to get a virtual address from a given physical address in a non-blocking fashion. On the Intel386 reference port, out of the available 4 Gigabyte virtual address space, the user address space was given 3 Gbytes and the kernel 1 Gbyte virtual space. The kernel virtual itself was divided at kernel boot time among different kernel segment drivers (*segkmem*, *segmap* and *segu* (which manages the simultaneous mapping of several processes' U areas at the same time)). After this division, only 256 Mbytes of physical memory could be mapped into the kernel virtual space. The default layouts of the kernel memory map could be changed to make this limit bigger but there would still be a limit. All the kernel functions that expect this non-blocking behaviour were modified in SVR4.2 to eliminate this restriction in one of the following two ways:

- Cache the needed virtual address.
- Create and destroy virtual mappings to a given physical page as needed.

The HAT layer was one of the primary users of this "magic mapping" in using it to get to the virtual address of a level 2 page table entry from the page frame number stored in the level 1 page table entry. It was changed to cache the virtual address in the HAT structure itself. Going into the details of all the changes in the kernel is beyond the scope of this paper.

[9]This decision was not influenced by any MP related issues.

Performance Data

Most features of SVR4.2 have been completed and performance measurements are beginning to be collected for the system. Thus the performance measurements presented here are by no means the optimal figures for SVR4.2.

Shootdown Measurements

Some measurements were taken on how well our shootdown algorithms (lazy and immediate) scale with respect to number of processors. Scalability of the basic cost of the shootdowns, the lazy shootdown algorithm and the immediate shootdowns algorithm were measured. The following measurement process was used in collecting the data:

- The measurements were taken on a Sequent Symmetry platform which has 6 Intel 386 processors at 20Mhz.
- Ten samples of each measurement were taken, and their mean value was used.
- A measurement tool called *casper* was used to measure the time spent in different windows of the kernel code paths in units of microseconds.
- All our measurements reflect the time spent by the initiator.

The time spent by the responders in the lazy algorithm would be a fixed time constant (time taken to flush its TLB). In the immediate shootdown case, the time spent by the responders would be upper bounded by the time spent by the initiator.

The basic cost of shootdowns was computed on the Symmetry by calling *hat_shootdown()* with HAT_NOCOOKIE as an argument, which shoots down all processors in the system without any other computation. Figure 1 is the graph that illustrates the results.

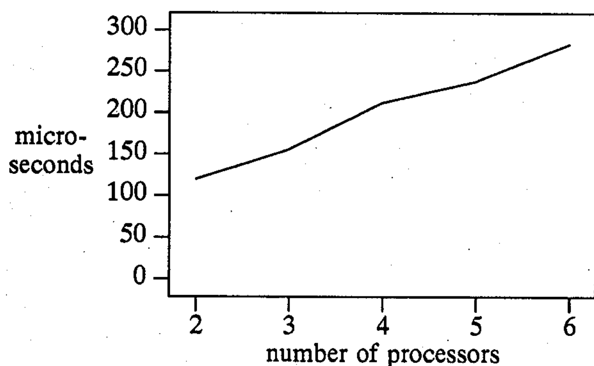

Figure 1: Basic cost of shootdowns

The graph shows that the measurements are not exactly linear. Two possible reasons for this is variances in the interrupt fanout facility on the Sequents and that even a slight disturbance in order of tens of microseonds for each collection of samples will perturb the linearity.

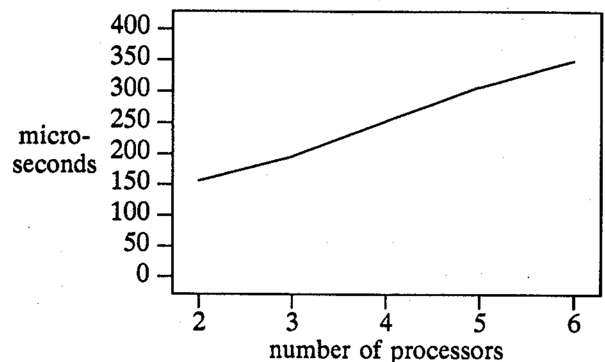

Figure 2: Cost of the Lazy shootdown algorithm (used for kernel address space)

The cost of the lazy shootdown algorithm is illustrated in Figure 2. Note that there is a fixed cost overhead over the basic shootdown cost in the range of 70 microseconds. The cost of the algorithm per additional processor is about 40 microseconds.

Measurements of the immediate shootdown algorithm were analyzed next. It was measured by running a kernel level test that handcrafted a user address space and spawned as many threads as the number of onlined processors. After the spawned threads waited spinning on a barrier, the parent unmapped a previously mapped page. This would generate a shootdown on all the other processors that were spinning on the barrier. Thus the initiator touches only one page table entry.

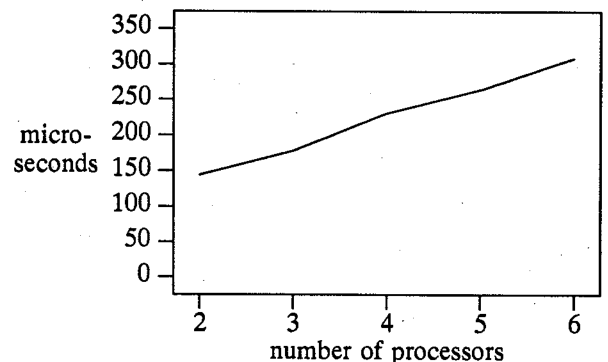

Figure 3: Cost of Immediate shootdown algorithm (used for user address space)

The overhead of the algorithm over the basic shootdown (Fig. 3) cost is about 30 microseconds. The fixed cost of each additional processor for the immediate shootdown algorithm is about 45 microseconds.

The data collected so far indicates that we should be able to scale well in the range of tens of processors for both the lazy shootdowns and immediate shootdowns. The cost of the lazy shootdown algorithm is slightly (about 40 microseconds) more than the immediate shootdown. This is due to all the accounting that is done to update the *cookie* for each

TLB in the lazy shootdown case. Other similar measurements [3] show that the bus contention may become a problem for algorithms that use cross processor interrupts when it deals with processors in the range of 15 - 20.

Another encouraging measurement about the effectiveness of the lazy unload policy of the *segkmem* driver (discussed above) shows that it makes only 1.1 calls to *hat_shootdown* per 100 memory allocation requests. Actual shootdowns will be even less frequent (as can be observed from the explanation of the algorithm). This data was collected by running a kernel level test that allocates and frees kernel memory repeatedly in different sizes. There was no other activity (such as the pageout daemon) in the system when this test was run.

Concurrent fork()/exec()/exit() measurements

Scalability of concurrent inter-address space *fork()/exec()/exit()* operations were measured through a benchmark program[10]. The benchmark consists of the following tests:

- *fork()/exit()* operations with *bss* size ranging from 0 to 192K.
- *fork()/exec()/exit()* operations with *bss* size ranging from 0 to 48K.
- *fork()/sbrk()/exit()* operations with *sbrk* size ranging from 0 to 192K.

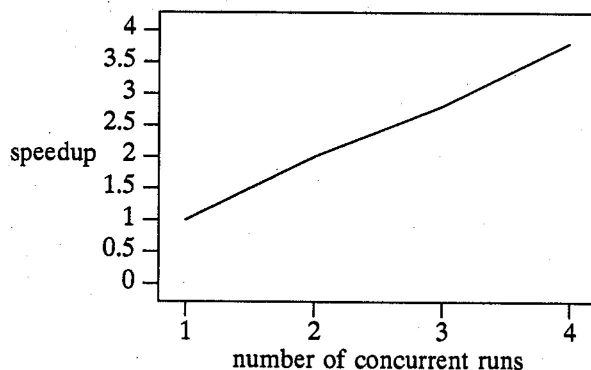

Figure 4: Scalability of fork()/exec()/exit() operations

The measurement process was as follows:

1. Scalability was measured by having a fixed processor configuration (4 processors) and varying the workload of the tests in the benchmark. The workload was varied by executing the same benchmark concurrently from 1 up to 4 times. The speedup was measured by the elapsed time of each work load and measuring it against the unit workload (one run of the benchmark).
2. Each of the above test was repeated for 10 times in a run.
3. The measurement was done on a 4 processor (Intel386 @ 20MHz) Sequent Symmetry machine.

As mentioned earlier, the measurements data (Figure 4) is used only to illustrate the scalability of SVR4.2 and should not be taken as the final performance data of the system.

[10] The benchmark program called S is one of the benchmarks used at USL.

Conclusions

A model of the HAT layer that is scalable with respect to processors and memory has been described in this paper. This model makes the porting effort simpler without losing sight of the scalability issues. A well defined multiprocessor management interface between the machine independent and the MMU dependent part of Virtual Memory subsystem and simple locking guidelines provide the keys in making a memory management feature portable. The TLB shootdown policy and algorithms in SVR4.2 adapt well to different architectures. With multiprocessor platforms becoming more common, preserving the ease of porting a kernel to different architectures without losing sight of scalability issues will be extremely critical.

Acknowledgements

The design and implementation of the VM subsystem was a joint effort by the SVR4.2 VM team members. The past and present members of this team include: Steve Baumel (also provided *segkmem* measurements), K. Doshi, Mike Lazar of Pyramid Technologies, Joe Lee, and Dave Lennert of Sequent Computer Systems. Dick Menninger was the initial implementor of the SVR4.2 HAT layer. Thanks also to Mike Miracle for his support in writing this paper.

References

[1] Richard Rashid et al., *Machine-Independent Virtual memory management for Paged Uniprocessor and Multiprocessor Architectures*, in IEEE Transactions of Computers, Vol.37, No.8, August 1988."

[2] Robert A. Gingell, Joseph P. Moran and William A. Shannon, *Virtual Memory Architecture in SunOS*, in Proc. USENIX Summer '87 Conference, Phoenix, AR, June 1987.

[3] Black, et. al., *Translation Lookaside Buffer Consistency: A software Approach*, December 1988, CMU-CS-88-201.

[4] Patricia J. Teller, *Translation-Lookaside Buffer Consistency*, June 1990, IEEE COMPUTER.

[5] Michael Y. Thompson et al., *Translation Lookaside Buffer Synchronization in a Multiprocessor System*, USENIX Winter Conference 1988.

[6] 80386 Programmer's Reference Manual

Author Information

Ramesh Balan is a Member of Technical Staff at UNIX System Laboratories in the Kernel Development group. He received his M.S. in Computer Science in 1989 from the school of Engineering and Applied Sciences in Columbia University. He can be reached via e-mail at ramesh@usl.com.

Kurt Gollhardt is a consultant at UNIX System Laboratories in the Kernel Development group. He received a B.S. in Computer Science and a B.S. in Electrical Engineering at Washington University in St.Louis. He can be reached via e-mail at kdg@usl.com.