To the light of extinguished STAR

VAX MP
A multiprocessor VAX simulator

Technical Overview

Sergey Oboguev
oboguev@yahoo.com

Initial edition: November 21, 2012
Last update: December 31, 2012 (rev. 3)

Content and intended audience:

This document describes internal design of VAX MP multiprocessor simulator.

It is intended for VAX MP developers or those interested in VAX MP internal architecture, not for end users. End users should refer to "VAX MP OpenVMS User Manual".

This document discusses VAX MP design principles and decisions, key problems met during VAX MP design and implementation, their possible solutions and rationales for chosen and rejected solutions.

The document does *not* attempt to describe VAX MP internal interfaces, data structures or other internal details, since that would have required a much longer document. Only a brief description is made of some basic interfaces introduced in the portability layer and key interfaces changed compared to the original uniprocessor SIMH, as well as interfaces relevant for modifying SIMH virtual device handlers for multiprocessing environment.

Introduction

VAX MP is a derivative variant of popular SIMH simulator[1] that extends original SIMH to simulate multiprocessor VAX system. The intent of VAX MP is to run OpenVMS and perhaps in the future also BSD Unix in SMP (symmetric shared-memory multiprocessor) mode.

In technical parlance this is also known as SMP virtualization.

VAX MP targets primarily hobbyist use of OpenVMS and possibly BSD Unix for the purposes of retro-computing, i.e. preservation of computer history and the past of computer and software technology.

While the intent of VAX MP is to provide reasonably stable SMP VAX system simulation environment for hobbyist purposes, multiprocessing poses unique challenges to quality assurance, as explained below – challenges that go well beyond the resources available within a single-developer free-time non-commercial project. Therefore VAX MP is not recommended for production use. If you are looking for a production solution, please examine commercial options, such as VAX simulator from Migration Specialties or Charon-VAX/nuVAX product ranges by Stromasys.[2]

* * *

There are two conceivable high-level approaches to simulating a multiprocessor.

One straightforward approach is for the simulator to use a single thread of execution that, during each single cycle of execution, loops across all active virtual processors executing one instruction at a time from every virtual processor's instruction stream (skipping VCPUs marked idle) and thus automatically keeps VCPUs in a lockstep. This approach may have advantages for the purposes of system testing during its development[3], but is less attractive if the primary interest is the use of legacy software, such as the exactly case with retro-computing. Single-threaded approach avoids many problems of multi-threaded approach described below, but at the cost of mostly defeating the purpose of multiprocessing.

[1] This document does not make semantic distinction between terms *simulator* and *emulator* and uses them interchangeably, as is the common practice in the recent years, save for terminological purists.

[2] In addition to increased stability, these commercial products offer improved performance, since unlike SIMH which is an instruction-level interpreter, they perform JIT dynamic binary translation. As ballpark figures, whereas single-CPU version of VAX MP executes on Intel i7 3.2 GHz based PC at about 30-40 VUPS, commercial version of Charon-VAX simulating single-processor MicroVAX according to the manufacturer executes at 125 VUPS, and Charon-VAX/66x0 with six processors delivers up to 700 VUPS. Some of the mentioned commercial products also offer increased memory size beyond virtual MicroVAX 3900 limit of 512 MB (for example Charon-VAX 66x0 simulator offers up to 3 GB), making them altogether more suitable for production use.

[3] Or rather, instruction-level simulator is just one of possible testing simulators. It may simulate real-world computer only to an extent: for example, clock drifts may be simulated but only somewhat, whereas inter-cache interaction would be much more difficult to simulate and would probably require register-transfer-level or logical-block-level or even gate-level simulator.

Another approach, one that VAX MP takes, is to simulate virtual multiprocessor on top of a real hardware multiprocessor. The advent of on-chip multiprocessors readily available in consumer PCs makes it an attractive solution for hobbyist use.

The targets of VAX MP project are:

- Primary and initial targeted guest operating system is OpenVMS, not only because it is the most venerable system associated with VAX, but also the only currently available operating system that supports VAX multiprocessors. (Save Ultrix, which we do not target as having much lower historical interest and uniqueness.)

- It might also be possible to add in the future support for OpenBSD and/or NetBSD Unix. These operating systems do support VAX and they do support some multiprocessor platforms, but they do not support multiprocessing on VAX platform, hence they would have to be updated to add multiprocessing support for VAX to them, which obviously would be *Das Glasperlenspiel* of the second order, hence the possibility of adding support for these systems is kept in mind but they are not targeted by the initial release of VAX MP which keeps focused solely on VMS. Thus the discussion in this document will be somewhat VMS-centric, however much of VMS-related notes do also apply to OpenBSD and/or NetBSD as well should they ever be updated to support multiprocessing on VAX.

- No support for Ultrix SMP is planned for VAX MP at this point, owing to the difficulties obtaining Ultrix source code and listings and also very low expected level of interest for running Ultrix in SMP mode. However, just like all other VAX operating systems, it is possible to boot and run Ultrix on VAX MP in uniprocessor mode.

- Host systems targeted by general design of VAX MP are, theoretically, any cache-coherent SMP systems[4] with no or low NUMA factors. VAX/VMS and VAX version of OpenVMS assume flat uniform symmetric memory access for all the processors and do not accommodate NUMA factors. Any significant NUMA factors would cause inefficient process-to-resource allocation within the guest operating system and, most importantly, jeopardize guest operating system stability; for this reason host systems with significant NUMA factors are not supported. However it might be possible to run VAX MP on high-NUMA system as long as VAX MP is allocated to a partition with low intra-region NUMA factor.

  It might still very well be that virtualized OpenVMS will run fine (perhaps somewhat sub-efficiently, but stably) on a multiprocessing host system with moderate NUMA factors, however this aspect had not been thought through for the initial VAX MP release and the author does not have NUMA system in his possession available for testing. Should the need arise, the issue of

[4] Host systems with no cache coherency are not targeted, as they are unable to virtualize existing guest operating systems.

adaptability of VAX MP to NUMA configurations may be revisited later.

- On a related matter, 2-way hyper-threaded processors are supported for the host system. The effect of 2-way hyper-threading is that virtual processor can execute quite a bit slower when the physical core it is mapped to is shared by two threads, compared to when the virtual CPU's thread has the core exclusively to itself.[5]

  The accommodation of temporal variance in VCPU speed induced by hyper-threading is made possible by two factors:

  - If hyper-threaded processors are detected on the host system, VAX MP paravirtualization layer adjusts operating system's busy-wait loop calibration counters increasing them approximately two-fold (more precisely, by 1.8).[6] This effectively increases sanity intervals for spinlock acquisition, for inter-processor request-response time and (less important) for virtual device interaction delays in the kernel to offset any possible CPU speed variability.[7]

  - Default values of VAX/VMS (OpenVMS) SMP sanity timers have large safety reserve margin to them, about 10x even on a processor with 1 VUPS performance, and effectively even larger on more powerful processors, as would be for today's hosting environments.[8] This creates ample safety reserve against false premature expiration of SMP sanity timers.

[5] Let us say, perhaps about 1.7 times slower as a ballpark figure, albeit the actual number can vary widely, depending on the behavior of the second thread executed by the hyper-threaded processor. The slowdown is caused by the threads competing for execution units, cache space and memory access bandwidth of the CPU core.

[6] For VMS these are kernel variables EXE$GL_TENUSEC, EXE$GL_UBDELAY and their shadow copies CPU$L_TENUSEC and CPU$L_UBDELAY in the per-CPU database.

[7] If host system is itself virtualized, such as Windows or Linux ran inside VMWare or other virtual machine monitor that conceals underlying physical topology from virtualized host machine, this VM monitor may fail to properly expose physical processor topology and packaging to guest OS, and hence to VAX MP running on top of that virtual OS instance, and falsely represent all of its virtual processors as independent cores, whereas in fact they are implemented as threads executing on hyper-threaded cores. Such a failure to convey information about processor topology might negatively affect VAX MP stability and reduce guest OS safety margin. When running such a configuration, it is therefore important to verify that SMT slow-down factor printed by VAX MP at startup matches actual host system processor topology. For 2-way SMT, SMT slow down factor should be about 1.8. If printed slow-down factor mismatches actual host topology, it should be adjusted manually via VAX MP SIMH console command

```
   CPU SMT <slow-down-factor>
```

[8] Single-CPU VAX performance of VAX MP running on the machine with Intel i7 3.2 GHz CPU is about 30-40 VUPS, whereas multiprocessor performance with cores hyperthreaded would be on the order of 15 - 25 VUPS per VCPU.

Taken together, these two factors should provide sufficient safety margins against variability in VCPU speed that may be observed on 2-way Hyperthreaded/SMT system.

However for 4-way and higher concurrency order hyper-threaded/SMT systems the issue becomes much more acute, therefore such systems are not supported by the initial release of VAX MP, and any future possibility of their support should be revisited later.[9]

- VAX MP expects host system to provide one of universal[10] synchronization primitives, such as CAS or LL/SC or (if perhaps available in the future) transactional memory. Host systems that provide only XCHG primitive may be supported, but with some limitations.[11] Host system is also expected to provide cache coherency and memory fence facilities.

- The initial release of VAX MP targets only x86 and x64 multiprocessors as supported host hardware architecture. Targeted host operating systems are Windows (32-bit and 64-bit), Linux (32-bit and 64-bit) and OS X (64-bit), with possible later addition of support for Solaris or FreeBSD and/or OpenBSD and/or NetBSD.

When simulating a multiprocessor on top of another multiprocessor, it is necessary to define two key mappings: (1) a mapping for execution model, i.e. mapping of virtual CPUs to host logical or physical CPUs and (2) mapping for memory-consistency model, as described in detail in subsequent chapters. These two mappings, especially memory-consistency model mapping, make multiprocessor simulator and its porting very much unlike the development and porting of uniprocessor system simulator.

For one thing, when porting simulator to yet another host platform, memory-consistency model has to be thoroughly re-examined from scratch. Simulating multiprocessor system with stronger memory model on top of host system with weaker memory model may present considerable challenges and sometimes be impossible altogether. While VAX MP tries to provide generic enough design seeking to accommodate a range of possible host systems, porting VAX MP to new host hardware architecture cannot be automatic and would have to involve thorough re-examination of memory consistency model

[9] Current version of VAX MP will by default issue error message and reject an attempt to configure multiple virtual VAX processors when ran on a host system with more than 2-way SMT/Hyper-Threading. This behavior can be overridden with `CPU SMT <slow-down-factor>` command described above, with understanding that system may not run stably.

[10]Maurice Herlihy, “Wait-Free Synchronization”, ACM Transactions on Programming Languages and Systems, Jan 1991; see also Michael & Scott, “Scalability of Atomic Primitives on Distributed Shared Memory Multiprocessors”, University of Rochester, Tech. Report UR CSD / TR528, 1994; also Maurice Herlihy, Nir Shavit, “The Art of Multiprocessor Programming”, Elsevir, 2008, chapter 6 (“Universality of Consensus”).

[11] One limitation would be a somewhat decreased efficiency of the locks and poorer concurrency characteristics. Another limitation is that 32-bit processors that support only XCHG instructions would have no way to efficiently generate system-wide sequence stamps for VAX instruction history trace buffer, hence instruction recording for debugging purposes would have to be limited to per-processor recording, without synchronizing per-CPU histories into a global instruction stream, since it would be impractical to implement the later due to very high level of contention.

mapping. This also includes mapping of atomic data types, use of VAX interlocked instructions, host-specific interlocked instructions and proper memory barriers.

Furthermore, simulator can provide proper mapping only for *systemic* features of VAX memory-consistency model. It is entirely possible that actual VMS code can sometimes go beyond the guarantees and constraints of VAX Architecture Standard memory-consistency model specification and implicitly rely on memory-coherency features of particular historically existing VAX hardware models that go beyond generic VAX architecture memory-consistency model guarantees. This, strictly speaking, would represent a bug (or, rather, over-stepping of the VAX Architecture Standard specification and good coding practice) – however a bug that lays dormant when VMS is executed on actual VAX machines or single-processor simulators and gets triggered only when VMS is ran on the multiprocessor simulator executed by host machine with weaker memory-consistency model than VAX historical models. Such use was not the intention of VMS developers and certainly was not covered by VMS release testing. It is therefore entirely possible that existing OpenVMS VAX codebase may contain such dormant bugs that will manifest themselves only on a multiprocessor simulator. Should such bugs get triggered, they would manifest themselves in obscure, very hard to debug system crashes (or, worse, data corruptions) occurring far away from the place of the bug origin and leaving no traces in the system dump[12]. Some of these crashes can be tracked down and fixed by patching or dyn-patching[13] the running system before activating SMP, whereas some possibly existing bugs can be found only by VMS code review – however general VMS code review or extensive QA testing with proper coverage are well beyond the resources of this single-developer non-commercial hobbyist project.

This makes one reason why virtualization of OpenVMS VAX SMP system on top of non-VAX host multiprocessor is not only non-trivial, but even risky and essentially unpredictable undertaking with no guaranteed success within constrained resources.

For the same reason, successful execution of OpenVMS on VAX MP running on x86/x64 multiprocessor does not mean it will be able to execute equally well on VAX MP ported to a host multiprocessing system with weaker memory model. Possible VMS memory-coherency related bugs not triggered on x86/x64 may get triggered on host system with weaker memory model.

If this did not scare you away yet, read on.

[12] Since by the time the dump is taken, memory change that did not reach target processor earlier would most likely already propagate through inter-processor cache synchronization and therefore will be reflected in the dump – which, thus, would not represent the processor's view of the memory at the time the crash was triggered, or may represent a view of another processor. Furthermore, in OpenVMS case the dump is performed by the primary processor, rather than the processor that originally triggered the bugcheck.

[13] Hereafter we use term *dynamic patching (dyn-patching)* for a technique that installs a patch in memory-resident copy of the operating system after it is loaded, without modifying operating system files on the system disk.

Overview

The intent of this version of SIMH is to provide the simulator for multiprocessor VAX system capable to run OpenVMS and VAX/VMS and perhaps in the future also BSD Unix in SMP (symmetric shared-memory multiprocessor) mode, with virtual processors mapped to the threads executed in parallel by the hosting environment that can leverage multiprocessing capability of the hosting system.

There are two possible general approaches to implementing VAX multiprocessor simulator on top of host multiprocessor. These approaches are not mutually exclusive and the changes required to the original uniprocessor SIMH for both approaches do overlap, to an extent.

The first approach would be to provide a veritable implementation of historically existing genuine DEC VAX SMP system natively recognized by OpenVMS as capable of SMP, such as VAXstation 3540 or VAX 6000 or 7000 series. The advantage of this approach is that OpenVMS would natively run on such a simulator "out of the box", without a need for any additional operating system level software. This approach however would require implementation of additional virtual buses such as MBus or VAXBI or XMI and various other hardware devices in the simulator, including buses, adapters and variety of IO devices, such as disk controllers, Ethernet cards etc. This would require hard-to-obtain technical documentation for legacy systems and also console firmware[14] and much of the focus would be on the implementation of these proprietary buses and IO devices and away from the SMP capability per se. Another potentially difficulty is that VMS might contain (as it does indeed) pieces of code that rely on strong memory-consistency model of historically existing VAX models, beyond what is guaranteed by VAX Architecture Standard, that cannot be properly simulated on host machines with weaker memory-consistency model without the loss of efficiency. Addressing this issue efficiently would require patches or dyn-patches to VMS code and thus negate "out-of-the box" approach.

The second approach, one that the current implementation of VAX MP takes, is a shortcut. It does not provide implementation of genuine DEC VAX SMP hardware system that would be natively recognized by OpenVMS. When OpenVMS is booted on VAX MP, VMS recognizes the simulator as MicroVAX 3900, a single-processor computer. However OpenVMS SYSGEN parameter MULTIPROCESSING is changed from default value of 3 to either 2 or 4, causing OpenVMS SYSBOOT to load multiprocessing version of the kernel. Once the system is started, we install and run inside OpenVMS a custom module that overrides a number of MicroVAX 3900 processor-specific routines with VAX MP specific SMP-capable versions, dynamically adds extra processor descriptors to the running system kernel and provides OpenVMS with a way to start and manage these virtual processors by communicating with the simulator[15]. Once created, these processors are then started at OpenVMS operating system level with the regular START

[14] Alternatively, unavailable documentation for the proprietary buses can be substituted, at least partially, by looking into VMS source code and/or listings and trying to understand the interaction between VMS and bus/adapter hardware.

[15] Communication of VAX-level code with the simulator is implemented via MTPR/MFPR to a special virtual privileged register. It would have been also possible to implement it via XFC instruction and a number of other signals.

CPU command, which in turn invokes overriden process-specific routines, but in a fashion transparent for the rest of the system and the user. The result is virtual MicroVAX 3900 with multiple processors running in an SMP configuration. The advantages of this approach are:

- Implementation of additional hardware (buses and devices) is not required.

- Number of configurable virtual processors is not limited by configuration of any historically existing VAX model, i.e. one can go beyond the physical limit of 4 or 6 processors of historically existing VAX systems to a theoretical maximum of 32 processors supported by OpenVMS – as long as host system provides required number of logical execution units and satisfactory scalability for memory throughput.

Even if one were to consider the former approach (i.e. simulating existing VAX SMP model with precise implementation of required buses) as more desirable, the second approach still has a value as essentially a necessary predecessor to the first approach because it performs groundwork on internal parallelization of SIMH with multiple virtual processors and multiple threads of execution, which is a required precondition for the first approach as well. Thus, the second approach is a stepping stone toward the first approach, and the first approach can subsequently be implemented on the top of SIMH code changes introduced by the second approach which we take here.

Therefore, under the chosen approach VAX MP does not simulate any historically existing VAX multiprocessor; instead VAX MP simulates a multiprocessor variant of MicroVAX 3900 or VAXserver 3900 – a machine that never historically existed. The purpose of VAX MP is not a simulation of specific historically existing multiprocessor hardware, but of general VAX multiprocessing architecture, and execution of OpenVMS in multiprocessor mode.

VAX MP is composed of two essential pieces. One part is modified version of SIMH VAX simulator executable that supports parallel execution of multiple virtual processors. The other part is paravirtualization module (called VSMP) that is installed inside OpenVMS.

VSMP is chiefly required because VMS does not natively recognize VAX MP as capable of multiprocessing. When OpenVMS boots on VAX MP, OpenVMS initially recognizes the machine as uniprocessor MicroVAX 3900. However OpenVMS SYSGEN parameter MULTIPROCESSING is used to force-load multiprocessing version of OpenVMS kernel mage regardless of being started on a virtual hardware that is initially recognized at boot time as uniprocessor. Still, even with OpenVMS multiprocessing-capable kernel loaded OpenVMS does not natively know how to perform basic multiprocessing functions on VAX MP, such as starting or stopping processors or sending interprocessor interrupts, since MicroVAX 3900 does not have such capabilities. Such knowledge has to be provided to OpenVMS by VSMP. VSMP loads VAX MP proprietory code into VMS kernel. This kernel-resident VSMP module takes over VMS processor-specific CPULOA (SYSLOA650) routines and thus provides OpenVMS with knowledge how to operate underlying virtual hardware as a multiprocessor.

During OpenVMS boot sequence or, at user's discretion, at any time after boot, VSMP is loaded into OpenVMS kernel and exposes VAX MP multiprocessing capability to OpenVMS. Once VSMP is loaded, it overrides CPULOA routines and reconfigures the system to multiprocessor by creating additional CPU descriptors inside VMS kernel and exposing VAX MP multiprocessing capability to VMS. After this point, system can be operated in regular way just as any VMS multiprocessor. CPUs created by VSMP are initially in inactive state, so the first step would be start them with DCL command START /CPU.[16]

In addition to this, VSMP LOAD command also dynamically modifies memory-resident copy of OpenVMS kernel and some drivers loaded into main memory by installing about 30 patches to the running, in-memory copy of VMS kernel and some VMS drivers. These patches are required to support multiprocessing.

There is a number of technical reasons for VSMP dynamic patches:

> Most of these patches are intended to address modern host machines' weaker memory-consistency model compared to VAX memory model. In VAX days there were no out-of-order instruction issuing, speculative execution, write/read combining or collapsing, asynchronous multibank caches and other high-performance designs that can reorder memory access and affect the order in which memory changes performed by one processor become visible on another. These differences do not matter as long as VAX code uses proper synchronization, such as spinlocks or interlocked instructions to communicate between parallel branches of the code, since VAX MP will issue appropriate memory barriers once VAX interlocked instruction is executed. However there are pieces of VMS code that do not issue appropriate memory barriers by executing interlocked instructions. Most importantly, these are PU and XQ drivers (drivers for UQSSP MSCP disk/tape controllers and Ethernet DEQNA/DELQA controllers) that communicate with their devices not only through CSRs but also by reading from and writing to shared control memory area – control area shared by the device and CPUs and read from and written to asynchronously by both the device and CPUs. PU and XQ driver do not (most of the time) issue memory barriers when accessing this memory area, relying instead on stronger memory-consistency model of historically existing VAX implementations that provide ordering of reads and writes to memory in the same order they were issued in VAX instruction stream. When executed on top modern host CPUs, such VAX code would not work properly and reliably since underlying memory access can be reordered by the host CPU at will, unless memory barriers are issued – and such barriers are missing in VMS PU and XQ code. Therefore VSMP installs patches in PU and XQ drivers to fix their memory accesses to the control memory regions shared by the devices and CPUs, by providing required memory barriers that force proper ordering of memory reads and writes to/from these regions.
>
> Some other patches deal with the fact that VAX MP VCPU threads are idleable. On a real VAX, when VMS is in idle loop, CPU will spin in idle loop executing it over and over again, until it finds that some work to do had arrived. On VAX MP, VCPU threads do not have to spin in this situation, consuming host processor resources for nothing. When VMS enters idle loop on a given VAX MP VCPU, this VCPU's thread instead of spinning in the loop can go into hibernated state on host OS so as not to consume host processor resources while idle. Some of the patches installed by VSMP are intended to support this capability,

[16] More precisely, at load time VSMP modifies only a mask of configured processors but does not create per-CPU database structures in OpenVMS kernel. Per-CPU database for a processor is created on demand when the processor is first started with START CPU command. START CPU invokes loadable processor-specific routine which is re-vectored by VSMP to the routine supplied by VSMP, and the latter creates per-CPU database entry if it does not exist yet (i.e. processor is being started for the first time) and then invokes SIMH layer to start the processor.

including both going into idle sleep and then, on scheduler events, waking only the number of VCPUs actually required for pending processes in computable (COM) state. When VMS scheduler detects there are some processes in computable state, it will normally “kick” all of the idling CPUs off the idle loop and let them reschedule themselves. VSMP install patches that will wake up not all of the idling VCPUs but only actually required number.

One of the patches modifies XDELTA system debugger (if loaded) to let it properly handle multiprocessing on VAX 3900. XDELTA has CPU-specific code that gets current processor's CPU ID, and since it sees VAX MP as VAX 3900, it assumes the machine is uniprocessor and always uses CPU ID #0. The patch modifies this code in XDELTA to actually read from CPU ID register.

One of VSMP patches fixes bug in SYS$NUMTIM system service where system time location EXE$GQ_SYSTIME is accessed without properly acquiring HWCLK spinlock, which may result in incorrect value being read, including totally corrupt value. This can happen even on a real VAX, but probability of this corruption increases when VAX processor is simulated on present-day host CPUs with weaker memory-consistency model, compared to the original VAX. Patch installed by VSMP provides appropriate spinlock acquisition by SYS$NUMTIM when reading system time variable.

VSMP does not change any files on OpenVMS system disk to implement described patches, it changes only in-memory copy of the loaded system. Thus, other than just installing VSMP on system disk in a separate directory, no modifications to OpenVMS system disk are required for VAX MP. OpenVMS system disk image remains intact, processor-independent and OpenVMS system files are not modified for VAX MP. Only in-memory copy of OpenVMS is patched by VSMP. Hereafter, these in-memory modifications are called “dynamic patches”, to oppose them to static patches of on-disk files that VAX MP does not perform.

Note that the critical importance of some of the listed patches for proper operation of multiprocessor system, such as for example patches for storage and Ethernet controllers that use shared-memory regions to communicate with the CPU, would create a challenge to an approach trying to run unmodified OpenVMS “out of the box”, without paravirtualization layer or patches, on a simulator emulating historically existing VAX multiprocessor model. Other than injecting corrections into the drivers that implement proper memory fences, an alternative approach that does not require modification of driver code would be to monitor every reference to device/CPU shared memory area and to execute memory fence on each access of the CPU to the area. This is a possible approach (and it also has a benefit of better OpenVMS version-independence), however such reference monitoring would need to be executed on every memory reference by the CPU and induce a penalty. Nevertheless it is still a viable approach if a bitmap of “hot” memory pages is used and main memory access path includes only a quick check against this bitmap, bypassing further detailed checks if the page is not marked as “hot”.[17]

[17] It might be possible that the issue is obviated on the systems that use “mailbox” interface to reference devices. Unfortunately, we do not currently have technical manuals for XMI peripherals (KDM, DEMNA, KZMSA) that would allow to assess whether “mailbox” interface provides complete insulation for XMI controllers or the problem of synchronizaton of memory transactions to shared memory areas still persists even under the “mailbox” interface.

* * *

With chosen VAX MP basic approach, maximum number of configurable virtual CPUs is limited by the number of host logical CPUs, including hyperthreaded/SMT units. For example, commodity quad-core i7 based host machine allows to configure and use up to 8 VAX MP virtual CPUs. It is not possible to run more VCPUs than the number of LCPUs available on the host machine.

Maximum number of configurable VAX MP virtual CPUs is also limited by VMS maximum supported limit of 32 processors. VAX MP had been successfully tested in 32-processor configuration on host machine (a node in Amazon EC2 cloud) with two 8-core Xeon sockets, for a total of 16 cores, each multithreaded, yielding a total of 32 LCPUs, which 32 VAX VCPUs are mapped to.

While, abstractly speaking, the number of simulated virtual processors in virtual SMP system does not necessarily has to be bound by the number of physical processors of the hosting system, however:

- In all-important OpenVMS case the number of configured and active virtual processors[18] should never exceed the number of logical execution units available on the host system. OpenVMS contains sections of code (such as for example code that invalidates TLB entries for system pages) that require simultaneous execution and interaction of all virtual processors. If all of the virtual processors are unable to execute simultaneously, the system as a whole will be unable to make forward progress and will appear hung.

- In any event, regardless of the guest OS, performance can be expected to suffer significantly (latencies increase) and synchronization and context switching overhead (wasted CPU time) rise sharply if the number of configured virtual processors exceeds the number of host system physical processors available for use, i.e. if the host system is overcommitted.

It is therefore not recommended (and in case of OpenVMS as guest OS is impossible, and this is likely to apply to other guest OS'es to) to configure the number of virtual processors (VCPUs) beyond the number of host logical processors (LCPUs) available for the execution of the simulator. This is discussed in more details in subsequent chapters.

In case multiple instances of the simulator are executed on the same host machine, combined number of VCPUs in all virtual multprocessor instances should not exceed the number of available host LCPUs.

We also did not try to research OpenVMS XMI device drivers source code to glean this information from the XMI device drivers codebase.

[18] More precisely, the number of virtual processors in OpenVMS active set, i.e. processors started with START CPU command and active simultaneously at any given time, should not exceed the number of host LCPUs. The number of configured but non-started (or stopped) VCPUs processors is irrelevant.

Supported host platforms

While the design of VAX MP is generic, initial release implements support for x86 and x64 hardware architectures, for Windows , Linux and OS X only. Support for other host operating systems and hardware architectures can be added with ease within the existing framework, as long as the host platform has required capabilities.

Two categories of platform-specific features would need to be chiefly taken into account when adding a support for another platform.

Specificity of host hardware architecture is defined by its memory consistency model, including cache coherence model, memory reference reordering permitted, atomicity of basic data type read and write operations, instructions for memory barriers and interlocked instructions available.

Specificity of host operating system involves threading API, memory barrier primitives and additional interlocked instructions primitives.

These aspects are discussed in detail in subsequent chapters.

As for the threading APIs, initial release of VAX MP provides support for WIN32 threads and PTHREADS. When making use of those we decided to utilize only basic features, partially to avoid an impediment for porting, but also due to other considerations:

For example, VAX MP implements proprietory version of basic locking primitive partially for portability, but also because existing host implementations do not provide locking with desired semantics:

- Composite lock that begins by spinwait (in the assumption that lock will be typically held by the holder for only very short time, much less than the cost of userland/kernel roundtrip, rescheduling and thread context switch), but after certain number of busy-wait cycles escalates to OS-level blocking wait.

  Spinwait interval can also be calibrated in microseconds, in addition to the number of busy-wait cycles.

- Gathering statistics, such as total number of lock acquisitions, percentage of no-wait acquisitions, percentage of acquisitions causing OS-level blocking and also average number of busy-wait cycles before acquisition happened in non-blocking case. Collected statistics can be used for monitoring and tuning the system.

- Support for nested locking.

- Optional thread priority elevation when acquiring the lock and thread priority demotion when releasing the lock and no other locks with priority protection are held by the thread and there are no other intra-application causes to keep the thread priority elevated.

- Although VAX MP does not currently implement self-learning locks that dynamically adjust the number of busy-wait cycles the lock primitive tries to spin before resorting to OS-level blocking, such a functionality was envisioned as possible future improvement, should it judged to be beneficial, and the algorithm is described in the comments to `smp_lock` primitive source code and also in chapter "Possible improvements for subsequent VAX MP versions" of the present document.

None of the described features would have been possible with host platform stock primitives, across the targeted platforms.

Likewise, when making use of PTHREADS, we decided to utilize only basic features and specifically abstain from relying on pthreads priority ceiling mechanism for thread priority control, because:

- it is optional and likely not be present in some pthreads implementations;

- where it is present, implementation of pthreads priority ceiling may be implemented in user space rather than in kernel, still having to perform context switch, and thus have no benefit over application-level thread priority control;

- relying on priority management by pthreads would have create VAX MP architectural dependency on pthreads and hence the need to construct a separate and parallel platform-specific logics for host platforms that do not use pthreads, such as Windows or pthreads implementations that do not support priority ceiling; as a result, not only low-level primitives, but large-scale structure of the simulator would have become platform-specific;

- pthreads does not provide a way to integrate application state (other than locks held) into thread priority management;

- pthreads does not provide a way to integrate non-pthreads synchronization primitives (such as custom adaptive critical section used by VAX MP) into thread priority management;

- Linux and Unix priority space is segmented into real-time priorities and timesharing priorities ("nice" value) managed by separate APIs without a way to uniformly specify a priority across the two spaces; pthreads priority mechanism works only with real-time priorities, there is no way to specify timesharing-space priority;

- pthreads implementations are likely not to cache current thread's known priority level in user space, thus resulting in excessive calls to the kernel; they also do not provide for invalidation of cached value when a priority of the thread is changed externally by another thread;

- pthreads does not have adequate means to manage thread priority changes by outside events, such as to elevate thread priority when an external signal is sent to a thread by other thread, and add this signal (until processed) to thread priority protection database.

Therefore, VAX MP does not rely on pthreads or other host-specific thread priority protection feature built in into host locking primitives, and opts instead to use explicit self-elevation (and self-reduction) of thread priority as decided by simulator’s own thread priority management logics.

Mapping execution context model: virtual processors to host system threads

One principal decision for SMP virtualization design is mapping virtual processors (VCPUs) to available host execution units (host logical processors). Two considerations that need to be addressed by the implementation of execution model mapping are:

- *Synchronicity.* Mapping should provide a way to keep VCPUs in a relative lockstep and prevent them from drifting apart from each other by too much in their forward progress. This is essential capability because guest OS or applications running on it may implement timeouts for interprocessor interaction and consider VCPU whose execution had been preempted for far too long as hung and trigger a failure – application abort or system crash.[19]

  Another aspect is a synchronicity between VCPUs and virtual devices: if synchronicity of interaction between them is unsatisfactory, guest OS may decide that device is not responsive and trigger a failure.

- *Lock holder preemption avoidance*[20]*.* When an execution context holds a lock for an important resource (represented e.g. by guest OS spinlocks or VM internal locks), it is undesirable to preempt this execution context before it finishes processing associated with holding the lock and releases the lock, as preempting it while the lock is held is likely to cause other virtual processors trying to lock the same resource to spin waiting on the lock, uselessly wasting host computational resources. Furthermore, if execution contexts spinwaiting on the lock already hold locks for other resources, this can also lead to convoying, with a whole chain of virtual CPUs spin-waiting for the previous VCPU in wait chain, which in turn waits for yet a previous VCPU etc. ultimately waiting on the preempted VCPU. Thus preemption of VCPU holding a lock is likely to cause other VCPU or multiple VCPUs to go into spinwait state waiting for the preempted VCPU and wasting host computational resources, and also (by occupying host LCPUs) not letting the preempted VCPU to complete its operation and release the resource lock. Therefore VCPU scheduling on the host should not be agnostic of VCPU state, and VM is responsible for implementation of lock holder anti-preemption measures.

[19] OpenVMS allows to disable some, but not all of SMP synchronicity and sanity checks in VMS kernel with SYSGEN parameter TIME_CONTROL. This is discussed further in more detail in sections of interprocessor synchronization window and implementation of interlocked instructions.

[20] For introduction to lock holder preemption problem see Volkmar Uhlig et. al. “Towards Scalable Multiprocessor Virtual Machines”, Virtual Machine Research and Technology Symposium 2004, pp. 43-56; also see Gabriel Southern, “Analysis of SMP VM CPU Scheduling” (http://cs.gmu.edu/~hfoxwell/cs671projects/southern_v12n.pdf). One associated study worth looking into is Philip M. Wells, Gurindar S. Sohi, Koushik Chakraborty, “Hardware Support for Spin Management in Overcommitted Virtual Machines”, Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT) (2006), pp. 124-133.

While the bulk of available research on the problem focuses solely on preemption of thread or VCPU holding a lock, such as a spinlock, this specific problem case is actually merely a part of a much larger pattern, where one thread or VCPU is spin-waiting for an activity to be completed by another thread or VCPU. Such waiting is not necessarily limited to lock acquisition, but can be for acknowledgement flag or signal – for instance, one VCPU waiting for the confirmation that its interprocessor request had been processed by another VCPU. Preemption of VCPU serving the request would cause the requesting VCPU to spin, at best wasting host resources and introducing latency, and at worst triggering a timeout condition and system failure.

Some interaction patterns in multiprocessor system involve multiple VCPUs interacting with each other, perhaps in multi-stage handshake. One example of such interaction in OpenVMS is TLB invalidation for system pages (pages in VAX S0 address region) explained in the subsequent section: it requires simultaneous interaction of all processors in the system, and if one of VCPU threads gets preempted, other VCPUs will all spin unable to make forward progress.

Some of existing SMP virtualization solutions seek to mitigate lock holder preemption problem by modifying guest OS code to delegate spinlock control to hypervisor, via paravirtualization calls. With OpenVMS this would not really solve the issue, since besides few busy-wait loops for spinlock acquisition OpenVMS also has large number of diverse busy-wait loops for various interprocessor interaction patterns dispersed throughout OpenVMS code. Besides the sheer number of busy-wait loops in OpenVMS, insides of these loops access system data and perform complex checks, branching and conditional actions (such as related to system debugger), and thus handling of OpenVMS busy-wait loops cannot be easily delegated to hypervisor level.

VAX MP chooses to implement execution model by statically mapping host threads to VCPUs. Each VCPU is allocated host thread statically associated with it.

In addition VAX MP also utilizes optional threads for some IO devices[21], optional thread for system clock that sends clock interrupts to all active VCPUs at each clock tick, and also simulator console thread.

Static mapping of threads to VCPUs is straightforward, but not the only thinkable approach.

Another thinkable, albeit (as discussed further) impractical approach is to use thread pool with dynamic mapping of threads to VCPUs. Under that approach each thread performs a certain number of VAX instructions (work chunk) of one VCPU execution stream and then (or on earlier signal from dispatcher module) either proceeds of stalls or possibly switches over to another VCPU context and instruction stream processing. Dispatcher module is responsible for dynamic mapping of available threads to waiting VCPUs, potentially in round robin like fashion (if the number of VCPUs exceeds the number of LCPUs), but accounting for synchronicity and lock

[21] Currently worker threads can be utilized by MSCP disk devices, MSCP tape devices and Ethernet controllers. SIMH-level handlers for these devices implement asynchronous IO operation processing with the use of device handler worker threads. Use of device handler threads is optional, but is recommended in multiprocessor configuration.

holder anti-preemption requirements, and each available pool thread goes through the dispatcher after completing its chunk of work or if signaled by the dispatcher and is assigned next chunk of work according to dispatcher's logics.

This approach however is impractical for a number of reasons. Most obviously, frequent switching of host threads between VCPU contexts would undermine various caches (data caches, physical TLB, virtual TLB and other VCPU structures etc.) and thus would negatively affect execution performance. The impact of cache thrashing[22] can be minimized by increasing the size of standard work chunk and also by coding the dispatcher to normally keep a thread to the previous VCPU and minimize thread switching between VCPUs unless a switching is required due to overriding considerations of synchronicity.

Described approach also has, at a first look, a number of advantage promises – that, however, at a closer consideration are either difficult or impossible to realize.

One obvious advantage promise is a possibility to configure the number of active VCPUs exceeding the number of host LCPUs, since VCPU worker threads are not "wired" to a single processor and thus each thread can serve more than one processor. This however would present a major problem for running OpenVMS and most likely other guest OS'es as well. OpenVMS contains sections of code that require simultaneous execution of *all* virtual processors.[23] If any of virtual processors is stalled, other VCPUs are unable to make forward progress and will spin-wait wastefully. The issue can be avoided if work chunk size is made small (so work thread quickly switches from moving ahead one VCPU to moving ahead another VCPU and pushes them through this code path bottleneck without too much of wasteful spinning), but making work chunk size small would be in direct contradiction with avoidance of cache thrashing. A possible solution could be modification of OpenVMS to recognize specific code paths and communicate their execution to the simulator that would, based on such communication, dynamically adjust work chunk size – but the need for such pretty involved modification of legacy OS is highly undesirable.

However the true attraction evoked – at a first glance – by general idea of discussed approach is a hint at the possibility of fine-grained intra-application scheduling of VCPUs that would help to address the requirements of synchronicity and lock holder preemption avoidance – a hint at the idea of cooperative system/application scheduling, with application having critical knowledge about current fast-changing needs of its threads and able to participate in fine-grained scheduling decisions. Cooperative system/application scheduling for trusted critical applications (such as database engines or multiprocessing VMs) is a capability missed in most general-purpose operating system, with general-purpose thread priority control provided by them being just too crude, expensive and low-bandwidth mechanism. Unfortunately, this hint is all that this

[22] Or, at best, cache partitioning between VCPUs the thread is executing.

[23] One example explained below in detail in this section is OpenVMS multiprocessor TLB invalidation code. However there are other paths with similar behavior.

approach and its description can actually offer. First, the lack of underlying preemption control facilities in targeted host OS'es makes the discussed approach untenable: if a pool thread executing VCPU stream gets preempted by the host OS, it will most likely be preempted in the middle of executing instruction stream (rather than at safe points, with designated exits to the dispatcher), with VCPU structures being left busy and other thread therefore unable to take over these structures. Given the lack of efficient low-overhead anti-preemption primitives in the targeted host OS'es, it is therefore impossible to implement the described approach in a practical and efficient way. Second, if discussed category of primitives were available, they would most probably lend themselves just as well for the use with threads statically mapped to VCPUs.

Thus the approach of dynamically mapped thread pool does not actually confer any likely advantages beyond the possibility of running higher number of VCPUs than the number of host LCPUs available… Even with the latter capability coming of course at the cost of a loss in actual system throughput due to context switching overhead and thus largely defeating meaningful purposes of multiprocessing other than playful simulation, and also at the cost of somewhat elaborate modifications required to OpenVMS pre-existing binary image.

For described reasons VAX MP opts for statically mapping host execution thread to VCPUs.

To address synchronicity requirements VAX MP implements interprocessor synchronization window mechanism described in subsequent chapter.

To implement lock holder preemption avoidance, VAX MP relies on VCPU thread priority control. VAX MP defines a range of logical thread priorities mapped to host thread priorities, as described in the following table, in ascending order[24]. Priorities utilized by VCPU threads are highlighted in green, priorities used by auxiliary (non-VCPU) thread are marked gray. Priority assignment for some levels depends on exact VCPU state, such as particular IPL range, and is configured by paravirtualization layer when multiprocessing is being initialized. The table describes priorities as they are configured by paravirtualization module for OpenVMS (VSMP). If multiprocessor support for another operating system is later implemented by VAX MP, such as perhaps for some BSD variants, exact priority assignment for that system may differ, but it is still configurable by paravirtualization module for the given guest OS.

[24] Ascension is within a category, such as within CPU priorities, or CONSOLE priorities. Priority interrelation across the categories may slightly vary. See another table further in this chapter for logical to host priority value mapping for particular host platforms.

Thread logical priority	*Description*
CPU_RUN	Normal priority for VCPU thread execution. VAX user mode code and low-IPL privileged code will typically be executed at this VCPU thread priority.
CONSOLE_PAUSED	Priority for simulator console thread while VCPUs are paused.
IOP	Priority for IO processing threads (virtual disk, tape drive or network adapter simulator handler threads).
CPU_CRITICAL_OS	Priority for VCPU thread when VAX code is inside OS critical sections. Under OpenVMS VCPU thread elevates to this priority if VCPU's IPL goes above IPL$_QUEUEAST level, which typically will happen if VCPU acquires a spinlock. IPL$_QUQUEAST is the lowest spinlock-holding IPL. It is also possible for VCPU to elevate IPL while not holding a spinlock, but in most cases elevation of IPL means execution of system-critical task, whether holding spinlock or not.
CPU_CRITICAL_OS_HI	Priority for VCPU thread inside OS high-priority critical sections. VCPU thread elevates to this priority while processing clock interrupt or interprocessor interrupt, or if any of these two interrupts is pending for the VCPU.
CONSOLE_RUN	Priority for simulator console thread while VCPUs are running.
CPU_CRITICAL_VM	Priority for VCPU thread holding VM internal locks.
CLOCK	Priority for clock strobing thread.
CPU_CALIBRATION	Priority for VCPU thread executing busy-wait loop calibration. Currently VAX MP considers processor to execute busy-wait loop calibration if SSC clock is active, since OpenVMS uses SSC clock to calibrate processor busy-wait loops, but it might also be possible to enter and leave calibration-priority state by direct calls via paravirtualization interface. This is (and must be) the highest level of priority, as discussed further down in this section.

As VCPU changes state, such as alters IPL or receives pending interrupt, VAX MP VM will adjust VCPU thread priority accordingly. For example acquisiton of a spinlock by OpenVMS will result in IPL elevation to the level of QUEUEAST or above, which in turn will cause thread priority to elevate to CRITICAL_OS, unless it was already higher. Likewise, if VAX MP VM observes that current VCPU has clock interrupt or interprocessor interrupt pending, it will elevate thread priority to CRITICAL_OS_HI.

When an interrupt is sent to VCPU (either by IO device, internal device such as clock, or by other VCPU), target VCPU's thread priority is boosted by the sender to CRITICAL_VM thus ensuring interrupt will be fetched by target VCPU as quickly as possible.[25] After noticing the interrupt, targeted VCPU reevaluates and adjusts its thread priority accordingly, typically either to CRITICAL_OS or CRITICAL_OS_HI, unless intra-VCPU thread priority management logics had previously set desired priority even higher. When interrupt handler processes the interrupt and interrupt gets dismissed, VCPU thread priority will be re-evaluated again and downgraded, if necessary.

Thread priority levels can also be associated with some types of VAX MP VM internal synchronization objects, including `smp_lock` and `smp_mutex`. Currently only CRITCAL_VM and CRITICAL_OS_HI are the levels that can be associated with synchronization objects. When VCPU thread acquires a lock object with associated criticality, thread priority is elevated to this level unless it was already higher. When VCPU thread releases the lock, thread priority is reevaluated and downgraded, unless the thread is holding other high-criticality lock objects, or VCPU has interrupts pending or is executing at elevated IPL. In the latter cases thread priority can be demoted, but only down to the highest level implied by any of the listed state conditions.

VAX MP provides two modes for the implementation of VAX interlocked instructions: "portable" and "native". In portable mode interlocked instructions are implemented via a set of VM internal locks. In native mode some interlocked instructions are implemented using host-specific interlocked instructions operating directly on VAX memory and thus executed atomically, in non-preemptable fashion, whereas other interlocked instructions are implemented as composite and VCPU thread can be preempted in the middle of the instruction, with VAX datum's (interlocked queue header) secondary interlock held by the thread. In those cases when the implementation of VAX interlocked instruction is not atomic on the host and can be preempted in the middle of the instruction, VAX MP temporarily elevates VCPU thread priority to CRITICAL_VM for the duration of the instruction, in order to reduce the likelihood of thread preemption while holding a lock required by the instruction[26].

Thus, VAX MP actively manages VCPU thread priority and it is changed often. VAX MP employs various optimizations and also caches logical priority in order to reduce the number of actual calls to the host OS kernel to change thread priority. As one special case, system calls to change thread priority are nearly completely eliminated when only one VCPU is active. Nevertheless under multiprocessor workloads that

[25] Unless thread priority management is disabled because VAX MP executes on dedicated host machine (configuration option HOST_DEDICATED is set to 1 by user), or because there is only one VCPU currently active in the system.

[26] Either VM-level lock (in portable mode) or VAX datum secondary interlock (in native mode).

involve heavy use of interlocked instructions and other inter-processor interaction, a notable fraction of time (e.g. 10%) can be spent on managing thread priority, although the cost is much less in more common workloads. On modern host computers (with 3 GHz x64 CPUs) a call to change thread priority takes about 1 microsecond on targeted host operating systems, thus the cost of VAX interlocked instruction that requires managing thread priority is approximately 2 microseconds , assuming the change in priority did not cause a rescheduling, and is equivalent to the cost of approximately 50-100 regular VAX instructions. Although this may sound as not too much for a specialized instruction, some workloads may involve very high rate of interlocked instructions. VAX MP provides two approaches to alleviate this cost. One is the use of native mode to implement interlocked instructions, which may be unavailable on some host processors, but is available on x86/x64 and allows to avoid thread priority management for interlocked instructions BBSSI, BBCCI and ADAWI, although still requires it for interlocked queue instructions. Nevertheless since BBSSI and BBCCI make a lion share in typical OpenVMS interlocked instructions workload, use of native mode eliminates the need for thread priority management for most common interlocked operations and drastically reduces the overhead caused by thread priority management. The other approach available only when VAX MP is executed on dedicated host system with no concurrent host load is VAX MP configuration option that disables thread priority management altogether, in assumption that preemption of VCPU thread on a dedicated system is an infrequent event.

Thus, the overhead of VAX MP multiprocessing implementation ranges, for most OpenVMS workload, from acceptable to negligible. Nevertheless it would have been desirable to have efficient low-overhead host OS anti-preemption primitive that would allow to avoid virtually any thread priority management overhead, for example host primitive for “deferred change thread priority” that would store desired thread priority in a page shared by userland and kernel, as a value available to the kernel on demand, but used only when a scheduling decision is actually being made by the kernel. In most cases, thread priority would be elevated only for a short time and downgraded back rapidly, before the scheduler would have a need to take a notice of pending change request and act on it. This would have allowed to eliminate the overhead of most of the remaining “set thread priority” calls.

* * *

Since according to the chosen VCPU thread priority scheme VAX kernel-mode code running at IPL below QUEUEAST (6) runs at VCPU thread priority CPU_RUN, even VMS non-preemptable code executing in IPL range [RESCHED … QUEUEAST-1], priority inversion is possible if the path executing at IPL RESCHED (3) acquires a spinlock or lock flag needed by VCPU thread executing at high IPL (at or above QUEUEAST). We are not aware of any paths in OpenVMS kernel that do this. One such path was in routines EXE$READ_TODR and EXE$WRITE_TODR that interlock flag SMP$V_TODR also used at higher IPLs, however VSMP overrides these routines. Overriden version elevates IPL to QUEUEAST before interlocking the flag.

* * *

The mapping of VAX MP logical thread priorities to platform-specific thread priorities is summarized in the following table. Notice that these mappings are implementation and release specific and are easily subject to change. The table is inteded just to give a general idea.

Thread logical priority	*Windows*	*Linux*	*OS X*
CPU_RUN	BELOW_NORMAL	0, nice 10	25
CONSOLE_PAUSED	NORMAL	0, nice 0	31
IOP	ABOVE_NORMAL	RR, 25	34
CPU_CRITICAL_OS	ABOVE_NORMAL	RR, 25	34
CPU_CRITICAL_OS_HI	HIGHEST	RR, 28	38
CONSOLE_RUN	HIGHEST	RR, 28	42
CPU_CRITICAL_VM	TIME_CRITICAL	RR, 31	42
CLOCK	TIME_CRITICAL if SMP is active, otherwise HIGHEST	FIFO, 34	42
CPU_CALIBRATION	TIME_CRITICAL	FIFO, 37	47

* * *

VAX MP addresses resource holder preemption problem for high-IPL critical section.

However this is not the only case the problem can manifest itself – albeit it is most urgent case with most severe consequences. Yet there are other, albeit milder cases too.

One kind of lock holder preemption not addressed by initial release of VAX MP is preemption of VCPU executing at low IPL (ASTDEL to QUEUEAST-1), but executing VMS process holding kernel-mode mutex.[27]

[27] Processes holding kernel mutexes execute in OpenVMS at IPL ASTDEL (2) or higher, and at VMS process priority elevated to 16, unless the process was already in real-time priorities range (16 to 31).

Since holding mutex or execution at enhanced VMS process priority does not translate into enhanced VCPU thread priority, VCPU thread in this case executes at low host priority, and its preemption is quite possible. Preempting VCPU in this state may cause priority inversion and would also cause increased latency for other VMS processes waiting for the mutex. However VMS processes do not spinwait for mutex acquisition, they enter wait state. Thus processes blocked waiting for the mutex will yield a processor, and VCPUs they might have used will either be used for useful computations by runnable processes (processes in COM state) or will enter idle loop and, if VAX MP idle sleep option is enabled (which usually will be the case), idling VCPUs threads will hibernate and yield host processor too, which will create the opportunity for preempted mutex lock holder VCPU to resume and complete mutex-holding critical section.

Thus preemption of VCPU executing mutex-holding process is a milder issue, albeit it is still undesirable, since priority inversion is very possible, and VCPU executing VMS process that holds a mutex may well get stalled in favor of VCPU executing low-priority process.

Other cases of lock holder preemption are possible too if host scheduler preempts VCPU that executes VMS process holding VMS locks managed with VMS ENQ/DEQ system services. Such preemption is possible even in VMS running on a real hardware VAX, but can be counteracted by elevating process priority. VAX MP however does not account for the priority of VMS process currently executing on VCPU and does not translate enhanced VMS process priority to enhanced VCPU thread priority, therefore preemption or effective preemption of ENQ lock holder is more likely under VAX MP than on a hardware VAX.

These are milder cases of resource holder preemption, therefore initial release of VAX MP chooses not to address them. Should they be deemed a problem, this issue can be revisited later. Two most obvious issues that any solution would face are that:

1. The scale of host system's thread priorities, to which VMS process priorities can be mapped, is very narrow. This is especially the case with Windows (which does not really have any space left on the scale that may be used for discussed purpose), and also may be the case with OS X and Linux, if VAX MP were to be restricted to a certain sub-range of host priorities.

   If VAX MP were to execute on top of "bare metal" hypervisor with sufficiently large priority scale, mentioned problem would not exist in that configuration.

2. There is certain overhead in tracking VMS process priority transitions and mapping them to VCPU thread priority, albeit in most cases likely to be a minor overhead, but it might become somewhat noticeable if system workload generates high rate of events resulting in process priority boosts.

* * *

Another host OS feature than could have been potentially beneficial for SMP virtualization is gang scheduling (co-scheduling)[28] of VCPU thread set, whereby if one VCPU thread gets preempted, all VCPU threads get preempted[29]. Gang scheduling of VCPU threads could have been considered as a means to address, to an extent, requirements of inter-VCPU synchronicity and spin-waste of computational resources due to lock holder preemption. However gang scheduling is not available on targeted mainstream desktop OS'es. Furthermore, it is not a common option even under "bare metal" hypervisors: VMWare ESX provides a form of gang scheduling, however Hyper-V does not, and production versions of Xen do no provide it as well. There had been proposals and even experimental patches for gang scheduling on Xen, however Xen development community believes that the issues that could be addressed by gang scheduling can better and with higher efficiency be addressed by modifying guest OS to explicitly support running under the hypervisors, by providing necessary conditional logics within the guest OS and also paravirtualization layer for guest OS interaction with the hypervisor delegating to the latter functions such as spinlock control and other busy-waiting. This is indeed a sound approach for guest operating systems that can be modified in their source code and rebuilt, but the situation is different with legacy guest systems, such as the case with OpenVMS, that in practice cannot be rebuilt, and have to be used in pre-existing binary form with no or very limited amount of modifications possible to the pre-existing binary code.

Nevertheless, even if gang scheduling were available, in its basic form it would create more problems for SIMH SMP virtualization than it solves and be at best of very limited, if any, help for the purpose. VCPU threads will get preempted now and then, either because of simulator page fault or because of synchronous and blocking IO operation incurred on VCPU thread, or because some system daemon or other host system service needs to execute briefly. Brief preemption of one or another VCPU thread can thus be a very frequent occurrence. For this reason, immediate (almost immediate) suspension of other VCPU threads from execution initiated once one of VCPU threads is preempted would lead to huge inefficiency, since VCPU thread set would get suspended all the time, which would drastically hamper system's forward progress. A solution to this is possible in the form of modified gang scheduling policy preempting the whole VCPU thread set only if one of the threads had been preempted for long enough, longer than certain threshold interval; however such modified policy goes beyond the conventional gang scheduling scheme. Furthermore, gang descheduling threshold would have to be introduced and managed very carefully, so as not to undermine the efficiency of ensuring synchronicity of VCPUs execution[30] and avoidance of wasteful spinning of VCPUs waiting for preempted lock holder[31] – the very

[28] See J. Ousterhout, "Scheduling Techniques for Concurrent Systems", Proc. 3rd International Conference on Distributed Computing Systems, October 1982, pp. 22-30. See also Orathai Sukwong, Hyong S. Kim, "Is co-scheduling too expensive for SMP VMs?", EuroSys '11 Proceedings of the sixth conference on Computer systems, ACM, 2011, pp. 257-272; also Dror Feitelson, Larry Rudolph, "Gang Scheduling Performance Benefits for Fine-Grain Synchronization", Journal of Parallel and Distributed Computing, December 1992, pp. 306–318.

[29] Except, perhaps, for VCPUs lagging behind.

[30] Due either to too large gang descheduling threshold or progressive drift accumulation due to it.

[31] Due to too large gang descheduling threshold.

issues gang scheduling would have been meant to address in the first place. In the end, even with modified gang scheduler accommodating a descheduling threshold, a mechanism functionally identical to interprocessor synchronization window described further in this document would still have been required for this reason. Therefore gang scheduling, even if it were available, would not be an automatic panacea and would be of limited benefit for SIMH SMP virtualization.

* * *

VAX MP can only support the number of active virtual processors that does not exceed the number of available host execution units (LCPUs). The reason for this is that OpenVMS contains sections of code that require simultaneous execution and interaction of *all* virtual processors in the system[32]. If any single VCPU is stalled, other VCPUs are unable to make forward progress and spin wastefully.

For an example of such codepath, consider how a processor in OpenVMS VAX SMP system invalidates TLB entry for a system page (page in S0 region):

1. Each processor on the system has per-CPU flags ACK and TBACK.
2. Processor requesting TLB entry invalidation for a given page acquires INVALIDATE spinlock and CPU database mutex.
3. Requestor VCPU sends INV_TBS request to all other active processors in the system (hereafter, responders) via inter-processor interrupts.
4. Requestor spin-waits till ACK flags are set by all the responders.
5. When each responding CPU receives INV_TBS request, it sets this CPU's ACK flag and enters spin-wait loop waiting for the requestor to set this responder's TBACK flag.
6. Requestor waits till ACK's are received from all the responders.
7. Meanwhile, responders spin waiting for their TBACK flags to be set.
8. Requestor modifies page PTE.
9. Requestor sets TBACK flag for each of the responders.
10. Requestor performs local invalidation, and releases INVALIDATE spinlock and CPU database mutex.
11. Each responder notices its TBACK set, clears it, and executes local TLB entry invalidation.

Thus, *all* processors on the system are involved in common multi-stage handshake, and *all* processors spin simultaneously as a part of this handshake. If any single VCPU is stalled, handshake cannot go through and other VCPUs keep spinning indefinitely.

If the number of configured and active VCPUs exceeds the number of LCPUs it will take very large time for this handshake to eventually clear, and for all practical purposes the system will appear hung.

The same effect would be observed if the host system is overcommitted in any other way, for example if the combined number of VCPUs in *all* VAX MP instances concurrently executing on the host exceeds the

[32] More specifically, all processors in OpenVMS active set, i.e. successfully started with START CPU command and not stopped.

number of host LCPUs. Like in the previous case, instances will effectively enter into a livelock as soon as two or multiple invalidations (or similar code paths) in them collide.

The same livelock behavior would also be observed if the host is executing concurrent workload that prevents some of the VCPUs from execution, effectively denying LCPUs to VAX MP for an extended time. VAX MP addresses this problem by elevating VCPU thread priority in relevant OpenVMS code sections above normal priority range (since such sections are executed at IPL above QUEUEAST, translating to VCPU logical priority CRITICAL_OS or CRITICAL_OS_HI). However high-priority host processes running non-stop and effectively making some of LCPUs unavailable for VAX MP will induce similar livelock behavior.

* * *

The key reason why inter-VCPU synchronicity is essential is that OpenVMS implements a variety of checks that monitor sanity of multiprocessor operations. Few major categories of checks are employed by OpenVMS to ensure that none of the processors in the system are hung (either hung at hardware level or hung unresponsive at high IPL due to software bug):

1. OpenVMS kernel will retry interlocked queue operations and some other interlocked operations only for a limited number of times. If secondary interlock on interlocked queue cannot be obtained after a specified number of retries or certain non-queue operations fail to complete after a specified number of retries, this is considered a fatal condition and OpenVMS will generate a fatal bugcheck. System will be forced to crash, system dump will be recorded, and system will optionally reboot.

2. All processors are linked inside OpenVMS kernel in circular sanity check chain. Each processor in the multiprocessor system receives clock interrupts with 100 Hz frequency and processes these interrupts, unless the interrupt cannot be delivered because the processor is currently executing at IPL higher than clock interrupt level (in which case the interrupt will be delivered as soon as processor's IPL drops below hardware clock interrupt level) or the processor is hung. On each clock tick, OpenVMS clock interrupt handler increments sanity monitor counter for the next processor in sanity monitor chain and resets sanity monitor counter for the current processor. If sanity monitor counter for the next processor in chain exceeds a threshold, this is taken as an indication that the next processor is unresponsive (had been unable to process clock interrupts for quite some time), either because it is hung at hardware level or hung at high IPL because of the bug in the system software. If this condition is encountered, current processor will trigger fatal bugcheck.

3. When OpenVMS code path tries to acquire a spinlock, it will spin-wait on spinlock object only a finite number of times. If spinlock cannot be acquired after a certain number of busy-wait cycles, OpenVMS will trigger fatal bugcheck.

4. When OpenVMS code path tries to make interprocessor request that requires synchronous response from the target processor, requesting code path will wait for a response only a finite amount of time, represented as finite number of busy-wait cycles.

5. Some inter-processor interaction code paths involve waiting on a flag or other datum that needs to be modified by another processor as a form of interaction acknowledgement. Similary to spinlock case, if acknowledgement is not seen after a certain number of busy-wait cycles, OpenVMS will trigger fatal bugcheck.

Categories (1) and (2) are analyzed in detail further in this document.

Check categories (3-5) are based on busy-wait loops. OpenVMS busy-wait loop are meant to allow the event being waited to happen within a certain maximum allowed time; if the event does not happen within the specified time limit, then a bugcheck is raised[33]. Time limit for busy-waiting the event is defined by OpenVMS SYSGEN parameters and is expressed as a multiple of 10 microsecond units. However during actual execution of busy-wait loops, loop code does not reference system clock to check for timeout condition[34], instead the basic prototype busy-wait loop is pre-calibrated at system initialization and desired temporal limits are translated into the number of loop iterations equivalent to the time limit for the given processor type. OpenVMS busy-wait loops do not reference clock, instead they execute for certain maximum number of iterations. The iteration count is derived from SYSGEN loop timeout value (expressed as a multiple of 10 microsecond units) multiplied by calibration data (roughly, the number of basic loop iterations executed by the processor per 10 microseconds).

For correctness of multiprocessor operation it is important that busy-wait loop calibration is accurate. If the loop is under-calibrated, this can cause trouble by resulting in too short actual busy-wait loops, leading in turn to bugchecks.

To ensure accurate calibration, VAX MP employs two techniques.

First, it does not rely on one-pass calibration measured natively by OpenVMS at system boot time, since one-pass calibration can easily be distorted by concurrent host load. Instead when VAX MP paravirtualization layer (VSMP) is loaded, it performs calibration multiple times (between 25 to 200 times, as may be necessary) to ensure calibration results converge to the fastest one measured. Produced results are then stored in OpenVMS kernel as active calibration before the multiprocessing is actually activated.

[33] More precisely, OpenVMS busy-wait loops spin waiting for the associated event for *at least* the amount of time defined as the limit; they can actually spin for somewhat longer time than the limit too.

[34] VAX clock does not have high enough resolution (not on all VAX models, anyway) and also continuously referencing the clock inside the loop would introduce unnecessary transaction on the bus and possible overhead.

Second, when performing loop calibration, VSMP elevates VCPU thread priority to CPU_CALIBRATION[35], the highest thread priority level used by VAX MP, also mapped to high host thread priority level. This ensures that calibration is unlikely to be undermined due to thread preemption.

Taken together, these techniques ensure satisfactory reliability of calibration results.

When VAX MP runs on hyperthreaded or SMT core, hyperthreading can also distort loop busy-wait calibration if the core is shared with other thread during the calibration or during actual loop execution. To compensate for possible distortion, calibration data in this case is multiplied (if running on 2-way hyperthreaded host processors) by a factor of 1.8. The exact value of adjustment factor can be overridden with VAX MP SIMH console command `CPU SMT <slow-down-factor>`.

Accuracy of busy-wait delay loops timing can also be undermined by the effects of memory caches, both instruction cache and data caches (OpenVMS busy-wait loops spin counting down on-stack variables). Such loops are obviously dependent on cold/warm cache line status and are vulnerable to cache flushes. Effects of cold cache delay on first counter access may also pro-rate differently depending on intended duration of actual delay loop vs. the duration of calibration loop. Indeed, OpenVMS initialization code for some VAX processors (such as for 3520/3540) imposes hardcoded constraints on measured calibration value – not a possibility for VAX MP. These effects will be even more pronounced for contemporary processors that have much higher non-cached vs. cached time access ratio than processors of the VAX era.

The impact of cache effects may be even further aggravated by the fact that VCPU threads are not tied to a physical CPU (core) and can migrate between host PCPUs, and thus find their caches being left behind and hence local cache being "cold" again.

Nevertheless our current assessment is that given that the number of both calibration cycles and actual wait cycles in SMP sanity timeouts is high to very high[36], the impact of cache effects in both cases will be negligible and will not undermine the validity of calibration.

Thread migration between PCPUs or preemption can be expected to affect calibration very significantly (and in this case overhead of migration will dwarf cache effects). The solution, as described above, is to elevate VCPU thread priority during calibration to minimize the chances of the thread being preempted, and to perform calibration multiple times, and select the highest readings in the produced set, so that probability of the impact by migration and preemption on calibration can be further minimized.

* * *

[35] VAX MP elevates VCPU thread to CPU_CALIBRATION priority level when KA650 SSC clock is enabled. OpenVMS and VSMP use SSC clock only to calibrate busy-wait loops.

[36] 400,000 cycles during inner loop calibration (VSMP uses higher calibration cycle counter than native VMS initialization code), 20,000 cycles during outer loop calibration, and typically very high loop counter (millions) for SMP sanity timeouts.

Besides synchronicity of VCPU execution contexts (i.e. VCPU-to-VCPU synchronicity), multiprocessor simulator should also maintain synchronicity of IO devices to VCPUs, otherwise IO device may appear to the guest operating system as not responding in expected time, which would cause guest OS to trigger false device timeout condition. Maintenance of IO devices synchronicity to VCPUs in VAX MP and prevention of false device timeouts is discussed in detail in chapter “Device timeouts”.

Interprocessor synchronization window

Some operating systems, including OpenVMS, implement timeouts for interprocessor operations. For example, OpenVMS executing on an SMP system will be trying to acquire a spinlock only for a limited time (measured as limited number of busy-wait loop iterations) before concluding that spinlock structure is either corrupt or the CPU holding it must be hung, and will trigger fatal system bugcheck (crash) once past the timeout, i.e. busy-wait loop iteration count had exceeded the limit.

Similarly, OpenVMS system code not preemptable by the scheduler (executing at IPL 3 or above) will retry operations on interlocked queues for only a limited number of times; if the code fails to acquire a secondary interlock on the queue header after a certain number of retries, it concludes that either queue header must be corrupt, or the CPU that acquired secondary interlock on this queue header must have hung before it could release the interlock; when detecting these conditions OpenVMS will trigger a fatal bugcheck.

These SMP sanity checks do not cause a problem for OpenVMS system operating on a real silicon processor (rather than software simulator) since hardware processors operate non-stop at fixed, predictable number of cycles per second, and can be expected to finish processing and release a spinlock (or release interlocked queue or other waitable resource) within a predictable interval, expressed as predictable number of busy-wait loop iterations by any other processor in the system waiting for the resource to be released – and if the resource is not released within this interval, then SMP sanity bugcheck is justified and valid.

Unlike hardware processors, software simulator’s VCPU threads do not run at guaranteed predictable number of cycles per second. VCPU threads can be preempted by the host system due to competing host load[37] or due to page fault on simulator data or code, including runtime libraries data/code utilized by the simulator, or when VCPU thread performs IO operation on the host etc. Thus, while real hardware CPUs run at steady pace and clock out nearly identical number of cycles per second and do not drift apart in their notion of CPU local time and forward progress by much, simulator’s VCPU thread cannot have such shared notion of virtual time flow (at least not at the level of guest OS not designed

[37] On some virtualization hypervisors this issue is addressed by gang scheduling, whereby all VCPU threads are either scheduled or de-scheduled as a unit together, and thus if one VCPU thread gets preempted, other VCPU threads get stalled too and do not perform any busy-wait cycles. However mainstream desktop operating systems targeted by VAX MP as host environment (Windows, Linux, OS X) do not implement gang scheduling. Nor it is implemented by most existing hypervisors, including Hyper-V and production releases of Xen. Furthermore, gang scheduling by itself does not help with stalls caused by page faults.

specifically to deal with virtualization) and can drift apart from each other in their notions of per-CPU time flow due to uncontrollable VCPU thread preemption.

Unless some special measures are taken about such drift, a VCPU in OpenVMS SMP system can perceive other VCPU thread's preemption as processor being hung and trigger a bugcheck crash.

Triggering OpenVMS SMP sanity bugcheck due to timing out on a spinlock wait can be disabled by VMS SYSGEN parameter TIME_CONTROL (or by increasing timeout values with SYSGEN parameters SMP_SPINWAIT and SMP_LNGSPINWAIT), however it is generally undesirable to disable system sanity checks, or at least undesirable having to disable them always and have no way to execute a system with checks enabled should the need to diagnose a problem occur.

Furthermore, OpenVMS timeouts on interlocked queue headers may not always be disabled via SYSGEN. Although in some cases timeouts can be extended by SYSGEN parameter LOCKRETRY, but in many cases the maximum number of busy-wait loop iterations is hardwired into VMS binary code.[38]

Therefore VAX MP employs a solution to this problem by constraining relative drift between VCPUs to the number of cycles that is safely under the threshold that causes OpenVMS to trigger a bugcheck. If some VCPU starts lagging behind other VCPUs by too much, other VCPUs are going to stall and wait for the lagging VCPUs to "catch up". We call this mechanism "synchronization window" since it seeks to ensure that all interacting VCPUs fall within a narrow window with respect to each other's notion of virtual time (as represented by each VCPU's counter of accrued cycles) – narrow enough to be safely below SMP sanity timeout interval.

A crude and naïve implementation of synchronization window would impose it on all VCPUs at all time. However this would limit system scalability, introduce significant overhead, and would not be justified since much of the code executed by the simulator, specifically user-level applications, do not need to be constrained by the synchronization window: such code is designed from the very start as preemptable and capable to deal with and successfully perform in preemptable execution environment – it does not matter whether this preemption comes from preemption by other OpenVMS processes or preemption of VCPU threads, the two are largely undistinguishable for a preemptable code.

VCPUs need chiefly to be constrained to a synchronization window only when they are executing non-preemptable code (in OpenVMS case it would be the code executing at IPL >= RESCHED, i.e. IPL 3), and furthermore not all such code, but only non-preemptable code that engages into interprocessor interaction (either directly or sometimes indirectly), or what we also call here "entering inter-processor interaction pattern".

> VAX MP additionally constrains code executing interlocked queue instructions INSQHI, INSQTI, REMQHI and REMQTI for the duration of this instruction only, regardless of CPU mode (KESU) and IPL level.

It can also very well be (and indeed will be typical) that in a multiprocessor system only some of VCPUs will be executing at a given instant within an interprocessor interaction pattern, while other VCPUs will

[38] See more details in chapter "Implementation of VAX interlocked instructions".

be running preemptable code or in other ways not currently participating in interaction. Only currently interacting processors need to be constrained to the synchronization window; while non-interacting VCPUs do not need to be held back or cause other VCPUs to be held back.

Therefore VAX MP imposes synchronization window constraints only on processors executing (at a given time) critical system code and engaged in active interaction with each other. VAX MP does not constrain VCPUs executing user-mode applications or non-critical system code in order to avoid unnecessary overhead and impact on system scalability.

> An exception to this, as explained further, is interlocked queue instructions (INSQHI, INSQTI, REMQHI, REMQTI) that temporarily enter VCPU into synchronization window for the duration of the instruction even when the instruction is executed at low IPL and in any processor mode, including user mode – assuming the use of SYNCW-ILK is enabled.

The whole synchronization window (SYNCW) constraint is composed of two parts named SYNCW-SYS and SYNCW-ILK.

> SYNCW-SYS is responsible for constraining busy-wait loops controlled by SYSGEN parameters SMP_SPINWAIT and SMP_LNGSPINWAIT. These are mostly spinlock acquisition loops and inter-processor interrupt or other communication (such as VIRTCONS) acknowledgement loops, although some other loops also use SMP_SPINWAIT or SMP_LNGSPINWAIT as timeout limits.
>
> SYNCW-ILK is responsible for constraining busy-wait loops controlled by SYSGEN parameter LOCKRETRY or by limit of 900,000 iterations encoded in macros such as $INSQHI etc. These loops are mostly used to perform interlocked queue operation on various queues with interlocked queue instructions INSQHI, INSQTI, REMQHI, REMQTI or for locking queue header with BBSSI instruction for scanning the queue, although some other loops are constrained this way too.

VCPU forward progress and VCPU "drift" related to other VCPUs in the system is constrained by a combination of these two constraints. Any of these two constraints, when triggered due to other VCPU/VCPUs falling too much behind, will temporarily pause VCPU getting too much ahead from execution until the constraint is cleared by VCPUs catching up and it is safe for the paused VCPU to resume execution.

The following conditions are considered to classify a processor as engaged in SYNCW-SYS interprocessor interaction pattern:

1. VCPU is holding a spinlock
2. VCPU is trying to acquire a spinlock
3. VCPU is processing IPI or CLK interrupts
4. VCPU has IPI or CLK interrupts pending but not delivered yet[39]

[39] Under the hood, "CLK pending" means logical "or" of the following conditions: SYNCLK interrupt pending, or CLK interrupt pending, or conversion from SYNCLK to CLK pending (as described in chapter "Timer control"). For brevity, in this chapter we designate a logical "or" of these conditions as "CLK pending".

5. VCPU sent an IPI request and is spinning in busy-wait look waiting for a response

Limiting future discussion to OpenVMS only, the easiest and most convenient proxy indicator for possibility and likelihood of condition (1) is checking for VCPU IPL >= QUEUEAST, the lowest of spinlock IPLs. A VCPU holding any of existing VMS spinlocks will always be executing at IPL >= QUEUEAST. It is possible sometimes for a VCPU to execute at high IPL without holding a spinlock, but most of the time when executing at IPL >= QUEUEAST it will be holding a spinlock, so simple checking for IPL level is a good, easy and cheap proxy indicator for the purposes of imposing (or not) the synchronization window.[40]

Check for IPL >= QUEUEAST also covers conditions 2, 3 and 5.

Condition 4 has to be checked by peeking at VCPU pending interrupts register.

The summary constraint based on conditions 1-5 is designated hereafter as SYNCW-SYS.

VCPU “enters” SYNCW-SYS synchronization window when it elevates its IPL from a level below QUEUEAST to QUEUEAST or above.

VCPU “leaves” SYNCW-SYS synchronization window when it drops its IPL back to a level below QUEUEAST.

Current VCPU state (“in” or “out” of window, as well as types of sub-windows active) is kept by VCPU in VAX MP VM structure called `syncw`. When VCPU enters synchronization window, it flags its state in field `syncw.cpu[].active` and initializes its position within the window. Position is maintained in `syncw.cpu[].pos`. When VCPU exits synchronization window, it clears the flag and wakes up any VCPUs that may have been waiting on this VCPU to make forward progress. The list of such VCPUs is kept in field `syncw.cpu[].waitset`.

While executing main instruction loop, simulator advances VCPU’s accrued cycle counter kept in CPU_UNIT data structure and if VCPU is “in” synchronization window, then simulator also *periodically* advances VCPU position within the interprocessor synchronization window by advancing the value of `syncw.cpu[].pos` which represents an *approximate* number of cycles accrued by VCPU since it entered the window. Since all `syncw` data is protected by a lock, `syncw.cpu[].pos` is advanced not at each VCPU cycle, but only once in a while, perhaps once in 100,000 cycles (exact syncw scale quant size is calculated depending on maximum size of the window itself). When advancing its window position, VCPU thread also performs a check of new VCPU position relative to current positions of other VCPUs that are also “in” the window, i.e. marked as active in synchronization window via their `syncw.cpu[].active` fields. VCPUs that are not “in”, e.g. executing user-mode code, are not taken into account and do not constrain current VCPU forward progress.

[40] A more complicated alternative would be for a paravirtualization code to inject interception hooks into kernel code where conditions 1-5 are actually entered and exited, but that would have required pretty involved coding of interception routines, and it is not easy to account for *all* places in the kernel where conditions 1-5 are entered and exited, therefore using IPL level as a proxy indicator is an attractive and efficient alternative.

In a word, VCPU will perform this check only if it is itself active in the synchronization window, and only against those other VCPUs that are also active in the synchronization window (marked as active in `syncw.cpu[].active`).

If during the check VCPU thread finds it drifted too far ahead of any other processor active in the synchronization window, VCPU will place itself on the `waitset` of that processor and enter sleep state. If however VCPU finds it had not been constrained from making forward progress, it will resume execution. Just before ending the periodic check routine and proceeding with forward execution, VCPU will also wake up any other VCPUs in its `waitset`, i.e. those VCPU threads that entered themselves in this VCPU's `waitset` during the duration of the previous quant and had been waiting for the current VCPU to make forward progress (i.e. advance its position on synchronization window scale) or leave the synchronization window.

VCPU threads that had been awaken will perform a recheck of their position in SYNCW relative to other VCPUs and will either resume execution or reenter `waitset` for some VCPU that falls too much behind and wait till it catches up or exits SYNCW.

Described drift-limiting checks and actions are performed by routine `syncw_checkinterval` that is invoked periodically by each running VCPU. After VCPU executes (yet another) number of instruction cycles equal to the size of syncw scale quantum, it performs `syncw_checkinterval.`

In addition to performing functions described above, `syncw_checkinterval` also checks whether any of VCPUs has IPI or CLK interrupts pending (which, per condition 4 of SYNCW-SYS definition, should define VCPU as being active in SYNCW-SYS) but is not yet marked as active in SYNCW-SYS in the `syncw` structure. If this condition is detected, found VCPU will be entered in synchronization window by VCPU performing the scan and notification will be sent to the entered VCPU via non-maskable interprocessor interrupt SYNCWSYS. The reason for these actions is as follows:

If IPI or CLK interrupt is fielded against a VCPU, this normally causes VCPU to mark itself active as in SYNCW-SYS by setting a flag in `syncw.cpu[].active` when it observes the interrupt. However there is a time span before VCPU can perform that. Most importantly, if VCPU thread gets temporarily paused because of thread preemption before it can fetch the pending interrupt and update VCPU state, it may take a while before the change of logical state is reflected in `active`. Nevertheless, per rule 4 of SYNCW-SYS definition, other VCPUs must classify that VCPU's state by logical "or" of `syncw.cpu[].active` as caused by other conditions and the content of its interrupt register. It is therefore necessary for a VCPU to be judged "in" window before it enters itself there and reflects its new state in `syncw` structure. To avoid multiple references to VCPU interrupt register content by other VCPUs throughout the synchronization window code (that would also make some optimizations impossible), synchronizaton window code performs single centralized reference in `syncw_checkinterval`, and if necessary, enters the VCPU with IPI or CLK pending but not marked "in" SYNCW-SYS yet into SYNCW-SYS, even before target VCPU had the chance to enter itself into SYNCW-SYS. This maintains the role of `syncw.cpu[].active` as the central hub for recording current syncw state, simplifies the code and allows some optimizations.

An alternative would be to enter VCPU into SYNCW-SYS in the code that sends IPI or CLK interrupts, however that would have required locking global `syncw` data when those interrupts are sent. It was deemed instead that in vast majority of cases transition into SYNCW-SYS will be processed by the receiving VCPU itself, rather than other VCPU doing this for the receiving VCPU (so the latter plays the role of only infrequent fallback), and further, that in a fraction of cases entering target VCPU into SYNCW-SYS would not be required since it would already be in SYNCW-SYS a certain percentage of time. Thus reliance on SYNCW-SYS transition processing (for transitions caused by IPI and CLK) within the target VCPU context as the primary venue eliminates some of the locking that would have been required in case if the transition were processed by the interrupt sender.

Nevertheless, in case VCPU is entered into SYNCW-SYS externally, rather than from the inside of the VCPU context itself, it is necessary to notify target VCPU of the state change, so it could reevaluate its syncw state and adjust its private syncw-related data kept in CPU_UNIT structure. This is done by fielding SYNCWSYS interrupt against the target VCPU when it is being entered into SYNCW-SYS externally. SYNCWSYS is a non-maskable high-priority interrupt that is processed entirely within SIMH level of VAX MP and is not exposed to VAX code level.

VCPU will also update its syncw state any time it tries to refer to syncw data and notices the discrepancy between `syncw.cpu[].active` and its privately kept syncw "active" mask copy, but posting SYNCWSYS interrupt expedites this process.

Because of the reliance on the described algorithm for externally entering paused VCPU with preempted thread into SYNCW-SYS (i.e. entering by other VCPUs performing `syncw_checkinterval`, rather than by IPI or CLK sender), it is possible that VCPU will be entered into SYNCW-SYS with some delay compared to the time IPI/CLK was sent, and that other VCPUs will make during this time forward progress in the maximum amount of instruction cycles equal to (*one syncw scale quantum* + `INTERCPU_DELAY`)[41]. To account for this possibility, maximum drift allowed between the VCPUs is reduced by the mentioned magnitude. `INTERCPU_DELAY` is the number of VAX instruction cycles safely exceeding the latency of inter-CPU memory updates propagation through the host system's cache coherency protocol, i.e. time between IPI or CLK interrupt is posted to VCPU interrupt register and before other VCPUs in the system can observe the change in the content of target VCPU register.

> Because access to VCPU interrupt register is not interlocked by syncw global data lock, and because inter-CPU memory updates propagation is not instantaneous, it is possible in extremely rare cases for other VCPUs to falsely see another VCPU*x* as having IPI or CLK interrupt pending, but not been "in" SYNCW-SYS. This can happen after VCPU*x* had delivered IPI and CLK to VAX level and cleared them, VAX level quickly processed the interrupt and dismissed it, VCPU REI'd to lower IPL and exited SYNCW-SYS – and have done all of this in a shorter time (number of cycles) than actual inter-CPU memory updates propagation delay. In a very unlikely case it happens, other VCPU or VCPUs may observe target VCPU*x* as still having IPI or CLK interrupts pending in its interrupt register, whereas in fact they had already been cleared there within

[41] VAX MP currently uses the hardwired value of 200 instruction cycles for `INTERCPU_DELAY`. At VM performance of 20-40 virtual MIPS (typical for 3+ GHz host computers) it translates to 5-10 usec, which should be ample for updates propagation through cache coherency protocol.

VCPU*x* memory context. Should this happens, external VCPU will enter VCPU*x* into SYNCW-SYS and send it SYNCWSYS interrupt. On receipt of SYNCWSYS, VCPU*x* will reevaluate its syncw state and remove itself from SYNCW-SYS if it actually should not be "in".

Thus, if IPI or CLK interrupt is fielded against VCPU*x*, but its thread did not have an opportunity to observe and process the interrupt yet, and VCPU*x* does not make forward progress for some time because of preemption, other VCPUs will still correctly see VCPU*x* as being "in" SYNCW-SYS and detect growing drift between them and VCPU*x* and eventually will enter the `waitset` of VCPU*x* until it catches up.

The size of SYNCW-SYS window, i.e. number of cycles that VCPUs can be allowed to drift apart, is derived from SYSGEN parameters SMP_SPINWAIT and SMP_LNGSPINWAIT (smaller of two values, which normally will be SMP_SPINWAIT). Default OpenVMS values of these parameters are defined to have approximately 10-fold or higher safety margin over minimally needed value – even on the slowest multiprocessors, and therefore have even much larger reserve on faster multiprocessors (VAX MP running on contemporary host CPUs falls into "faster" category).[42] Default OpenVMS values of SMP_SPINWAIT and SMP_LNGSPINWAIT provide ample reserve over the time (and instruction cycle count) any actually existing code needs to execute while holding a spinlock or being otherwise busy and unresponsive during interprocessor interaction pattern. VAX MP allocates time space to a synchronization window by borrowing a part of this reserve. By default, synchronization window size is set to 66% of the number of cycles equivalent to SMP_SPINWAIT.[43] In the worst case when interacting VCPUs drift to opposite sides of SYNCW-SYS scale, this borrowing leaves OpenVMS with 3x instead of 10x safety reserve (and more than 3x on modern host CPUs). This is deemed to be sufficient and safe enough. Should some system component require a larger reserve, it is always possible to increase SMP_SPINWAIT and/or SMP_LNGSPINWAIT up from their default values.

In addition to SYNCW-SYS, there is also another synchronization window constraint arising out of synchronization needs due to OpenVMS pattern of use of VAX interlocked instructions, chiefly instructions used for managing interlocked queues. This constraint is called SYNCW-ILK and is examined in more details in chapter "Implementation of VAX interlocked instructions".

The whole synchronization window is thus composed of these two constraints: SYNCW-SYS and SYNCW-ILK.

Each of SYNCW-SYS and SYNCW-ILK constraints has associated events that cause VCPU either to enter corresponding subwindow (either SYS or ILK) or to leave it.

In certain cases (configurations) use of both of these constraints (SYNCW-SYS and SYNCW-ILK) is a must and must be enabled. However in other configurations one or both of these constraints are optional and

[42] SMP_SPINWAIT and SMP_LNGSPINWAIT are expressed in time units (10 microsecond time units), so on a faster processor the same parameter value is equivalent to a larger number of instruction cycles available for the code to execute during time interval defined by SMP_SPINWAIT or SMP_LNGSPINWAIT.

[43] Exact size of synchronization window is configurable with VSMP tool options.

can be either disabled or enabled at user's discretion. This is described in more details in chapter "Implementation of VAX interlocked instructions" that contains a list and explanation of all valid VAX MP synchronization window configurations. VAX MP stores current configuration in field `syncw.on` that has two bitflags: SYNCW_SYS and SYNCW_ILK. These flags are set each to either 1 or 0 by VSMP when multiprocessing is started and can be configured by user with VSMP command options.

If `(syncw.on & SYNCW_SYS)` is set, then entry-events associated with SYNCW-SYS constraint will cause VCPU to enter SYNCW-SYS subwindow and raise SYNCW_SYS flag in `syncw.cpu[].active`. However if `(syncw.on & SYNCW_SYS)` is cleared, these entry-events will be ignored and will not cause VCPU to enter SYNCW-SYS window and mark itself as active in SYNCW-SYS by setting the flag in `syncw.cpu[].active`.

Similarly, if `(syncw.on & SYNCW_ILK)` is set, then entry-events associated with SYNCW-ILK constraint will cause VCPU to enter SYNCW-ILK subwindow and raise SYNCW_ILK flag in `syncw.cpu[].active`; otherwise these events will be ignored.

When both SYNCW-SYS and SYNCW-ILK are in effect for the VCPU, they act as logical "or" condition: VCPU is considered to be "in" SYNCW when any of these two constraints is currently in effect and is triggered for the current VCPU by actual execution, and is "out" of SYNCW only when both of these conditions are cleared. In other words, if both SYNCW-SYS and SYNCW-ILK are enabled by configuration, and CPU during its execution detects any entry-event that causes *either* of SYNCW-SYS or SYNCW-ILK to be entered, then VCPU is considered to be in composite SYNCW. VCPU is considered to be out of composite SYNCW only when it is out of *both* of SYNCW-SYS and SYNCW-ILK.

SYNCW-ILK implies window size constraint distinct from SYNCW-SYS. The size of SYNCW-ILK subwindow is limited to a fraction of 2,700,000 cycles, which derives from hardwired retry counter for interlocked queue instructions macros (900,000) multiplied by the busy-wait loop size (3 instructions).[44] VAX MP takes by default 66% of that reserve for SYNCW-ILK subwindow.[45] It is deemed that if high-IPL non-preemptable code takes an interlock on queue header with BBSSI instruction, 900,000 instruction cycles should suffice that code to complete the processing of locked queue and release the queue.

When SYNC-ILK is active, it behaves similarly to SYNCW-SYS in that VCPU performs periodic checks of its position in the window and enters wait state if it gets too much ahead of some other VCPU that is also "in" the window.

[44] See chapter "Implementation of VAX interlocked instructions" for more details. In addition, retry counter is limited by OpenVMS SYSGEN parameter LOCKRETRY with default value if 100,000. However VSMP LOAD command by default boosts LOCKRETRY to 900,000.

[45] Exact value is configurable with VSMP tool.

However SYNCW-ILK uses a different set of conditions that cause VCPU to enter or leave SYNCW-ILK synchronization subwindow:

- VCPU enters SYNCW-ILK when it executes interlocked instructions BBSSI or ADAWI at IPL >= RESCHED (3). Unless VCPU was already marked as active in SYNCW-ILK, this event will cause it to be marked as such and start position counter (unless the counter was already started by prior unexited SYNCW-SYS event, as explained below).

- VCPU enters SYNCW-ILK when it executes interlocked instructions (INSQHI, INSQTI, REMQHI, REMQTI) at IPL >=3 *and* instruction fails to acquire secondary interlock. Unless, of course, VCPU was already active in SYNCW-ILK.

- Any interlocked queue instruction (INSQHI, INSQTI, REMQHI, REMQHI) executed at any IPL and in any CPU mode (KESU) will cause VCPU to temporary enter SYNCW-ILK for the duration of this instruction, unless VCPU was already active in SYNCW-ILK.

- If VCPU's IPL drops below RESCHED (3), VCPU leaves SYNCW-ILK, i.e. is marked as non-active in SYNCW-ILK by clearing flag SYNCW_ILK in `syncw.cpu[].active`. If VCPU was also inactive in SYNCW-SYS, and thus have left composite synchronization window, all waiters on this VCPU are waked up.

- If VCPU executes OpenVMS idle loop iteration, it will also exit SYNCW-ILK.

- If OpenVMS executes process context switch (SVPCTX or LDPCTX instructions), VCPU will also exit SYNCW-ILK.

Since conditions and purposes for SYNCW-SYS and SYNCW-ILK are largely independent, it may at a first glance be tempting to handle them independently, i.e. processors in SYNCW-SYS constrain only other processors also in SYNCW-SYS but not in SYNCW-ILK, and vice versa, and keep two distinct scales for each processor's position within ILK and SYS sub-windows that will be started/stopped by respectively ILK or SYS conditions only, independently of each other. However this may cause a deadlock: two processors may be active in both ILK and SYS scales, and VCPU1 may get ahead of VCPU2 on ILK scale, while VCPU2 may get ahead of VCPU1 on SYS scale. Then VCPU1 will enter wait state for VCPU2 for the latter's progress on ILK scale, while VCPU2 will enter wait state for VCPU1 for the latter's progress on SYS scale. VCPU threads will then deadlock. More complex circular deadlock scenarios are also possible. Therefore, to avoid such deadlocks, VAX MP combines SYNCW-SYS and SYNCW-ILK into a composite SYNCW condition. SYNCW is started when *any* of SYNCW-SYS or SYNCW-ILK is signaled, and terminated when *both* of SYNCW-SYS and SYNCW-ILK are terminated. There is thus just only one "position" scale, and all VCPUs can be consistently ranked as being ahead or behind each other on this single scale. This eliminates the problem of deadlocking between subwindows.

Per-CPU state `syncw.cpu[].active` holds a combination of two bit-flags, SYNCW_SYS and SYNCW_ILK, that may be set or cleared independently, however processor is considered to be in composite SYNCW when *any* of these flags is set, and be out of composite SYNCW when *both* of them are cleared. There is just one scale for VCPU position in the composite SYNCW, rather than two separate scales for positions in SYNCW-SYS and SYNCW-ILK: position counting is started when processor enters composite SYNCW and is stopped when processor leaves composite SYNCW. While processor is in composite SYNCW, its forward progress is constrained by comparing its position along the composite SYNCW scale against other VCPUs positions on the composite SYNCW scale. The checking is done only among VCPUs that are active in the composite SYNCW, i.e. their `syncw.cpu[].active` have either SYNCW_SYS or SYNCW_ILK flags set.

Maximum drift allowed between the processors is thus determined *initially* as

vsmp_winsize = min(winsize_sys, winsize_ilk)

where:

winsize_sys = 66% of min(SMP_SPINWAIT, SMP_LNGSPINWAIT) * cpu_speed_calibration
winsize_ilk = 66% of 900,000 * 3

(66% is the default, adjustable with VSMP command line options)

Calculated value however needs to be further slashed in half. Here is why:

Consider syncw scale that limits maximum relative “drift” between the processors to SW. Suppose there are two processors A and B active on the scale. B initially is near the top of the scale and A is near the base of the scale.

```
top       |      B             ^
          |      |             |
          |      |             |
          |      |        (SW cycles)
          |      |             |
          |      |             |
base      A      |             v
```

Processor A performs interaction event (such as sending IPI to B or trying to acquire a spinlock held by B) and begins spin-waiting for B. B’s thread is preempted and makes no progress. Syncw scale will allow A to make forward progress until it gets “SW” cycles ahead of “B”:

```
top       A      |             ^
          |      |             |
          |      |             |
          |      |        (SW cycles)
          |      |             |
          |      |             |
base      |      B             v
```

Thus until A is suspended by syncw scale, it will be able to make forward progress by (2 * SW) cycles, which must not exceed window size value configured by VSMP: 2 * SW <= vsmp_winsize.

In other words, SW must be <= vsmp_winsize / 2.

To arrive to the value of maximum drift allowed between the processors, SW must be further reduced by:

- the value of (*one syncw scale quantum* + `INTERCPU_DELAY`), as described earlier in this chapter, to account for maximum possible latency between IPI or CLK interrupt is fielded to VCPU and target VCPU is entered into SYNCW-SYS;

- the value of (*one syncw scale quantum*), to account for the quant at the end of the scale, i.e. additional drift that can be accrued between the processors in between the successive invocations of `syncw_checkinterval`: the drift may not be allowed to get to the maximum closer than one scale quantum, since another check is not coming (in the guaranteed way) sooner than a quantum later after the preceding check;

- small extra safety margin to account for rounding effects during the calculations.

Another issue to be decided is the initial position of VCPU on syncw scale when VCPU enters synchronizaton window. Slashing syncw in half, as discussed above, makes the choice of initial position arbitrary from the standpoint of ensuring the safety of operations, i.e. prevention of false SMP timeouts. The choice of initial position in syncw, however, can matter somewhat from the standpoint of performance. If VCPU is always placed initially at the base of the scale (around minimum of current `syncw.cpu[].pos` of all other VCPUs currently "in" the window), that could potentially put a drag on forward progress of VCPUs in the higher part of the scale and cause them to enter syncw wait state too early. Likewise, if VCPU is placed initially at the top of the scale, that would put a drag on that VCPU's forward progress caused by VCPUs at the lower end of the scale. In the initial release of VAX MP, VCPU enters syncw (or is entered externally into syncw) at the lower end of the scale, however in the subsequent release it might be better to change this by placing VCPU initially at the average position of other VCPUs currently "in" the window.[46] However in practical terms this choice has minor significance in vast majority of cases, since in most cases VCPU running OpenVMS would not stay in syncw long enough to cause the drag on other VCPUs: typical number of instructions expected by OpenVMS code between entering syncw and leaving syncw in vast majority of cases can be expected to be much less than typical syncw size (or than maximum drift allowed by syncw). Typical value of *maxdrift* under default OpenVMS parameters is over 700,000 instruction cycles. Critical sections of OpenVMS code in vast majority cases are much shorter (in fact, most of them shorter than one quant of the scale, let alone *maxdrift*). Therefore in most cases syncw will cause VCPUs to be paused waiting for another VCPU only if that VCPU had been stalled due to reasons such as thread preemption or simulator blocking wait such as page fault. (This is consistent with the chosen syncw design goal to impose as little serialization and as little impact on scalability as possible.)

Another point of the design is whether maximum drift allowed should depend on VCPU membership in *particular* syncw subwindows or not. Since the sizes of SYNCW-SYS and SYNCW-ILK subwindows are different (under default configuration and typical modern host processor speeds, SYS subwindow is about 10 times longer than ILK subwindow), it may be tempting (but would be incorrect) to define VAX MP to use the most constraining drift limitation between two VCPUs that applies at the end of current quantum, i.e. at `syncw_checkinterval`, but not use the constraint that does not apply at the time. For example, it may (incorrectly) appear that when two VCPUs are active in SYNCW-SYS only, they should be constrained by SYNCW-SYS size only, but not by typically much shorter size of SYNCW-ILK and may be allowed to drift apart from each other by the number of cycles implied by SYNCW-SYS size, and beyond the constraint of SYNCW-ILK.

Drifting limit, according to this logics, would then be calculated according to the following table, depending on bit-flags set in the current VCPU's `syncw.cpu[].active` and `active` field for the VCPU being compared against. *w_sys* and *w_ilk* designate the sizes of SYNCW-SYS and SYNCW-ILK subwindows correspondingly:

[46] In cases when VCPU enters syncw as the result of IPI interrupt sent to it, it is possible to enter target VCPU at the position of the sender (interrupting VCPU) at the time of sending the interrupt by the latter, but generally syncw code and design is agnostic of the sources that cause VCPU to enter into syncw.

(note: table's implied would-be use is wrong!)

This VCPU's `syncw.cpu[].active`	Other VCPU's `syncw.cpu[].active`	effective window size
SYS, ILK	SYS, ILK	min(w_sys,w_ilk)
SYS, ILK	SYS	w_sys
SYS, ILK	ILK	w_ilk
SYS	SYS, ILK	w_sys
SYS	ILK	max(w_sys,w_ilk)
ILK	ILK, SYS	w_ilk
ILK	SYS	max(w_sys,w_ilk)

Such design however would be inherently incorrect, since if two VCPUs initially active in SYNCW-SYS but not SYNCW-ILK are allowed to drift apart from each other by (typically larger) *w_sys*, beyond the constraints of *w_ilk*, and subsequently one or both of them enter SYNCW-ILK, they turn out to be beyond the constraints of *w_ilk*, breaking the logics and constraints of synchronization window mechanism. For this reason, VAX MP always calculates maximum allowed drift based on *min(w_sys,w_ilk)* regardless of VCPU membership in particular subwindows.

The downside of this approach is that it over-constrains those parts of OpenVMS code that are actually safe within *w_sys* to a much shorter *w_ilk*, however this is inevitable with single composite syncw scale design, which in turn is the only way to avoid deadlocking between multiple independent scales.

In particular, since SYNCW-ILK typically has much smaller size than SYNCW-SYS[47], chosen approach over-constrains spinlock-holding code, which is the most common case in inter-CPU synchronization.

> As a side note, requirements of composite scale are not the only cause for over-constraining spinlock-holding code paths. There is also an additional cause:
>
> Entrance to SYNCW-ILK on BBSSI is required to synchronize code that locks interlocked queue headers with BBSSI against code branches that use INSQHI, REMQHI, INSQTI and REMQTI instructions to access the queue. Since VAX MP has no way of telling whether the target of BBSSI instruction is a queue header (that requires entrance into SYNCW-ILK) or spinlock or other synchronization flag (that do not require entrance into SYNCW-ILK and require only already pre-existing SYNCW-SYS)[48], VAX MP has therefore to force VCPU into SYNCW-ILK indiscriminately, since it does not know the semantics of target operand of BBSSI instruction. Thus, also due to described additional cause, most of spinlock holding code would have to be constrained not to SYNCW-SYS window whose size derives from SYSGEN parameter SMP_SPINWAIT, but to much more narrow SYNCW-ILK window.

[47] With default system settings and typical modern host CPU speed, about 10 times smaller.

[48] It is possible to inform VAX MP of the semantics of BBSSI and BBCCI operations in many common (but certainly not all) cases by paravirtualizing OpenVMS spinlock handling routines and some other common OpenVMS SMP synchronizaton routines. However at this point we want to avoid the amount of tampering with OpenVMS that would be required by that approach. But even if we did, spinlock-holding code would still have to be over-constrained to SYNCW-ILK due to the use of single composite scale.

However tightening down the effective size of composite SYNCW to the size of SYNCW-ILK is unlikely to be much of a problem in practice. SYNCW-ILK imposes requirement of 2,700,000 cycles before the timeout is triggered by VAX code. At 66% of this space delegated to syncw use, thus translates to *winsize_ilk* of about 2,000,000 instruction cycles, which in turn translates to *maxdrift* calculated to about 700,000 cycles. Vast majority of OpenVMS spinlock-holding code paths are going to be much shorter than 700,000 instructions, therefore VCPU will typically stay in synchronization window for only a very small fraction of 700,000 cycles before exiting the synchronization window, and thus even smaller SYNCW-ILK size will not be too constraining in practice. (However this assumption needs to be validated by experience actually using SYNCW under different workloads.)

Nevertheless if it were deemed that SYNCW-ILK size was a constraining factor influencing system performance, it may be possible to stretch SYNCW-ILK size upward to match SYNCW-SYS size, by using the following technique. Typical interlocked queue operation retry loop is structured the following way:

```
        INSQHI    OPERAND1, OPERAND2
        BCC       DONE
        CLRL      -(SP)
LOOP:
        INSQHI    OPERAND1, OPERAND2
        BCC       OK
        AOBLSS    #900000, (SP), LOOP
        BUG_CHECK BADQHDR, FATAL
OK:
        BLBC      (SP)+, DONE
DONE:
```

Three instructions highlighted in bold represent the retry loop. The size of this loop's each iteration is 3 instructions. It is possible to modify the implementation of interlocked queue instructions (INSQHI, INSQTI, REMQHI, REMQTI) in such a way that in case SIMH instruction implementation routine fails to acquire secondary interlock on the queue header, it retries the instruction *internally* multiple times before returning a falure. The following pseudocode illustrates internal structure of possible new instruction implementation:

```
QxI()
{
    repeat (N times)
    {
        execute Qxi_core()
        if queue header was not busy, return success (CC)
        advance CPU cycle counter
        decrement countdown of cycles remaining to syncw_checkinterval
        if result is low, return failure (CS)
        cpu_relax()
    }
    return failure (CS)
}
```

where `QxI` is the new implementation of interlocked queue instruction, and `QxI_core` is old (existing) implementation.

Some details are omitted: for example, on retries it would be better to use non-interlocked check first to see if the queue is still busy and call `QxI_core` only if it was no longer busy.

Another possible optimization is moving thread priority downgrade out of `QxI_core` to `QxI`, so thread priority change is performed just once inside `QxI`, rather than multiple times.

Also, to better the support of solitary non-retry queue status checks, it may be possible on the very first (non-loop) `QxI` invocation to return success/failure status after just one `QxI_core` invocation, not repeating it N times. Non-loop invocation (or very first invocation in the loop) can be recognized by detecting that there was no interlocked instruction failure within the last few instructions on this VCPU at this PC. This condition can be tracked by recording `QxI` failure's PC and cycle count in CPU_UNIT.

The result of this change is that retry loop instruction count effectively changes from 3 to (2 + N), and SYNCW-ILK window size can correspondingly be enlarged by (2 + N) / 3 times. By selecting sufficiently large value of N, SYNCW-ILK size can be made to match or exceed SYNCW-SYS, and no longer be an over-constraining factor compared to SYNCW-SYS.

However the need for such change is doubtful. SYNCW-ILK implied *maxdrift* of nearly 700,000 cycles represents a significant time (taking roughly about 30 microseconds on modern host processors not accounting for thread priority management, and much more accounting for the latter), and if VCPU thread finds itself spinning on queue header for that long, it may be better for it to yield host LCPU resources and wait for other VCPUs to clear the synchronization window and release the queue header.

If nevertheless such a change is ever implemented, additional checks need to be done before implementing it:

- OpenVMS code need to be double-checked to ensure that interlocked queue instructions are the only "culprit" and there are no retry loops in VMS code that timeout after 900,000 or LOCKRETRY cycles and re-attempt BBSSI or BBCCI, rather than interlocked queue instruction.

- Specifically, the use of LOCKRETRY in [CLIUTL.SRC]SHWCL_SNAP.MAR and [SYS.SRC] needs to be double-checked. Same for CHKQ and $QRETRY.

- The use of $QRETRY and QRETRY macros in [DRIVER.SRC]PEDRIVER.MAR need to be double-checked.

Use of synchronization window has implication for the scalability of the system. Every cross-over of IPL level QUEUEAST – and such events are common in OpenVMS – causes SYNCW event, causing VAX MP to acquire `cpu_database_lock` required to update `syncw` data. Other events that cause VAX MP to enter or leave SYNCW-SYS or SYNCW-ILK are common too. Thus it can be expected that when the use of synchronization window is enabled, `cpu_database_lock` will be a highly contended lock. Furthermore, each acquisition of this lock causes VCPU thread to elevate its priority to CRITICAL_VM and then demote it.[49] The bulk of overhead is due to the calls to enter and leave synchronizaton window (`syncw_enter` and `syncw_leave`), rather than due to quantum-end drift checks (`syncw_checkinterval` calls), since under default OpenVMS configuration parameters syncw quantum size is about 90,000 instruction cycles. Most OpenVMS critical sections are shorter than 90,000 instructions, and thus a majority of `syncw_enter` calls will be terminated by `syncw_leave` earlier than the end of the quantum, never incurring `syncw_checkinterval` in between.[50] Pessimistically assuming the cost of one `syncw_checkinterval` call to be equivalent to 30-60 regular VAX instuctions, each VCPU still spends less than 1/1000 of its time inside `syncw_checkinterval`, correspondingly its practical cost is low, and contention caused by it is low too.

On the other hand, the rate of `syncw_enter` and `syncw_leave` invocations can be quite high under heavy guest OS activity, but their execution time is very short, so in practice they should not cause too much of `cpu_database_lock` contention too.

Note further that one of SYNCW-ILK entrance conditions (namely, entering SYNCW-ILK when BBxxI or ADAWI instructions are executed at IPL >= RESCHED) means that spinlock-acquiring code will first enter SYNCW-SYS (due to elevation of IPL) and then almost immediately will also enter SYNCW-ILK (due to BBSSI at elevated IPL), locking and releasing `cpu_database_lock` twice in a rapid succession, so it could update field `syncw.cpu[].active` with flag SYNCW_SYS first time and then SYNCW_ILK second time – instead of locking `cpu_database_lock` just once. Given just how frequent this situation is, it might be worthwhile to consider the following change for future VAX MP release:

- Instead of keeping SYNCW_ILK and SYNCW_SYS flags in `syncw.cpu[].active`, keep them rather in per-CPU private storage (not protected by `cpu_database_lock`) in CPU_UNIT structure.

- Use `syncw.cpu[].active` to store only logical "or" of SYNCW_SYS and SYNCW_ILK, this way when one of these flags is set or cleared while another flag is present, no extra locking of `cpu_database_lock` will be required to change `syncw.cpu[].active`. VCPUs will be able to see only composite SYNCW conditions in any VCPU other than current, but not individual SYNCW_SYS and SYNCW_ILK conditions.

[49] One possible optimization that may be implemented later is that if VCPU thread is already at high levels of priority, such as OS_HI, elevation and demotion do not happen. This covers many, albeit not all cases of processing SYNCW events.

[50] Were it not the case, it would have been desirable to make the design of `syncw_checkinterval` more lockless. For example, instead of checking the drift every 100,000 cycles in locked mode, perhaps it might be possible to check it every 50,000 cycles in lockless mode and enter locked mode only if the drift is suspected to have become large – assuming it would have been possible to design position checking algorithm that is safe with respect to data races and also assuming required check interval was not too short compared to inter-processor memory updates propagation, i.e. latency of host machine's cache coherence protocol does not delay too much the visiblity of changes in global data when it is accessed in lock-free mode. However since `syncw_checkinterval` overhead is of minor importance compared to the overhead of `syncw_enter` and syncw_leave, there is no purpose in pursuing this line of improvements.

- This would reduce almost in half the number of calls to `syncw_enter` and `syncw_leave` in spinlock acquisition and release cases, by far the most common and frequent interprocessor synchronization operation. Required `cpu_database_lock` locking rate will be accordingly reduced.

- Described window membership-agnostic approach requires the use of *min(w_sys,w_ilk)* for effective window size. This will over-constrains those parts of OpenVMS code that are actually safe within *w_sys* to a much shorter *w_ilk*, however as explained above, this over-constraint is inevitable anyway because of fundamental single composite syncw scale design, and also because the semantics of BBSSI operand is unknown, and it is also unlikely to represent a performance problem.

- However the difficulty of implementing this design is combining it with entering VCPU into syncw from the outside of VCPU context as a consequence of IPI or CLK sent, by other processors performing the scan in `syncw_checkinterval`. Perhaps this difficulty can be resolved by storing both flags in interlocked cells. Or perhaps the whole design for entering VCPU into syncw externally can be reviewed (with perhaps instead entering VCPU into syncw by IPI/CLK sender).

It may also be worthwhile to consider for next VAX MP release a conversion from NCPU per-CPU position scales to NCPU x NCPU relationship. This should improve scalability for cases when VCPU or multiple VCPUs enter and exit synchronization window while other VCPUs stay in synchronization window and are at opposite sides of the scale. With presently implemented version of NCPU scales approach, when VCPU enters synchronization window, it enters the window at initial position of most lagging behind of all other VCPUs in the window; it might be thus a drag on all other VCPUs also active in the window – in most cases, unnecessary drag. It may be better to keep NCPU x NCPU scales and at entry time record position of every other VCPU in the window and subsequently use deltas against this initial position. This would be less constraining, but maintenance on NCPU x NCPU scales would be more expensive. It is however uncertain (and, indeed, at the moment looks improbable) whether VCPUs would actually be a drag on each other under common circumstances, i.e. when VCPU threads are executed without preemption by host OS and other stalls such as caused by page faults. As discussed above, it is expected that the stay of VCPUs with non-preempted threads in synchronization window will be very short, typically under one quantum of scale and certainly much less than *maxdrift*, so in vast majority of cases syncw is not going to effect a drag on running VCPUs as long as no VCPUs “in” the window are preempted or blocked from execution by page faults or other host system level blocking events, so described possible improvement is likely to be not worthwhile. Experience using synchronization window under different workloads, along with monitoring performance counters, may provide a more definitive guidance.

Mapping memory consistency model

One of the most critical and intricate aspects of mapping shared-memory virtual multiprocessor to a host multiprocessing system with a different hardware architecture is the mapping of shared memory consistency model. In case of significant mismatch, such as mapping multiprocessor system that relies heavily on strong memory consistency model to a host system with weak consistency model, such a mapping can be impossible in practical terms, since memory synchronization between VCPUs would have to be executed often, essentially after every instruction, with devastating results for the performance.

Do note however that what matters is not only actual memory model of virtualized system, but also the way its software deals with it: while actual VAXen may have relatively strong memory model compared to some of contemporary systems such as IA64, but VAX and VMS software coding standards limit assumptions that software can make about VAX memory model.

Yet, whether actual code follows coding standards or breaks them is yet another helix of the issue.

It is beyond the scope of this document to discuss memory-consistency model notions or functioning of memory barriers. For general overview of memory-consistency model problem and summary information on specific processors see a reference list below[51].

In a nutshell nevertheless, when CPU1 executes instruction to update memory location, the update is propagated to CPU2 via approximately the following path (described here with some simplifications):

$$\text{CPU1} \rightarrow \text{SB1} \rightarrow \text{interprocessor interconnect} \rightarrow \text{INVQ2} \rightarrow \text{CACHE2} \rightarrow \text{CPU2}$$

where SB1 is a Store Buffer for processor 1, INVQ2 is cache invalidation message queue for processor 2, and CACHE2 is cache memory for processor 2. Cache for CPU1 and its interaction with SB1 is not depicted.

Processors try to keep caches coherent, but this coherence is not achieved instantly, and memory update notification takes time to propagate between the processors. When CPU1 changes memory cell, CPU2 will see the change only *eventually*. For some relatively extended time after CPU1 issued memory update instruction, CPU2 may keep seeing the old value of the cell.

Furthermore, although changes made by CPU1 will eventually become visible on CPU2 and other CPUs in the system, there is no guarantee that they will become apparent on CPU2 in the same order they were executed on CPU1.

Unless special synchronization is used in the software that accesses shared memory, writes and reads

[51] Appendix A.

can be reordered by processors leading to CPU2 seeing updates coming from CPU1 out of order compared to how they were perceived by the program executing on CPU1. This reordering is executed for performance enhancement reasons and may be caused by mechanisms like:

- out-of-order instruction issue by CPU1 (and CPU2), with the processor reordering the instructions for parallel execution or execution ahead, assuming no logical dependency between physically disjoint data;
- the order of memory accesses in particular can be rearranged to promote better use of the CPU buses and caches, for example by:
- write combining, i.e. combining writes to the nearby cells to one longer write;
- read combining;
- write collapsing, i.e. eliminating write if followed by another write to the same memory location;
- loads can be performed speculatively , leading to data being fetched at the "wrong" time from a program's standpoint (unless the program took care to issue memory barriers to ensure ordering of access to data);
- buffered writes can be passed by reads, since writes can wait and read data is needed ASAP for instruction execution;
- splitting cache into multiple banks (such as for even and odd addresses) working asynchronously to each other; if some bank is more loaded than others, updates to this bank will be committed later than to other bank or banks; therefore even if update A arrives to INVQ ahead of update B, it can be committed to CACHE2 after B;
- and so on.

Thus memory updates can be executed out of order compared to program sequence; pushed out to interconnect out of order compared with their execution; and processed by receiving processor (CPU2 in the example) out of order compared to their delivery to CPU2.

To serialize update sequence across the processors, and for memory updates to appear sequenced on the receiving side in the same way a producing side software is executing them, a special synchronization is required in the form of *write memory barrier* on the data writer side and *read memory barrier* on the receiving side.

Hereafter, we designate write memory barrier as WMB, read memory barrier as RMB and full memory barrier as MB.[52]

[52] There is also a weaker form of read memory barrier called *data dependency barrier*, that will not concern us here. In a nutshell, data dependency barrier places read mark in INVQ past any update messages in there for data that had been *already* touched by preceding reads. All invalidation messages before this mark are guaranteed to complete before any other read operation is allowed to execute. Thus, dependency barrier places the mark somewhere in the middle of the INVQ whereas RMB places it at the entrance of the INVQ. Hence it takes shorter time for dependency barrier to "pump through" than for general-purpose RMB, but dependency barrier does not cover more general range of memory consistency issues VAX MP has to deal with.

It is beyond the scope of this document to describe functioning and exact semantics of memory barriers.[53] We will provide here only a very basic and simplified description, that nevertheless will be sufficient for the purposes of VAX MP implementation.

Memory-consistency models utilized by most commonly used CPUs imply that:

- caches are kept coherent between processors through inter-processor cache coherency protocol; this protocol takes time to propagate changes and, by itself, does not guarantee that sequencing of changes will be retained;

- explaining purposefully simplistically and generally speaking incorrectly, WMB flushes pending store buffers of the local CPU to the interconnect making memory changes executed by the current processor prior to execution of WMB visible to other processors bus interfaces (but not to the programs running on those processors yet! only to other processors INVQs!), by sending cache invalidation messages to them;

  more precisely, WMB *sequences* writes by placing a mark in the store buffer of local processor indicating that writes executed in a program sequence after this barrier mark cannot be reordered with writes executed before the mark and cannot be flushed out to other processors (i.e. onto the interconnect) and made visible to outside processors ahead of the writes that preceded the mark;

  note however that it may take a while for other processors to process these messages and merge received updates into their caches; hence the need for RMB;

- explaining purposefully simplistically and generally speaking incorrectly, RMB stalls current processor (instruction stream execution) until all cache invalidation messages that arrived from other processors before the RMB instruction was issued are handled by local processor and corresponding updates are committed to the cache of the local processor;

  more precisely, RMB *sequences* reads by placing a mark in the load buffers and/or incoming invalidate message queue of the local processor indicating that reads executed in a program sequence after this barrier mark cannot be reordered with reads executed before the mark and hence these reads cannot be executed until previous reads complete;

- MB is a *super*-combination of WMB and RMB[54]. It sequences both reads and writes by placing a mark in the queues that separates load and store requests issued before the mark from the requests issued after the mark.

[53] Interested reader can refer to bibliography in appendix A. Also see 200-page fascinating discussion thread between Paul E. McKenney, Alan Stern and few other participants titled “Uses for memory barriers” in Linux kernel mailing list archives for September-October 2006.

VAX MP assumes and uses only the most basic feature of memory barriers, the sequencing of updates between the processors:

- if CPU1 executes sequence S1 `{write A=A1; WMB; write B=B1}`
- and if CPU2 executes sequence S2 `{ read B; RMB; read A}`
- and if CPU2 during "read B" observed the results of write to B by CPU1/S1
- then "read A" in S2 executed by CPU2 will observe the results of write to A by CPU1/S1

Note that coupling between WMB and RMB is crucial. WMB and RMB are local operations, taken separately they perform sequencing only within the current processor and do not impose sequencing within another processor or processors. Only a paired combination of WMB and RMB executed correspondingly on producer (writer) and consumer (reader) processors ensures proper sequencing of memory updates communication between the processors.

Also notice that propagation of updates to B in the above example is only *eventual*. After CPU1 executes S1 update to B, it will take time[55] before CPU2 notices an updated value of B. For some time CPU2, when referencing memory location of variable B, will see its previous value (such as B0). Hundreds of CPU2 cycles may pass before the circuitry makes updated value of B (i.e. B1) visible to a program executing on CPU2. Furthermore, memory updates become visible to different processors at different time. If CPU2 sees an update, it does not mean CPU3 already sees it too.

To reiterate once again, unpaired WMB and RMB taken by themselves are *strictly local* operations. All they do is *local sequencing*. WMB does *not* cause memory updates to be instantaneously pushed to other processors. RMB does *not* cause current processor to instantaneously fetch pending updates from other processors.[56]

* * *

When speaking about VAX memory-consistency model, a distinction should be made between:

[54] Note the *super* part! MB is stronger than a simple combination WMB + RMB placed straightforwardly in the program code. The latter combination guarantees that any preceding read will be sequenced before any following read, and that any preceding write will be sequenced after the following write, however it does not guarantee that any preceding read will be sequenced before any following write, whereas MB does guarantee that.

For example, in sequence `{ load A; store B; rmb; wmb; load C; store D }` other CPUs might see the store to D preceding the load from A, or, conversely, the store to B following the load from C.

Whereas in sequence `{ load A; store B; mb; load C; store D }` other CPUs will always see accesses to A and B as preceding accesses to C and D.

[55] Unpredictable amount of time, perhaps few hundred or even few thousand CPU cycles.

[56] Exact semantics and functionality of memory barrier instructions is machine-dependent. We are talking here of the basic platform-independent functionality of memory barrier primitives.

- Memory consistency models for various actually existing released hardware systems in the VAX family (and also DEC VVAX, internal DEC virtual VAX machine).

- Memory consistency model defined by VAX Architecture Standard as the lowest common denominator for all existing VAX systems and any future intended VAX systems, at the time of writing the Standard.

- Assumptions about memory consistency model made by actually existing software, and the pattern of reliance on memory consistency model exercised by the software (OpenVMS, Unix, applications).

- Veering of actually existing software from the model defined by VAX Architecture Standard and reliance on stronger memory-consistency models of historically existing VAX systems, beyond the guarantees of the model defined by VAX Architecture Standard.

While memory consistency model for VAX was not well-defined in public technical documents [57], there is a somewhat (albeit not fully) sufficient description in the internal document titled "DEC STD O32 VAX Architecture Standard"[58]. Parts of VAX memory consistency model, as defined by the VAX Architecture Standard, that are most relevant for VAX MP emulation are summarized below.

Summary of VAX memory-consistency model as defined by the VAX Architecture Standard:

VAX CPU caches are coherent between CPUs and IO devices as follows:

- An IO transfer from memory to a peripheral device started after a program write to the same memory, will output the updated memory value.
- A program memory read executed after the completion of an IO transfer from a peripheral to the same memory, must read the updated memory value.

Memory modifications by a processor are guaranteed to be propagated to other processors and their caches (in modern parlance, store buffers flushed and write memory barrier WMB executed) under the following conditions:

[57] As complained already in Kourosh Gharachorloo, "Memory Consistency Models for Shared-Memory Multiprocessors", DEC WRL Research Report 95/9, pp. 79, 278. See a copy in *aux_docs* subdirectory or at http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-9.pdf. For another discussion of ambiguities in VAX memory model and their interpretation, albeit of little practical use for the purposes of VAX MP project, see S. Adve, J. Aggarwal, "A Unified Formalization of Four Shared-Memory Models", Technical Report CS-1051, University of Wisconsin, Madison, 1992, abbreviated version also in IEEE Transactions on Parallel and Distributed Systems, June 1993.

[58] "DEC STD O32 VAX Architecture Standard", EL-00032-00 (1990 edition), Digital internal document. See a copy in *aux_docs* subdirectory or at http://bitsavers.org/pdf/dec/vax/archSpec/EL-00032-00-decStd32_Jan90.pdf. The most relevant part is chapter 7, "Memory and I/O".

- In a multiprocessor VAX system, software access to all memory areas shared between processors must be interlocked by one of the interlocked instructions (BBSSI, BBCCI, ADAWI, INSQHI, INSQTI, REMQHI, REMQTI). Important specification of these instructions is that if their operand is in memory (rather than a register or IO space), they perform full write memory barrier (WMB) on the *whole* memory, not just on their immediate operands. In addition, of course, interlocked instructions guarantee atomic read-modify-write cycle on their operand.
- If a processor writes and modifies memory and then executes HALT instruction or stop, changes become visible to other processors, i.e. HALT and error stop execute WMB.
- Any interrupt request from a processor to another processor.
- Any interrupt request from an IO device to a processor.
- Processor entry to console mode as a result of a HALT or on error halt.
- Any console function that changes memory state.
- Any processor write operation to an IO space.
- Since some QBus, Unibus and VAXBI devices begin transfer when their register is accessed for read, therefore processor read from IO space must also execute WMB.

Memory read and write operations are atomic on:

- byte-sized operands
- word-sized operands aligned in memory on a word boundary
- longword-sized operands aligned in memory on a longword boundary
- bit fields contained within one byte[59]

[59] This latter guarantee does not mean that bit fields are accessed in an interlocked fashion, but simply that when multiple concurrent updates (INSV instructions) are performed to a bit field residing in a single byte, one of concurrent instructions is going to win wholly and the resultant value field won't contain values of different field bits coming from different concurrent INSV instructions, but rather all values coming from single (most latest, winning) INSV instruction. This is essentially just a re-iteration of atomic byte-level write guarantee expressed for INSV instruction. VAX architecture standard however provides no write atomicity guarantee for a bit field spanning multiple bytes.

"VAX MACRO and Instruction Set Reference Manual" provides the following comment for INSV instruction:

> *"When executing INSV, a processor may read in the entire aligned longword or longwords that contains the field, replace the field portion of the aligned longword with the source operand, and write back the entire aligned longword. Because of this, data written to the nonfield portion of the aligned longword in memory by another processor or I/O device during the execution of INSV may be written over when the INSV is completed."*

"VAX Architecture Reference Manual" (ed. Richard A. Brunner, 2nd edition, Digital Press, 1991, p. 73) similarly explains:

> *"When executing INSV, a processor may read in the entire aligned longword(s) that contains the field, replace the field portion of the aligned longword(s) with the source operand, and write back the entire longword(s). Because of this, data written to the non-field portion of the aligned longword(s) in memory by another processor or I/O device during the execution of the INSV may be written over when the INSV is completed."*

Access to all other operands is not guaranteed to be atomic. Specifically, access to operands that are not naturally aligned, access to quadwords, to octawords, character strings, packed decimal strings and bit fields not contained within one byte is not guaranteed to be atomic.

Adjacent atomically writable elements can be written or modified independently. For example, if a processor executes INCB on a byte at address *x*, and another processor executes INCB on the byte at address *x+1*, then regardless of the order of execution, including simultaneous execution, both bytes will be incremented by 1 from their original values.[60]

Instructions that perform read-modify operations, other than special interlocked instructions, do not guarantee atomicity of a read-modify cycle. Read and write parts of an instruction that modifies its operand are two separate accesses to the same address; unless they are a part of an interlocked instruction, another processor or IO device can access the operand between the read access and the write access, even if both are atomic. Moreover, any required or permitted memory operand reference (read, write or modify) may be performed by an instruction more than once, except for references to IO space which are made exactly once. Attempt to access the same cell concurrently with non-interlocked instructions executed in parallel on different processors results in an undefined and unpredictable behavior, except for atomicity guarantees mentioned above.

If a processor accesses a variable with an interlocked instruction and another processor accesses the same memory location with a non-interlocked instruction, the results are unpredictable.

Section of the VAX Architecture Standard on "The ordering of Reads, Writes and Interrupts" unfortunately is left blank (it just says "This section will be added by a later Engineering Change Order"), but the tenor of the whole chapter leaves it clear that the intention is to provide weak guarantees on ordering and require the software to perform memory access synchronization via interlocked instructions or one of the other memory-synchronizing signals described above, and not otherwise rely on any implied ordering.

VAX memory-consistency model as described by VAX Architecture Standard is obviously underspecified: it speaks when WMB is performed, but does not speak about RMB and leaves it undefined, tending to coalesce the two together. Technical manuals for VAX 6000 and 7000 multiprocessor system suffer from the same omission of not describing read-side synchronization within the processor[61]. VAX Architecture

[60] For explanation of atomicity of access and its guarantees by VAX see also "VAX/VMS Internals and Data Structures" (version 5.2), chapter 8.1.1 ("Synchronizaton at the hardware level").

Few examples of locations in OpenVMS kernel that rely on atomicity of access to them: CPU$L_MPSYNCH, EXE$GL_ABSTIM_TICS, CPU$L_NULLCPU.

[61] VAX Architecture Standard simply says: "The maximum delay from a program issuing a [WMB] signal until the signal takes effect in the other processors is machine-dependent" (ch. 7.1.1).

Standard model also says nothing about transitive properties of the model when more than two processors are involved.[62]

VAX MP implementation has to take care of these omission based on synchronization methods and patterns actually employed by OpenVMS source code.

So, how do we implement this model in VAX MP?

- Perhaps at a glance the easiest and most straightforward part to implement is VAX interlocked instructions (ADAWI, BBSSI, BBCCI, INSQHI, INSQTI, REMQHI, REMQTI). These instructions can be implemented in a portable way by acquiring a VM-level lock (from a hashed set of locks) when simulating the instruction and executing appropriate memory barriers before and after. They can also be simulated in a host-specific way with host interlocked instructions whenever those are available and powerful enough.

  Yet notice how we said "at a glance": there are more issues here than meet the eye at the first moment, and simulation of VAX interlocked instructions, along with those issues, is discussed in more details below in the chapter "Implementation of VAX interlocked instructions".

- HALT instruction and error stops generate WMB.

- When a processor sends an inter-processor interrupt, VAX MP generates WMB.

- When a processor receives an inter-processor interrupt, VAX MP generates RMB.

- When an IO device completes the transfer or changes its register state and sends an interrupt to a virtual processor other than selected for execution in a current working thread, the device executes WMB.

- When an interrupt is received from an IO device by a virtual processor, VAX MP generates RMB, unless the device handler[63] is known to have been executing within the current thread.

- While it may be possible to execute WMB on each and every virtual processor's access to the IO space (SIMH routines `ReadIO`/`WriteIO` and `ReadReg`/`WriteReg`), besides coming at a cost, it would have made little sense per se, since the device IO initiation code is executed in the context of the current virtual processor's thread anyway.

[62] Thus the software following the model has to rely on locking if it wants to implement safe interaction in the transitive case.

[63] Hereafter we use term *device handler* to refer to SIMH code that implements virtual device, as opposed to guest OS *device driver* (implemented as VAX executable code) that interacts with virtual device represented and simulated by *device handler*.

Most per-CPU devices will not need memory barrier at all, since they operate entirely (or almost entirely) within the context of local virtual processor, and hence do not require memory barrier.

Most system-wide devices (i.e. non-per-CPU devices) will perform writing to their internal data structures in their CSR *xx_rd*/*xx_wr* routines that will, hence, most likely require eventual execution of another memory barrier[64] that can be combined with any master memory barrier to be executed on IO space access, therefore there is usually no need for double execution of memory barrier by generic IO space access routines first and then by the device's *xx_rd* or *xx_wr* routines afterwards. Hence, in the case of system-wide devices, memory barrier at IO space access level would have been redundant either.

Therefore, instead of trying to impose indiscriminate WMB for every device on every IO space access, and thus hampering performance, VAX MP leaves it up to device handler to decide when memory barrier should (or should not) be executed.

See more details below in the section "Design notes and changes to SIMH codebase".

- On entry to SIMH console mode, every virtual processor thread generates WMB and console thread executes RMB. On resumption of execution, console thread executes WMB and all VCPU threads execute RMB.

- Target x86 and x64 architectures guarantee atomic access to naturally aligned byte, word and longword (x86/x64 dword) sized operands of the instructions, identically to guarantees by VAX architecture[65].

  Independence of writes to adjacent naturally aligned elements (thus each falling entirely within a cache line) is guaranteed on x86/x64 by a combination of cache coherency protocol and atomic access guarantee.

- One aspect ignored by VAX Architecture Standard and actual VMS code is timely propagation of updates to SCB to other processors.

  Suppose CPU1 changes a vector in SCB shared by all processors, and interrupt occurs immediately after SCB vector change, with interrupt being delivered to CPU0. One scenario (albeit not relevant for VMS) would be CPU1 configuring a device and setting the vector for it in

[64] Usually when acquiring a lock on device handler's internal structures.

[65] See "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A & 3B): System Programming Guide", P/N 325384-039US, section 8.1.1, "Guaranteed Atomic Operations". For earlier processors starting with i386 see "Intel Architecture Software Developer's Manual, Volume 3: System Programming", P/N 243192, section 7.1.1.

SCB, with device immediately triggering an interrupt dispatched to CPU0, that may not have yet received an updated view of SCB delivered by cache coherency protocol.[66] Another scenario would be a kernel debugger (such as OpenVMS XDELTA) running on CPU1 changing the SCB and an exception or trap occurring immediately afterwards on another CPU.

To ensure timely propagation of updates to SCB to other virtual processors even on host systems with weaker memory-consistency model, VAX MP monitors memory writes against SCB address range. When VAX MP detects a write to SCB being performed on the current VCPU, it performs WMB and broadcasts to all other running VCPUs a request to perform RMB on them. This request is dispatched as non-maskable interrupt (internal to VAX MP and is not directly visible at VAX level) and is processed with virtually no delay on the target processors, and in any event before VAX MP interrupt handler fetches SCB vector for processing of any "real" interrupt.

- Special case is the update of PTE "modified" bit performed by VAX processor when memory paged is being written to and PTE "modified" bit is not set in the copy of PTE maintained in the TLB.

  VAX Architecture Standard states: "The update of PTE<M> is not interlocked in a multiprocessor system" (ch. 4.3.1).

  When VAX MP performs write of PTE with modified bit set back to page table in main memory, VAX MP issues WMB to ensure that PTE<M> gets set in main-memory copy of the PTE before any actual update to page content takes place.

  We can expect that VMS will handle this update correctly on read side since the latency in propagation of PTE<M> update through cache coherence protocol is not fundamentally distinguishable from the effects of asynchronicity between instruction streams executing on different processors, so to cover for the latter VMS should already have in place proper logics to ensure synchronization between PTE<M> writers and PTE<M> readers.

In addition, the following table summarizes memory access reordering that can be performed by various popular processors. The table is from an article by Paul McKenney[67], leaving only the columns for several most popular CPUs in it, and adding the column for VAX. As mentioned above, the section on "The ordering of Reads, Writes and Interrupts" in the available copy of VAX Architecture Standard was

[66] This scenario is not relevant for OpenVMS, as SYSGEN loads the drivers and connects them to SCB interrupt vectors after acquiring first an affinity to the primary processor. All device interrupts in OpenVMS VAX are also serviced on the primary processor.

[67] Paul E. McKenney, "Memory Barriers: a Hardware View for Software Hackers", at http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2009.04.05a.pdf. x86_32 OOStore is Linux designation for some non-Intel x86 clone processors, such as from Cyrix and Centaur, that unlike Intel performed store reordering.

left unfilled, so the values in VAX column represent an interpretation of the standard by present author based on reading the whole chapter that contains the section in question. They do not reflect actual reordering by any particular VAX model, but only more lax general Architecture Standard that thoroughly emphasizes the need for software to use interlocked instructions and other listed means of synchronization that effectively implement paired memory barrier, i.e. WMB on the updating CPU or IO device side and implied RMB on the consumer side. Actual VAX processors may (and many certainly do) implement stronger memory consistency model than suggested by the Architecture Standard[68], and there is a chance that some of OpenVMS code may contain implicit reliance on this stronger memory model, since the system was never ran and tested on VAX computers with ordering truly as weak as suggested by the Architecture Standard, and hence trying to run the simulator on host systems with weaker ordering may trigger these dormant issues[69].

	VAX	Alpha	x86_32	x86_32 OOStore	x86_64	IA-64
Loads reordered after Loads	Y	Y	Y	Y	Y	Y
Loads reordered after Stores	Y	Y	Y	Y		Y
Stores reordered after Stores	Y	Y		Y		Y
Stores reordered after Loads	Y	Y	Y	Y	Y	Y
Atomic reordered with Loads	(1)	Y				Y
Atomic reordered with Stores	(1)	Y				Y
Dependent Loads reordered	?	Y				
Incoherent Instruction cache pipeline	n/a (2)	Y	Y	Y		Y

(1) NO for interlocked atomic instructions. YES for writing bytes and naturally-aligned words and

[68] "Before Alpha AXP implementations, many VAX implementations avoided pipelined writes to main memory, multibank caches, write-buffer bypassing, routing networks, crossbar memory interconnect, etc., to preserve strict read/write ordering." (Richard L. Sites, "Alpha AXP Architecture", Digital Technical Journal, 1992, № 4).

Goodman observes that VAX 8800 follows *processor consistency* model (James R. Goodman, "Cache Consistency and Sequential Consistency", Computer Science Department, University of Wisconsin, Madison, Tech Report #1006, Feb 1991, a replica of IEEE Scalable Coherent Interface (SCI) Working Group TR 61, March 1989), on the basis of J. Fu et. al, "Aspects of the VAX 8800 C Box Design", Digital Technical Journal, № 4, Feb 1987, pp. 41-51.

Processor consistency model means that processor maintains ordering of loads/loads, stores/stores and also loads/stores for loads preceding stores (i.e. these operations appear on interprocessor interconnect in the same order as in the executed program), however processor can reorder stores/loads for stores preceding loads, in order to partially hide the latency of store operations. In the latter case processor can execute load operation even if preceding store had not been completed yet in case there are no dependencies. Thus a load operation can be performed even if the effects of store operation are not visible yet to other processors. A close variant of processor consistency model is also known as TSO. (For more detailed explanation of *processor consistency* model see Thomas Rauber, Gudula Rünger, "Parallel Programming for Multicore and Cluster Systems", Springer, 2010, p. 87.)

[69] One might have hoped that weak ordering in Alpha processors could help to detect these dormant issues, but VAX and Alpha versions of OpenVMS actually were different code bases. See Clair Grant, "Porting OpenVMS to HP Integrity Servers", OpenVMS Technical Journal, vol. 6 at http://h71000.www7.hp.com/openvms/journal/toc.html and also actual source listings for both VAX and Alpha/Itanium codebases of OpenVMS.

longwords.

(2) SIMH retrieves all VAX instructions from SIMH data memory, not from SIMH instruction stream.

Table above, by its very nature, captures only few aspects of memory-consistency model. For more thorough description of memory-consistency model of particular host system refer to the documentation on this system. For Intel x86 and x64 platforms, refer primarily to the articles on x86-TSO model by Sewell, Owens et. al. listed in Appendix A and also Intel note 318147-001 "Intel 64 Architecture Memory Ordering White Paper" (August 2007).

One known specific area where VAX operating system software does indeed rely on stronger memory model is drivers for devices that use shared memory area for communication between the driver and the device. For QBus based MicroVAX this includes Ethernet controllers (DEQNA, DELQA and DELQA-T) and the driver for them (VMS XQDRIVER)[70]. It also includes UQSSP port devices (MSCP and TMSCP controllers) and VMS driver for UQSSP ports (PUDRIVER)[71]. XQDRIVER and PUDRIVER assume that read/write accesses to the shared area are ordered, and that memory accesses to the data buffers are also ordered relative to the accesses to the shared area. This ordering assumption holds true on real silicon VAXen because of their strong memory model, but it does not hold true under a simulator running on a modern processor with weaker memory model. To resolve this problem, VAX MP introduces two sets of matching memory barriers to provide appropriate synchronization between the device's driver in the operating system and device's handler implementing virtual device in the simulator. One set of barriers is provided within SIMH virtual device handlers implementing XQ, RQ and TQ virtual devices. The other set of handlers is implemented as dynamic patches applied to the running copy of PUDRIVER and XQDRIVER loaded into operating system memory and inserting required memory barriers into the running driver code.

Compiler barriers

In addition to memory access reordering by computer hardware, reordering can also be performed by the compiler trying to achieve optimizing code transformation. Compiler normally assumes that updates to global variables (as well as any other variables exposed via pointers or other references and having external accessibility outside of the synchronous flow of execution) cannot be performed asynchronously by another thread. "Not under my feet!", thinks the compiler. Hence the compiler can load such a variable in a register and keep using its "stale" value, unaware that the variable might have actually been updated by another thread in the meanwhile. There are two ways to make compiler to drop the assumption on "cacheability" of a variable in a compiler-managed local storage. One is to declare the variable *volatile*, which makes the compiler to avoid caching it in a register or on-stack storage, and rather read it directly from memory and write it back directly to memory every time the variable is updated. Another approach is to use compiler barriers. Calling any function whose body

[70] DEQNA/DELQA shared area used for communication with host contains TX BDL and RX BDL lists.

[71] UQSSP port shared area used for communication with host contains COMM area, command ring and response ring.

(code) is unknown to the compiler will force the compiler to assume that the function might have changed all global variables.

For older compilers, any external function would serve as a compiler barrier. However modern compilers, such as recent versions of GCC and Microsoft Visual Studio, perform project-wide global compilation and optimization, with some parts of it being carried out as late as during linking stage. You cannot trick such a compiler into execution of compiler barrier just by calling an external function with empty body located in one of the project's files: compiler is likely to optimize such call away and assume no barrier had been performed. For this reasons, modern compilers (such as current versions of MSVC and GNUC) provide special intrinsics that force memory barrier.

VAX MP code uses a function for compiler barrier called `barrier()`, that internally contains such an intrinsic.

In addition, all memory barrier primitives (`smp_wmb()`, `smp_rmb()` and `smp_mb()`) are also made to double as compiler barriers in VAX MP. The latter was a design decision in order to avoid having to declare plenty of global variables *volatile*, as well as various direct and indirect pointers to such variables, and also to avoid bugs when overlooking to do so. This decision, admittedly, affects in minor ways performance of compiler optimization for relevant sections of VAX MP code, as it invalidates compiler's knowledge of *all* shared variables, but present author felt it was better to take a safer approach for the current implementation, at least in the initial version. This decision can be revisited later if desired, with memory barrier primitives possibly losing the status of compiler barriers and instead individual relevant variables and some pointers to them declared *volatile*; but there are also strong arguments against it, as doing so would prevent the compiler from optimizing memory access to the variables in question *within* the critical section. [72] This is a matter of balance, and the balance is unclear a priori, without more detailed examination.

For more extensive discussion of compiler barriers see the references in the footnote.[73]

[72] See file "Why the "volatile" type class should not be used" in Linux kernel distribution, file Documentation/volatile-considered-harmful.txt. Also see "Linux: Volatile Superstition" at http://kerneltrap.org/Linux/Volatile_Superstition.

[73] "Handling Memory Ordering in Multithreaded Applications with Oracle Solaris Studio 12 Update 2: Part 1, Compiler Barriers", Oracle, 2010, at http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-compiler-barriers176055.pdf; Hans-J. Boehm, "Threads Cannot be Implemented as a Library", at http://www.hpl.hp.com/techreports/images/pdf.gif, also slides at http:://www.hpl.hp.com/personal/Hans_Boehm/misc_slides/pldi05_threads.pdf.

Implementation of VAX interlocked instructions

VAX interlocked instructions (ADAWI, BBSSI, BBCCI, INSQHI, INSQTI, REMQHI, REMQTI) make a key part of VAX inter-processor synchronization.

When executing these instructions, real hardware VAX processor asserts interlock ("primary interlock") signal on the bus to ensure atomicity of read-modify-write cycle on the operand.

Interlocked queue instructions {INS|REM}{QHI|QTI} additionally utilize secondary interlock kept in the lowest bit of the queue header operand. These instructions begin by setting secondary interlock bit under the protection of the primary interlock, in a manner similar to BBSSI. If they find secondary interlock was already set, they terminate with condition code (C bit in PSL set) indicating the queue was busy and thus the caller must retry executing the instruction. If they are able to successfully assert secondary interlock, they release primary interlock at this point and proceed with non-interlocked memory accesses to the queue structure (or rather proceed with primary interlock released, but under logical protection of secondary interlock). Once queue structure had been modified, they clear secondary interlock bit in a manner similar to BBCCI.

Although interlocked instructions can be functionally categorized into three subsets

- ADAWI
- BBSSI, BBCCI
- INSQHI, INSQTI, REMQHI, REMQTI

these subsets do not make independent locking domains: data entity can be accessed by instructions from more than one subset and these accesses are expected to be mutually interlocked. For example, some interlocked queues are accessed in OpenVMS code with both {INS|REM}{QHI|QTI} and BBSSI/BBCCI:

> For instance, non-paged pool lookaside queues (from the array of queues holding pre-allocated blocks of different sizes, according to the index into the array) pointed by EXE$AR_NPOOL_DATA can have their headers temporary interlocked with BBSSI/BBCCI for entry counting by routine COUNT_LIST_PACKETS callable by DCL SHOW MEMORY command and SYS$GETSYI system service. While the header remains locked, memory blocks of corresponding size are allocated from general pool (via EXE$ALONONPAGED) rather than from the lookaside list.
>
> Likewise queue header EXE$GQ_SNAP_FORK_LIST in [SNAPSHOT.SRC] is locked with BBSSI and subsequently unlocked by BBCCI by routines that search the queue or count entries in it.
>
> Also some SCS port drivers in [DRIVER.SRC] may use BBSSI and BBCCI instructions (see macro CHKQ) on queue headers PDTQ_COMQL, PDTQ_COMQH, PDTQ_RSPQ, PDTL_DFQHDR, PDT$L_MFQHDR in

order to verify the integrity of these queues. However this verification is disabled in release version of the system.[74]

Similarly, data entity can be accessed by both ADAWI and BBSSI/BBCCI:

For instance IPC$GL_SEMAPHORE in [IPC.SRC] is locked with BBSSI on bit 0 but is unlocked with ADAWI -1 to it (apparently, bored code author wished to try something original).

Field UCB$W_DIRSEQ is accessed in [RMS.SRC] and [SYS.SRC]RMSRESET.MAR using both ADAWI and BBSSI/BBCCI instructions: it is structured to contain both the sequence number and "AST armed" flag.

Fields CSB_W_CSVAL, CSB_V_LOCK, CSB_V_DELETE, EV_V_LOCK IN [PPLRTL.SRC] are also accessed with both ADAWI and BBSSI/BBCCI.

Thus, whatever is the implementation of interlocked instructions, the implementation must ensure all instructions are interoperable with each other and are not segmented into functional sub-groups that are not mutually interoperable.

As had been discussed, {INS|REM}{QHI|QTI} are distinct from other interlocked instructions in that they utilize secondary interlock and may terminate with status "queue header busy", in which case the caller must retry the instruction until it succeeds or the caller concludes (after certain number of retry attempts) that the queue header must be corrupt and secondary interlock bit set in error or current queue's interlock owner must have locked up or otherwise failed to release the queue within maximum legitimate interval of time. It is up to particular application to decide the number of retry attempts, however existing OpenVMS system code embodies retry limit in several ways. Some code, mostly user-level code (such as in PPLRTL) will reattempt queue operation indefinitely. Most of kernel and executive mode code however encodes retry cycle with one of ${INS|REM}{QHI|QTI}, QRETRY and $QRETRY macros. ${INS|REM}{QHI|QTI} will reattempt the instruction up to 900,000 times and failing that will trigger a bugcheck.[75] QRETRY and $QRETRY have two versions. One version will reattempt the instruction the number of times specified by SYSGEN parameter LOCKRETRY[76] with three instructions in the wait loop (queue instruction, followed by conditional branch, followed by AOBLSS or SOBGTR loop instruction), similarly to ${INS|REM}{QHI|QTI}. The other version will execute 10 usec wait inside the wait loop (terminated early if the loop notices, with non-interlocked access, that secondary interlock bit had been cleared).[77]

[74] It also appears to be the only place where wait loop is two instructions (BBSSI, AOBLSS), rather than 3, and is thus limited to 1,800,000 instruction cycles (rather than 2,700,000) assuming SYSGEN parameter LOCKRETRY is set to 900,000.

[75] Retry loop consists of three instructions (queue instruction, BCC, AOBLSS/SOBGTR), so altogether 2,700,000 instruction cycles will pass before the bugcheck.

[76] With default value of 100,000. VSMP.EXE boosts it at load time to 900,000 unless disabled to do so by VSMP command line override option NOPATCH=LOCKRETRY.

[77] Thus, this version of the loop will attempt to re-execute interlocked queue instruction for up to 1 second of accumulated VCPU run time (i.e. corresponding number of cycles, as measured by calibration loops) if LOCKRETRY is 100,000 or for up to 9 seconds if LOCKRETRY is 900,000, before triggering a failure.

There are two basic ways VAX interlocked instructions can be simulated in a multiprocessor simulator:

- Portable cross-platform way, with primary interlock being simulated with VM software lock or, rather, a set of locks indexed by operand address hash value used instead of a single master lock in order to reduce contention between VCPUs trying to access distinct data locations in interlocked fashion.

- Host-specific ("native mode") way, with VAX interlocked instructions simulated by host interlocked instructions, assuming host instruction set is powerful enough. On most modern processors only a partial native implementation is possible – full implementation for ADAWI, BBSSI, BBCCI, but only mixed-mode implementation for interlocked queue operations, as discussed below.

VAX MP, being a somewhat experimental project, includes both cross-platform portable implementation for VAX interlocked instructions and partial native implementation for x86/x64. Availability of native implementation is selectable with build option SMP_NATIVE_INTERLOCKED. Once available, native mode can be activated with VSMP LOAD command option INTERLOCK=NATIVE. Implications and ramifications of both implementations are discussed below.

(Portable implementation)

Under the portable approach, VAX interlocked instructions are implemented by:

- acquiring a global lock within VAX MP VM;
- and executing appropriate memory barriers before and after the interlocked instruction simulated: full MB before and full MB after (see detailed discussion of memory barriers at the end of this chapter).

To reduce contention between processors concurrently executing interlocked instructions trying to access operands at different addresses, VAX MP uses a set of locks, rather than a single lock, to implement portable-mode primary interlock.[78] An entry in the lockset to use is selected by a hash function computed on an operand address. Thus interlocked instructions trying to perform interlocked operations on operands with different physical addresses will, most likely, use different locks and avoid contending with each other. However since the size of lockset is limited, it is still possible for different addresses to map to the same lock index, thus leading to false sharing of the same lock by different operand addresses.

According to VAX Architecture Standard, locking entity is identified by the lowest byte address of a cell pointed by an operand, which should be longword-aligned. Granularity of the locking may range from a longword to all of memory and even be discontiguous, e.g. every 2^{16}-th longword can be locked.

[78] Locks are held in array named `InterlockedOpLock_CS`, with number of entries `InterlockedOpLock_NCS` currently defaulting to 64.

Locks are implemented as hybrid spinwait/blocking-wait critical section. It is expected that in vast majority of cases the lock will be held only for a very short interval, so if the contender is unable to acquire the lock immediately, it will assume at first the lock will become available shortly and try to spin-wait on the lock without yielding processor. However if the lock is not released fast, contender will assume the holder had been preempted. Contender will then exit spin-wait loop, enter blocking wait and yield processor resources to host operating system.

An obvious problem with the portable approach is that VAX interlocked instructions become potentially blocking. On a real silicon VAX, interlocked instructions are always guaranteed to complete within small number of cycles and both primary and secondary interlocks are never held for long. In contrast, on a simulated VAX using portable implementation of interlocked instructions, thread executing the instruction can get preempted in the middle of instruction simulation code, while holding the lock, and then another thread trying to access either the same data (or data aliased to it via false sharing of the lock index[79]) will also block waiting for the lock.

While this problem cannot be avoided altogether as long as host operating system does not provide effective preemption avoidance control, such as gang scheduling, it can largely be mitigated by elevating thread priority while holding the lock, and thus reducing *probability* of preemption and thus *nearly* avoiding preemption by normal concurrent time-sharing and interactive load on the host system. Even then, preemption can still happen if operating system allocates a fraction of CPU time to low-priority processes (such as to avoid starvation). With properly designed host system, however, the intervals of preemption of high-priority thread must be short.

Preemption can also happen due to page faults incurred by the simulator's VCPU thread while executing the code that simulates interlocked instructions and accesses relevant data, including page faults on VM runtime data, VM code and operands of interlocked instructions. Like with scheduler-induced preemption, page faults cannot be prevented altogether, however their incidence can be reduced by allocating adequate working set to the simulator. VAX MP provides facilities for such allocation.[80] In short, while total preemption avoidance and total avoidance of blocking on the interlock simulation locks is not achievable, a meaningful "soft" goal is to make such blocking waits short and infrequent enough so they do not significantly hamper performance of the system. A meaningful "hard" goal is to maintain system stability in face of thread preemption when "soft" measures fail to prevent it.[81]

Simulated interlocked instructions present two potential threats to system stability:

---

[79] I.e. addresses of two different operands mapping to the same lock index and thus sharing the lock.

[80] See below in chapter "Effects of host paging".

[81] Yet another possible source of thread preemption on a desktop system is an antivirus that may decide it has to scan process space or file being opened and suspend process for an extended time while doing this. There is not much we can do about this issue other than to suggest turning off such antivirus scanning if it is noticed to present a problem. While "soft" impact of the issue (such as stalls and performance reduction while antivirus is blocking a thread) is impossible for VAX MP itself to eliminate, still "hard" mechanisms are meant to prevent large relative inter-VCPU drift leading guest OS to observe false sanity timeout and crashing.

- Long blocking wait can lead to expiration of OpenVMS SMP sanity timeouts. This issue, along with other kinds of preemption, is taken care about by synchronization window mechanism, as discussed in chapter “Interprocessor synchronization window”.

- If secondary interlock on an interlocked queue (represented by the lowest bit in the queue header) cannot be acquired after certain amount of retry attempts, OpenVMS will bugcheck with BADQHDR code, assuming queue header is corrupt.[82]

The latter threat normally does not apply to portable implementation of interlocked instructions, since interlock is always set and cleared entirely under the protection of VM-level lock and is never left set by interlocked queue instructions beyond the duration of holding the lock within the instruction. Since all contenders need to acquire VM-level lock to operate on a queue header, they never see (under normal circumstances, see below for an exception) secondary interlock bit set. This applies both to “raw” {INS|REM}{QHI|QTI} instructions and macros ${INS|REM}{QHI|QTI}, QRETRY and $QRETRY. These macros begin by issuing “raw” instruction and thus, similarly to “raw” instructions, will either immediately acquire VM-level lock or wait on this lock and after lock acquisition will get control over the queue with secondary interlock bit cleared.

> The only “legal” exception being is if secondary interlock in the queue header was intentionally set with BBSSI/BBCCI instruction, as in cases outlined above in the discussion of cross-group interoperability.

To ensure lock holding periods are short, and also corresponding blocking waits are short too and infrequent enough so they do not significantly hamper system performance and do not jeopardize system stability, VAX MP relies on three kinds of safeguards against the preemption of VM lock holding thread. First two of these safeguards are “soft”, merely reducing *probability* of preemption (and hence actual performance impact of preemptions), but unable to *guarantee* that preemptions will not happen:

- VCPU thread priority is elevated when executing interlocked instruction and holding VM-level lock for the operand interlock. This applies to any interlocked instruction, including queue instructions, BBSSI/BBCCI and ADAWI.

- Simulator working set size is enlarged enough so page faults are infrequent.

Third safeguard is “hard” kind and is intended to guarantee system stability in the worst case when VCPU thread does get preempted in spite of “soft” anti-preemption safeguards. This safeguard is implemented by the synchronization window component SYNCW-ILK that is triggered by the following events:

[82] The amount of retries is hardcoded to be 900,000 in macros $INSQHI, $INSQTI, $REMQHI and $REMQTI. Some places in system code use instead the count set by dynamic SYSGEN parameter LOCKRETRY with default value of 100,000. When VSMP.EXE starts multiprocessing, it increases by default the value of this parameter to 900,000 unless VSMP command line option specifically forces it not to. Since there are at least 3 instructions in the retry loop, this timeout corresponds to 2,700,000 instruction cycles.

- As a fallback for cases when queue header is intentionally interlocked with BBSSI instruction, synchronization window component SYNCW-ILK is used, if enabled. VCPU will also enter SYNCW-ILK synchronization window if the use of SYNCW-ILK is enabled *and* IPL is >= 3 *and*:

    - VCPU executes instructions ADAWI or BBSSI.

      It is assumed that VCPU might have interlocked queue header or other structure accessed via busy-wait retry loop with iteration count limited by SYSGEN parameter LOCKRETY or by hardwired value of 900,000, whichever is smaller.

  or

    - VCPU executes interlocked queue instruction (INSQHI, INSQTI, REMQHI, REMQTI) *and* it fails to acquire secondary interlock.

      It is assumed as possible (and indeed likely) that VCPU will enter busy-wait retry loop at this point.

  Note that this applies only for VMS code executing at IPL RESCHED (3) or above, since any VMS code executing below this level is preemptable and thus cannot expect any predictable maximum duration between the secondary interlock is set and the time it is cleared. Synchronization window width in this case is presumed to be 3* min(LOCKRETRY, 900000) instructions.

- If the use of SYNCW-ILK synchronization windows component is enabled, interlocked queue instructions INSQHI, REMQHI, INSQTI and REMQTI will enter VCPU into SYNCW-ILK for the duration of the instruction, regardless of CPU mode (KESU) or IPL level. This means that processors executing interlocked queue instructions cannot drift from each other by more than 3 * min(LOCKRETRY, 900000) instruction cycles. If the thread for VCPU1 executing interlocked queue instruction gets stalled because of thread preemption or page fault or other similar cause, processor VCPU2 trying to execute interlocked queue instruction on the same queue operand will be prevented by SYNCW-ILK from drifting ahead of stalled VCPU1 by more than described number of instruction cycles.

Note that thread priority must be elevated regardless of the caller's processor mode, whether kernel or user. It may be tempting to limit elevation to kernel mode only, however it won't work properly, both because lock indexes for kernel-mode and user-mode data entities can map to the same lock index due to false sharing, and also because of the following scenario:

1. VCPU1 executing user-mode code (and thus at low thread priority) acquires lock for interlocked data access. While executing simulator code for interlocked instruction, VCPU1's thread gets preempted by concurrent workload on the host. This preemption can be long-lasting.

2. VCPU2 trying to access either the same data (or perhaps data aliased to the same lock through false sharing) blocks (or convoys) on the same lock. Priority inversion can already happen at this point.

3. VCPU3 sends an inter-processor request interrupt to VCPU2 that needs urgent servicing. VCPU2 priority gets elevated (assuming it was low), however VCPU2 still remains blocked waiting for VCPU1 thread, whose priority is still low, and that remains preempted, now with VCPU2 and VCPU3 having to wait for it indefinitely.

To avoid this situation happening, portable implementation of interlocked instructions elevates thread priority while the primary interlock's lock is held – for holder threads executing VAX code of any mode (user or kernel) and at any IPL level. Thread priority is elevated to CRITICAL_VM.

This priority elevation for anti-preemption purposes does not come for free and is not a cheap solution. Measured cost for the system call changing thread priority is about 1 microsecond on both Windows 7 and Linux on 3.4 GHz i7 machine (this machine executes VAX code at about 30 MIPS). Since interlocked instruction will typically require thread priority elevation before execution of the instruction and thread priority demotion after its execution, this means two system calls per interlocked instruction with summary overhead of about 2-2.5 microseconds, equivalent to approximately 70 regular VAX instructions. While this might sound an acceptable cost for a specialized instruction, under certain workloads OpenVMS kernel can execute interlocked instructions at very high rate. For example, heavy paging will cause a very high rate of MMG spinlock acquisition and release cycles, and under these circumstances cost of thread priority control inside the portable implementation of BBSSI/BBCCI instruction can amount to a significant fraction of the total workload.

As a practical example, typical boot sequence of basic OpenVMS configuration (OpenVMS with TCP/IP, LAT and C installed) results in 2.2 million interlocked instructions executed over 55 seconds: a rate of 40,000 interlocked instructions per second! Vast majority of these instructions are BBSSI and BBCCI instructions, predominantly for MMG and SCHED spinlocks, very remotely followed by other spinlocks; a comparatively small part is made of interlocked queue instructions and ADAWI instructions. To diminish this overhead during boot time, current VAX MP release does not perform priority elevation and locking for interlocked instructions when only primary processor is active. This change alone speeds up bootstrap time by 11%. However locking is a must when multiple processors are active; and then (at least under relatively light contention) vast majority of overhead is contributed by system calls to change thread priority, not as much by locking a hashed-set lock.

VAX MP therefore provides configuration parameter `HOST_DEDICATED` settable via SIMH console command that inhibits thread priority elevation by VCPU threads during execution of interlocked instructions and also for most of CPU state transitions. This parameter may be enabled when VAX MP is executed on a host system where preemption is deemed unlikely, such as on dedicated host system, not having to contend with host load, and thus with guaranteed availability of host CPUs for VAX MP, making anti-preemption measures not essential. Under such set up, it may be possible to forgo anti-preemption measures and their overhead in favor of increased performance. Similarly a host system with gang scheduling would allow to do away with the need for thread priority elevation.

Additionally, some host architectures, in particular ubiquitous x86/x64 based host machines, allow another workable solution for the reduction of the overhead of system calls to change thread priority while executing interlocked instructions. This solution is the use of native-mode implementation for interlocked instructions, as described below.

> Since in native-mode implementation essential parts of BBSSI, BBCCI and ADAWI execute atomically at host processor hardware level, directly mapping VAX interlock to host hardware interlock, and cannot be preempted in the middle of instruction execution, they do not require thread priority elevation.
>
> Interlocked queue instructions nevertheless are implemented as composite and still can be preempted inside the instruction, so they still do require priority elevation for the duration of the instruction, as anti-preemption measure. However interlocked queue instructions make relatively insignificant part in the whole interlocked instruction workload issued by OpenVMS kernel. OpenVMS uses predominantly BBSSI/BBCCI instructions (and can use a lot of them), only remotely followed by interlocked queue instructions. Therefore the possibility to limit thread priority elevation/demotion to only interlocked queue instructions – possibility accorded by x86/x64 native mode implementation – eliminates the bulk of overhead associated with "set thread priority" system calls required for the implementation of interlocked instructions.
>
> On the other hand, use of native mode necessitates the use of fairly tight synchronization window when running OpenVMS (sized to a fraction of 2,700,000 instruction cycles), as discussed below in sub-chapter "Native implementation", whereas with portable implementation it is normally possible to forego the use of synchronization window.
>
> Even though synchronization window applies only when VCPU is executing system-critical code running at elevated IPL or has interrupt pending, still synchronization window somewhat constrains system scalability, therefore being able to forego its use is advantageous from the performance standpoint. Furthermore, when checking for its synchronization window position and when entering and exiting synchronization window, VCPU thread has to acquire a lock on VM internal data structures, and for that it has to elevate its thread priority level as anti-preemption measure while holding this lock. Still, this lock may have to be acquired and released much less often than BBSSI/BBCCI instructions that may get executed approximately every 500'th instruction under some workloads (such as system boot sequence of heavy paging)[83]. Therefore native implementation also does hold advantages. The balance of

[83] Typical VMS bootstrap sequence on an installation with VMS 7.3, TCP/IP, LAT and C takes about 1 billion instructions. About 2 million of them are interlocked instructions, mostly BBSSI and BBCCI. This translates to approximately 1:500 ratio between interlocked and non-interlocked instructions. When executing typical user-mode application, incidence of interlocked instructions will likely be much lower. However if the applications pages very heavily or performs very intensive I/O, it might be even higher.

advantages and disadvantages for both modes is not clear a priori and may also depend on the workload. VAX MP therefore implements both modes, so preferred mode can be selected as we gain more experience with using VAX MP and also to allow a user to select preferred mode depending on the user's predominant workload.

It may also be possible to collapse VCPU thread priority levels into just two (or three) priority levels, corresponding essentially to IPL < IOPOST and IPL >= IOPOST. Since vast majority of interlocked instructions get executed at IPL >= IOPOST, this would greatly reduce the need for frequent priority elevations and demotions when executing interlocked instructions. Do note however that of 2.2 million of interlocked instructions executed during VMS bootstrap, 63 thousand are still executed at IPL 2, 31 thousand at IPL 0 in kernel mode and 100 thousand in user mode, still comprising together 9% of the total number.

(Native implementation)

On some host machines it might also be possible to implement VAX interlocked instructions with functionally similar host instructions when available – thus mapping VAX primary interlock to host hardware interlock instead of software locks used in portable approach.

For example, VAX ADAWI instruction can be implemented on x86/x64 using LOCK ADD host instruction. While BBSSI and BBCCI cannot be implemented using x86/x64 LOCK BTS and LOCK BTR instructions since the latter operate only on 16-bit or 32-bit operands (that may also have to be aligned), whereas VAX Architecture specification requires that BBSSI/BBCCI change only single byte and operate on any alignment, but these VAX instructions can be implemented with x86/x64 8-bit LOCK CMPXCHG instruction.

In general, ADAWI, BBSSI and BBCCI can be natively implemented on any host computer having CAS or LL/SC instructions, combined with appropriate memory barrier instructions, wherever required, as long as these host instructions have a variant able to operate on a 16-bit aligned word (ADAWI) and unaligned byte (BBSSI/BBCCI).

VAX interlocked queue instructions can then be implemented by:

1. asserting secondary interlock bit in queue header using native-mod implementation of BBSSI,
2. performing operation on the queue structure with non-interlocked access,
3. and finally releasing secondary interlock with native-mode BBCCI or native-mode CAS on queue's header first longword.

Obvious difference with portable implementation is that native implementation of ADAWI, BBSSI and BBCCI is lock-free, non-blocking. Similarly to real hardware VAX, native implementations of ADAWI, BBSSI and BBCCI are guaranteed by host hardware to complete within a very short fixed time. Same applies for setting and clearing of secondary interlock on the queue header. In addition, ADAWI, BBSSI and BBCCI do not require elevation and subsequent demotion of VCPU thread priority.

This attractive similarity with hardware VAX, however, ends here and does not extend to the whole implementation of {INS|REM}{QHI|QTI} instructions with native interlock. On a real VAX, interlocked queue instructions are non-interruptable and are guaranteed to complete within a very short time, and therefore secondary interlock bit temporarily asserted by these instructions is also normally cleared within a very short time. On a simulator, VCPU thread executing interlocked queue instructions can be preempted by the scheduler or due to page fault in the middle of the instruction while secondary interlock is asserted, leaving secondary interlock bit set for an extended time.

Under the portable implementation, in this situation contenders trying to access "busy" queue header will enter blocking wait on indexed lock representing the interlock for queue address, and then will be granted ownership of the queue header on a first attempt to execute the interlocked instruction, once the blocking wait completes; there will be no need for retries (except if the queue was explicitly interlocked with BBSSI/BBCCI, but this is a rare occasion).

Under the native implementation, contenders will spin on the queue header until the secondary interlock is cleared. This spin happens at VAX instruction stream level and raises possibility of spin-wait timeout leading to BADQHDR bugcheck (or similar failure in the applications running in non-kernel mode), unless proper safeguards are implemented and maintained. For system code, these safeguards (as well as methods to avoid performance loss due to preemption of secondary interlock holder) are the same as for the portable implementation case and include two "soft" safeguards:

- elevation of VCPU priority while holding the secondary interlock;
- and ensuring simulator's working set size is adequate, so paging will be infrequent;

and ultimately a "hard" safeguard of the synchronization window.[84] However importance of crash protection provided by the synchronization window greatly increases in a native-mode implementation, since not only relatively marginal cases now would rely on its protection from timing out the busy-wait loop, but it becomes a mainstream mechanism safeguarding all interlocked queue operations in the kernel.

These safeguards are sufficient to ensure adequate operation of OpenVMS system code under native-mode implementation of interlocked instructions. However extending this mechanism to protect applications and third-party privileged mode code is potentially problematic. First, while we can have a look at the operating system code, we cannot know interlock queue retry/timeout limits used by various user-level applications and privileged third party components that may utilize interlocked queues. Second, applying synchronization window to user-mode code in order to put VCPUs executing user-mode code in a relative lockstep (rather than doing it only for VCPUs executing high-IPL system code) would reduce scalability and performance of the system.

[84] Refer to sub-section on the use of synchronization window for interlocked instructions later in this chapter.

Therefore, while using native-mode interlocked implementation has some benefits compared to portable mode[85], it may cause instability and retry loop timeouts in third-party components (user-mode applications and privileged code) in the following cases:

- If the component contains preemptable code (user-mode code or privileged mode code executing at IPL < 3) that uses retry loop count on interlocked queue instructions (INSQHI, INSQTI, REMQHI, REMQTI) smaller than min(900000, LOCKRETRY).

- If preemptable code uses finite retry count on interlocked queue instructions and queue header also gets locked with BBSSI instruction.

- If preemptable code tries to lock interlocked queue header with BBSSI instruction and uses finite retry count for this BBSSI.

- If non-preemptable code (kernel-mode code executing at IPL >= 3) uses smaller retry count for interlocked queue instructions than is hardwired in VMS macros $REMQHI, $REMQTI, $INSQHI or $INSQTI, i.e. 900,000 times, or set by SYSGEN parameter LOCKRETRY, whichever is less, i.e. smaller than min(900000, LOCKRETRY).

- If non-preemptable code tries to lock interlocked queue header with BBSSI instruction and uses retry count yielding total number of instructions executed in retry loop lesser than 3 * min(900000, LOCKRETRY).

We are not aware at the moment of any applications or components that do this, but they might exist. Under native-mode interlock implementation, such code would be secured from failing only by “soft” safeguards, but not by “hard” safeguard of the synchronization window, and may fail.

For this reason, VAX MP provides both implementations of interlocked instructions (on platform where native-mode interlock can be implemented) – both portable and native mode implementations – but with configurable option that allows user to switch between the implementations. Default is to use portable implementation. Native-mode implementation can be enabled at the time VSMP is started with the following option:

```
VSMP  LOAD  INTERLOCK=NATIVE  SYNCW=ALL
```

and is likely to be preferable from performance viewpoint when system does not run applications and components that use tighter interlocked queue retry limits than OpenVMS itself does.[86]

[85] Chiefly, eliminating overhead associated with system calls to elevate and demote VCPU thread priority when executing BBSSI, BBCCI and ADAWI instructions (but not interlocked queue instructions). Also avoidance of false sharing of lock index, i.e. situation when different interlocked operand addresses map to the same lock index.

[86] We also considered an option for hybrid mode, based on native mode but supplemented with interlocked queue instructions (but not BBSSI, BBCCI and ADAWI instructions) acquiring locks from a hashed set of locks indexed by

[Interlocked interaction stability in native mode]

Here is the summary of timing synchronization guarantees (i.e. guarantees against false premature timeout of busy-wait loops) that native mode provides, and those it does not. We use below BBxxI to designate any instruction in group (BBSSI, BBCCI, ADAWI). We use QxI to designate any interlocked queue instruction (INSQHI, INSQTI, REMQHI, REMQTI).

Some cases of potential instability, as designated below, can be eliminated by interlocked queue instructions temporarily activating synchronization window constraint for the VCPU (unless it was already active) and leaving synchronization window after the completion of the instruction, i.e. temporarily entering VCPU into synchronization window for the duration of interlocked queue instruction regardless of CPU mode (kernel/user) or IPL level. This is what VAX MP actually does.

There are four groups of possible cases, depending on instructions executed by both interacting processors, two last cases further subdivided each into 4 sub-cases depending on whether each processor was executing preemptable (P) or non-preemptable (NP) code, i.e. whether it was in active synchronization window or was outside of it:

(1) BBxxI ↔ BBxxI – protection against timeout is ensured by synchronization window (SYNCW-SYS); can also be ensured by SYNCW-ILK if sized not larger than SYNCW-SYS

(2) QxI↔QxI – can fail in native mode (but not portable mode) if any of the caller does not use SYNCW-ILK (or both callers do not), unless QxI forces temporary SYNCW-ILK for the duration of the instruction

(3) BBxxI on first processor, successfully locks element
QxI on second processor, spin-waiting

	QxI NP	QxI P
BBxxI non-preemptable (NP, IPL >= 3)	1	3
BBxxI preemptable (P, IPL < 3)	2	2

1 → protected by SYNCW-ILK

2 → QxI may not use finite loop in a valid software when interactive with preemptible BBxxI

3 → can fail, unless QxI forces temporary SYNCW-ILK for the duration of QxI instruction

operand address, same as in portable mode case. Analysis of all possible inter-instruction interaction cases shows that this hybrid mode provides exactly the same stability guarantees and has exactly the same vulnerabilities as the native mode and does not provide more reliable operation compared to the native mode.

(4) QxI on first processor, successfully locks element
BBxxI on second processor, spin-waiting

	BBxxI NP	BBxxI P
QxI NP (IPL >= 3)	1	4
QxI P (IPL < 3)	3	4

4 → can fail if finite wait loop is used, syncw won't help

(Memory barriers)

Implementation of VAX interlocked instructions should generate full memory barrier before the instruction and full memory barrier after the instruction. These MBs cannot be relaxed to either WMB or RMB. Primary reason for this is that simulator cannot be aware of exact purpose and intended semantics of operation implied by the interlocked instruction within a higher-level code. Simulator has to assume the possibility that interlocked instruction may delineate either the beginning or the end of a critical section protecting other data. Furthermore, simulator cannot possibly know what side of instruction the critical section lays on. While it is true that in most cases BBSSI begins a critical section and BBCCI terminates it, but it is not guaranteed to always be so. Therefore simulator has to assume that any VAX interlocked instruction can delineate the boundary of the critical section protecting other data and that this section may be laying on any side of the interlocked instruction.

Since neither read nor write references to protected data may be allowed to leak out of the critical section, through its boundaries, implementation of VAX interlocked instruction should be sandwiched the following way:

[possible critical section 1]
full MB
interlocked operation
full MB
[possible critical section 2]

where shaded background covers the implementation of VAX interlocked instruction. In other words, any VAX interlocked instruction should be implemented by the simulator the following way: "MB-op-MB", where "op" is the implementation of atomic operation.

In addition to this, an implementation of interlocked instruction itself must also be wholly contained within that instruction's boundaries. Host instructions that make up the implementation of simulated

VAX interlocked instruction may not be allowed to be reordered by the processor with host instructions from neighbor VAX instructions.

The most obvious case of this is portable implementation of VAX interlocked instructions, where manipulation of interlocked data entities is performed with regular host instructions under the protection of primary interlock represented by host critical section – such as manipulations on interlocked queue structure or atomic addition etc. that owe their VAX-level atomicity to being executed under the protection of the VM-level host lock. These manipulations may not be allowed to leak (via processor reordering) out of the critical section. Therefore, implementation of VAX interlocked instruction must be bracketed by full MBs on both sides, for its own sake. This applies even to native-mode implementations, whose instruction issues and data references may not be allowed to be reordered by the host processor with instruction issues and data references of neighboring VAX instructions. (Some host processors, including x86/x64 may automatically issue memory barriers when host interlocked instruction is performed – but even on these processors the barrier may be only partial and not always sufficient; e.g. on x86/x64 automatic barrier issued by the processor is valid only as long as the simulator code does not use non-temporal SSE/3DNow instructions.)

(Use of synchronization window)

When executing OpenVMS, both native and portable implementations of interlocked instructions necessitate constraining the "drift" between virtual processors to a fraction of 2,700,000 cycles when processors execute at IPL >= RESCHED (3).

This constraint is necessary to prevent a scenario where VCPU1 acquires secondary interlock on the queue header via BBSSI and then VCPU1 thread gets preempted while holding the interlock, then VCPU2 thread tries to acquire an interlock on the same queue header for 900,000 iterations (each iteration being a step in the busy-wait loop composed of three instructions) and triggers a bugcheck after being unable to acquire the interlock within the maximum number of iterations.

> This constraint is optional, as explained below, since the kind of code it is meant to constrain is not actually present in baseline OpenVMS 7.3, however such code might be present in some layered products. Use of layered product containing such code would require the use of described constraint.

Synchronization window constraint to prevent this scenario (both for native and for portable implementation) is named SYNCW-ILK and is defined as follows:

- If VCPU is executing at IPL >= 3, it cannot "run ahead" of any other VCPU executing at IPL >= 3 by more than a fraction of 2,700,000 instruction cycles – provided both VCPUs are "in" SYNCW-ILK window.

    - "Fraction" should not be too close to 1.0, so as to give the processor holding the interlock reasonable number of cycles to do the intended work.

- Drift constraint periodically compares per-CPU counters reflecting a "position" of VCPUs in the synchronization window.

- Per-VCPU counter is started (unless it was already active) when VCPU executes interlocked instructions BBSSI or ADAW at IPL >= 3. Processor is then said to "enter" SYNCW-ILK window.

- Per-VCPU counter is started (unless it was already active) when VCPU executes interlocked queue instructions (INSQHI, INSQTI, REMQHI, REMQTI) at IPL >= 3 *and* instruction fails to acquire secondary interlock on the queue. It is expected that retry loop will follow.

- This counter is also temporarily started for the duration of QxI instruction (unless it was already active) when QxI instruction is executed by this VCPU at *any* IPL, including low IPLs and *any* processor mode, including user mode. The counter is then terminated at the completion of the instruction assuming the counter was inactive at the start and VCPU IPL was below RESCHED (3).

- This counter is reset each time a VCPU drops its IPL below 3 or when VCPU executes an OpenVMS idle loop. Processor is then said to "leave" SYNCW-ILK window.

- The counter is also reset each time OpenVMS executes process context switch (SVPCTX or LDPCTX instructions). VCPU will then exit SYNCW-ILK.

For native implementation this constraint is supplemented by another constraint intended to prevent VCPU1 acquiring a secondary interlock on queue header with {INS|REM}{QHI|QTI} instruction, and VCPU1 thread subsequently getting suspended while owing the interlock, then VCPU2 trying to acquire secondary interlock on the queue header for 900,000 iterations and, failing this, triggering a bugcheck. Though the scenario for this problem is different (secondary interlock being temporarily held within the boundaries of single interlocked queue instruction, rather than during a longer section of code), the remedy is the synchronization constraint identical to one described above.

Of these two described scenarios, the former type of code does not actually exist in baseline OpenVMS 7.3 code. There is a possibility it may exist in some layered products, and in this case the use of such product would require the use of SYNCW-ILK.

The latter scenario however is ubiquitous, but luckily in portable mode locks acquired internally by interlocked queue instructions make inter-processor drift impossible: if VCPU1 is holding the lock for the operand and VCPU1 thread is preempted, and VCPU2 tries to execute interlocked queue instruction on the same queue operand, VCPU2 will try to acquire lock and will wait on the lock until VCPU1 completes the instruction and releases the lock. Thus VCPU2 won't be able to drift ahead of VCPU1 if the latter's thread is preempted. Thus portable interlock mode does not require the use of SYNCW-ILK to address the second scenario. However native-mode implementation does require the use of SYNCW-ILK to prevent the second scenario.

Note that descried SYNCW-ILK constraint is different and distinct from synchronization window constraint SYCNW-SYS meant to prevent guest OS (OpenVMS) SMP sanity timeouts, as described in chapter "Interprocessor synchronization window". These two constraints are distinct. To disambiguate, we name them SYNCW-SYS and SYNCW-ILK correspondingly.

The whole synchronization window (SYNCW) constraint is thus composed of two parts: SYNCW-SYS and SYNCW-ILK.

> SYNCW-SYS is responsible for constraining busy-wait loops controlled by SYSGEN parameters SMP_SPINWAIT and SMP_LNGSPINWAIT. These are mostly spinlock acquisition loops and inter-processor interrupt or other communication (such as VIRTCONS) acknowledgement loops, although some other loops also use SMP_SPINWAIT or SMP_LNGSPINWAIT as timeout limits.
>
> SYNCW-ILK is responsible for constraining busy-wait loops controlled by SYSGEN parameter LOCKRETRY or by limit of 900,000 iterations encoded in macros such as $INSQHI etc. These loops are mostly used to perform interlocked queue operation on various queues with interlocked queue instructions INSQHI, INSQTI, REMQHI, REMQTI or for locking queue header with BBSSI instruction for scanning the queue, although some other loops are constrained this way too.

VCPU forward progress and VCPU "drift" related to other VCPUs in the system is constrained by a combination of these two constraints. Any of these two constraints, when triggered due to other VCPU/VCPUs falling too much behind, will temporarily pause current VCPU from execution until the constraint is cleared by other VCPUs catching up and it is safe to resume execution.

Constraint SYNCW-SYS is optional and can be disabled if OpenVMS SMP sanity checks are disabled too via setting SYSGEN parameter TIME_CONTROL to 6. Doing so enhances system scalability. The downside is that if VMS locks up because of the bug in operating system, driver or third-party privileged mode component, setting TIME_CONTROL to 6 will prevent OpenVMS sanity self-check from generating the bugcheck to crash and restart the system. However since OpenVMS VAX is stable at this point, such risks are very low and it may usually be desirable to disable sanity checks and associated SYNCW-SYS constraint in favor of improved scalability.

Constraint SYNCW-ILK is a must in native mode and cannot be disabled when executing simulator with native mode enabled (by VMSP LOAD option INTERLOCK=NATIVE).

However when executing the simulator in portable mode (VSMP LOAD INTERLOCK=NATIVE is *not* set), SYNCW-ILK is optional and usually can be disabled for the sake of improved scalability and performance. This is chiefly because in portable mode (unlike native mode) interaction QxI↔QxI cannot cause BADQHDR bugcheck. It is still possible for the system to fail because of BBxxI↔QxI interaction in case SYNCW-ILK is disabled (i.e. if BBSSI locks out queue header on one VCPU and the other VCPU tries to access the queue with QxI instructions), however we are not aware of any actually existing code sequences in base OpenVMS VAX 7.3 that can lead to such failure.[87] Thus it appears to be safe (and beneficial for scalability) to disable SYNCW-ILK when using portable mode, as least as long as only base OpenVMS is used. Some layered and third-party privileged mode components might possibly contain code sequences that perform BBxxI↔QxI interactions and rely on timing of such interactions (i.e. use finite retry count) – albeit this is not too likely, it is possible. If user observes failures caused by failing timing of such interactions (such as BADQHDR crashes), he should enable SYNCW-ILK even in portable mode.

Table below summarizes valid combinations of SYNCW related parameters.

Three relevant parameters are:

- VSMP LOAD command option INTERLOCK=*value*, where *value* can be PORTABLE or NATIVE.

  Default is PORTABLE. On some host systems that do not provide adequate instruction set for native-mode emulation of VAX interlocked instructions, setting to NATIVE can be impossible. It is possible for the ubiquitous case of x86/x64 host processors.

- VSMP LOAD command option SYNCW that can be any combination of SYS and ILK: ILK, SYS, (SYS, ILK) also expressible as ALL, and NONE.

- OpenVMS SYSGEN parameter TIME_CONTROL.

[87] The only known suspect place is [SNAPSHOT.SRC] component and its handling of EXE$GQ_SNAP_FORK_LIST. However it appears that EXE$CANRSTFORK is not actually called from anywhere. EXE$SNAP_NOTIFY_FORK is also not called and furthermore according to the comment should be called in uniprocessor environment only.

INTERLOCK = PORTABLE
 	
SYNCW = NONE, TIME_CONTROL = 6
	This is the default setting.
	
	Synchronization window is not used.
	
	This setting provides the best scalability in portable mode.
	
	It is safe with base OpenVMS, but may potentially cause OpenVMS to crash under some code sequences that do not exist in base OpenVMS but might exist in layered or third-party privileged products.

INTERLOCK=PORTABLE
 	
SYNCW = ILK, TIME_CONTROL = 6
SYNCW = SYS, TIME_CONTROL = 2 or 6
SYNCW = (SYS,ILK), TIME_CONTROL = 2 or 6
	These are the safest settings.
	
	These settings engage one or both components of synchronization window.
	
	They reduce scalability and performance compared to the default setting.
	
	Use these settings only when running layered or third-party privileged components that are known to crash for SMP-timing related reasons (such as bugchecks CPUSPINWAIT or BADQHDR) under the default configurations. Currently no such components are known.

INTERLOCK=NATIVE
 	
SYNCW = ILK, TIME_CONTROL = 6
SYNCW = (SYS, ILK), TIME_CONTROL = 2 or 6
	Using native mode.
	When using native mode, ILK is a must.
	
	Native mode is experimental at this point and only limited testing had been performed for it.
	
	Native mode might offer less overhead than portable mode when system makes heavy use of BBSSI and BBCCI interlocked instructions, such as heavily using spinlocks, which in turn may occur during heavy paging or intense IO.
	
	Actual performance benefits of native mode have not been assessed yet. Native mode may be faster than portable mode on some workloads.

Before VSMP tool starts multiprocessing, it performs a check that combination of configuration parameters being used is valid and will refuse any incorrect combination with appropriate diagnostic message.

Per-CPU devices and console commands

SIMH is composed of simulated devices. In a multiprocessor configuration some of these devices become per-CPU, i.e. there is a separate instance of a device for each virtual CPU. Other devices stay system-global, shared by all virtual processors. Device descriptor remains global, representing essentially a device class, but actual instance data for per-CPU device is kept in per-CPU context area, whereas for global devices it resides in global or static variables.

VAX MP adds console commands to configure multiple CPUs into the system, and to switch between different CPU contexts when applying other console commands. For example the following console commands create 6 virtual processors on the system, i.e. extra five CPUs in addition to the primary CPU #0, and start the execution:

```
sim> cpu multi 6
sim-cpu0> boot cpu
```

After the “halt” key (Ctrl-E) is pressed on the console, all CPUs are brought to halt and emulator displays console prompt again. User can switch between processor instances and examine their state:

```
sim-cpu0> examine PSL
[cpu0]  PSL:    041F0000
sim-cpu0> cpu id 2
sim-cpu2> examine PSL
[cpu2]  PSL:    03C00009
```

Command SHOW DEVICES will list the same device names for every CPU. Some of these devices will be system-wide, and listings appearing in the context of different current CPUs will refer to the same shared, global device instance, whereas other devices (such as TLB) will be per-CPU and any references to such devices mean a reference to device *per-CPU instance*.

Dispatching device interrupts

DEC VAX multiprocessors provide hardware capabilities for dispatching IO device interrupts to various CPUs in the system for servicing. These hardware capabilities are model-specific. For example, VAXstation 3520 and 3540 feature MBus FBIC controller chip that allows for mapping QBus interrupt lines to various MBus interrupts that can be subsequently selected for servicing (or not) by individual processors in the system. VAX 9000 series provides configurable choice between broadcasting IO device interrupts to all processors in a selected set defined by a mask in configuration register or dispatching incoming interrupts in round-robin fashion among the processors in the set, for load-balancing of interrupt handling. Configuration of interrupt dispatching is performed by a processor-specific part of operating system code (SYSLOAxxx). Since VAX MP is recognized by OpenVMS as MicroVAX 3900, such specific code is not activated and dispatching interrupts is left to VAX MP. Technically VAX MP could implement one of the mentioned approaches such as dispatching interrupts in round-robin fashion among running virtual processors, or dispatching all interrupts to the primary processor. However when running on an SMP multiprocessor, VAX/VMS and OpenVMS VAX actually serve all IO device interrupt requests on the primary processor[88], for reasons of overhead reduction and contention reduction. Since existing VMS builds expect to receive all interrupts on the primary processor, and dispatching interrupts to processors other than primary could thus potentially result in unexpected behavior and consequences, VAX MP honors this VMS design decision and system expectations and dispatches all IO device interrupts to the primary processor in the system, i.e. CPU #0. This applies to system-wide devices, such as QBus devices. Interrupts from per-CPU devices are dispatched to the processor that owns this particular device instance.

Device timeouts

Besides synchronization between virtual processors, there is also an issue of synchronization processors and virtual devices.[89]

A part of this problem is inter-thread interlocking simulator's data structures representing the state of a virtual device when accessing the device, i.e. synchronization *among* VCPU threads when accessing a shared IO device for the purpose of maintaining consistent state of device representation. This aspect is discussed below in the chapter "Lesser design notes and changes to SIMH codebase".

Present chapter will address another aspect – a *timing* of interaction between virtual processors and virtual device.

In SIMH, each virtual device is executed in the context of one or another virtual processor, with a part of execution possibly also delegated to an auxiliary IO processing thread (IOP thread).

For example, when VCPU (any VCPU, either primary or secondary) initiates an IO operation on virtual disk device, VCPU typically begins by accessing device CSR. This access is performed in the context of VCPU thread. Simulator's device handler[90] implementing device logics gets invoked in the context of the VCPU thread trying to read or write device CSR. Device handler may at this point either initiate actual underlying host IO operation (such as reading or writing host file representing virtual disk's container) right away or schedule SIMH clock queue event to respond with some delay to the command received via CSR[91]. In the latter case, scheduled clock queue entry will be triggered and processed soon, also in the context of the same virtual processor. Device's event processing routine may at this time initiate

[88] "VAX/VMS Internals and Data Structures", version 5.2, p. 1040 (chapter "Symmetric Multiprocessing", section "Device Interrupts"); "OpenVMS AXP Internals and Data Structures", version 1.5, pp. 1279-1280. CPU to IO device affinity, including interrupt dispatching, was revised for Fast Path scheme that was however implemented only for OpenVMS Alpha and I64, starting with OpenVMS version 7.3-1 for Alpha, and was never introduced for OpenVMS VAX; see HP "OpenVMS I/O User's Reference Manual OpenVMS Version 8.4", p. 322 ff.

[89] IO devices in SIMH do not directly interact with each other, so there is no issue of device-to-device synchronization.

[90] To remind, we use term *device handler* to refer to SIMH code that implements virtual device, as opposed to guest OS *device driver* that interacts with virtual device represented and simulated by *device handler*.

[91] Or perhaps via request structure passed indirectly through CSR, such as MSCP disk/tape port controller request/response ring or Ethernet DEQNA/DELQA buffer descriptor list.

actual host IO operation. Once the host operation completes, device will signal virtual interrupt to the primary CPU, which will again access device CSR and execute device handler's completion code. Thus, virtually all device activity happens in the context of one or another VCPU (except only for asynchronous part executed on IOP thread for device handlers that utilize IOP threads). However these VCPUs can be *different*. Device IO operation can be initiated by CPU1, but its interrupt will be processed by CPU0.

What is important in this context is that before initiating an IO operation guest OS device driver in most cases schedules a timeout for that operation. Whereas the operation may be performed by let us say VCPU1, OS clock-driven timeout handler may (and in VMS does) execute on a *different* processor (in case of VMS, primary CPU). When guest OS executes on a real hardware multiprocessor, this works fine, since both the processors and the device are implemented in silicon and comply with hard real-time constraints expected by the driver, so timeout does not get triggered unless the device malfunctions or is not ready (printer out of paper, media unloaded etc.). When simulated on a time-sharing system however, this scheme falls apart, since threads simulating VCPUs and devices can get preempted by the scheduler or delayed due to page faults, file system delays and similar effects, therefore hard real-time constraints assumed by the driver cannot be satisfied.

Therefore, timing of interaction between VCPUs and IO devices requires special analysis and handling that account for the impossibility for the simulator to provide hard real-time constraints that are provided by real VAX hardware and are presumed by guest OS drivers and other components.

Dealing with the impossibility to implement such real-time constraints in the simulator running on top of mainstream commodity host OS is the subject of the present chapter.

There are two broad categories of checks employed by VAX/VMS when checking for device timeouts.

One is for shorter timeout intervals (typically in sub-millisecond range)[92]. VMS will initiate device activity (typically by writing or writing device CSR) and then perform busy-wait loop for a certain number of calibrated loop cycles producing delay of intended duration[93], to let device perform expected work and possibly checking device CSR state change during the loop or after the loop. Usual SIMH implementation of device activity is via primitive `sim_activate` that places scheduled invocation of device handler into simulator's clock event queue. Delay for SIMH device handler invocation is expressed as the number of VCPU instruction cycles past the current cycle. VAX MP design decision is to place clock queue entry produced by `sim_activate` into the event queue of *local* VCPU – the same VCPU that executed `sim_activate` and that subsequently performs busy-wait (TIMEDWAIT) loop[94]. This ensures automatic intended synchronization in timing between VCPU instruction stream and device activity and its state transitions since both the instruction stream and device handler get executed in this case by the same

[92] During system boot time synchronous busy-waiting can also be used for longer intervals, into multi-second range.

[93] This loop is typically implemented in VMS drivers' source code with TIMEDWAIT macro.

[94] For more detailed description of SIMH clock event queues and changes made to them in VAX MP refer to the description in the chapter "Lesser design notes and changes to SIMH codebase" below.

VCPU thread and are timed to the same time scale of executed VCPU instruction cycles. VCPU thread preemption (whether by scheduler, due to page faults or other kinds of delays) affects all activities timed to this scale equally and thus does not disrupt their relative synchronization. There may be (and will be) a drift in synchronization *between* VCPUs, however this drift does not affect co-synchronization of events *intra* each processor. Placing device's event scheduled by device handler with `sim_activate` on the clock event queue of *local* VCPU ensures co-synchronization of device activity with busy-wait (TIMEDWAIT) loop on that VCPU, and that no false device timeout will be signaled because of VCPU thread preemption.

The other category of device timeout checks employed by VMS is for longer interval timeout checks, in the range above a second. When driver initiates an IO operation, it sets request for timeout measured in seconds since current time, initiates IO operation and returns control to the system. Under normal circumstances device will field an interrupt and driver's interrupt service routine will cancel pending timeout request. If timeout request was not cancelled by the time it is scheduled to expire, it is fired by VMS once-a-second timer routine and aborts IO operation as failed, and also depending on the driver may mark device as malfunctioning or even trigger system crash.

What is essential is that IO operation can be initiated on one processor (let us say, VCPU1 or VCPU2), and invocation of device handler also happens on this processor. However VMS system clock events are processed on the primary CPU (under VAX MP, VCPU0). In addition, some device handlers can relegate a part of their activity to separate IO processing threads (IOP threads).

If VMS device driver is invoked in the context of VCPU1, schedules a timeout using one of the mechanisms (considered below) deriving from the operating system timer routines, then starts IO operation on the device and VCPU1 thread get preempted[95], then VCPU0 can falsely trigger device timeout before the virtual device had a chance to perform required operation.

Similarly, if VMS device driver is invoked on any VCPU (either primary or secondary), starts IO operation and the handler relegates its execution to an IOP thread that gets subsequently preempted, then driver will experience false timeout.

Such preemptions cannot happen on real, silicon VAX with real hardware devices, but they can happen on the simulator running on top of general-purpose commodity operating system and therefore unable to provide the same hard real-time constraints as provided by the hardware.

[95] Preemption at this point may appear to be somewhat unlikely, since when accessing the device VCPU is likely to do so at elevated IPL and hence at elevated VCPU thread priority. However unlikely does not mean impossible: even high priority thread may still get preempted by the scheduler or incur page fault. Moreover, some SIMH device handlers do not initiate actual IO operation on the host system right away, instead they just schedule this operation to begin in a certain number of VCPU cycles. By the time scheduled activity actually starts, VCPU instruction stream may have already relinquished device spinlock and dropped the IPL, and hence also dropped thread priority to more easily preemptable by concurrent host workload. In the future it may be worth considering retaining elevated thread priority if short-term device events are pending in VCPU SIMH clock queue. Current version of VAX MP does not implement such retention, however it implements partially similar (albeit weaker) mechanism in the form of SYNCLK device protection window, as described in chapter "Timer control" below.

While thread preemption does not matter for uniprocessor simulator that does not utilize IOP threads, and where all activities therefore occur on the same single thread and thus are paused, resumed and delayed together and share the same virtual time scale, it matters for the simulator utilizing multiple threads, including IOP threads, and especially for the multiprocessor system simulator, where one VCPU thread can get paused while another continues to execute. Special measures need to be taken to address this problem unique to multiprocessor simulator and, to a smaller extent, also uniprocessor system simulator utilizing IOP threads.

These measures can fall into two possible categories. One possible solution would be to keep virtual processors in a lockstep, synchronized in their forward progress to a sub-second range. If any VCPU gets stalled, the other will wait for it to catch up. This will resolve the problem in a generic way (but only for devices that do not use IOP threads), and can also be implemented with little modifications to the existing VAX MP synchronization window mechanism (indeed, even with simplification of this mechanism) by applying it for VCPU execution either at all time or at any time there is IO device activity pending, rather than only during critical periods. However such tight and continuous synchronization between the VCPUs will introduce significant overhead and also limit scalability of the system, undermining the benefits and purpose of having a multiprocessor system in the first place. In addition, it won't cover virtual devices that utilize IOP threads anyway.

Therefore we opt for another, less generic but more efficient and SMP-friendly approach:[96]

- To consider the requirements of particular OpenVMS drivers servicing those IO devices that are supported by VAX MP.

- Extend driver timeouts where possible and easy.

  One way this can be achieved is by patching VMS EXE$TIMEOUT routine in the running system to intercept two instructions that check for timeout, and by marking device as having extended timeout (using one of reserved flags in CRB, DDB, IDB or DPT). [97]

  However this method would extend all timeouts for a given device indiscriminately and therefore would work only for device drivers that use timeouts *solely* to safeguard against device going non-responsive and do not intentionally use short timeouts in order to cancel pending IO operation, poll for device becoming ready, retry failed resource allocation or other similar legitimate purposes.

[96] Yet another conceivable approach would be to override timeout handling centrally, by intercepting invocations and modifying the behavior of guest OS timeout signaling routine (in VMS, EXE$TIMEOUT), and extending all device timeouts by safe margin. E.g. checking due time not against current time, but let us say current + 120 seconds. However this would have interfered with legitimate purposeful timeouts declared sometimes by the drivers in order to retry failed allocation attempts etc.

[97] VSMP tool includes two dyn-patches named CRBTMO and UCBTMO to hook on these instructions, however since this interception is not currently utilized, these dyn-patches are disabled by default.

- Where easy timeout extension is not possible and device does *not* use IOP thread, mark device as having affinity to the primary CPU. The latter will cause OpenVMS to always initiate IO operations on the device in the context of primary CPU only. If a process executing on a secondary CPU requests IO operation on the device, VMS IO subsystem will route the request to the primary processor and invoke the driver in the context of the primary processor. Thus, all device activity – including IO initiation, all handler activity, interrupt processing and timeout processing – will always take place on CPU0, thus eliminating the issue of synchronizing multiple processors to the device and preventing false driver timeouts.

- Where above approaches do not yield a solution, active intervention into particular driver is required as a fallback, such as modification of driver structures or driver's running code.

There are five ways VMS driver can request a timeout.

1. Driver can request device controller-wide timeout by setting controller request block (CRB)'s field CRB$L_DUETIME to target timeout expiration time typically computed by the driver as (EXE$GL_ABSTIM + N), for example:

   ```
   ADDL3    #N, G^EXE$GL_ABSTIM, CRB$L_DUETIME(R4)
   ```

   Location EXE$GL_ABSTIM contains the number of seconds since system boot time and gets incremented once a second. After incrementing this location, VMS scans (in a routine EXE$TIMEOUT executed once a second on the primary processor only) all CRBs linked in the timeout chain and compares CRB's CRB$L_DUETIME field with new EXE$GL_ABSTIM. If CRB$L_DUETIME is less or equal than EXE$GL_ABSTIM, device timeout is triggered and EXE$TIMEOUT invokes driver's controller timeout handling routine pointed by CRB$L_TOUTROUT.

   Note that since timeout can be scheduled shortly before the end of the current second, actual earliest timeout value for the device is not N, but (N – 1) seconds.

2. Driver can request per-unit timeout by setting unit control block (UCB)'s field UCB$L_DUETIM to target timeout expiration time calculated similarly to CRB's and also setting flag UCB$V_TIM in UCB$W_STS. Like in the CRB case, routine EXE$TIMEOUT executed once a second on the primary processor will scan all UCBs marked for timeout and invoke timeout handler (or resume fork process if handler was not specified).

   Driver typically requests per-unit timeout with WFIKPCH or WFIRLCH macros, for example:

   ```
   WFIKPCH TIMEOUT_HANDLER, #NSECS
   ```

   The argument to the macro is specified in seconds, and like in the CRB case actual earliest timeout value is not NSECS, but rather (NSECS – 1) seconds.

3. Terminal multiplexer drivers and console driver also request timeouts using TIMSET macro that ultimately also sets UCB$L_DUETIM.

4. Terminal drivers also have an additional way to request timeout that is utilized for read timeouts, such as for example expiration of login prompt.[98] It does not concern us here.

5. Driver can keep custom VMS timer queue entry (TQE) scheduled with routine EXE$INSTIMQ and do its own timeout processing.

We are now going to explore which timeout mechanisms and timeout values are used by VMS drivers of the devices supported by SIMH MicroVAX 3900, and plan the solutions.

SIMH VAX (virtual MicroVAX 3900) and VAX MP based on it support the following virtual devices:

Device ID	SIMH name	VMS name	VMS driver	Category
CR11	CR	CR	CRDRIVER	punch card reader
LPV11	LPT	LP	LPDRIVER	printer
RL11 (RLV12)	RL	DL	DLDRIVER	disk
DZV11	DZ	TT	DZDRIVER TTDRIVER	terminal multiplexer
DHV11	VH	TX	DZDRIVER TTDRIVER	terminal multiplexer
RQDX3 (KDA50)	RQ	DU PU	DUDRIVER (unit) PUDRIVER (controller/port)	MSCP disks
RX211 RXV21 RX02	RY	DY	DYDRIVER	floppy disk
MSCP tapes	TQ	MU PT	TUDRIVER (unit) PUDRIVER (controller/port)	tape
TS11 TSV05	TS	MS	TSDRIVER	tape
DEQNA DELQA	XQ	XQ	XQDRIVER	Ethernet controller
CPU console	TTI, TTO	OP	OPERATOR	operator console

Early prototype of VAX MP was based on SIMH 3.8.1 codebase. In SIMH 3.8.1 codebase all devices execute IO operations synchronously, without utilizing IOP threads. Handlers for disk devices (RQ, RL, RY), tape devices (TQ, TS), LPT and CR perform blocking IO: VCPU thread is frozen until actual underlying

[98] By setting flag UCB$V_TT_TIMO in UCB$W_DEVSTS and target expiration time in UCB$L_TT_RDUE.

host operation completes. Terminal multiplexers (VH, DZ), console port (TTI, TTO) and network adapter (XQ) execute synchronous but non-blocking IO on underlying sockets.

Synchronous blocking IO is undesirable even in the uniprocessor case, since it stalls VCPU execution until host IO is complete, thus effectively reverting simulator architecture back to early computer era times before interrupts were invented exactly to allow CPU utilization by other tasks (or perhaps the same task) in parallel to the IO, instead of making CPU to wait for IO devices.

For multiprocessor system however, synchronous blocking IO is even much worse and is extremely undesirable, since VCPU blocked while executing underlying host IO operation is likely to hold a spinlock for the device and also possibly some associated spinlocks (such as FILSYS spinlock for file IO or MMG spinlock if IO operation performs paging). In a real silicon VAX multiprocessor, CPU initiates IO asynchronous operation and quickly releases held spinlocks. In a simulator that implements IO operation via synchronous blocking IO to host files or other host objects, VCPU will hold the spinlocks for the duration of the operation, and other VCPUs will quickly serialize on the held spinlocks, with drastic impact on scalability and interactive response. The larger the number of VCPUs, i.e. the degree of parallelism, the higher is the contention, the more noticeable can expected to be the effects of holding spinlock for extended time. In our experience, interactive response on a loaded system with four VCPUs and synchronous blocking IO to the disk drive was tolerable if unspectacular, but at six VCPUs it was absolutely unacceptable (even output of “SHOW SYSTEM” was chunky, few lines at a time). MMG spinlock in this particular workload was a particular “offender”, remotely followed by FILSYS spinlock and PUA spinlock[99].

Therefore it was deemed a priority for VAX MP development to integrate SIMH 3.8.2 codebase changes that provide asynchronous IO for RQ and TQ devices (using IOP threads).[100] Such integration had been performed.

SIMH 3.8.2 also utilizes IOP threads for XQ handler, replacing network polling – that in 3.8.1 was performed by the primary VCPU thread every TMXR_MULT ticks, typically on every clock tick (i.e. 100 times a second) – with IOP thread. This change is less critical than the change for RQ and TQ, but is also welcome: it reduces polling overhead when network device is not actively used, and cuts response latency when device is actively used (especially for VAXcluster).

[99] Apparently some of VMS paging paths hold spinlock when initiating disk IO operation and get caught with spinlock held when disk read/write operation gets initiated and blocks the VCPU thread.

[100] For non-technical reasons these SIMH codebase changes were not actually enabled in SIMH itself in 3.8.2 and 3.9 releases (current at the time of writing this). SIMH asynchronous IO facilities are available in VAX MP. It is expected that they will be also released in a version of regular SIMH coming after 3.9.

Thus underlying host IO methods utilized by various VAX MP IO devices can be summarized, for initial intended VAX MP public release, as follows:

Device kind	SIMH name	Host IO method and use status
MSCP disks	RQ	Asynchronous (IOP), preferred kind of disk device to use
MSCP tapes	TQ	Asynchronous (IOP), preferred kind of tape device to use
Ethernet	XQ	Either asynchronous (IOP) or synchronous non-blocking polling
terminal multiplexers, console port	VH, DZ TTI, TTO	Synchronous non-blocking polling
floppy disk	RY	Synchronous blocking, but to memory buffer, so actually no blocking for extended time
tape	TS	Synchronous, blocking, intense use is discouraged (at least until the handler is modified to use IOP)
disk	RL	Synchronous, blocking, intense use is discouraged, certainly not as system disk
printer	LPT	Synchronous, blocking, intense use is discouraged
punch card reader	CR	Synchronous, blocking, intense use is discouraged

Now let's turn to particular devices and timeout utilization by their VMS drivers.

For most system configurations and usage patterns, MSCP disks are typically by far the most important kind of device in terms of the number and rate of IO operations performed. Luckily, their affinity does not have to be constrained to the primary processor, and they can be accessed for IO initiation by any virtual processor and use IOP threads without concern for risks of device timeout.[101] VMS MSCP/TMSCP port driver (PUDRIVER) uses a variety of timeouts during initialization, but they cannot cause the early timeout problem we are discussing here, because MSCP controller initialization on VAX MP is performed before multiprocessing is started and do not involve actual disk IO, and thus do not involve IOP threads: all controller initialization is performed within the context of primary CPU. VMS PUDRIVER also utilizes CRB timeout mechanism to establish periodic 15-second polling that checks for uncorrectable errors signaled in the controller's CSR, however there is no harm if some of these polls cycles will get triggered earlier (relative to device activity) than intended. VMS DUDRIVER uses timeout defined by the value of MSCP controller timeout reported by the controller at initialization. SIMH module PDP11_RQ.C(CPP) defines it via preprocessor symbol RQ_DCTMO with value of 120 seconds in the original uniprocessor SIMH version. This value is generally sufficient to safeguard against false early timeout due to VCPU

[101] Note that here and below we are discussing QBus and Unibus controllers, not BI controller or HSC.

thread preemption under the circumstances that can be considered valid for running the simulator. However to provide additional safety margin RQ_DCTMO can be increased up to 255 seconds.[102]

> *Important:* The analysis in this chapter does not take into account timeouts used by the drivers (any of discussed device drivers) during initialization. Therefore, drivers on VAX MP should never be reloaded while multiprocessing is active. Should the need occur to reload the driver, stop secondary processors first (via DCL command STOP /CPU/ALL), reload the driver with only the primary processor active, and then start secondary processors again (with DCL command START /CPU/ALL).

Similarly, TQ (MSCP tape) controlled by VMS TUDRIVER can be used without restricting device affinity to the primary processor or practical concerns about IOP thread preemption. TQ controller timeout is defined by symbol TQ_DCTMO in file PDP11_TQ.C(PP) that has value of 120 seconds in the original uniprocessor SIMH version. This value is normally more than adequate to avoid the problem of false early device timeout, but can be further increased up to 255 seconds for an additional safety margin.

RL handler currently does not use IOP threads, for this reason its use with VAX MP in multiprocessor mode is strongly discouraged, as RL would significantly degrade multiprocessor performance and responsiveness. Use RQ (MSCP) disks instead of RL. Even though the use of RL on a multiprocessor VAX MP is not advised, it is possible if for some reason is absolutely required, such as perhaps for lightly accessed data disks or if VAX MP is executed in uniprocessor mode, with only one active processor enabled. VMS RL driver uses timeouts of 1 and 5 seconds; we did not analyze if they can all be extended indiscriminately. Current solution is that DL device affinity must be set to the primary processor.

> Note that some OpenVMS components (such as e.g. MSCP server serving local disks to the cluster and others components[103]) perform IO operations on local disks that bypass $QIO interface. However these operations are still executed via EXE$INSIOQ(C) or IOC$INITIATE and therefore will be routed, if necessary, in accordance with device affinity. Therefore disk devices with restricted affinity can be safely used in the system, including being served to other nodes the cluster.

RY handler does not use IOP, however it performs IO to in-memory buffer holding floppy disk data, and thus blocking is not a significant issue. VMS DYDRIVER uses a range of timeouts down to 1 second. DYDRIVER does not depend on short timeouts and it would be possible to extend all timeouts on DY device indiscriminately. However the significance of this device is very marginal, therefore our current solution is to have DY device affinity set to the primary processor.

[102] With existing DUDRIVER code controller timeout can in practice be increased to even higher value (0xFFFF - 10), however MSCP specification states that no controller should ever return controller timeout value greater than 255. If controller returns 0 (no timeout), DUDRIVER will default to 255 seconds.

[103] Including MSCP server, F11 ACP/XQP, volume shadowing, quorum disk monitor, cluster cache, cluster IO to multi-host drives, access to cluster incarnation file.

TS handler currently does not use IOP threads, for this reason heavy use of TS with VAX MP in multiprocessor mode is strongly discouraged, as it would degrade multiprocessor performance and responsiveness. Use TQ (MSCP) tape instead of TS. Nevertheless, albeit heavy use of TS is not advised, light or occasional use of TS is possible, as well as the use of TS in uniprocessor mode. VMS TS driver uses a range of device timeouts down to 1 second; we have not analyzed whether these timeouts can be extended indiscriminately. Current solution is that TS must have device affinity set to the primary processor.

> Similarly to disk drives, some system components including TMSCP server and MTAACP perform IO operations on tape devices bypassing $QIO interface, via EXE$INSIOQ(C). However appropriate inter-processor routing is still performed in case of restricted device affinity. Therefore it is safe to use tape drives with restricted affinity both locally and served via cluster.

VMS VH driver uses a range of device timeouts down to 1 second to detect multiplexer's failure to perform the output of characters and acknowledge transmission[104] with the interrupt. VH devices (VMS TX*xx* units) must:

- Either have their affinity set to the primary processor.
- Or have timeout use disabled altogether by setting flag TTY$V_PC_NOTIME in UCB$W_TT_PRTCTL. But in this case driver will fail to properly timeout and terminate the IO request if for whatever reason transmission fails to complete, e.g. due to disruption of telnet connection to the line.
- One conceivable option could have been to extend VH timeouts indiscriminately by intercepting the check for UCB$L_DUETIM in EXE$TIMEOUT. YFDRIVER itself does not use timeouts for any reason other than to detect device failure to respond due to malfunction. However TTDRIVER does have an instance of purposeful use of timeouts with different intent: when character frame error is detected (under VAX MP it happens when BREAK command is received on telnet connection), TTDRIVER toggles to 600 baud for 2-3 seconds. Extending timeouts indiscriminately would have extended this timeout too.
- Another problem with TTDRIVER is race condition between routines TT$TIMER and MODEM$LINK_CRB that do not properly synchronize access to variable TT$L_DIALUP. This variable is accessed only when a modem used, so does not really apply to VAX MP. However were it utilized, problems arising out of this race condition would more likely manifest themselves if there were several multiprocessor threads of control executing TTDRIVER.

Therefore on the balance the choice is against trying to extend VH timeouts indiscriminately in order to run VH with unrestricted affinity (it may be possible, but would have required an effort to address mentioned issues). Current choice is to limit VH (TX) units affinity to the primary processor.

VMS DZ driver uses a range of device timeouts down to 1 second to detect multiplexer's failure to perform the output of characters and acknowledge transmission[105] with the interrupt. It also uses

[104] This includes any kind of transmission, such as echoing of typed characters etc.

timeouts to purposefully abort (cancel) the transmission; this prevents treating DZ timeouts indiscriminantly since indiscriminate extension would change abortion time of the request for burst transmission from few seconds to minutes in case telnet connection to DZ line is aborted, which is unacceptable. In addition, DZDRIVER shares for problem of race condition in TTDRIVER and further exacerbates it by adding similar race condition inside DZDRIVER itself, on variable DZ$L_DIALUP between routines DZ$TIMER and DZ$SET_MODEM. Current solution therefore is that DZ devices (VMS TTxx units) must have their affinity set to the primary processor.

VMS CR driver uses timeouts down to 1 second, some to poll for device coming online. It might be possible to extend device timeout by a bit, since timeout leads to retry, not failure. However device is very marginal and is not worth the effort. Current solution therefore is to set device affinity to the primary processor.

VMS LP driver uses timeouts in range 1 to 11 seconds. Current solution is to set affinity to the primary processor.

VMS console driver. VAX MP also always sets operator's console (OPA0) affinity to the primary processor since VMS assumes only primary processor has access to the console.[106]

VMS XQ driver presents the most complicated case. It uses a range of timeouts for multiple purposes; some of these timeouts apply only during initialization; but the remaining ones still cannot be extended indiscriminantly, since they include intentional short timeouts, such as to retry failed resource allocation. Of all timeouts used by XQDRIVER, the only one that can be falsely triggered by preemption of VCPU or IOP threads is the transmit timeout. XQDRIVER uses transmit timeouts of 5 seconds, and if controller fails to acknowledge transmission within this interval then the driver shuts the controller down, so obviously false early timeout prevention is a must. XQ is also a device that on many common configurations can generate a high rate of IO requests, perhaps rivaling or even surpassing local hard drives (think of Local Area VAXcluster, possibly LAD and FTP transfers and so on), so restricting its affinity to the primary CPU is undesirable from the performance standpoint.

> In addition to this, many components that use XQ do not go through regular SYS$QIO interface rerouting invocation of driver's START or ALTSTART entry point according to device affinity. Instead, these components use fast internal interfaces (FFI and VCI) that call driver's internal routines directly on the local processor and thus do not respect device affnity mask.[107] It may be possible to augment VAX MP with capability to intercept invocations on these interfaces and re-route the invocations for execution of the primary processor, but it would be cumbersome and

[105] This includes any kind of transmission, such as echoing of typed characters etc.

[106] In practice under VAX MP any VCPU can write to the console via PR$_TXCS and PR$_TXDB or routines that access them, but only the primary VCPU receives console input.

[107] These components include SCS (including generic VAXcluster communication messages and MSCP/TMSCP traffic via PE NI-SCA port), LAT, TCP/IP stack, DECnet, most likely LAD and possibly DFS and so on.

prone with variety of synchronization issues. This is another reason why restricting XQDRIVER affinity is undesirable.

Therefore our solution is not to restrict XQ affinity, but instead to enlarge XQ driver's timeout from 5 seconds to a larger safe value by patching driver's structures and cod. This patching is implemented as a part of VSMP tool. No modifications are done to on-disk copy of the driver. Only active, loaded and running copy of the driver and its structures in system memory is modified. VSMP patch id for XQ driver is XQTIMXMT. Extended transmit timeout can be set with VSMP LOAD command option XQTIMEOUT and can range between 5 and 255 seconds; default value is 255. The patch modifies value of field LSB$G_QNA_TIMXMT in LSB structure of every XQ controller on the system and also modifies instruction in XQDRIVER's routine SUB_START_CTRL_TIMER that initializes this field. After these modifications are performed, XQDRIVER reloading is disabled. Therefore XQDRIVER should be loaded and devices configured before VSMP LOAD command is executed.

To simplify device management as described, VAX MP includes command file DEVICES.COM that should be executed before multiprocessing is started and that assigns appropriate affinities to the devices found on the system and may also in the future assign extended timeouts to the devices.

Timer control

One important goal of SIMH/VAX MP clock event processing is to provide timely delivery of 100 Hz virtual VAX clock interrupts to virtual processors and their execution, so that (1) guest system time could be maintained accurate and with as small drift as possible and (2) virtual processors won't miss clock tick interrupts too often and could stay well-coordinated. Proper and accurate timing is desirable for the uniprocessor case, but is even more important for the multiprocessor configuration.

Implementing prompt and timely delivery of virtual clock interrupts is a tricky matter made even more complicated by the fact that most of the time VCPU threads run at below-normal priority, and therefore:

- are easily preemptable by higher priority threads running on the host system, and
- are also scheduled in round-robin fashion with other threads of the same priority range[108] that may be running in the workload of the host system.

Original SIMH design implements clock events processing within the regular control flow of the thread executing VCPU instruction stream. This means that if the VCPU thread gets preempted by a higher priority thread, delivery of virtual clock interrupts to guest OS running on this VCPU will be delayed until higher priority thread relinquishes the host CPU. Likewise, if a VCPU thread gets preempted by the thread of similar priority, virtual clock ticks won't be processed until that other thread's scheduling

[108] Since many host systems implement dynamic priority adjustment of the processes, "neighbor priority range" for a given thread includes processes and threads of slightly lower base priorities, and also of slightly higher base priorities, that may get co-scheduled in more or less round-robin fashion with respect to each other.

quantum expires, or perhaps the sum of scheduling quanta for multiple processes contending for the execution expire.

To appreciate numeric magnitude of the problem, consider that virtual VAX clock interrupts should be delivered to the guest OS every 10 ms and preferably not missed or delayed much, whereas "normal" round-robin scheduling quantum for processes in Microsoft Windows host environment may range anywhere between 20 to 400 ms[109]. Scheduling timeslice can be decreased for the process in question on Microsoft Windows when a process uses multimedia timers, such as SIMH/VAX MP does, however there may be no similar facility on other host systems. On OS X default preemption rate is 10 ms[110]. Linux behavior depends very much on kernel build configuration.[111]

VAX MP attempts to alleviate this problem through the use of optional clock strobe thread. This feature is enabled if conditional USE_CLOCK_THREAD is defined as TRUE during VAX MP build or if global variable `use_clock_thread` is set to TRUE before the simulation is started. Under currently targeted host systems (Windows, Linux and OS X) this is the default VAX MP build option.

Clock strobe thread runs at high priority and wakes up every 10 ms to broadcast clock interrupt requests to all VCPU threads for all active virtual processors.[112] VAX MP interrupt delivery system ensures that target VCPUs' thread priorities are elevated too when an interrupt is posted to them and that interrupt processing happens at an elevated thread priority, until an interrupt is dismissed and there are no other interrupts pending and processor reverts back to low IPL.

> Broadcasting clock interrupt every 10 ms to every virtual processor is a simplified picture.
>
> To reduce the overhead of unnecessary context switching, VAX MP clock thread may not broadcast clock interrupts to *idle* secondary virtual CPUs on every tick. Instead it broadcasts clock interrupts to idle secondary processors at much lower frequency, in case multi-click sleep time is permitted by paravirtualization layer running inside guest OS kernel.
>
> When paravirtualization module signals to VAX MP that given VCPU is in idle state and its thread can go to idle sleep, the module can also signal its willingness to stay in idle sleep for multiple

[109] Mark Russinovich, "Windows Internals", 5th edition, Microsoft Press, 2009, chapter 5, esp. section "Thread Scheduling".

[110] Amit Singh, "Mac OS X Internals", Addson-Wesley, 2007, ch. 7.4 ("Scheduling"), p. 775-777.

[111] Key relevant parameters are CONFIG_PREEMPT and various timer-related configuration parameters: CONFIG_NO_HZ, CONFIG_HZ, CONFIG_HZ_100, CONFIG_HZ_250, CONFIG_HZ_300, CONFIG_HZ_1000, CONFIG_HIGH_RES_TIMERS, CONFIG_HPET_TIMER, CONFIG_X86_PM_TIMER, CONFIG_SCx200HR_TIMER, CONFIG_TIMER_STATS etc.

[112] Actually clock strobe thread broadcasts not CLK interrupts, but non-maskable SYNCLK interrupts that are visible only inside VAX MP runtime and are not directly visible at VAX level. When received, SYNCLK interrupt is eventually converted by the receiving VCPU to CLK interrupt. This conversion may happen with an intentional delay, as described later, to provide a measure of virtual time elasticity. It may not happen at all if CLK interrupts are disabled on this VCPU via ICCS register.

clock ticks, without being woken up on every clock tick. Paravirtualization layer checks pending system activity on this VCPU, and if there is none for several next ticks, paravirtualization module will signal to VAX MP that this VCPU is able to hibernate for the specified number of clock ticks.

Only secondary VCPUs can go into multi-click idle sleep.

Primary processor under VMS is unable to perform multiple-tick sleep when idle, since it is responsible for managing system-wide structures (such as system clock queue and others), and also for maintaining system-wide time shared by all the processors. Therefore primary processor has to receive clock interrupts with 100 Hz frequency in order to maintain proper system state, but secondary processors, when idle, can, with proper paravirtualization assistance and back-to-back clock interrupt reissue by VAX MP simulator (as described later), afford to be brought out of hibernation at a frequency of just several Hz.

To pass on the intent for multi-tick sleep signaled by the paravirtualization module, VAX MP VCPU thread sets, before hibernating, a counter of maximum clock interrupts it is willing to skip without being immediately woken up (*multiple-tick-sleep*), and also sets to zero another per-CPU counter – of pending clock interrupts for this CPU (*tick- skip-counter*). Whenever clock strobe thread runs subsequent clock cycle, it treats hibernated VCPUs that have *multiple-tick-sleep* counter set in a special way. Clock strobe thread broadcasts clock interrupt to all other processors, however for hibernated VCPU threads with *multiple-tick-sleep* set, clock thread increments *tick- skip-counter* and compares the result with *multiple-tick-sleep*. If resultant value of *tick-skip-counter* is less than *multiple-tick-sleep*, clock strobe thread does not post clock interrupt to the target VCPU on this clock cycle and lets this VCPU thread stay hibernated. If however the counter of skipped clock interrupts reaches the limit indicated by *multiple-tick-sleep* counter, clock strobe thread posts an interrupt to the target VCPU. When target VCPU thread finally awakes, either because its idle sleep counter expired and clock interrupt was finally delivered or processor was woken up in other ways[113], target processor checks *skipped-tick-counter* and if it is not zero, executes multiple clock interrupts "back to back". "Back to back" execution allows to maintain guest OS statistics, such as performance counters, and to perform system maintenance functions, such as VMS SMP sanity checks – at low overhead, since it incurs no context switch on the host system.[114]

[113] Such as either because an inter-processor interrupt was posted by another VCPU to this processor, or because VMS modified idle processors mask SCH$GL_IDLE_CPUS and the modification was intercepted by the simulator and so on. For a full list of possible wakeup causes see invocations of routine `wakeup_cpu` in VAX MP source code.

[114] Multi-tick idle sleep optimization is not implemented in the initial release of VAX MP. OpenVMS VSMP resident code does pass number of ticks to sleep to VAX MP SIMH layer, however routine `sim_idle` in the simulator code currently ignores passed "maxticks" argument and always sleeps up to next clock tick only at maximum. It had been decided to withhold multi-tick sleep feature from the initial release of VAX MP since potential benefits do not justify introduction of additional complexity for required state tracking in the simulator, for the initial release. This feature may be implemented in a subsequent VAX MP release.

> Described optimization is intended to reduce the overhead by eliminating unnecessary context switches on the host system, typically reducing them 10-fold or more for idle secondary processors. It also allows host operating system to implement efficient power management of physical CPUs, shifting idling cores into power state with lower power consumption and heat dissipation. This, in term, may allow on-chip thermal management to boost frequency and speed of active cores.
>
> Possibility of multiple-tick sleep for secondary processors is indicated by paravirtualization layer executing in the kernel of guest OS and able to detect when a secondary processor is idle and does not have any pending processing that cannot be postponed and executed during “back-to-back” clock interrupt processing.
>
> Multiple-tick idle sleep optimization is implemented whether clock strobe thread is used or not and is not bound to the use of separate clock strobe thread. Secondary processors will execute multiple-tick sleep whenever possible regardless of whether clock strobe thread is employed.

Delivery of clock interrupts via clock strobe thread, running itself at elevated priority and elevating target VCPU threads’ priority when dispatching clock interrupts to VCPUs, seeks to ensure that clock interrupts are delivered timely and are processed promptly, and also to ensure better clock synchronicity between the processors.[115]

> Note that clock strobe thread does not broadcasts raw VAX CLK interrupts directly, it broadcasts instead internal SYNCLK interrupts that are not directly visible at VAX level. When SYNCLK interrupt is received by target virtual processor, the recipient VCPU performs various internal tasks, such as local clock event queue management (triggering events that were co-scheduled with the clock), recalibration of processor speed etc., and in addition to these activities also translates received SYNCLK to CLK, assuming CLK interrupts are enabled in the ICCS register – sometimes performing this interrupt translation with intentional delay as described below in this section.

However the usability of clock strobe thread approach is dependent on host system having preemptive scheduler and being able to process timer events and perform thread rescheduling and context switches at sufficient temporal resolution (better than 100 Hz). These are the features normally expected of contemporary SMP-capable host OS’es. However it may potentially be possible that some host OS may lack priority-based true preemptive scheduler and limit preemption points to the boundary of scheduling time slice only; i.e. high-priority process may become runnable, but will not be actually scheduled for the execution until host scheduler’s current time slice is expired, and currently executing lower-priority process will continue running till the end of current time slice. If the duration of time slice is large, this will effectively limit possible frequency of context switching and therefore frequency of clock interrupt delivery and processing. This would defeat clock-strobe thread scheme and make it either unusable or of limited utility.

[115] Clock strobe thread loop is also a natural place for sampling performance counters.

In this case VAX MP has no choice but to fall back on the original SIMH clock interrupt dispatch scheme within the execution flow of preemptable low-priority thread (selected via `use_clock_thread` set to FALSE) and embrace the consequences of delayed/missed clock interrupts. From guest operating system's viewpoint it will be equivalent to some "black hole" code blocking interrupts for an extended time.

> As minor alleviation for this situation, VAX MP tries to reduce the probability of VCPU thread staying in a voluntary idle sleep for much longer than it bargained for. If idle sleeping is enabled and VCPU thread intends to go into idle sleep, such as till the next clock tick (or other pending clock queue event), the thread elevates its priority before entering timed sleep. This increases the probability of the thread gaining control back as soon as timed wait expires, and not being delayed by round-robin scheduling with competing processes. Once the VCPU thread gets control back, it checks if virtual clock interrupt is about to happen. If not, VCPU thread drops its priority back to below-normal and may be either rescheduled or let run at this point , depending on host operating system whims. If virtual clock interrupt is imminent however, VCPU thread retains its priority until after the clock interrupt had been processed and there are no more interrupts pending. This technique does not improve clock interrupt strobing for compute-bound VCPUs, but it may somewhat improve it for idling or lightly loaded VCPUs.

One significant distinction between the scheme with dedicated clock strobe thread and original SIMH scheme for clock interrupts is greater elasticity of virtual time in the latter:

> In SIMH original scheme clock interrupts are timed to CPU clock cycles counting, based on a measured rate of cycles per second that is adjusted once a second. Thus (unless competing host system load is very adverse), interval between clock interrupts is adjusted gradually[116] and there is usually certain number of CPU cycles available between successive clock interrupts (equal to last measured cycles per second divided by 100 Hz clock frequency).
>
> In contrast, SYNCLK interrupts generated by clock strobe thread are delivered rigidly, at fixed real-time intervals or close. If VCPU thread is starved out of execution, it may receive SYNCLK interrupts with few or none cycles available between them.
>
> *In SIMH original scheme CLK interrupts are like pen marks on the rubber-band of execution stream, a section of which may stretch and contract as a whole, as PCPU time allocation to VCPU changes, with distance between the marks changing more or less proportionally. In clock strobe thread scheme, SYNCLK interrupts are like nails through the rubber-band of execution stream nailing it to the real-time scale: resultant virtual time scale is not elastic.*
>
> In the original uniprocessor SIMH elasticity of virtual time scale and its semi-independence from real wall clock time scale provides a buffering that creates an illusion of uniform virtual time: simulator events can be scheduled in terms of this elastic uniform virtual time. For

[116] Or rather, it can be made to adjust gradually if moving average by at least two measurements is used instead of single last virtual second sampling.

multiprocessor case exactly this buffering (as long as it is not constrained to small limits) is undesirable, since virtual processors need to be coordinated, however their time scales are independent and have independent elasticity.

The latter effect, induced by the impact of adverse host system load, could *potentially* affect the stability of the simulator in two ways, unless compensatory measures are taken:

1. SIMH device events could fire later (in real time terms) than desirable. They could also fire in different order with respect to events co-scheduled with clock interrupts or clock itself than was intended by virtual device designer.[117]

2. System activities executing at IPL levels below CLK could be starved – activities like VMS fork processes and timer queue entries, with potential undesirable consequences for system stability. This is also an unlikely development, since VAX MP executes these activities at elevated thread priority level (CRITICAL_OS or CRITICAL_OS_HI), thus guarding them from starvation.

Non-elasticity of virtual time under SYNCLK strobing has no *apparent* and *known* destabilizing effect. Nevertheless as an extra margin of safety, VAX MP adds a limited measure of virtual time elasticity to the scheme utilizing clock strobe thread. After SYNCLK interrupt is processed[118], simulator is protected from processing subsequent SYNCLK interrupt for about 2000 VAX instruction cycles. If another SYNCLK event is delivered to the VCPU before protection interval had expired, VAX MP will mark SYNCLK event as pending, elevate VCPU thread priority to CRITICAL_OS_HI (so VCPU could clear protected interval as quickly as possible), but will not process SYNCLK interrupt until approximately 2000 VAX instructions had been executed by this VCPU since previous SYNCLK had been processed. This limited measure of virtual time elasticity should provide ample "breathing space" for SIMH device handlers' events to execute properly, and also to prevent the starvation of system activities at IPL levels corresponding to CRITICAL_OS.

Protection interval is composed as a union of two intervals tracked by respective count-down counters: device protection counter and OS protection counter. When protection interval is started,

- OS protection interval counter is initialized with the value of SIMH configuration variable SYNCLK_SAFE_CYCLES.

  This setting is accessible via SIMH console commands DEPOSIT and EXAMINE and is expressed as the number of VAX commands that defines the duration of OS protection interval. Default value

[117] Under extremely adverse host load, this could even potentially trigger guest OS level timeouts before device event could have been signaled and processed – albeit this is not realistically possible, since the number of cycles taken by VMS clock interrupt handler is higher than SIMH short-span device timers such as RQ_XTIME, but illustrates one way potential problems could occur if not watched after.

[118] That may trigger raising CLK interrupt and execution of SIMH device events co-scheduled to be executed with clock tick.

of SYNCLK_SAFE_CYCLES is 2000 VAX command cycles.

- Device protection interval counter is initialized with the longest short-timer scheduling interval for any device in the clock queue entry, ignoring CPU device, throttling device, any device co-scheduled with the clock, and any device scheduled for time beyond SYNCLK_SAFE_CYCLES cycles.

Interval of OS protection from SYNCLK is terminated when one of the following events occurs:

- Protection interval cycles counter counts down to zero, i.e. 2000 VAX instructions had been executed since previous SYNCLK event had been processed.

- Processor exits kernel mode.

- Processor IPL drops to a level below minimum level of starvation-protected system activities. For VMS this is IPL 6 (IPL_VMS_QUEUEAST). The value is configured by paravirtualization module when it is loaded in guest OS kernel. Before the module is loaded, i.e. during the execution of firmware, system bootstrap and any other operations performed before loading the paravirtualization module, VAX MP uses for checks the value of simulator register SYNCLK_SAFE_IPL that can be set with SIMH console DEPOSIT command and that defaults to IPL_VMS_QUEUEAST (6).

- Processor tries to enter idle sleep.

Interval of device protection from SYNCLK expires when device protection counter counts down to zero.

Summary protection interval is terminated or expires when both OS and device protection intervals terminate or expire.

Once summary protection interval expires or terminates and processor is out of the protection interval, newly received SYNCLK event can be processed immediately. Any pending SYNCLK event (received during protection interval) will also be processed on the expiration or termination of protection interval. Once protection interval terminates or expires, VCPU thread priority will be recalculated and, if no high-priority activity is pending for this VCPU, dropped below CRITICAL_OS_HI level.

* * *

Under particularly adverse conditions, current initial release of VAX MP may allow SMP sanity bugcheck to be triggered by OpenVMS, unless this particular bugcheck control is relaxed or disabled by OpenVMS SYSGEN parameters.

There are two basic kinds of SMP sanity bugchecks in VMS. One, with bugcheck code CPUSPINWAIT, is bugchecks generated when a code is unable to acquire a spinlock within a specified maximum amount of time or receive a response from another processor to a synchronous inter-processor request within certain time limit. Timeout limits are actually represented by busy-wait loop counters calibrated at system start; i.e. the system times out not when wall clock time expires, but instead waits in a busy-wait loop for a maximum number of loop iterations (established in advance by loop calibration and SYSGEN parameters) and times out when loop counter expires. VAX MP protects against this kind of bugchecks using synchronization window mechanism: if a processor holding the lock or needing to respond to an IPI request falls too much behind in its forward progress because its thread was preempted (either because of scheduler decision due to competing system workload, page fault or blocking IO delay), other processors will stall waiting for the lagging processor to catch up. Synchronization window monitors a divergence in VCPUs cycle count, not wall clock, and is capable to guard against CPU cycle based busy-waiting thresholds.

However VMS also has another kind of built-in sanity bugchecks. All active CPUs in the VMS multiprocessing system are linked in a circular chain, and every processor in the chain monitors next processor down the chain. On each clock tick, monitoring processor increases a counter in the per-CPU database of the monitored processor. Once this counter exceeds the threshold, a bugcheck is triggered. Monitored processor resets its own counter on each of its clock ticks. Thus a bugcheck (with code CPUSANITY) is generated if some processor failed to process a clock tick for an extended time. The amount of this time is defined by SYSGEN parameter SMP_SANITY_CNT with default value of 300 ticks, i.e. 3 seconds.

This means that if some VAX MP VCPU was never scheduled in execution for about 2 seconds[119] and failed to process any virtual timer event during this interval, while VCPU monitoring it had been able to successfully receive and process over 200 virtual clock interrupts during this time, then CPUSANITY bugcheck will be triggered. This scenario can occur in three circumstances:

- VCPU thread is preempted by host scheduler for an extended time due to competing host system workload. When SYNCLK is used, this is an unlikely cause in practice, because it would mean that one VCPU thread with priority boosted to CRITICAL_OS_HI had been completely or virtually completely preempted for a long time, while other VCPU thread was allowed to run during this time and perform 200 clock interrupts. This cannot, or rather should not, happen under a scheduler with any measure of fairness and honoring thread priorities.

  Further, it is also made very unlikely by existing synchronization window mechanism based on VCPU cycle counters – unlikely but not altogether impossible yet: while cycle-based synchronization window prevents compute-bound monitoring VCPU from getting too much ahead of the monitored VCPU, it will not block clock processing on non-compute-bound monitoring VCPU.

- VCPU thread gets blocked due to synchronous IO performed by the simulator in the context of this thread. This reason is very possible with initial VAX MP codebase derived from SIMH 3.8.1

[119] 3 seconds less possible effects of multiple-tick sleep less possible legitimate VMS blocking of the clock interrupts.

that performs disk and other IO as blocking operations in the context of VCPU thread. Under heavy IO load, it may be quite possible for a VCPU thread to get blocked by synchronous IO operation for an extended time if the host system is overloaded heavy IO. It can also happen if thread tries to access virtual disk or virtual tape container file located on host physical disk that had been spun down due to host system power-saving settings. Spinning such disk back up can easily take up to several seconds. VCPU will be effectively hung during this IO operation time.

Implemented solution for this problem is to move blocking IO operations out of the context of VCPU thread to a dedicated IO processing (IOP) thread, so that VCPU thread does not get blocked by IO operations. Currently this solution is implemented for MSCP disk and tape devices only, and also for Ethernet controllers, and must be enabled with the command in VAX MP startup script.

- VCPU can also get stalled due to page faults. This cause is not preventable. VAX MP can and does try (if enabled by simulator settings) to reduce incidence of page faults by locking VAX main memory segment and simulator code and data into working set or physical memory[120], however such locking may not always be possible and is optional, and in any event VAX MP is unable to lock paged host operating system code or host operating system paged pool. Therefore intent of memory locking performed by VAX MP is to *reduce* paging, in order to reduce serialization and its negative impact on scalability and responsiveness in a virtual multiprocessing system, rather than to provide a guarantee mechanism for stability – such mechanism has to be complementary. VAX MP is unable to *guarantee* that page faults won't happen at all – they can and they will.

Therefore, a complementary mechanism needs to be provided that guarantees against CPUSANITY bugchecks. This mechanism is not incorporated in the current release of VAX MP simulator and will be provided as a built-in feature in a later release.

Shorter term workarounds for current VAX MP release are:

- Increase VMS SYSGEN parameter SMP_SANITY_CNT (maximum number of clock ticks processor is allowed not to clear its sanity counter) from its default value of 300 (3 seconds) to a higher value, such as 3000 (30 seconds) or higher, up to 65000 (yielding 650 seconds).

- Alternatively, disable inter-CPU sanity monitoring altogether via a flag in SYSGEN parameter TIME_CONTROL. Set VMS SYSGEN parameter TIME_CONTROL to either 2 (to disable processor chain SMP sanity checks) or 6 (to disable all SMP sanity checks).

Later VAX MP release will address this deficiency by including built-in protection against clock interrupt divergence (in addition to CPU cycle count divergence) as a part of synchronization window mechanism. It is not included in the initial VAX MP release because straightforward control for CPU clocks interrupts

[120] See chapter "Effects of host paging" below.

received in combination with CPU cycle divergence control may cause a deadlock: VCPU A may be waiting for VCPU B to process clock interrupt, while VCPU B may be blocked waiting for VCPU A to make some forward progress in terms of cycles executed. Therefore a more subtle synchronization window algorithm is needed, which was deemed to be a lower priority compared to other tasks for the initial release.[121]

Effects of host paging

In a real hardware VAX, physical memory is always accessible within short predictable interval. Likewise, microcode is always resident and each micro-instruction takes short predictable time to execute.

In a software simulator, virtual VAX's main memory is located in the virtual memory of the simulator process running on the host operating system and is pageable. Likewise, simulator code and control data, runtime libraries, memory heaps, runtime libraries internal data and so on – are all also located in virtual memory and are pageable.

If VCPU thread incurs page fault while this VCPU is holding VMS spinlock or mutex, other VCPUs will have to stall as soon as they need this spinlock, and busy-wait for original VCPU thread to complete its page fault, which may take an extended time. Likewise, otherwise runnable VMS processes will have to stall waiting for mutex. Similar stall will occur if VCPU thread is waiting to complete its page fault while other VCPUs are sending inter-processor interrupt requests to it. Likewise, if VCPU incurs page fault while holding a VAX MP VM-level lock on internal VM data structures, other VCPU threads will serialize.

This serialization will reduce scalability, degrade virtual multiprocessing system performance, and its responsiveness.

Furthermore, stalls that started as singular (pair-wise) can lead to waiting chains (convoys) further degrading performance, scalability and responsiveness of the system.

It is therefore essential for efficient and responsive VSMP system (much more so than for an uniprocessor simulator) that the simulator does not page much, including the simulator itself and its internal data, and "physical" memory segment provided by the VM.

VAX MP provides user-controllable mechanisms intended to reduce paging. These mechanisms are specific to host operating system.

Note that these mechanisms are not intended to prevent paging altogether, but merely to reduce paging rate to a level that does not cause any significant degradation of system scalability and responsiveness. Some paging will always happen – at a minimum of pageable operating system code and paged system pool accessed on behalf of the process.

[121] One possible approach may involve coalescing two scales (timer ticks scale and the scale of forward progress in terms of instruction cycles) into a single composite scale. Since it is ultimately the existence of multiple scales that leads to the possibility of deadlock between them.

Under Windows, VAX MP provides a way to increase working set size of the simulator process. Working set size is controllable with two configuration variables accessed via SIMH console commands DEPOSIT and EXAMINE. These variables, named WS_MIN and WS_MAX, define minimum and maximum size of the working set expressed in megabytes. As a rule of thumb, it is recommended to set WS_MIN to the size of VAX "core memory" as defined by SET CPU command (e.g. SET CPU 256M) plus extra 40 MB for WS_MIN and extra 100 MB for WS_MAX. For example, for 256 MB VAX memory configuration, execute

```
sim> DEP  WS_MIN  300
sim> DEP  WS_MAX  360
```

Alternatively, setting variable WS_LOCK to 1 causes both WS_MIN and WS_MAX set to appropriate values:

```
sim> DEP  WS_LOCK  1
```

That will set WS_MIN and WS_MAX to heuristically reasonable values, as described above. However if you want fine-grain control, you can set WS_MIN and WS_MAX directly instead. Note that setting WS_MIN or WS_MAX to non-zero value will also set WS_LOCK to 1.

Linux does not have a notion of per-process working sets. It has one system-global working set for all the processes in the system and the operating system itself. LRU-like page replacement is performed on this system-wide working set as a whole. It is possible to lock process pages into this working set and thus effectively in memory. This is achieved by "DEPOSIT WS_LOCK 1" as described above. For locking to work on Linux, caller's account must have high RLIMIT_MEMLOCK (definable on per-account basis in /etc/security/limits.conf), or VAX MP executable file should be granted privilege CAP_IPC_LOCK.

If you want to have a cross-platform VAX MP configuration script file usable both on Windows and Linux machines, you can define either WS_MIN/WS_MAX or WS_LOCK in it. Setting WS_MIN or WS_MAX to non-zero value causes WS_LOCK to be set to 1, and inversely, setting WS_LOCK to 1 causes WS_MIN and WS_MAX being set to a heuristically calculated values.

If VAX MP is unable to perform extending working set or locking code and data in memory, it will display warning message at first BOOT CPU command, or first RUN, CONTINUE, GO and STEP command.

More on the effects of Hyper-Threading

If host system uses hyper-threaded (SMT) processor or processors,, some VCPUs may come to share the same core in hyper-threaded mode of execution – especially if the number of configured and active VAX VCPUs exceeds the number of host cores.

This will have significant observable consequence: high-priority process executing inside guest OS can be effectively slowed down by a low-priority process.

While these processes will be executing on separate VCPUs, but host thread for these VCPUs may come to share the same physical processor core, and the core itself (in most existing modern processors) treats all active streams of execution as having equal priority. Therefore despite the processes have different priority within guest OS, high-priority process will share host core's execution resources on an equal basis with lower priority process and will be effectively slowed down by it.

To avoid such slow-down, it may be desirable to configure virtual VAX multiprocessors to have at most the same number of VCPUs as the number of physical cores on the host system, rather than the number of host logical processors created by hyper-threading. For example, for host computers utilizing Intel i7 CPU chip that has 4 cores, each of these cores further hyper-threaded to a total of 8 logical processors, avoidance of sharing the core via hyper-threading would translate to a configuration maximum of 4 VAX VCPUs, rather than 8 VAX VCPUs.

Furthermore, host operating system may be smart or not smart enough about allocating VCPU threads to distinct cores when free cores are available instead of lumping two threads to the same core, while another core may have no threads assigned to it. Ideally, host OS scheduler should be smart enough about spreading active threads among the cores, trying to avoid sharing the core by two threads, but whether specific operating system is performing it is undocumented. It appears Windows 7 does it and Linux does it too (HT-aware passive and active load balancing).

It can be reasonably expected that mainstream modern OSes will be smart about HT/SMT-aware scheduling. Nevertheless just in case, VAX MP includes code (currently defunct) to set affinity mask for VCPU threads to a mask that has only one logical processor per core, thus preventing VCPUs from being co-scheduled on the same core. This code can be enabled by console command

```
sim> DEPOSIT  VCPU_PER_CORE  1
```

This setting is accepted only if number of VCPUs does not exceed the number of cores.

This code had not been debugged and made functional yet. It has been observed that enabling it on Windows causes – for some mysterious as of yet reason – one of VCPU threads to be virtually stalled and executed at near-zero speed, despite the core and logical CPU assigned for it were not utilized. It had not been tried on Linux yet.

Essential OpenVMS bugs

The latest released version of OpenVMS for VAX was 7.3 (released in 2001) and is available on OpenVMS VAX Hobbyist CD. Unfortunately with VAX already well underway departing into sunset at that time, 7.3 was not tested as thoroughly as previous versions. There is a bug in OpenVMS 7.3 relevant for VAX MP.

This OpenVMS 7.3 bugs show up even on a regular, original uniprocessor version of VAX SIMH.

Setting MULTIPROCESSOR to 4 apparently does not cause multiprocessing system image to be properly loaded, as indicated by SHOW CPU. What's worse, it causes system crash on DISMOUNT of any disk volume (other than system volume), often wiping out the home block of that volume for a good measure.[122] The crush is caused by the system trying to release a spinlock that is not held. Conclusion: do not use MULTIPROCESSOR set to 4.

Setting MULTIPROCESSOR to 2 does cause multiprocessing system image to be properly loaded and system execute stably.

* * *

On a related note, when MULTIPROCESSOR is set to 2[123], DZDRIVER, the driver for terminal multiplexer, causes a crash when VMS is booted on the original SIMH 3.8.1, immediately on trying to log into the system via this multiplexer or performing any IO operation on it. This is not OpenVMS bug, this is SIMH 3.8.1 bug fixed in VAX MP and expected to be fixed in SIMH 3.8.2. The problem is specific to SIMH 3.8.1 and earlier versions and is related to SIMH incorrectly defining DZ as a device interrupting at QBUS request level BR5 translating to IPL 21. After redefining DZ to be a BR4 and IPL 20 level device (along with all other QBus non-custom devices that should be BR4 too), as VAX MP does, the problem goes away.

The crash is triggered by TTDRIVER trying to release a device spinlock that is not held, which in turn is caused by improper assignment of BR level for the device by SIMH 3.8.1.

If you expect to move system disk image between VAX MP and older versions of SIMH (pre-3.8.2), you need to be aware of the issue, since if you boot a system with MULTIPROCESSOR set to 2 on older versions of SIMH and try to use DZ, the system will crash.

If you need to move VAX MP system disk image back to SIMH 3.8.1 and boot it on SIMH 3.8.1, follow one of these two solutions:

- Either boot on SIMH 3.8.1 with SYSGEN/SYSBOOT parameter MULTIPROCESSOR set to 3 or 0. DZ driver works fine when MULTIPROCESSOR is set to 3 or 0.

- Or use MULTIPROCESSOR set to 2, but disable DZ device in your SIMH configuration file and use VH device instead:

  ```
  set dz disable
  set vh enable
  attach vh 10025  ; or whatever telnet port you'd want to use
  ```

  Luckily, the driver for VH multiplexer works just fine both in VAX MP and SIMH 3.8.1, as well as the driver for console and VMS TCPIP telnet connections. You can use any of these connections instead of DZ.

[122] SPLRELERR bugcheck, “Spinlock to be released is not owned”. The spinlock in question is FILSYS. This bugcheck is triggered inside F11BXQP, in routine DEL_EXTFCB, the last SYS_UNLOCK(FILSYS, 0) in this routine. It is curious that the bugcheck is triggered with the streamlined version of SMP code, but not with the full-checking version.

[123] The problem does not happen with MULTIPROCESSOR set to 4.

Lesser miscellaneous design notes and changes to SIMH codebase

This section outlines the changes to the original SIMH codebase done to implement VAX MP. It is written in assumption that the reader is familiar with original SIMH codebase. Therefore changes to existing structures are described in a somewhat greater detail, to alert the reader that familiar things had changed, than new additions that this document cannot afford to describe at the same level of detail.

The following most essential changes to SIMH code were required:

- Due to extensive changes to SIMH code, VAX MP is a separate code branch from regular SIMH. This code branch currently covers only VAX MP and no other simulated computers. Trying to keep VAX MP within the original SIMH code branch shared with other simulated computers would have potentially destabilized the implementation of other simulated computers, and regression testing for existing implementations is not a practical possibility. Code not compiled conditionally for VAX MP, such as code for FULL_VAX had been partially changed, but not fully. There was no attempt to keep changes to shared source files such as sim_*.* or pdp11_*.* files compatible with the use of these files by other simulators, including PDP10, PDP11, single-processor VAX and VAX780 simulators or any other simulators.

  VAX MP original codebase was derived from SIMH version 3.8-1, however most 3.8-2 changes were subsequently merged to it, based on 3.8-2 release candidate dated 2012/2/1 with some of subsequent most important changes up to late pre-3.9 release candidate as well.

  Some of 3.8-2 changes are handy (such as improved access to TAP/TUN network devices on Linux), but some are also important specifically for multiprocessing, in particular asynchronous disk IO and, to a smaller extent, network IO performed on a separate thread. SIMH device IO was implemented in 3.8.1 as synchronous file access and this distorts calibration of virtual processor speed which is calculated on the basis of the number of virtual processors cycles executed per interval of time obtained from the host system clock. If part of this interval was spent waiting for host system IO to complete, this can distort calculated "cycles per second" speed of the processor[124], which in turn will distort inter-processor synchronization in virtual SMP system.

  While VAX MP is designed to cope with such distortion through synchronization window mechanism, and some of the distortion sources will inevitably stay[125], it was deemed very desirable to reduce their influence by weeding out the most important of distortion sources, disk IO, which 3.8-2 provides for MSCP disks (RQ), and in addition also TMSCP tape (TQ) and Ethernet controller (XQ).

[124] Not to mention data displayed by system status/statistics utilities such as VMS MONITOR command.

[125] Such as page faults that will similarly delay the execution of the thread, or polling on network connections simulating multiplexer IO.

In addition, implementing disk and tape IO in synchronous fashion may cause virtual CPU to hold on to guest OS level synchronization objects such as spinlocks and kernel mutexes for the duration of underlying host operation, while the VCPU thread is blocked in a synchronous host system call, and cause other VCPUs to serialize and stall on the lock. Implementing IO in asynchronous fashion allows VCPU to release spinlocks as soon as possible, just like when the operating system executes on a real hardware computer.

Most of 3.8-2 changes had been merged into VAX MP codebase, with the following exceptions:

- New TODR management code changes (added in 3.8.2) in vax_stddev.c were not merged. Before merging this code it is necessary to analyze its implication and consistency with multiprocessor clock management.

- Changes to sim_timer.h and sim_timer.c had not been merged since VAX MP had already implemented equivalent functionality, but in VAX MP specific and extended way.

- Changes to breakpoint actions (in scp.cpp) were not merged since breakpoints are handled differently for VAX MP. VAX MP implements a redesigned version of 3.8.1 breakpoint package and merging 3.8.2 changes would have required code review and reconciliation of changes.

- Changes to provide console commands GOTO, RETURN and ON (in file scp.cpp) had not been merged.

- AIO changes had been merged in modified, VAX MP specific form.

- Name of the simulator VM was changed from VAX to VAX MP to identify new virtual machine and also to prevent VAX MP SIMH save-files from being loaded by VAX SIMH, with destructive results. While it could have been possible to implement reading VAX save-files by VAX MP, it was decided it's not worth the effort. Should one want to migrate an existing system from VAX to VAX MP, it should be shut down properly at OpenVMS level and existing disk images then attached to VAX MP.

  Save/Load functionality needs to be significantly revised for VAX MP, due to many changes in structures. This revision is not implemented in the original release of VAX MP, as low-priority task, and will be implemented in a subsequent release. In the original release of VAX/MP Save and Load commands are disabled and will issue error messages.

- The code base was changed from C to C++.

- Use of setjmp/longjmp was replaced by sim_try framework that can conditionally compile either to a chained setjmp/longjmp or native C++ try/catch/throw blocks, depending on whether symbol USE_C_TRY_CATCH is defined, and allows intermediate frame cleanup handlers and C++ object destructors be called during stack unwinding. This framework is used by VAX MP to implement:

    - interlocked instructions with locks orderly released in case any nested routine fails to access VAX memory, such as either because the target page is not currently mapped or protection disallows access to it;
    - convenient acquisition of resource locks, mostly for IO device resources, and also safe and automatic release of held locks (RAII handling of lock acquisitions);
    - orderly cleanup in case of runtime errors.

  For detailed description of how to use sim_try framework refer to the file `sim_try.h`.

  The “try” part of C++ try/throw/catch blocks is typically very inexpensive, however throw/catch on some system can be much slower than setjmp/longjmp. If it is, this will find reflection in slower handling of page faults and slower execution of VAX emulated instructions (such as CIS COBOL-oriented instructions or H-sized floating point arithmetics) that are not implemented by the MicroVAX 3900 processor in hardware and instead are emulated in software by trap handler.[126]

  Measured performance of simple *throw*/*catch* is about 150 times slower in Win32/MSDEV environment and 80 times slower on Linux/GCC than *longjmp*, and 5 times slower on OS X, however this is misleading, as actual performance during real execution depends on the content of the stack and simple stacks may be unrepresentative of more complex stacks. Also, while *throw* can be more expensive than *longjmp*, but *setjmp* can be much more expensive than *try* (since depending on the system setjmp can save signal mask and establish alternate stack); since most *try*/*catch* blocks invocations do not result in actual *throw*, the cost of unnecessary context saving in *setjmp* (executed every single time) can well outweigh the added cost of *throw* (executed only infrequently) compared to *longjmp*.

  It is therefore better to benchmark system with USE_C_TRY_CATCH defined and then undefined to determine which setting is better for a given host environment.

[126] Try/catch (sim_catch) is also executed if device interrupt or inter-processor interrupt is fielded during processing of MATCHC instruction *and* full VAX instruction set is enabled (conditional FULL_VAX is defined during compilation), which is not a default for MicroVAX 3900. All other code paths where exceptions are raised and sim_catch handling is performed are much less frequent and are unable to affect simulator performance in major ways. However one specific path where any slowdown due to the overhead of non-local jump slowdown can be observed directly is ROM console command SHOW QBUS that generates massive amounts of machine check exceptions while scanning QBus space, handled in simulator via non-local jumps.

OpenVMS boot sequence makes a good measurement target since it incurs many page faults. Our measurements indicate that native C++ exception handling is very marginally more performant than chained *setjmp*/*longjmp* both for Win32/MSDEV and Linux/GCC/GLIB and OS X host environments.

In addition to described above, the reason why this is so under Win32/MSDEV is obvious since in this environment *longjmp* is just a wrapper around either native C++ unwind mechanism or *RtlUnwind* mechanism, depending on the stack structure. On Linux *longjmp* does not perform C++ unwinding[127], so the reason for unstellar longjmp performance on Linux is not immediately apparent (other than the added cost of *setjmp*, of course), however measurements do show that even under Linix/GCC/GLIB as well as OS X *longjmp* does not outperform C++ *throw*/*catch*.

For this reason Linux, OS X and Windows versions of VAX MP utilize native C++ exception handling (via symbol USE_C_TRY_CATCH disabled).

- Multiprocessor / multithreading synchronization primitives were provided.
    - `smp_wmb()`, `smb_rmb()`, `smp_mb()` are write, read and full memory barriers. They are executed according to processor capabilities auto-detected at initialization. They also double as a compiler barrier.
    - `barrier()` is a compiler barrier.
    - Class `smp_lock` implements lock with spin-waiting. Thread will try to spin-wait for specified number of cycles on `smp_lock` object before hibernating on host OS-level synchronization object within it.

      `smp_lock` object has `lock()` and `unlock()` methods. In addition to locking or unlocking the lock object itself, both `lock()` and `unlock()` execute full memory barrier.[128] Lock object also implements internal performance counters.

      A normal way to create a critical section objects in most SIMH source files that do not include system-specific headers such as <windows.h> is by invoking a section creation function:

[127] Albeit it performs some pthreads related cleanup, but not in actual VAX MP code paths.

[128] This may be excessive for general locking purposes (see e.g. Hans-J. Boem, "Reordering Constraints for Pthread-Style Locks", http://www.hpl.hp.com/techreports/2005/HPL-2005-217R1.pdf and slides in http://www.hpl.hp.com/personal/Hans_Boehm/misc_slides/reordering.pdf), however current code relies on it in some places. Barrier on unlocking can be relaxed later, with adding "issue full barrier" option where it is really required.

```
smp_lock* lock = smp_lock::create(uint32 cycles = 0);
```

- o VAX MP also provides cross-platform classes `smp_semaphore`, `smp_barrier`, `smp_mutex`, `smp_condvar` and `smp_event`. Semaphore class is pollable, implements interface `smp_pollable_synch_object` and can be used in "wait for multiple objects"/select/poll constructs in conjunction with file objects such as console keyboard input.

  In most cases you'd want to use `smp_lock` over `smp_mutex` since the former is more powerful and efficient, however `smp_mutex` can be used in conjunction with `smp_condvar`.

- o Classes `smp_lock` and `smp_mutex` support the notion of critical locks (such as VM-critical locks) via method set_criticality. Locks marked as critical will cause acquiring thread to elevate its priority on acquisition and then drop it back on release of the lock.

- o Thread-local storage is created and managed with the following three wrapper functions:

```
t_bool smp_tls_alloc(smp_tls_key* key);
void tls_set_value(const smp_tls_key& key, void* value);
void* tls_get_value(const smp_tls_key& key);
```

- o VAX MP runtime defines data types for interlocked operations:

```
smp_interlocked_uint32
smp_interlocked_uint16
smp_interlocked_uint64
volatile t_byte
```

  These data types are declared *volatile* and have proper memory alignment attributes. Interlocked operations can only be invoked on entities of one of these types.

- o VAX MP runtime defines a number of primitives that operate on interlocked variables. Currently these primitives are implemented for x86/x64 instruction set, however implementation for LL/SC instruction set is straightforward.

  Functions

```
uint32 smp_interlocked_increment(smp_interlocked_uint32* p);
uint32 smp_interlocked_decrement(smp_interlocked_uint32* p);
```

  provide interlocked increment and decrement capability. On 64-bit host systems, interlocked increment and decrement for 64-bit operands are also provided:

```
uint32 smp_interlocked_increment(smp_interlocked_uint64* p);
uint32 smp_interlocked_decrement(smp_interlocked_uint64* p);
```

These functions return the value of their operand *after* the operation.

- A range of compare-and-swap (CAS) functions is provided for various argument types:

```
uint32 smp_interlocked_cas(smp_interlocked_uint32* p,
                           uint32 old_value, uint32 new_value);
uint16 smp_interlocked_cas(smp_interlocked_uint16* p,
                           uint16 old_value, uint16 new_value);
t_byte smp_interlocked_cas(volatile t_byte* p,
                           t_byte old_value, t_byte new_value);
uint64 smp_interlocked_cas(smp_interlocked_uint64* p,
                           uint64 old_value, uint64 new_value);
```

Variant for `smp_interlocked_uint64` is available on 64-bit systems only.

These functions return old value of the datum; if it was equal to expected `old_value`, they will also set the datum to `new_value`, otherwise leave it unchanged. Comparison and value exchange are performed in the interlocked fashion, as a single atomic operation.

A variant of these functions is also available that returns boolean value indicating if value swap had been performed successfully (TRUE) or if the original value of datum was not equal to `old_value` and the swap had not been performed (FALSE):

```
t_bool smp_interlocked_cas_done(smp_interlocked_uint32* p,
                           uint32 old_value, uint32 new_value);
```

and similar function variants for other operand types.

- Interlocked bit-test-and-set and bit-test-and-clear functions are provided:

```
t_bool smp_test_set_bit(smp_interlocked_uint32* p, uint32 bit);
t_bool smp_test_set_bit(smp_interlocked_uint16* p, uint32 bit);

t_bool smp_test_clear_bit(smp_interlocked_uint32* p, uint32 bit);
t_bool smp_test_clear_bit(smp_interlocked_uint16* p, uint32 bit);
```

These function set or clear the specified bit in the operand and return the original value of the bit: TRUE if the bit was originally set, FALSE otherwise.

- It is often needed to execute memory barrier along with interlocked operation. For example, the thread releasing a logical lock via low-level interlocked operation may want to execute write memory barrier before releasing a lock, so thread's changes to other memory locations become visible to other processors, while the thread acquiring the same logical lock may want to execute read memory barrier right after acquiring it, so memory changes executed by other processor(s) become visible in the cache on the local processor.

On some systems, such as x86 and x64, interlocked operations double as general memory barriers, and thus no separate additional memory barrier is needed, on other systems a separate barrier has to be executed.

To provide a portable way to cover the hardware differences, VAX MP defines and uses the following primitives:

```
void smp_pre_interlocked_wmb();
void smp_post_interlocked_rmb();

void smp_pre_interlocked_mb();
void smp_post_interlocked_mb();
```

On x86 and x64 these primitives are defined as empty macros, on other hardware targets they may expand into proper processor-specific primitive implementations.

- In certain cases when accessing a value of a variable shared among threads (and virtual processors) it is not necessary to know the *most* uptodate value of the variable. Any recent value of the variable would suffice, as long as it is possible to read any *valid* recent value and avoid reading the value with partial updates to it, i.e. with some bytes coming from one old value while other bytes from another old value. For example, if the value of the variable was 0x1111 and it is changed by another processor to 0x2222, we want to read either of these values, but not 0x1122. This can be achieved using *atomic* data types. VAX MP runtime defines atomic data types `atomic_int32` and `atomic_uint32`.

  Variables of this type can be updated and read atomically. No memory barrier and associated overhead is required as long as the code is content with using element value that may be not the most uptodate, but represent one of the previous recent values.

  To emphasize in the code that the value may be not most uptodate, the convention is to reference such variables in the code via “weak read” operation macro `weak_read`:

  ```
  atomic_int32 tmr_poll;
  . . . . .
  sim_activate(uptr, weak_read(tmr_poll) * clk_tps);
  ```

  `weak_read` expands to the value of its argument. The intention of the macro is to visually emphasize in the code that “weak read” operation is performed on the shared and concurrently updated variable, and that the value used may be not the most uptodate.

- If interlocked variable shares cache line with other data, access to interlocked variable by processors will interfere with access to other data by other processors and cause ping-pong cache coherency traffic on inter-processor bus, including repetitive expensive Request For Ownership messages. To avoid this traffic, it is better to allocate memory for most interlocked and some atomic variables in such a fashion that no data shares cache line with the variable, and each variable actively shared by processors has a whole cache line to itself and is located at the start of this line.[129]

  To address the issue, VAX MP provides container data types for interlocked and atomic variables. These container data types are declared to be aligned on cache line size and padded to the size of the cache line. Size of cache line is defined by `SMP_MAXCACHELINESIZE` and is processor-dependent. For x86 and x64, `SMP_MAXCACHELINESIZE` is defined to have value of 128, which is the maximum currently used by existing processors of these architectures.[130]

  These container data types have names

  ```
  smp_interlocked_uint32_var
  smp_interlocked_uint16_var
  smp_interlocked_uint64_var

  atomic_uint32_var
  atomic_int32_var
  ```

  Access to actual interlocked variable inside the container is provided via macros

  ```
  smp_var(cx)
  atomic_var(cx)
  ```

  where `cx` is the container varaible. Similarly, translation of pointers is performed by convenience macros

  ```
  smp_var_p(pcx)
  atomic_var_p(pcx)
  ```

[129] Besides hurting performance on all processors, on some processors that implement LL/SC interlocked semantics interference of separate variables within cache line may cause LL/SC instructions being aborted and having to retry them, possibly multiple times before they can succeed, making interlocked variable isolation even more critical.

[130] See “Intel 64 and IA-32 Architectures Software Developer’s Manual”, Volume 3 (3A & 3B, System Programming Guide), Table 11-1 “Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors”; also see section 8.10.6.7 “Place Locks and Semaphores in Aligned, 128-Byte Blocks of Memory”.

Initialization of the variable in container is done with the help or macros

```
smp_var_init(init_value)
atomic_var_init(init_value)
```

The following convenience macro perform interlocked operation on interlocked variables held inside container variables:

```
smp_interlocked_increment_var(p)
smp_interlocked_decrement_var(p)
smp_interlocked_cas_var(p, old_value, new_value)
smp_interlocked_cas_done_var(p, old_value, new_value)
```

Convenicnce macro `weak_read_var(m)` is a version of `weak_read` that accesses an atomic variable held inside the container type, for example:

```
atomic_int32_var tmr_poll = atomic_var_init(...);
. . . . .
sim_activate(uptr, weak_read_var(tmr_poll) * clk_tps);
```

- `sim_device` is changed from C structure to C++ class.
- `UNIT/sim_unit` is changed from C structure to C++ class.
- In original SIMH there was one CPU unit defined as UNIT `cpu_unit`.

  VAX MP introduces multiple CPU units, one per virtual processor. Primary processor unit is redefined as `CPU_UNIT cpu_unit_0`, other CPU units are dynamically created at run time.

  CPU descriptor type is changed from `struct UNIT` to C++ `class CPU_UNIT`, with the following structure:

```
class CPU_UNIT : public UNIT
{
public:
    uint8 cpu_id;
    . . . . .
    CPU_CONTEXT cpu_context;
};
```

  All per-CPU registers are hosted in CPU_CONTEXT. In original SIMH they were stored in global or static variables, VAX MP moves them to per-processor CPU_CONTEXT. This includes:

- o General-purpose processor registers, such as R0, R1, … AP, FP, USP, KSP etc.

- o Privileged processor registers, such as PCBB, SCBB, IPL, ASTLVL, SISR etc.

- o Other per-CPU registers.

There is one CPU_UNIT object per each virtual processor. Within the code current CPU object is always pointed by variable

```
CPU_UNIT* cpu_unit;
```

This variable is passed either as a parameter to a function or is a local variable. Most functions making use of CPU structure were modified to accept this variable as an argument and to pass it down as to other functions that need to use it. This is implemented with macro RUN_DECL for declaration of cpu_unit as an argument in function prototypes and declarations, and with macro RUN_PASS to pass it down. For example, routine op_extv that in original SIMH looked like this

```
int32 op_extv (int32 *opnd, int32 vfldrp1, int32 acc)
{
    . . . . .
    wd = Read (ba, L_LONG, RA);
    . . . . .
}
```

is modified for VAX MP as

```
int32 op_extv (RUN_DECL, int32 *opnd, int32 vfldrp1, int32 acc)
{
    . . . . .
    wd = Read (RUN_PASS, ba, L_LONG, RA);
    . . . . .
}
```

and similarly for many other routines. Whenever execution loop or other top-level routine selects a virtual processor to be executed on a current thread, it sets local variable cpu_unit and passes it down to invoked subroutines and recursively their children subroutines via RUN_DECL/RUN_PASS macros. In some cases it is impractical to use RUN_DECL/RUN_PASS mechanism as there is a number of infrequently used routines invoked deep down in the calling chain where higher-level routines do not make use of CPU_UNIT and do not pass it down. For those cases top-level routine also stores pointer to current CPU_UNIT in a thread-local structure via method run_scope_context::set_current. Any routine that does not receive cpu_unit via RUN_DECL can retrieve it from thread-local storage using macro RUN_SCOPE that defines cpu_unit as a local variable for this routine, for example

```
t_stat dbl_wr (int32 data, int32 addr, int32 access)
{
    cqipc_wr (addr, data, (access == WRITEB)? L_BYTE: L_WORD);
    return SCPE_OK;
}
```

is changed to

```
t_stat dbl_wr (int32 data, int32 addr, int32 access)
{
    RUN_SCOPE;
    cqipc_wr (RUN_PASS, addr, data, (access == WRITEB)? L_BYTE: L_WORD);
    return SCPE_OK;
}
```

Likewise,

```
t_stat tlb_ex (t_value *vptr, t_addr addr, UNIT *uptr, int32 sw)
{
    . . . . .
    *vptr = ((uint32) (tlbn? stlb[idx].pte :
                             ptlb[idx].pte));
    . . . . .
}
```

is changed to

```
t_stat tlb_ex (t_value *vptr, t_addr addr, UNIT *uptr, int32 sw)
{
    RUN_SCOPE;
    . . . . .
    *vptr = ((uint32) (tlbn? cpu_unit->cpu_context.stlb[idx].pte :
                             cpu_unit->cpu_context.ptlb[idx].pte));
    . . . . .
}
```

- The following elements had been moved to per-CPU context area:
    - `sim_interval`, reverse counter of executed cycles;
    - General-purpose processor registers (R0-R11), AP, FP, PC, PSL, stack pointers for all four modes and ISP, SCBB, PCBB, P0BR, P0LR, P1BR, P1LR, SBR, SLR, SISR, ASTLVL, mapen, pme;
    - backing storage for registers TRPIRQ, CRDERR, MEMERR, PCQ, PCQP;
    - `int_req` mask of requested interrupts – replaced by per-CPU InterruptRegister;
    - TLB register banks `stlb` and `ptlb`;
    - microcode registers `mchk_va`, `mchk_ref`, `d_p0br`, `d_p0lr`, `d_p1br`, `d_p1lr`, `d_sbr`, `d_slr`;
    - `in_ie` flag ("inside exception or interrupt");
    - instruction recovery queue `recq`, `recqptr`;
    - instruction prefetch buffer `ibufl`, `ibufh`, `ibcnt`, `ppl`.
    - fault parameters have been renamed from p1, p2 to `fault_p1` and `fault_p2` and moved to per-CPU context area, along with `fault_PC`;
    - last bad abort code `badabo`;
    - CQBIC registers SCR, DSER, MEAR, SEAR were moved to per-CPU context area;
      CQBIC MBR (QBus map base register) is left shared between the virtual processors, however MBR is made writable only by the primary/boot CPU (CPU0), attempt to write

MBR by a non-primary processor would cause the machine to stop to console with appropriate diagnostic message;

  - CADR and MSER registers, `ka_cacr` cache control register, cache data `cdg_dat`.
  - console terminal IO registers: `tti_csr`, `tto_csr`; references to `tto_unit.buf` have been changed to use `tto_buf` that is also made per-CPU;
  - clock device CSR: `clk_csr`;
  - clock calibration data `rtc_*`;
  - most, but not all SSC registers: `ssc_bto`, `ssc_otp`, `ssc_adsm`, `ssc_adsk` and all SSC timer registers `tmr_*`.

- Some of devices are global and there is only one instance of the device (and device unit set) per simulator shared by all virtual CPUs. Other devices are logically per-CPU and there is, effectively, a separate instance of the device and all its units per every virtual CPU in the system.

  To simplify the structure, DEVICE and UNIT descriptors are not replicated when multiple CPUs are created. These descriptors are considered metadata, rather than instance data. Actual device/unit state is kept, for system-wide global devices and units, in global or static variables; whereas for per-CPU devices the state is kept in per-CPU context area.

  Devices that are designated per-CPU have flag `DEV_PERCPU` set in their `DEVICE.flags` descriptor field.

  The following devices are considered to be per-CPU in VAX MP: CPU, TLB, SYSD, CLK, TTI, TTO.

  The following devices are considered to be system-wide: ROM, QBA, and all QBus devices: DZ, VH, CR, LPT, RL, RQx, RY, TS, TQ, XQx.

  NVR is declared system-wide. However its write routine will deny any CPU other than CPU0 write access to NVR memory. On a MicroVAX 3900, only boot ROM firmware and early stages of VMS SYSBOOT write to NVR.

  TTI and TTO are per-CPU. Any CPU can write to console, however only CPU0 is able to read from it, other CPUs will receive no data incoming (attempt by a non-primary processor to read RXDB will return no data and error bit set).[131]

  CSI and CSO are global but non-functional. Console storage is not available on a MicroVAX 3900. Attempt to connect it with SYSGEN CONNECT CONSOLE command does not create CSA device.

[131] OpenVMS fully assumes that in VAX SMP systems secondary processors do not have direct access to the console and must route all input/output to the console via “virtual console” by sending interprocessor requests to the primary processor that handles IO to the actual console registers on behalf of secondary processors.

"KA655 CPU System Maintenance" manual[132] places registers CSRS, CSRD, CSTS, CSTD in an undescribed category that follows category "accessible but not fully implemented, accesses yield unpredictable results". Just in case, routines that implement access to CSI and CSO registers had been proofed with synchronization locking.

CQBIC is split: QBus map base register (MBR) is shared between the processors, other registers are kept per-CPU. IPCR is not employed for interprocessor interrupt functionality by VAX MP.

CDG (cache data) is made per-CPU.

KA is split: CACR register is made per-CPU, but BDR is kept system-global.

CMCTL, CADR and MSER registers are made per-CPU.[133]

SYSD is declared per-CPU. SSC (System Support Chip) registers are split. `ssc_cnf` remains system-global, shared by all the CPUs, but the access to it is protected by a lock. `ssc_base` remains system-global too, access to it is protected by a lock; in addition writing to `ssc_base` from non-primary CPUs is disallowed and will cause the system to halt with diagnostic message. `ssc_bto`, `ssc_otp`, `ssc_adsm`, `ssc_adsk` were moved to per-CPU area.

SYSD/SSC timer-related registers `tmr_*` have been moved to per-CPU area too. However SSC timer is only used by the firmware, OpenVMS SYSBOOT and SYSINIT and terminal phase of VMS shutdown, always in the context of the primary processor. SSC is not accessed by firmware of VMS at operating system run time when multiprocessing is active. Therefore, while SYSD/SSC clocks could operate in each processor (assuming operating system perceives them per-CPU and not system-wide), VAX MP currently restricts access to SYSD/SSC clocks to the primary processor only. Attempt to access clock registers by a non-primary processor will trigger stop-to-console condition with diagnostic message.

TODR is made per-CPU, but is not actually maintained on secondary processors. Operating system is expected to access TODR for read or write only on the primary processor for any significant functions. OpenVMS does not use TODR on the secondary processors for time-keeping and does not access TODR on the secondary processors for these purposes; instead when secondary processor need to read or write TODR, it makes interprocessor call to the

[132] EK-306AA-MG-001, DEC, 1989, pages B6-B7, available at http://bitsavers.org/pdf/dec/vax/655/EK-306A-MG-001_655Mnt_Mar89.pdf and also in *aux_docs* subfolder of the project.

[133] Once a second OpenVMS executes processor-specific routine ECC$REENABLE provided by SYSLOA image. This routine is invoked on the primary processor only. It reenables CRD interrupt via CMCTL17 and sets up for managing CRD errors. It's SYSLOA650 version is not SMP-aware and acts as if CPU0 was the only CPU in the system. There is no need to SMP-ize it however and make guest OS aware that "virtual cache is per-CPU" since VAX memory cache parity errors never happen on VAX MP and it does not generate CRD interrupts.

primary processor that performs requested operation on its TODR.[134] If secondary processor tries to access TODR on VAX MP, value is read from the primary's copy of TODR. Writes to TODR on secondary processors are ignored.

Registers CONPC and CONPSL are meant for firmware use only that executes on the primary CPU. An attempt to access them by a non-primary CPU will trigger a stop-to-console condition with diagnostic message.

Logic throughout the code had been modified to account for split into per-CPU and system-wide devices. One particularly affected area is device reset logics. The convention decided is that `DEVICE.reset()` resets the *instance* of per-CPU device associated with the current processor, as defined by `run_scope_context`. Hence, a code that needs to reset the device on all processors has to loop through the processors and invoke `DEVICE.reset` within the context of every processor.

- New console command "CPU" has been added:

  CPU MULTI{PROCESSOR} <n> will create virtual processors in SIMH.

  CPU ID <n> will set default context for execution of other console commands. Thus, for example

  ```
  CPU ID 2
  EXAMINE STATE
  DEPOSIT PSL …
  STEP
  ```

  will display the state of CPU2, change its PSL and execute single instruction on CPU 2 without resuming other processors. Explicit CPU unit specification in a command, such as for example "EXAMINE CPU3 STATE" overrides default context.

  Reference to device "CPU" without specifying explicitly its unit number means a reference to currently selected CPU, for example

  ```
  CPU ID 2
  EXAMINE CPU STATE
  ```

  means a reference to the state of CPU2, not CPU0.

  Console command CPU ID displays currently selected default processor for other console commands context.

[134] See "VAX/VMS Internals and Data Structures", version 5.2, p. 1027. Also OpenVMS VAX (VAX/VMS) source listings for the use of routines EXE$READ_LOCAL_TODR, EXE$READP_LOCAL_TODR, EXE$WRITEP_LOCAL_TODR.

Command CPU INFO will display VCPUs state information of use for a developer, including summary of current VCPU modes, IPL levels, pending interrupts and synchronization window state.

Code for other console commands had been instrumented to honor the setting of current CPU ID as a context for these other console commands. The setting is kept in global variable `CPU_UNIT *sim_dflt_cpu` and also inside `run_scope_context* sim_dflt_rscx` for use by `RUN_SCOPE` macros down the call chain.

- It was deemed desirable to expose virtual CPUs units as regular SIMH devices so available SIMH console commands can be used with them, just like with any other device, for example

```
EXAMINE CPU2 STATE
```

and so on. This means that CPU unit descriptors should be pointed by `cpu_dev.units`. Unfortunately this field is declared in original SIMH as pointing to an array of UNIT's and since CPU_UNIT is larger than UNIT, it cannot be accommodated by the existing structure.

One possible solution could have been to describe CPU by the existing UNIT structure rather than extended CPU_UNIT and to keep per-CPU context in a separate area, outside of UNIT and pointed from it. This however would have led to very frequent dereferences via this pointer, since all registers, TLB and many other per-CPU elements are kept in per-CPU context and every reference to them would have incurred an extra dereference via the pointer, affecting performance. Alternatively, it could have been possible to pass the pointer of per-CPU extension area as an extra argument included in `RUN_DECL/RUN_PASS` macros, but besides extra instructions to push the argument for about each and every call this would have intensified competition for host system registers, which is especially important for register-starved architectures like x86. Finally, it could have been possible to pad UNIT to the maximum size of any subclass, but besides been an ugly hack, the size of the padding would have to be large: in per-CPU context area, the size of two TLB banks alone is 64K, CDG adds another 64K, with the total coming to over 130K; it is obviously undesirable to bloat every UNIT in the system to such a size without any real need for it.

Therefore, it was decided to place per-CPU area extension into CPU_UNIT itself and change declaration of `sim_device.units` and the code that references it.

`DEVICE.units` has been changed from "a pointer to an array of UNIT descriptors" to "an array of pointers to UNIT descriptors". It has been changed from `UNIT*` to `UNIT**`. This necessitated the following changes in the code:

  - UDATA macro had been changed from static initializer list to new `UNIT(…)`.

- `xx_unit` entries for multi-unit devices had been changed from
  ```
  UNIT xx_unit[] = { UDATA(…), … UDATA(…) };
  ```
  to
  ```
  UNIT* xx_unit[] = { UDATA(…), … UDATA(…) };
  ```

- Multi-unit devices (in the sense of their code structure only) are rl, rq, ry, tq, vh, tlb, sysd and xq. Single-unit devices are cr, dz, lpt, ts, qba, rom, nvr, csi, cso, tti, tto, clk, sim_step_unit, sim_throt_unit.

- Initializers for single-unit devices had been changed to use, instead of `UDATA`, other macros: `UDATA_SINGLE` or `UDATA_SINGLE_WAIT`. They also use macro `UNIT_TABLE_SINGLE` that generates unit table with a single entry pointing to the device. This table if referenced from corresponding device structure, with old references to `&xx_unit` replaced by `xx_unit_table`.

- Code that assumed `units` array structure for indexed access to units list had been altered.

- References in the code were replaced according to the following patterns, wherever appropriate:

old	*new*
`dptr->units + i`	`dptr->units[i]`
`dptr->units`	`dptr->units[0]`
`&xx_unit[i]`	`xx_unit[i]`
`xx_unit[i].field`	`xx_unit[i]->field`
`uptr – dptr->units`	`sim_unit_index(uptr, dptr)`

- Per-unit register descriptor macro URDATA had been renamed to URDATA_GBL and altered to handle the changes, along with register accessor method `sim_reg::getloc` and couple of references in the code to registers of the type `REG_UNIT`.

- Structure `sim_reg` is changed to C++ class of the same name with additional field `loctype` describing the location of the register's backing store. It can be `REG_LOCTYPE_GBL` for registers shared by all CPU units and still residing in global variables, or `REG_LOCTYPE_CPU` for registers moved to `CPU_UNIT.cpu_context`, in which case field `loc` stores byte offset of the register backing store inside `CPU_UNIT` structure. Method `sim_reg::getloc(RUN_DECL)` returns the address of the register regardless of its location type. All code making use of `sim_reg.loc` had been changed to use `sim_reg::getloc` instead.

Per-unit registers described by UDATA_GBL have loctype set to `REG_LOCTYPE_GBL_UNIT` and are accessed with accessor method `sim_reg::getloc_unit_idx(RUN_DECL, uint32 unit_index)` whenever indexing is required.

- Macros that define backing store location of virtual registers (HRDATA, DRDATA, ORDATA, FLDATA, GRDATA, BRDATA) had been split into two versions: xxDATA_GBL defines location for registers shared by all CPU units, xxDATA_CPU defines location of per-CPU registers private to each processor.

  For BRDATA registers, BRDATA_CPU used together with BRDATA_CPU_QPTR additionally define a per-cpu location for `qptr` field. Methods `sim_reg::getqptr` and `sim_reg::setqptr` had been provided to access `qptr` regardless of the kind of baking store in use.

- Virtual device interrupt raising and dispatching mechanics has been revised to support multiprocessor environment.

  Some IO devices in the initial release of VAX MP normally do not have threads of their own and their implementation code executes in the thread context of one virtual processor or another, i.e. the context of the worker thread for some virtual processor. Other devices do have worker threads associated with them and can send interrupts from these threads (although in most cases they signal the completion of IO operation to the part of SIMH handler executing in the context of primary VCPU and performs post-processing there, such as queuing device event entries, before sending an interrupt to VAX level).

  Accordingly there are several scenarios for raising interrupts in VAX MP:

  - When a per-CPU device raises an interrupt, this happens in the context of the thread executing the device's owning processor. In this case processor's thread sends the interrupt to the current processor.

  - Interrupts from system-wide (not per-CPU) devices are handled, by design, by the primary CPU (CPU0).[135] When a system-wide device raises an interrupt, the device can be executing in the context of any virtual processor in the system (unless guest OS affinity mask "pins" device to a particular virtual processor or processors), leading to this processor's thread sending an interrupt to the CPU0. Thus, when system-wide device raises an interrupt, it can be sent from the execution context of any processor to CPU0.

[135] See discussion in section "Dispatching device interrupts" above.

- Any virtual processor in the system can send inter-processor interrupt (IPI) to any other virtual processor in the system, including self.[136]

If dedicated clock strobe thread is employed, it adds additional scenario:

- Clock strobe thread sends SYNCLK interrupt to all running VCPUs. This interrupts is processed inside VAX MP runtime code and is not directly visible on VAX level. SYNCLK is handled as non-maskable interrupt. When SYNCLK is received, processor converts it to CLK interrupt (in case CLK interrupts are enabled in system clock CSR), schedules for immediate execution all pending clock queue events that were marked co-scheduled with clock, and performs various housekeeping tasks like clock recalibration.

Some devices execute a part of their processing on a separate dedicated thread, in a virtual IO processor. Accordingly, this adds another possible scenario:

- IO processor (IOP) thread sends an interrupt to the primary CPU.

Device interrupts can by their nature be either synchronous or asynchronous with respect to VAX guest instruction stream. Few interrupts (such as machine check or software interrupts via SIRR register) must be raised immediately, without any delay, on executing an instruction, before next instruction is executed, and thus are timed precisely to the instruction stream, effectively synchronous to it. System logics and guest operating system code expects them to be synchronous.

Most interrupts however require some prior processing time within the device and are not logically synchronous to the instruction stream, they are asynchronous and the system code fully expects them to happen with some indeterminate delay with respect to instructions that logically caused them, perhaps a small delay, but not within the same instruction cycle.

Accordingly listed scenarios break down into four cases:

- any VCPU can send asynchronous IPI to any other[137]
- any VCPU can send synchronous IPI or other synchronous interrupt to itself
- any VCPU can send per-CPU device interrupt to self
- any VCPU can send system-wide device interrupt to self

[136] VAX MP employs also a special variant of interprocessor interrupt called IPIRMB. IPIRMB is sent to target processor or processors to make them execute memory read barrier within their contexts. This interrupt is not propagated to VAX level and is not directly visible at VAX level. It is handled within VAX MP runtime itself. Currently this interrupt is used when a VCPU detects a write to SCB, it then performs WMB and sends IPIRMB to all other VCPUs in the system. IPIRMB is handled as non-maskable interrupt.

[137] Inter-processor interrupts are considered asynchronous because instruction streams on different CPUs are different (if sometimes tightly coupled) logical "timelines" and therefore, by definition, are asynchronous to each other.

For devices that have threads of their own, such as for handling asynchronous disk IO or arrival of network packets, the fundamental picture still remains: there are synchronous interrupts that must be raised in the context of execution of the current processor, and interrupts that are raised in external context and are delivered to this processor asynchronously. Synchronous interrupts under present VAX MP design will always be generated by the same thread that is executing instruction stream of the target processor, whereas asynchronous interrupts must be delivered from the context of the thread that can be either the same or different from the thread executing instruction stream of that processor.

To handle these scenarios, pending interrupt mask `int_req` was replaced by per-CPU object `cpu_intreg` of type `InterruptRegister` located in `CPU_UNIT` structure.

```
class CPU_UNIT : public UNIT
{
    . . . . .
    InterruptRegister cpu_intreg;
    . . . . .
}
```

Internally `cpu_intreg` contains interrupt mask very similar to original `int_req`, only replicated on per-CPU basis, and modified via interlocked instructions, plus a “changed” flag for the mask also raised and cleared via interlocked instructions. InterruptRegister provides methods to raise and clear interrupts. Macros `SET_INT` and `CLR_INT` had been changed to call functions `interrupt_set_int` and `interrupt_clear_int`. These functions examine the type of the device indicated by their argument. If a per-CPU device tries to raise or clear an interrupt, interrupts are directed to `cpu_intreg` of local processor, otherwise they are directed to `cpu_intreg` of CPU0. When an interrupt is raised, interrupt mask inside the target CPU’s InterruptRegister is altered using host interlocked instructions, and in addition “changed” flag in the InterruptRegister is also raised using host interlocked instructions. If a target processor is different from the current processor, write memory barrier is executed[138] and `wakeup_cpu` is called for the target virtual processor, making it runnable and scheduled for execution, in case processor was in an idle sleep state.

On the receiving side, instruction loop examines “changed” flag before execution of every instruction. This variable is examined here in a “lazy” way, without using interlocked read or execution of a read memory barrier prior to reading it, i.e. it is read with “weak” read operation, relying only on eventual update propagation through host cache coherency protocol. Thus examining this flag is cheap, but it may take some time for cache coherency protocol to propagate the raising of “changed” flag to local processor in case the flag was raised by a thread executing on another processor or inside non-VCPU thread (such as IOP thread or clock strobe thread). Given typical host inter-CPU cache coherence latencies and the number of host cycles required to implement typical VAX instruction, this delay can conceivably be as long as several

[138] WMB (if required) is executed *before* interlocked operations described above.

VAX instruction cycles, although usually will complete within just one or at most two VAX cycles. This is perfectly acceptable for asynchronous interrupts. For synchronous interrupts, the "changed" flag will be raised in the context of the current processor and since local cache is self-coherent, the update to the flag will be visible immediately.

Once instruction loop of the target processor notices that "changed" flag of its InterruptRegister had been raised, the loop invokes `SET_IRQL` macro that will perform full read on InterruptRegister, involving necessary memory synchronization and access to the pending interrupt mask using proper host interlocked instructions and memory barriers. In the process of mask scan, "changed" flag will be cleared (until another interrupt arrives) and `trpirq` of current processor properly updated and IRQL field within it set accordingly. After noticing that IRQL part of `trpirq` contains pending interrupt, instruction loop will handle this interrupt. SIMH functions `eval_int` and `get_vector` had been modified accordingly to access processor's InterruptRegister (instead of `int_req` in the original SIMH version) using proper inter-processor / inter-thread synchronization.

Invocation of `SET_IRQL` in `ReadIO` and `WriteIO` functions that access QBus space and internal processor registers located in IO space had been limited in the SMP version to a subset of cases it served previously. Calling `SET_IRQL` at the end of `ReadIO` and `WriteIO` was effective for device interrupt detection in the uniprocessor version of SIMH, that had only one instance of pending interrupt mask and updates to it were always being executed on the current (single) thread and completed by the time `ReadIO` or `WriteIO` would exit, but none of these conditions hold anymore for the multiprocessor version: in the SMP version, devices may send interrupts to other CPUs (usually CPU0) rather than the current CPU. Therefore it does not make sense to re-evaluate pending interrupt state mask of the current processor by invoking `SET_IRQL` at the end of `ReadIO` or `WriteIO`, since the device-generated interrupt may have been directed to the processor other than the current one, and hence `SET_IRQL` executed at the current processor just won't catch this interrupt, it should be noted and evaluated by another processor, not the current processor. Checking for raised device interrupts at exit from `ReadIO` and `WriteIO` had therefore been replaced by polling instead for CPU's InterruptRegister "changed" flag in the instruction loop. `SET_IRQL` is still invoked at exit from `ReadIO` and `WriteIO`, but only when per-CPU *synchronous* condition is detected, currently limited to `mem_err`, i.e. attempted access to non-existing location in the IO space.

The whole interrupt signaling and dispatching scheme is implemented using non-blocking logics.

Interrupt signaling/dispatching code also takes care of properly boosting target VCPU's thread priority if an interrupt is posted at request level that is at or above `sys_critical_section_ipl` and `sys_critical_section_ipl` is enabled. When an interrupt is sent to a VCPU:

- if the sender is the same as the target, and `sys_critical_section_ipl` is enabled, and request level is at or above `sys_critical_section_ipl`, sender/target will raise its own thread priority to CRITICAL_OS or CRITICAL_OS_HI unless it is already above this level;

- if the sender is different from the target, and listed `sys_critical_section_ipl` related conditions are met, sender will raise target's priority to CRITICAL_VM and signal the target to re-evaluate its thread priority ASAP.

  Priority elevation on posting the interrupt safeguards VCPU with pending interrupt from being preempted by system load of lower or equal priority from timely dispatching this interrupts and timely entering CRITICAL_OS or CRITICAL_OS_HI level implied by the interrupt. If target thread priority were not boosted on interrupt posting (and was boosted on interrupt dispatch alone), it would have been possible, for example, for a VCPU with pending interprocessor interrupt (IPI) to retain for a long time low pre-interrupt thread priority (corresponding to user-mode or low-IPL execution) and stay for a long time preempted by competing threads before this target VCPU could even dispatch the interrupt and reevaluate its thread priority on dispatching.

On the receiving side, there are three ways to read interrupt state of the local VCPU:

- Using "weak" read, as described above. "Weak" reading is cheap, but provides a delayed[139] view of the state of the pending interrupts set and the "changed" flag for the mask for the interrupts raised by sources external to this VCPU. This is the type of access executed on every VAX instruction cycle.

- Using interlocked instructions, such as CAS on the "changed" flag. This access is likely to be cheap in terms of low or no interconnect traffic, since *unless* an external interrupt had been actually posted to the local VCPU, the cache lines for the interrupt mask and "changed" data are likely to be already owned by the local PCPU's cache[140] and thus interlocked instruction causing no interconnect traffic; whereas if the interrupt *had* been posted, additional interconnect traffic necessary to process it is justified. However interlocked instruction is likely to cause slow-down on local processor, due to the fence in processing of store buffers and invalidation queue. Therefore the use of interlocked instruction to check VCPU interrupt state is avoided in the code that routinely executes on every VAX cycle, and "weak" read is used there instead; but interlocked instruction can be used at essential VCPU state change junctions (such as REI instruction processing etc.), when there is a reason to check for the most current interrupt state before

[139] Delayed – with respect to async interrupt sources external to this VCPU. Interrupts raised within the context of the local VCPU are visible for VCPU's thread immediately even with weak read operations, due to cache self-coherence. Delays due to cache coherency protocol processing can be as long as few simulated VAX instruction cycles, but in most cases are likely to be under one VAX instruction cycle.

[140] I.e. being in Modified or Exclusive states of MESI cache coherence protocol on the local processor.

adjusting VCPU thread priority and performing similar functions.

  - Current view of pending interrupt mask state can be also accessed with read memory barrier, because when an external interrupt is posted, the sender performs write memory barrier after raising interrupt bits and before raising “changed” flag. This type of access does not cause interconnect traffic[141], but it may cause local PCPU slow-down until its cache invalidation queues are processed. Therefore this type of access is avoided for routine checks executed on every VAX instruction cycle, but can be employed at essential VCPU state change junctions, and on x86/x64 systems is likely to be cheaper than access via interlocked instructions, since unlike the latter RMB performs fencing only on invalidation queue, but not on the store buffers. VAX MP employs this type of access when it re-evaluates current VCPU thread priority depending on whether various kinds of interrupts are pending.[142]

- Switching from `int_req` to per-processor `InterruptRegister` structure necessitated the change in how SIMH “INT” registers of various devices are accessed by console commands. These registers are no longer located at fixed static addresses or indeed any location visible outside of `InterruptRegister` structure, so references of the kind `FLDATA(INT, ...)` had to be converted to new kind of register descriptors named `IRDATA_DEV(...)`. Likewise, references to QBus pending interrupt mask for each IPL level in the form `HRDATA(IPLxx, ...)` had to be converted to new kind of references named `IRDATA_LVL(...)`. New kinds of references provide access to register via getter/setter routines specified in the descriptor, rather than through a pointer to static memory location. Under the hood, new type of registers access is designated by code `REG_LOCTYPE_DYN` in `sim_reg.locinfo.loctype`. SIMH routines `get_rval` and `put_rval` had been modified to recognize this new kind of register mappings and invoke accessors specified in the register descriptor instead of trying to access the register via some kind of direct memory pointer, as for other kinds of mappings.

  Register list of CPU devices had been extended with registers IPL14 - IPL17 too, so console commands like

  ```
  EXAMINE CPU2 STATE
  ```

  or

  ```
  EXAMINE CPU2 IPL17
  ```

  or

  ```
  CPU ID 2
  EXAMINE CPU STATE
  ```

[141] Assuming snoopy bus or similar interconnect, which is the case for host systems targeted by VAX MP.

[142] For example in routine `cpu_reevaluate_thread_priority`.

will display IPL*xx* registers for CPU2 or other specified CPU.

Console commands that display QBA registers IPL14 – IPL17 take their values from InterruptRegister of the currently selected CPU.

- To help user easily distinguish between per-CPU registers and system-wide registers, console commands such as EXAMINE had been modified to display a CPU ID tag when displaying a register value, for example:

  ```
  sim-cpu0> CPU ID 2
  sim-cpu2> EXAMINE QBA STATE
  [cpu2]  SCR:    8000
  [cpu2]  DSER:   00
  [cpu2]  MEAR:   0000
  [cpu2]  SEAR:   00000
          MBR:    00000000
  [cpu2]  IPC:    0000
  [cpu2]  IPL17:  00000000
  [cpu2]  IPL16:  00000000
  [cpu2]  IPL15:  00000000
  [cpu2]  IPL14:  00000000
  ```

  In this example, MBR register is system-wide, while other displayed registers are per-CPU and displayed tag indicates CPU ID their values are displayed for.

- VAX interlocked instructions (ADAWI, BBSSI, BBCCI, INSQHI, INSQTI, REMQHI, REMQTI) have two implementation modes: *portable* and *native*. In *portable* mode they are implemented by acquiring a global lock within SIMH (logically representing VAX shared memory interlock) and executing appropriate memory barrier before and after.

  For efficiency reasons, the lock is implemented as a spinlock, converting after short spin-wait to a host OS-level critical section (composite object is implemented as class `smp_lock`). Amount of code executed inside lock-holding critical section is very small, just few references to memory location and memory barriers. Therefore if the thread trying to acquire shared memory lock has to spin-wait more than short amount of time, it means that lock-owning thread is suspended out of execution and it is not worth continue spin-waiting for the spinlock but instead the contending thread should convert to blocking wait on OS-level critical section.

  To reduce contention during the execution of VAX interlocked instructions, VAX MP uses an array of spinlocks, rather than a single spinlock. An entry in the array to lock is selected by a hash function on the interlocked operand address rounded down to the longword, i.e. two lowest bits stripped.

  When BBSSI instruction handler finds that the bit was already set, it executes host processor's PAUSE command. This is based on assumption that BBSSI is likely to be a part of spinlock or other busy-wait loop, and to relieve host system by preventing generation of multiple (and useless) simultaneous out-of-order read requests.[143] PAUSE inside busy-wait loop may also be beneficial when spin-waiting on a Hyper-Threaded processor by releasing shared resources to a thread doing useful work, perhaps exactly the lock holder or the thread performing other activity that the current thread is spin-waiting to complete.

Native mode relies on host interlocked instructions to simulate VAX interlocked instruction set or a part of it in more direct way, reducing the incidence of preemptable locking.

- Access to QBus devices and other IO devices requires proper synchronization between the processors. In a real hardware system, this synchronization is achieved at three levels:

    - Processors synchronize between themselves with the use of spinlocks or other primitives.
    - IO bus allows only one access at a time, per bus cycle.
    - Firmware or microcode inside an IO device that sits on the other side of IO register has its own execution engine, its own event dispatch loop and its own internal synchronization mechanisms.

  Even if we assumed we could rely on robustness of guest OS synchronization (mechanism №1), which we do not necessarily want to (do we want to “damage” virtual hardware just because there may sometimes be a bug in the operating system?), mechanism №3 is still missing in the original uniprocessor SIMH and is not always covered by №1. For example, in SIMH several device controllers may share common data, and hence access to it needs to be synchronized. Operating system won’t know these controllers actually represent a single device, it will consider them as separate independent devices, lockable independently. Another example: SIMH device interrupt-acknowledgement routine may be invoked on CPU0 just as CPU2 is initiating another IO operation on the device, and as these routines will access the same data, such as device’s “interrupt pending” state, synchronization is needed. Async-proofing of the devices needs to be implemented for each device, to ensure integrity of device data in case of concurrent access to the device.

  The only case when a device does not need async-proofing is per-CPU devices, since they are always accessed in the context of their owning virtual processor[144] and there is no concurrent access to them.

  However for all system-wide devices, shared by the CPUs and accessible concurrently, their SIMH implementation code needs to be reviewed and appropriate locking performed on the

[143] See Intel application note AP-949 "Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor", P/N 248674-002 (http://software.intel.com/file/25602).

[144] Including their timer routines (*xx_svc*) that are also executed by the thread of the owning processor and within the context of the owning processor.

access to device state. A straightforward approach to this is to look at device interface routines exported via DIB, DEVICE and UNIT structures (*xx_rd*, *xx_wr*, *xx_svc*, *xx_ack*, *xx_reset*, *xx_boot*), as these are the only routines the device is accessed through by the outside world, and implement appropriate locking inside these routines.

Granularity of locking is left up to particular device handler.

SMP version of SIMH introduces helper macros to perform device state locking: AUTO_INIT_DEVLOCK, AUTO_LOCK and AUTO_LOCK_NM.

In the present version, device handlers had been modified to perform locking as follows:

> CR, DZ, LP, RL, RY, TS, TQ are single-controller devices. They have single per-controller lock that locks data controller-wide.
>
> There are two per-controller locks in XQ, one for XQA controller, another for XQB. The code inside `sim_ether.cpp` invoked by XQ devices is based on PCAP that is thread-proof (in WinPcap, since version 3.0), except just one routine. Some minor required thread-proofing had been added to `sim_ether.cpp`.
>
> VH exposes multiple controllers, but underneath these controllers are just facets for single TMXR structure, therefore VH currently uses single lock for all the VH controllers.
>
> RQ uses per-controller locks, separate for RQ(A)/RQB/RQC/RQD controllers, but they share master interrupt state, so the code for interrupt handling had to be additionally modified to be thread-safe.

- For per-CPU devices, all their interface routines (*xx_rd*, *xx_wr*, *xx_svc*, *xx_ack*, *xx_reset*, *xx_boot*) are always invoked and always execute in the context of the owning processor and thus do not require locking to mutually synchronize against their invocations.

  For system-wide devices, most of these routines (in particular *xx_rd* , *xx_wr* , *xx_svc* and *xx_reset)* can execute in the context of any virtual processor in the system and be invoked simultaneously. To avoid race condition between the invocations of these routines, proper synchronization logic must be followed. Although the exact implementation of locking is up to particular device handler, but as a rule of thumb device handler must ensure proper locking of its device data, unit data and its own global data when being called through any of interface routines. For most devices, locking will be per-controller and utilize helper macros AUTO_INIT_DEVLOCK, AUTO_LOCK and AUTO_LOCK_NM.

  Handler must always designate a lock that is responsible for synchronizing access to particular UNIT, hereafter called *unit lock*. In most cases this will actually be per-controller lock (with all units in the controller sharing the same lock), but it can also be module-wide lock or per-unit lock. In any event, there is 1:1 or 1:N association between UNIT and corresponding lock. This lock must be held when performing operations on unit's data, to ensure exclusion while accessing it. Pointer to unit lock is stored in field UNIT.lock for use by SIMH runtime parts other than device handler itself.

In particular, unit lock **must** be held when invoking routines `sim_activate`, `sim_activate_abs`, `sim_activate_cosched`, `sim_cancel` and `sim_is_active` for system-wide (non per-CPU) device units. [145]

Device handler is responsible for binding unit lock to UNIT structure by storing the pointer to the lock in UNIT.lock, normally within *xx_reset.* If all units of the device share the same lock, it can be bound to all of the device's units with helper routine `sim_bind_devunits_lock`, for example:

```
    static t_stat vh_reset(DEVICE  *dptr)
    {
        AUTO_LOCK(vh_lock);
        sim_bind_devunits_lock(&vh_dev, vh_lock);
```

`sim_bind_devunits_lock` binds specified lock as unit lock for all units of the device.

When *xx_svc* is invoked for a unit of system-wide device (rather than per-CPU device), it must acquire unit lock and then invoke the following macro: RUN_SVC_CHECK_CANCELLED().

Thus, typical *xx_svc* entry point for a system-wide device will look like:

```
        t_stat xq_svc(RUN_SVC_DECL, UNIT* uptr)
        {
          CTLR* xq = xq_unit2ctlr(uptr);

          AUTO_LOCK_NM(xq_autolock, *xq->xq_lock);
          RUN_SVC_CHECK_CANCELLED();

          . . . . . .

          return SCPE_OK;
        }
```

xx_svc will be entered with no unit lock held, will acquire unit lock and invoke RUN_SVC_CHECK_CANCELLED. The latter macro will check if invocation is still valid or event queue entry had been invalidated since being created (see more on event queues in VAX MP below). Holding unit lock is required for this check, so RUN_SVC_CHECK_CANCELLED must be preceded by acquisition of the unit lock.

[145] These routines can also be invoked by console thread, in this latter case holding the lock is not required since multiprocessing is suspended while SIMH console is active.

If the queue entry had been invalidated, RUN_SVC_CHECK_CANCELLED will exit from *xx_svc*, which will cause unit lock to be released and the whole call and associated event queue dismissed.

If event queue entry is still valid and current, *xx_svc* will proceed with processing the event while holding unit lock.

Invocation of RUN_SVC_CHECK_CANCELLED is required only for system-wide (non-per-CPU) devices. Per-CPU devices do not require checking with RUN_SVC_CHECK_CANCELLED, nevertheless the macro can also be safely invoked from per-CPU device's *xx_svc* routine too, in the latter case it will be a no-op.

- As discussed in the section about VAX memory consistency model, VAX MP simulator does *not* impose automatic write memory barrier (WMB) on every access to IO space, leaving it instead up to individual device handlers to decide whether to execute memory barrier and when to execute it.

  In general, device handlers should be aware about the need to properly execute memory barriers in the routines they export to SIMH via device description tables (*xx_rd*, *xx_wr*, *xx_svc*, *xx_ack*, *xx_reset*, *xx_boot*).

  For example, if a device CSR is written by CPU2 and device *xx_wr* handler queues internal data request for writing data in the supplied buffer that will eventually be processed by *xx_svc* invoked at CPU0, write memory barrier would be required inside *xx_wr* and read memory barrier inside *xx_svc* (so that *xx_svc* code executed on CPU0 will see valid data values in the buffer written by CPU2 before calling *xx_wr*), and also obviously a proper locking on *xx* internal data structures, such as internal *xx* request queue.

  It is responsibility of the device handler code to properly execute memory barriers.

  This task is much simplified by the fact that provided locking primitives automatically execute memory barriers, so wherever internal data structure locking is present, proper memory barriers are automatically present as well.

  All VAX MP QBus device handlers (DZ, VH, CR, LPT, RL, RQ, RY, TS, TQ, XQ) do not require any special code changes for memory barriers beyond data structures locking described in the previous sub-section.

  For per-CPU devices (TLB, SYSD, CLK, TTI, TTO) memory barriers are either not required at all or, wherever needed, are properly executed by locking primitives.

  In addition, routine `get_vector` that calls *xx_ack* or retrieves device interrupt vector, will execute RMB – either itself or any of its predecessors that are invoked after device interrupt is noticed but before interrupt is processed and. This ensures that read memory barrier is always executed before processing IO device interrupt. In addition, for devices that dynamically change vectors, such as XQ or RQ, it ensures that primary CPU (CPU0) processing the interrupt will see the latest value of assigned device vector even if it had been just modified by another CPU.[146]

- QBus reset capability had been constrained to the primary CPU only (CPU0). Non-primary processors are not expected to perform QBus adaptor resets. Attempt by a non-primary processor to write to IORESET register will cause `ioreset_wr` to halt to the console with diagnostic message.

  Console command "reset QBA" is disallowed by command handler if currently selected CPU ID is not 0. If nevertheless `qba_reset` gets somehow executed in error by a non-primary CPU, it will stop the machine to the console with diagnostic message.

- Some devices, including XQ and RQ in the present SIMH configuration, allow their interrupt vectors to be changed dynamically, by operating system writing vector change command to device's CSR. This change of interrupts vectors had been factored out of XQ and RQ device code files into routine `io_change_vec(…)` that takes care of SMP issues during vector change.

- SIMH clock event queue (`sim_clock_queue`) is split into per-CPU event queues.

  Access to event queues does not require locking since the queues are always accessed within the context or owning VCPU or in the context of console thread when VCPUs are paused to SIMH console. One exception is access of primary VCPU to the queues of stopped secondary VCPUs when handling the SECEXIT request described below. In neither of these cases there is any concurrent access to the queue by multiple VCPUs, so no queue locking is required.

  Use of event queue by device handlers depends on whether the device is per-CPU or system-global.

  When a *per-CPU* device handler requests the unit to be inserted into the clock queue, this unit gets inserted into the queue of currently executing virtual CPU that it belongs to and that is owned by the current thread. When timer event occurs, it is processed subsequently within the

[146] Compare the code for `io_change_vec` and `get_vector` for synchronization between them. When changing the vector, device handler will call routine `io_change_vec` that obtains interrupt vector table lock, changes the vector and releases the lock; the latter executes write memory barrier. After that it can send the interrupt. On interrupt receiver's side, `get_vector` or its callers execute read memory barrier some time after noticing the interrupt request but before fetching the vector. Thus on the sender's side change to vector table is guaranteed to be ordered before sending the interrupt, while on the receiver side fetching the vector is guaranteed to be ordered after receiving the interrupt. While `get_vector` does not obtain the lock on the vector table, described sequence suffices for properly propagating vector change notification.

context of this VCPU as well.

Several options were considered for *system-global* (non per-CPU) devices.

Two possible designs considered and rejected were the use of separate queue for system-wide devices with events from it processed by the primary processor, and the use of primary's processor own event queue for this purpose (with introduction of locking for it). These designs were rejected since they would incur overhead, and also because there is no system-wide notion of fine-grained time shared by all VCPUs, therefore scheduling fine-grained events would under any of these designs would result either in imperfect event scheduling[147], or significant overhead, or complexity, of all of these vices.

Another design that was considered and rejected was sending event scheduling request to the primary processor via interprocessor interrupts. While event queue operations are performed in this case within the context of primary processor owning the queue, this scheme still suffers from the flaws of two designs described above: overhead, misalignment and poor synchronization between fine-grain device activity and busy-wait loop (in case the driver had triggered device state transition and is busy-waiting for it to complete), especially if primary VCPU thread is not currently running instruction loop – and is either preempted or is blocked in a synchronous system call or is in idle sleep state. This design also has additional flaws: while it does address requests that can be performed asynchronously, it does not address requests that need to be performed synchronously, such as a satisfactory way to check whether event for a unit is pending nor query meaningful value for event scheduled time. Furthermore, since requests with synchronous interface (such as `sim_is_active`) need to be serialized in sequence of execution with "asynchronous" requests (such as `sim_activate` and `sim_cancel`) the latter de facto become synchronous too. More importantly, because all queue manipulation requests need to be synchronized with access to UNIT and other device data (that is synchronized by unit locks and possibly other handler locks), even requests with asynchronously-looking API have actually to be synchronous in their real semantics and implementation. Thus, the requestor cannot just "post" them to the primary processor and proceed with execution: it has to wait for request completion being acknowledged by the primary, which creates a potential for deadlocks. Furthermore, described design would lead to latencies in the execution of clock manipulation routines due to IPI roundtrip and handling delays, and it would also have kept primary processor being woken up from idle sleep by other processors all the time. Thus, it is also not a suitable approach to follow.

Instead, a different design is used by VAX MP.

Timer events for *system-wide* (non per-CPU) devices are inserted into the queue of VCPU in

[147] That could open a worm can of problems. Misalignment between fine-grained device activity and busy-wait timed (TIMEDWAIT) loops is one obvious issue.

whose context sim_activate or sim_activate_abs had been invoked and queued events are subsequently executed on that processor. Since it is possible for device handler to be executed on any of the processors in the system, this can result in events for the same system-wide device (unit) queued on multiple processors. Only one of them, of course, should be actually handled – one that represents the result of the latest sim_activate or sim_activate_abs invocation for the unit. This is achieved by adding an extra field to UNIT structure named clock_queue_cpu. When VCPU inserts an event queue entry for the unit in its event queue, it records VCPU identity into clock_queue_cpu. As explained earlier, routines that manipulate event queue (including sim_activate, sim_activate_abs, sim_activate_cosched, sim_cancel and sim_is_active) must always be called by VCPU while holding a lock that protects access to UNIT structure (*unit lock*). Thus it is possible to tell which VCPU holds the latest (and thus the only logically valid) event queue entry for the unit. When event queue entry for the unit expires on VCPU and is being processed by it, the handler acquires appropriate lock that protects access to the unit and uses macro RUN_SVC_CHECK_CANCELLED described earlier in this chapter. RUN_SVC_CHECK_CANCELLED checks whether unit's clock_queue_cpu field matches current VCPU. If it does, event is processed. If it does not, the event is discarded: held unit lock is released and handler event service routine exits without processing an event.

Note that successive events for a given system-wide unit may get inserted into the queues of different processors (i.e. first on CPU2, then CPU0, then CPU1 and so on) and thus executed each time on a different processor as well.

When secondary processor stops, it is possible for unprocessed events for events for system-wide devices to be left in its event queue. To ensure these events get properly processed, secondary VCPU being stopped sends special interprocessor interrupt named SECEXIT to the primary processor. On receipt of SECEXIT request, primary processor acquires a lock for CPU database and scans all the stopped secondary processors (their event queues) for pending event queue entries for system-wide devices. If found, such queue entries are migrated to the primary VCPU.

Since there is only one UNIT structure for all per-CPU instances of the unit, events can no longer be maintained in UNIT.next and UNIT.time. These fields were therefore removed from UNIT. Clock events are maintained instead in a dedicated clock queue structure, with entries allocated from a lookaside list. Each clock_queue_entry has fields next, time and UNIT* uptr.

Various event handling routines had been modified accordingly to the described changes.

noqueue_time had been moved into per-CPU area.

sim_gtime and sim_grtime had been moved to per-CPU area. The values represent a local processor's view of time. There is no notion of single "current time" shared by all processors in

VAX MP, rather time view is processor-specific, but running processors try to keep within a narrow synchronization window with respect to each other's time.

- Clock queue infrastructure had been revamped to provide better support for co-scheduling device events with the clock when clock strobe thread is used. The purpose is to allow accurate timing of execution of events co-scheduled with the timer to actual SYNCLK timer strobes, and not based on cycle count that may diverge from actual SYNCLK strobes. The intent is to allow idle VCPUs to sleep full inter-strobe quantum without interrupting the sleep in the middle and therefore without associated extra thread context switch overhead (when VCU thread has to wake up in the middle of the quantum between successive clock interrupts instead of being able to sleep the full quantum and wake up just once per quantum) and possible VCPU's inability to sleep the remaining part of the quantum (because this part may be below sleep timer resolution) and having to spin instead.

  Accurate device event alignment with clock strobes had been done for DZ, VH, RQ, TQ, XQ and TTI.

- Routine `process_event` had been modified to temporary elevate thread priority while processing pending events, if any of these events are for devices whose handlers can be expected to acquire locks. These are mostly system-wide (non-per-CPU) devices, however TTI and TTO are also included in this list. The purpose is to avoid holding the lock at low thread priority, risking preemption, and at the same time minimizing the number of "set thread priority" calls to ideally at most one pair per loop (one elevation and one downgrade) instead of elevation and dropping priority by each individual handler.

  Note that we currently do not mark device locks with any specific criticality, so when device handlers' read and write routines acquire the locks they do not elevate thread priority. This approach was chosen because handlers' read and write routines can normally be expected to be invoked by guest OS at high IPL (at device interrupt IPL or higher) and thus already executing at elevated thread priority level, so there is no need to elevate thread priority. The case of `process_event` is distinct because it may be called at low thread priority (for example, while executing VAX user-mode code), and thus thread priority needs to be elevated on demand while processing the event that requires invocation of handler routine that will in turn acquire unit lock. Since unit locks are not marked with criticality, thread priority management is performed inside `process_event` itself.

- Whenever a virtual page data is modified and its PTE's "modified" flag is set (PTE_M), VAX MP executes WMB.

  It might be possible to try to postpone execution of WMB in this case until an interrupt, but this could entail risks, depending on OS behavior and the use of PTE_M by guest OS code: looking into OpenVMS source code and Unix source code would be required to understand their exact use of PTE_M and whether there are cases when access to this bit is not protected enough by other existing synchronization mechanisms. We chose instead a safer and guest OS agnostic approach of not trying to save the cost of WMB.

- Original SIMH memory reading routines did provide required atomicity when reading or writing naturally-aligned longwords, words, and bytes, when executed on x86 or x64 host system.

  However original SIMH data writing routines did not provide required atomicity and write isolation from neighbor data elements. These routines were adequate for uniprocessor, single-threaded version of SIMH but had to be adjusted for multiprocessor (multi-threaded) version. Affected routines are `Write`, `WriteW` and `WriteB`. Implementation currently provided for these routines is processor-dependent and specific to x86 and x64 architectures[148], however is sufficiently generic and should be, if needed, easily generalizable for most other common host systems with different endianness and write alignment requirements.

  Nevertheless atomicity and isolation of datum read/write operations are fundamentally processor-specific; therefore mapping of VAX memory consistency model to the model of other host multiprocessor would have to be an essential part of any port to yet another new host architecture.

- As a related change, to improve performance:

  - Basic routines for memory access were inlined and made to use conditionally compiled native constructs when built on x86/x64.

  - Reference to `cpu_unit->capac`, which is usually of type `int64`[149], was cast to `uint32` in `MEMSIZE` definition, to let compiler's optimizer do away 64-bit arithmetics in `ADDR_IS_MEM` macro.

  - Under GCC, `ADDR_IS_MEM` is hinted with `__builtin_expect`.

  - Instruction stream prefetch via GET_ISTR macro was inlined.

  - VAX MP improves the speed of VAX instruction stream parsing by using streamlined instruction stream fetch code. This optimization can be enabled or disabled with

[148] If not compiled for x86 or x64, a compilation error message would be emitted inside these routines: "Unimplemented".

[149] When USE_ADDR64/USE_INT64 is enabled, which is the case even for most 32-bit builds.

conditional symbol VAX_DIRECT_PREFETCH. If disabled, VAX MP falls back to regular SIMH instruction prefetch code (except GET_ISTR is inlined). By default VAX_DIRECT_PREFETCH is enabled. When VAX_DIRECT_PREFETCH is enabled, VAX MP fetches bytes, words and longwords from instruction stream by directly dereferencing them and avoiding function calls in majority of cases. Prefetch also stays valid till virtual page end or program jump to another page Jumps within the same page adjust existing prefetch pointer but do not revalidate virtual page (this is important since typical VAX code often executes short branches within the page). By contrast, regular SIMH implements only 8-byte prefetch buffer and virtually every read from it incurs the cost of a function call, with some compute-extensive logics. SIMH VAX essentially mimics the actual microcode of MicroVAX 3900 (KA655 processor), whereas VAX MP with VAX_DIRECT_PREFETCH enabled behaves like an architectural, rather than historical hardware simulator.

Overall, these changes produced speed-up of execution by about 25% on compute-intensive tasks such as compiling.

Profiling of the simulator after this change had been implemented indicates that the bulk of CPU time is spent inside the instruction loop itself, evenly spread across it with no identifiable significant "hotspots", while the rest of supporting routines, including virtual memory access routines, contribute only a minor fraction.

- Relevant memory locations are declared as *volatile* in C++ code. However we generally avoid using *volatile*, relying on compiler barrier functions instead. See discussion of compiler barriers use above in section "Compiler barriers".

- VAX MP adds CPUID and WHAMI registers to base KA655 privileged processor register set.

- CRT file I/O in SIMH was modified to make it thread-safe. All `FILE` objects were changed to type `SMP_FILE` that contains a critical section (`smp_lock`) associated with the file and wrappers were provided for various CRT I/O functions that perform locking of the critical section object while an I/O operation is being executed on the file. References to `stdin`, `stdout`, `stderr` were also changed to wrapping objects `smp_stdin`, `smp_stdout`, `smp_stderr`.

- Instruction history recording has been modified to support simultaneous recording for multiple virtual processors.

  There are several possible approaches how recording buffer can be implemented for multi-processor case. Leaving out approaches that rely on locking of a shared buffer, which is obviously most undesirable for a resource with very high contention, three lock-free possible approaches initially considered were:

- Use separate history buffers for each of the virtual processors and do not try to record shared sequence stamp. This makes it possible to display later instruction stream history individually for each of the virtual processors, but not their real-time order relative to each other, thus making diagnosing multiprocessing synchronization issues more difficult.

  On the other hand, this option provides the least impact on execution performance.

- Use per-CPU history buffers, but also record shared sequence stamp into each history record. This allows to display instruction stream history for each of individual virtual processors, and also view instruction stream for the system as a whole, for all the processors, sequenced the same way instructions were executed in real time.

  This way, in addition to the analysis of separate per-processor instruction streams, it is also possible to easily track parallel execution of instructions by all the processors and their timing relative to each other.

  Sequence stamp is obtained by interlocked increment operation and thus there is the cost of one host system interlocked operation (and, frequently, memory barrier that goes along with it) per VAX instruction executed. This definitely impacts the performance, but to an extent usually bearable for debugging purposes.

  Generation of sequence stamp requires host interlocked instruction or instructions that can operate on 64-bit data, either 64-bit interlocked increment returning the results or other interlocked instructions that would have sufficient power to implement 64-bit interlocked increment (such as 64-bit CAS or LL/SC). Such instructions are typically available on modern 64-bit host processors, including x64 processors. On 32-bit processors required instruction is often, but not always available in the form of DCAS. DCAS is available on x86 processors starting with Pentium and is named CMPXCHG8B.

  However some 32-bit host platforms may have no DCAS nor any other 64-bit interlocked increment capability. They can natively support only 32-bit sequence stamps. The consequence of using 32-bit stamps would be that for those virtual processors that went into idle sleep for a minute or more, there will be no easy and simple way to tell from their history stamps alone if these stamps belong to the current or one of the previous overflow cycles of 32-bit stamp counter (counter “generations” or “incarnations” or “epochs”). Assuming instruction speed execution of 30 million instructions per second and 4 active processors, counter overflow will be achieved in about half a minute. To be able to relate “counter generations” to each other, either some form of monitoring code would be required, with provisions for reliability in cases if any worker thread gets stalled for whatever reason by host operating system while other threads continue to execute, or a trick to somehow extend the counter to 64 bits.

- Use single system-wide circular history buffer shared by all the processors for recording of their instruction streams. Three interlocked operations are required per VAX instruction recording: one interlocked instruction to obtain the stamp, that also determines the slot to use in the buffer by taking the stamp value module buffer size; and then extra two interlocked instructions to lock and unlock the slot – the locking is required in case recording thread gets stalled during the recording and other processors in the meantime run around the buffer and arrive to the same slot.

  Limited ring buffer size alleviates the 32-bit counter incarnation problem mentioned above, but per se does not solve it fully. The problem with “counter generation aliasing” could however occur only if the thread is stalled by host operating for several minutes right in the middle of recording – theoretically possible, but improbable event, and impossible altogether in VAX MP when synchronization window mechanism is employed (however, synchronization windows is optional mechanism and is not activated for all possible VAX MP execution modes). Events outside of recording routine, such as processor’s idle sleep, do not affect the recording.

After considering these options, the approach that was finally settled on was to provide user with a selectable choice between the first approach and a modified version of the second approach. Recording method is selected by user via console command, for example

```
SET CPU HISTORY=200/SYNC
```

selects recording with synchronized interprocessor stamping (second approach), while

```
SET CPU HISTORY=200/UNSYNC
```

selects recording that is not synchronized across the processors (first approach). Default is SYNC.

Likewise, to display instruction streams from all processors, rather than just the currently selected processor, use console command

```
SHOW CPU HISTORY=SYNC
```

or

```
SHOW CPU HISTORY=10/SYNC
```

to display ten last commands from the whole system-wide sum of instruction streams.

Without SYNC option, `SHOW CPU HISTORY` command applies to currently selected processor’s instruction stream only.

Use of first approach is straightforward. Modified second approach works as follows. On x64, interlocked 64-bit increment operation is used for stamping. It is also straightforward on any 32-bit x86 processor with DCAS instruction, including Pentium and later processors. For processors that do not have DCAS, including x86 processors earlier than Pentium[150], the challenge is to find a solution for generating 64-bit sequence stamp with the lowest possible contention.

Generating sequence stamp has to be lock-free and wait-free and very efficient since it is an operation performed at very high rate and is a spot of very high contention.

The following trick is employed for pre-Pentium x86 and other similar DCAS-less hosts to generate 64-bit stamp (actually 63-bit stamp), hinged on the expectation that among the threads with the same base scheduling priority, computable thread cannot be stalled by operating system for far too long (minutes), while other threads of the same base priority continue to execute, further enforced by synchronization window when the latter is enabled:

> 64-bit (actually 63-bit) counter is represented by two 32-bit words. Low half is kept in the variable `hst_stamp_counter` that is incremented using 32-bit interlocked instruction. High part (hereafter, *epoch*) is kept in the variable `hst_stamp_epoch`, in its high 31 bits. Low bit of `hst_stamp_epoch` is used as descried below. `hst_stamp_counter` is accessed only via interlocked increment. `hst_stamp_epoch` is read as a regular longword (declared *volatile*, aligned on 4-byte boundary for atomicity, but not causing memory barrier) but modified with CAS instructions, as described below.
>
> When `counter` reaches the value of 0x40000000, the incrementing thread stores the following value in `hst_stamp_epoch`: (*epoch*+1, 1), where left part inside the brackets represents bits 31…1, right part represents the lowest bit, and *epoch* is high 31 bits read from `hst_stamp_epoch`.
>
> When counter reaches the value of 0xC0000000, incrementing thread stores the following value in `hst_stamp_epoch`: (*epoch*, 0), where epoch is high 31 bits read from `hst_stamp_epoch`.
>
> For all other `counter` values between 00000000 and 80000000 *epoch* is calculated from `hst_stamp_epoch` value the following way:

```
epoch = hst_stamp_epoch;
if (0 == (epoch & 1))
    epoch = epoch/2;
else
    epoch = epoch/2 – 1;
```

> For `counter` values between 80000000 and FFFFFFFF *epoch* is calculated from `hst_stamp_epoch` value the following way:

[150] It is, of course, extremely unlikely that x86 multiprocessor systems with pre-Pentium processors still exist around, do support current version of Windows or Linux, and will be used for running VAX MP.

```
epoch = hst_stamp_epoch;
epoch = epoch/2 – 1;
```

Safety guard zones are placed around `counter` values 70000000 to 90000000 where the reader expects low bit of `hst_stamp_epoch` to be set. Another zone is placed from `counter` values 00000000 to 10000000 and from F0000000 to FFFFFFFF. In this second zone readers expects `hst_stamp_epoch` low bit to be cleared. If expected condition in a safety zone is not met, history recording is deemed to fail.

As an additional safety measure, updates of `hst_stamp_epoch` are performed with CAS instructions. If CAS fails, history recording is deemed to fail.

If history recording fails to generate a valid sequence stamp, diagnostic message is printed to the console and further recording is suspended.

The failure can happen only if a processor is stalled in the middle of recording, while other processors continue to run and pass collectively over 0x3000000 VAX instruction recording cycles in the meanwhile – an event extremely unlikely per se, and made impossible altogether by virtual CPUs enforcing synchronization windows relative to each other and pausing/yielding if some virtual processor falls behind the window (in case the use of synchronization window is enabled by configuration).

Nevertheless recording failure can happen (and shut off instruction stream recording) if the use of synchronization window is disabled, one of VCPU threads get stalled and other VCPUs collectively accumulate 0x3000000 instructions. Under typical speed of modern x86/x64 host processors (but then, described algorithm is essential only for older legacy processors that do not have 64-bit capability and do not feature DCAS), on configuration with 10 virtual VAX CPUs this can happen if the thread gets stalled for about 2-3 seconds (and much more, of course, for older processors). Such stalling is *very* likely, but nevertheless quite possible. Should it ever become a problem, algorithm can be further enhanced: each VCPUs may record its current epoch. Each VCPU may periodically check that it's idea of current epoch is consistent with other VCPUs notion of epoch and if it detects that some VCPU falls behind, then stall in the cycle

```
while (epochs are out of sync)
    { sleep(1 usec); }
```

until desynchronization clears.

- VAX MP provides a setting that allows instruction history recording be stopped when processor executes BUGW or BUGL pseudo-instruction. This is helpful for debugging VMS system crashes, since recording gets stopped at the point bugcheck was generated, and history buffer is not flushed by subsequent instructions. To activate this setting, execute SIMH console command

```
DEPOSIT  BUGCHECK_CTRL  1
```

- STEP command handing had been reworked for VAX MP. Variable `sim_step_unit` had been replaced with `CPU_UNIT.sim_instrs` counter and `CPU_UNIT.sim_step` to make instruction counting independent from timer facilities, make instruction counting semantics more clear and robust and allow timers to be changed (such as after real-time wait) without affecting instruction counting.

- Breakpoint package (routines and data entities named `sim_brk_`*`xxx`*) required a rework for the multiprocessing version. Some data elements were moved to per-CPU area. Multiple nearly-simultaneous breakpoint hits on multiples processors are supported. During execution of breakpoint actions, CPU context is set to the CPU that hit a particular breakpoint. If multiple CPUs hit multiple breakpoints almost simultaneously, all triggered breakpoint actions will be invoked in their appropriate CPU contexts. Other miscellaneous clean-up and minor fixes were performed to the breakpoints code.

  Commands RUN, GO and BOOT cannot be executed directly or indirectly (in the invoked script files) from a breakpoint action context. Attempt to execute them will be rejected with diagnostic message.

  If breakpoint action invokes STEP command, it will be executed for the processor that triggered the breakpoint. If multiple processors triggered breakpoints with STEP in the action, STEP will be executed for all of them.

  If breakpoint action invokes CONT(INUE), this command will be executed after all other commands listed in the action complete.

  The logic for breakpoint handling in multiprocessor case is as follows. Once a breakpoint is triggered on one or more processors, all processors are stopped and control is transferred to the console thread. Console thread examines processors for triggered breakpoints and builds in-memory script with the following content (assuming, by the way of example, that processors 3 and 4 triggered breakpoints and console's default context was CPU0):

  ```
  CPU ID 3
  … breakpoint action commands triggered by CPU3 …
  CPU ID 4
  … breakpoint action commands triggered by CPU4 …
  CPU ID 0
  ```

  Console thread then executes this script. If any of the invoked actions contained CONT(INUE), it will not be executed immediately, but recorded as a flag till all other commands specified in the actions complete. If "continue" flag is set after the execution of the script, console thread will execute CONT(INUE) after the script had been performed.

  STEP, on the other hand, is executed immediately, in-sequence with respect to other breakpoint action commands. This allows to define breakpoint handling scripts that examine the state, execute STEP on the paused processor and examine the state again after the step.

If execution of STEP triggers another breakpoint, console thread will build another script and execute it as a nested script. There can be a stack of nested scripts for breakpoint invocations. As a safety measure against runaway scripts, script nesting depth is limited.

- CPU throttling had not been implemented yet for SMP version and had been disabled for VAX MP virtual machine, for now.

- `sim_flip` buffer variable in file sim_fio.cpp had been made thread-local.

- Decrementing `sim_interval` had been replaced throughout the code with macro `cpu_cycle()` that also advances `CPU_UNIT.cpu_adv_cycles`.

- MicroVAX 3900 console ROM is not SMP-aware and can wreak havoc on the system if executed while multiprocessing is active. For this reason VAX MP disables access to console ROM (>>> prompt) when more than one virtual processor is currently active. It is possible to access console ROM when only one processor is active, even after SMP had been enabled; however if multiple processors are currently active, BREAK action (such as telnet BREAK command or console WRU character such as Ctrl/P) will be ignored. Access to console ROM prompt is reenabled when primary VCPU is the only CPU in the active set, such as before START /CPU command, after STOP /CPU/ALL command, after system shutdown or during and after system bugcheck.

  One firmware console function that is sometimes needed during system operation is the access to VAX SIRR (software interrupt request) register that is used to request VMS to cancel mount verification on a device or trigger XDELTA debugger. Since firmware console is inaccessible when multiprocessing is active, SIRR access functionality had been replicated in SIMH console.

  To trigger XDELTA (assuming XDELTA is loaded), bring up SIMH console using Ctrl-E, then:

```
    sim-cpu0> DEP SIRR E
    sim-cpu0> CONT

    1 BRK AT 800027A8
    800027A8/NOP
```

  To access mount verification cancellation prompt, do:

```
    sim-cpu0> DEP SIRR C
    sim-cpu0> CONT

    IPC> C DUA1
```

- When VCPU enters OpenVMS idle loop, VAX MP can hibernate associated VCPU thread instead of causing it to spin in the idle loop (as OpenVMS does on a real hardware processor), in order to conserve host computational resources and power consumption by the core. Hibernation is controlled by VSMP option IDLE that can be initially set during VSMP startup with command VSMP LOAD IDLE=*value* and subsequently dynamically changed with VSMP SET IDLE=*value*.

  Valid values for IDLE are ON, OFF, NEVER.

  ON enables VCPU threads hibernation when OpenVMS executes idle loop.
  OFF disables it.

  NEVER can be used only in VSMP LOAD command (not in VSMP SET IDLE) and means that VCPU threads will never be hibernated. Effect of NEVER is the same as OFF, but when NEVER is used, VSMP does not tap into OpenVMS idle loop. If VSMP LOAD=NEVER is used, it is impossible to change the setting to ON subsequently without prior OpenVMS restart.

  NEVER is incompatible with the use of ILK synchronization window, since SYNCW-ILK requires VSMP tapping into OpenVMS idle loop. If VSMP LOAD command specifies option IDLE=NEVER along with SYNCW=ILK or SYNCW=ALL, a warning message will be printed and the value of IDLE will be overriden to OFF.

- One of the data items used by OpenVMS scheduler is longword `SCH$GL_IDLE_CPUS`. This longword holds a bitmask of CPUs in the system currently executing idle loop. Modifications to `SCH$GL_IDLE_CPUS` are performed while holding SCHED spinlock. When CPU performs scheduling, it acquires SCHED spinlock, tries to select process for execution, and if no matching process with satisfactory affinity found (and process rearrangement across CPUs is not possible), CPU will set the bit corresponding to its CPU ID in `SCH$GL_IDLE_CPUS`, release SCHED spinlock and go into idle loop. While spinning in the idle loop, CPU monitors its bit in `SCH$GL_IDLE_CPUS` reading the latter in non-interlocked fashion without any locking, and if it notices its bit had been cleared (typically, by other CPU), idling CPU will reattempt scheduling.

  When another CPU changes the state of some previously waiting process to computable, that CPU will check if there are any idle processors in the system, and if so, clear their bits in `SCH$GL_IDLE_CPUS`, in order to cause them to execute scheduler and pick up the process for execution.

  Thus, one `SCH$GL_IDLE_CPUS` use is interprocessor communication to effectively wake up idling processors out of OpenVMS idle loop.

  This use assumes that idling processors are actively spinning in OpenVMS idle loop and keep checking the content of `SCH$GL_IDLE_CPUS` all the time. Under VAX MP this is not the case if idle VCPU's thread had been hibernated: idle virtual processor then is effectively paused and is not spinning in OpenVMS loop, and hence is not checking for the changes in SCH$GL_IDLE_CPUS. Therefore, when SCH$GL_IDLE_CPUS is changed by OpenVMS by clearing the bits for VCPUs that are currently hibernated by VAX MP in idle sleep state, these VCPUs must be woken up out of sleep state so they can recheck the value of SCH$GL_IDLE_CPUS and respond to its change.

To restate, clearing bits in SCH$GL_IDLE_CPUS should cause VCPUs with corresponding CPU IDs be woken out of sleep state.

To implement this, when VSMP kernel-resident module is being loaded into VMS kernel, it reports virtual address of SCH$GL_IDLE_CPUS to SIMH layer of VAX MP. Reported address is stored by VAX MP in variable sys_idle_cpu_mask_va. VAX MP then monitors changes to this virtual address, i.e. to SCH$GL_IDLE_CPUS. To reduce overhead of this monitoring, it is performed only within the implementation of those VAX instructions that OpenVMS actually uses to clear bits in SCH$GL_IDLE_CPUS: CLRL, BICL2, BBSC. When VAX MP detects that any of these instructions had changed SCH$GL_IDLE_CPUS, VAX MP will wake up VCPUs with CPU IDs corresponding to cleared bits out of idle sleep.

sys_idle_cpu_mask_va is reset by VAX MP on operating system shutdown or primary processor reset. Care is also taken to temporary disable its handling while processor is interrupted to console ROM firmware via CTRL/P or BREAK and to resume it on CONTNUE from console ROM.

- Milliseconds as a unit of time measurement had been replaced by microseconds, wherever available, to provide a greater accuracy of current CPU speed calibration and forward progress calculation after idle sleep.

- SIMH dynamically calibrates virtual processor execution speed to provide a conversion from milliseconds to the number of instruction cycles. This conversion is used to create event queue entries scheduled in terms of time domain, but tracked in terms of VCPU instruction cycles remaining till the event should be processed. SIMH adjusts calibration data every second. VAX MP modifies SIMH VCPU speed calibration algorithm by averaging calibration data across all the active VCPUs. When performing the averaging, VAX MP takes into the account calibration data only for those VCPUs that had been sufficiently active during last one-second cycle. Data for those VCPUs that spend virtually all time of their last cycle in idle sleep (rather than executing) is not used for averaging, since its precision can be poor and distort the average.

- XQ handler was modified to issue appropriate memory barriers (smp_mb and smp_wmb) when accessing BDL.

- MSCP controller handlers (RQ and TQ) were modified to issue appropriate memory barriers (smp_mb and smp_wmb) when accessing UQSSP COMM area.

- SSC clocks have been redesigned to source timing data from host OS real-time clock.

  SSC clock is used on machine running OpenVMS in two cases:

  One case is firmware ROM performing hardware self-test at power-up time. In this case the original SIMH code is still used that obtains virtual timing data from VCPU cycle counter. One of ROM self-tests (test 31) used to fail sporadically even with that code, especially when clock strobe thread was in use, because tolerances within that test were too high (less than 0.15%). This failure did not have any functional impact on ability of system to execute OpenVMS, but caused ROM to print ugly-looking message potentially scary and confusing to the user saying the machine was not in usable condition. We modified routine `tmr_tir_rd` to recognize that this particular test is being executed (by checking that MAPEN is off and PC matches one of two known locations within ROM) and returning pre-cooked "good" values that satisfy the test.

  The other use case is when OpenVMS or VSMP use SSC clock to perform processor calibration. In this case VAX MP relies on an alternative code that source timing data from host OS real-time clock to provide timing data reported by the SSC clock.

- VAX MP adds console command to monitor performance counters.
  Format of this command is:

  ```
  PERF [ON / OFF / SHOW / RESET] [counter-id]
  ```

  ON enables specified counter or counters.
  OFF disables it.
  RESET resets accumulated counter data.
  SHOW displays accumulated values.
  PERF with no arguments is equivalent to PERF SHOW.

  If *counter-id* is omitted, operation is performed on all counters defined in the simulator.

  Currently performance counters are implemented for critical section locks (`smp_lock` objects).

- VAX MP implements "fast paths" for the case when VAX MP is running in uniprocessor mode (with only one VCPU currently active) and does not incur in this case the overhead of multiprocessor synchronization code and runs at about the same speed as uniprocessor SIMH (or even a tad better, due to optimized memory access routines).

- Secondary processors are supposed to be properly shut down by operating system before its termination. However in emergency cases, when guest OS fails badly, secondary processors may be left unstopped. As a safety net for this case, when primary processor executes HALT instruction or jumps to firmware ROM, VAX MP stops any secondary processors that are still running by sending then non-maskable interprocessor STOP interrupt request and waiting for them to come to a stopped state. STOP IPI request is handled within SIMH level and is not exposed to VAX level.

- Various optimizations to VAX MP had improved per-CPU computational speed about 2.5-3x compared to SIMH VAX 3.8.2, however most of these optimizations had since then been transferred back to regular SIMH VAX as well in version 3.9. VAX MP still has about 20% better per-CPU computational speed than SIMH VAX 3.9 due to streamlined instruction prefetch code.

Bugs noticed in original SIMH code and fixed for VAX MP

- QBus vector auto-configuration was not working properly in original SIMH 3.8-1 when XQB was configured in the system. Fixed the code for auto_config. Declared xqb_dev as DEV_FLTA (floating address device).

- In file pdp11_tq.cpp four code lines

  ```
  { UDATA (&tq_svc, UNIT_ATTABLE+UNIT_DISABLE+UNIT_ROABLE, 0, INIT_CAP) }
  ```

  were changed to discard erroneous extra argument “0”.

Miscellaneous minor fixes to SIMH

- Fixed SET CPU IDLE={…} command handler so it no longer produces false error message.

- Added detection of IAC-DO and IAC-DONT telnet sequences in sim_tmxr.c. These sequences are still not really processed, but at least they are removed from the data stream, so no garbage characters appear when trying to connect to SIMH with terminal emulators such as IVT VT220.

Requirements for build environment (Windows)
Requirements for build environment (Linux)

[The content of these sections had been moved to “VAX MP OpenVMS User Manual”.]

Requirements for execution host environment

80486 or older processor is required on host system executing VAX MP. Intel 80386, its clones and few early sub-models of 80486 will not work.

On Linux, the reason that earlier processor won’t work is that many GCC intrinsic functions for interlocked operations use instructions not available before 80486 (XADDL, CMPXCHG).

On Windows, CMPXCHG is used as well.

VAX MP will check CPU version at initialization time and if the processor does not implement CPUID instruction introduced in 80486 (some earlier 486 sub-models may not support it), VAX MP will exit with diagnostic message.

In addition, for Linux/GCC builds, default build option used for 32-bit (x86_32) builds is `-mtune=pentium` since GCC does not implement instruction scheduling optimization for 80486. If 32-bit x86 Linux build of VAX MP detects that it runs on a processor earlier than Pentium and was not compiled with options `-mtune=i386` or `-mtune=i486`, it will abort at initialization with appropriate diagnostic message.

Thus, on Linux builds, default requirement for target system is Pentium. The build will not work on 80386 or 80486 unless compiled with `mtune` options for these processors.

> (These requirements, of course, are pretty academic. It would be very hard to find pre-Pentium multiprocessor system supported by any of the targeted host OS'es.)

On Linux, glibc 2.13 or older is required. One particular important dependency is NPTL version of PTHREADS library.

On Windows, WinPCAP version 4.0 or later is required for networking support.

On Linux, libpcap is required for networking support (the latest released version is recommended; testing had been done with lbpcap0.8 version 1.1.1-2).

For host-to-VM bridging on Linux, bridge-utils and uml-utilities are required.

When debugging VAX MP under Microsoft Visual Studio, disable tracing of exception `sim_exception_ABORT` in the debugger, otherwise execution of VAX MP under debugger will slow down to a crawl because every single ABORT will require roundtrip between the simulator and the debugger, and debugger output window will also be flooded with messages about this C++ exception being thrown. To disable tracing of `sim_exception_ABORT`, execute menu Debug → Exceptions; after the Exceptions dialog comes up, select "C++ exceptions" and press "Add…" button; select Type as "C++ exceptions" and enter name "sim_exception_ABORT". Once this exception had been added to the "C++ exceptions" group, uncheck the check box for it in "Break when exception is thrown" column. Then, while Visual Studio debugger is active, right-click on the Output window and uncheck "Exception messages" to disable tracing of exception messages to debugger output window.

Note that running VAX MP under debugger is bound to be slower than standalone since debugger intercepts `longjmp` and `throw` operations and processes them internally even if exception tracing is disabled. These operations happen on every single guest OS page fault, among other cases, so slowdown when running under debugger can be quite noticeable.

Stress tests

VAX MP codebase includes several stress tests that can be executed under OpenVMS running on a multiprocessor. Supplied stress tests create parallel processes that perform various activities stressing the system and verifying continuing integrity of its operations under stress conditions.

[The content of this section had been moved to "VAX MP OpenVMS User Manual".]

Stress tests so far had been performed on dedicated host machine with little concurrent host load. Stability of the simulator in the presence of significant concurrent host load had not been subject to testing yet.

Correctness of network operations (involving proper memory barriers in interaction between XQDRIVER and DEQNA or DELQA Ethernet controllers) can be verified in two was. Smaller test involves parallel FTP of multiple large file into VAX and then out and comparing resultant files with the original. Larger test involves running ST_FILE test over the cluster. We typically configure two nodes, each with two drives on separate controllers (DUA, DUB) and run criss-cross cluster IO traffic to remote disks between the nodes.

Possible improvements for subsequent VAX MP versions

It may be worthwhile to consider implementation of the following improvements in a subsequent version of VAX MP:

- Review and improve synchronization:

    - Once VAX MP is deemed basically stable, investigate whether it is possible to replace LFENCE/MFENCE/SFENCE based memory barriers in the x86/x64 version with more performant barriers as described in the comment in sim_barrier.cpp (by flipping flag `SMP_X86_USING_NONTEMPORAL` to FALSE) after verifying that target compilers do not generate non-temporal SSE/3DNow instructions and also that runtime libraries used by VAX MP do not utilize such instructions without properly bracketing them with LFENCE/SFENCE.

      Setting `SMP_X86_USING_NONTEMPORAL` to FALSE can cut overhead of VAX interlocked instructions and other inter-thread operations related to memory barriers 2-fold to 3-fold.

      However at the moment status and policy of MSVC/CRTL and GCC/GLIBC on the use of non-temporal instructions is undocumented and unclear.

      Besides being undesirable for the reasons and purposes described above, lack of compiler and runtime library clear policy on the use of non-temporal instructions is dangerous since it may create a situation where compiled code ends up using non-temporal instruction on a datum that shares cache line with a lock or other variable accessed with x86/x64 interlocked instructions. Such overlapping would disrupt memory coherency protocol and invalidate interlocked access to the latter variable.

    - Implement additional performance counters, including counters tracking CPU and time resources spent on synchronization, such as blocking in inter-CPU synchronization window;

    counters for monitoring frequency of VCPU thread priority transitions; etc.

    - Review new performance counters and existing performance counters (exposed via PERF command) for insight into possible optimizations.

        - Note that during VMS bootstrap generates about 1.5 million VH lock acquisitions. This is legitimate and normal, since during controller initialization VH purposefully makes YFDRIVER to spin on "initialized" bit for 1.2 seconds, before setting this bit in CSR.

        - Also note that typical basic VMS bootstrap (system with minimum of layered products: TCP/IP, LAT, C) generates about 2.2 million interlocked instructions, vast majority of them BBSSI/BBCCI on MMG and SCHED spinlocks (mostly MMG), remotely followed by other spinlocks. Interlocked queue operations, by comparison, are much less numerous. About 100K interlocked instructions are issued by user mode code.

        - Specifically review the counters on cpu_database_lock when running in SYNCW mode.

    - Consider replacement of existing locks with MCS queue locks or M-locks or composite locks, as described in the comment in the header for method smp_lock_impl::lock().

- Implement multi-tick idle sleep as described in "Timer Control" chapter of this document and in the comment inside routine sim_idle. Multi-tick idle sleep is currently implemented at VSMP level, but SIMH level ignores passed `maxticks` argument. While consumption of CPU computational resources when idling is already low, and its further reduction will not be of much practical significance, multi-tick sleep may allow better power management by host system (core parking), reducing power consumption by idling cores and increasing power consumption and operating frequency (and hence execution speed) by active cores. Be sure to think through the interaction of multi-tick sleep with synchronization window.

- Consider possible improvements to synchronization window algorithm suggested in chapter "Interprocessor synchronization window".

- Extend synchronization window mechanism to limit, in addition to inter-CPU cycle divergence, also clock interrupt count divergence, as described in section "Timer Control".

- Modify MSCP disk handler (RQ) to support multiple simultaneous disk operations executed on the host within the same unit. Right now MSCP controller holds a queue of requests, but only one host disk request per unit is actually outstanding at any given time. Under heavy disk IO performance can be improved if multiple requests were actually queued to host disk. On the other hand, for smaller data sets performance improvement is likely to be negligible since the whole data set will reside in host file system cache, and queued requests are not going to improve IO throughput by much due to seek optimization and intra-(host)-controller pipelining. (However transfer bandwidth may improve if multiple IOP threads per unit are used.) If parallel execution of IO requests is implemented, it is important that in case of overlapping data requests, subsequent requests wait for the completion of previous requests. In overlapping situation, guest OS implies certain ordering of operations. If overlapping requests are handed off to host without serializing them, host OS may reorder them (with disastrous results), since its IO ordering semantics can be different from guest OS IO ordering semantics. For example, it may order requests based on current thread's priority (which in turn may be dynamic because of priority boosts) or based on strategy routines sorting etc. – with resultant ordering distinct from what was implied by guest OS sending requests to MSCP controller. Thus, before passing requests to the host in parallel fashion, it is necessary to check if it overlaps any of active requests or prior requests still pending in MSCP queue, and if so, wait for their completion before passing the request to the host.

- There is currently a race condition between routines `cpu_reevaluate_thread_priority` and `interrupt_set_int`, as explained in the comment to the former. The impact of this race condition is minor when SYNCLK thread is used. However it would be desirable to fix it, especially for any build that does *not* use SYNCLK.

- Implement Save and Load commands for virtual machine state.

- Implement debugging code that dumps system state when VCPU generates a bugcheck, and possibly produces dump file in virtual machine save-file format.

- Optimize MOVC3 and MOVC5 instructions to use memcpy/memset within a page range.

- Optimize QBus Map_Read*X* and Map_Write*X* transfers within a page range.

- Implement self-learning and self-tuning auto-adaptive spinlocks for `smp_lock`. Right now `smp_lock` implements spin-waiting for a certain number of cycles before resorting to OS-level blocking wait. `smp_lock` is meant for short critical sections with usually low contention, when the overhead of OS level blocking may be undesirable and unjustified in most cases, but still resorting to OS level blocking may be necessary if the thread that is holding the lock was scheduled out of the execution in the middle of the critical section (either preempted by another thread or because of a page fault), while holding the lock.

  Right now spin-wait timeout (spin count threshold) before transitioning to blocking wait is assigned somewhat arbitrary as fixed value by the developer at section initialization time (expressed as either number of spin cycles or spin microseconds) and stays fixed throughout the lock lifetime. It may be worthwhile to implement an adaptive, self-learning and self-tuning spinwait locks within the section.

Starting from some initial timeout value and staying within a certain range, spinlock will dynamically histogram wait times and learn empirically how long it takes for a typical "successful" (i.e. not timed-out) spin-wait to complete and adjust its timeout accordingly[151]; the algorithm should also measure and account for the cost of a context switch as busy-wait limiting factor. This will also take care of dynamic adjustment to the effects of dynamically changing host system load.

There is no shortage of research in hybrid locks that transition from spin-waiting to blocking wait and on the choosing a threshold for such transition.[152] In our opinion however real systems exhibit unpredictable patterns of workload resulting in unpredictable and dynamically shifting distribution of wait times, making mathematical models for hybrid locks of little practical utility. Robust approach may only be provided by very simple and robust heuristics, not by complex models that are unable to capture real system behavior. Furthermore, even simple heuristics may be able to cover certain workloads (ones with relatively stationary, slowly-changing distribution of lock holding time and contention level), but not the workloads with erratic, rapidly and randomly changing behavior. Thus first goal for heuristic is to discriminate workload patterns that it can reasonably handle from those it cannot. The second goal is to apply heuristic to the former category of workloads, while for the latter category shift down to even simpler fallback scheme.

Our proposed design for self-tuning hybrid lock, along these lines, is as follows:

- `smp_lock` can collects statistics over certain number of lock cycles, similar to what `smp_lock` does now for PERF command, but delta-statistics, rather than continuously running one. For example, statistics can be kept over a sampling period sized to recent 1000 and/or 10000 lock acquisition cycles. Statistics includes:

    - the number of non-nested "acquire lock" requests that resulted in immediate lock acquisition in the first attempt, without any spin-waiting;
    - the number of requests that led to OS level blocking wait;
    - the number of requests that were accomplished with non-zero number of retry spin cycles, but did not result in blocking wait;

[151] Another possible input is a sampling of lock holding time on the side of the *holder*.

[152] On related matters, see A. Karlin et. al, "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor", SOSP'91 Proceedings of the thirteenth ACM symposium on Operating systems principles (ACM SIGOPS Operating Systems Review, Oct. 1991, pp. 41-55); Beng-Hong Lim, Anant Agarwal, "Waiting Algorithms for Synchronization in Large-Scale Multiprocessors", ACM Transactions on Computer Systems, August 1993; Ryan Johnson, "A New Look at the Roles of Spinning and Blocking"; Ryan Johnson, "Decoupling Contention Management from Scheduling"; and L. Boguslavsky et al. "Optimal Strategies for Spinning and Blocking"; A. Nikolaev, "Exploring Oracle RDBMS latches using Solaris DTrace", Proceedings of MEDIAS 2011 (http://arxiv.org/abs/1111.0594); A. Nikolaev, "Exploring mutexes, the Oracle RDBMS retrial spinlocks", Proceeding of MEDIAS 2012 (http://andreynikolaev.files.wordpress.com/2012/05/exploring_mutexes_oracle_11_2_retrial_spinlocks.pdf); and further bibliography in the last two articles.

    - for the later case, an average number of spin cycles before the lock was acquired.

  Statistics will be acted on either at the end of period, or with higher frequency (at points intra period). In the latter case statistics is maintained as running moving average.

- Based on acquired statistics, adjust spin count to target very low (but non-zero) percentage of lock acquisitions resulting in going to OS-level blocking wait, such as 0.1% or 0.01% of all recent acquisitions.

- If recent blocking wait rate was too low (well below the target), then tighten spin count. If it was too high (well above the target), then loosen spin count.

- Tighten slow, loosen fast. Tighten spin count by a small amount, gradually, based on delta-statistics taken over many acquisitions. However if a contention burst comes in and blocking wait rate goes high, detect it quickly (based on smaller sampling interval, i.e. statistics across a smaller number of recent acquisitions) and loosen spin count quickly by larger amount.

- If lock contention has bursty pattern, detect it and cope with it by shifting down to a fallback. After having to back the tightening off by large amount for several times, give up on tightening and allow spin count to stay large.

    - After large number of sampling periods (e.g. 10 to 100 sampling cycles) it may be possible to resume self-tuning towards lock tightening. However if continuing bursty nature of current workload is confirmed, lock can shift into fallback to non-tuning mode even faster.

- Never increase spin count beyond approx. 1/2 of the cost of rescheduling wait. If spin count reaches 1/2 cost of the context switch, drop it to zero. (Meaning future acquisition requests will try lock acquisition once and if lock is busy proceed straight away to OS blocking wait without retrying in spin loop).

    - After sufficiently large number of sampling periods (e.g. 10 sampling periods) resume self-tuning. However if early observations confirm that current average spin count will stay above the cost of context_switch/2, terminate sampling attempt early and reset spin count back to 0; sampling can be retried again after yet another 10 or more sampling periods, possibly starting with 10 and gradually increasing retry delay in back-off fashion to 100 sampling periods.

- Never tune down spin count too low (except when setting it purposefully to 0). Account for variations in cache/main memory access times etc. Specifically, it does not make sense to set spin count below 10. Also it does not make sense to set spin count below 1-5% of rescheduling cost.

Although as described the design is intended for TATAS locks, with minor adjustments it is also applicable to ticket locks and to queued locks as well[153].

Note that preemption-safe and scheduler-conscious synchronization algorithms suggested by Kontothanassis, Wisniewski and Scott in "Scheduler-conscious synchronization" are a complementary solution, rather than an alternative to adaptive spin/block threshold adjustment, since algorithms proposed in "Scheduler-conscious synchronization" do not provide for a timely spin-waiter cutoff in case of lock holder preemption. However these algorithms are complementary and can be combined with adaptive spin/block threshold adjustment. Added value of adaptive spin/block threshold adjustment is that it reduces processor resource waste due to useless spinning by focusing on dynamic cut-off adjustment due to changes in system load, variance in legitimate (non-preempted) lock-holder execution time, changes in lock contention level and waiters queue length, and environmental factor changes like the change in CPU cores power state.

Likewise, time-published locks[154] (where a holder updates time stamp periodically to let waiters know the holder is not preempted) do not make a practical alternative to described algorithm since in time-published lock approach:

- Code executed while holding a lock needs to be instrumented to update a lock with current time stamp value , and furthermore should perform updates frequently enough for the algorithm to be operational and efficient, but not frequently enough to degrade performance by the updates.

  Such instrumentation would in most cases be either very impractical or outright impossible.

- Time-published lock approach relies on availability in system hardware of high-resolution timer that can be accessed cheaply (similar to x86 RDTSC instruction) and whose value should somehow be shared between the processors, i.e. timer should either be global and running at fixed frequency, or can be per-processor, but then operating system should provide inter-CPU timer synchronization tables offsetting for inter-CPU timers drifts and differences in frequency due to physical variability in processor strobes and also intended variability due to differences in processor power states etc. Most systems currently do not provide such a facility.

[153] Cf. M. Scott, W. Scherer, "Scalable Queue-Based Spin Locks with Timeout" // PPoPP '01 Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming; M. Scott, "Non-blocking timeout in scalable queue-based spin locks" // PODC '02 Proceedings of the twenty-first annual symposium on Principles of distributed computing, pp. 31-40; Prasad Jayanti, "Adaptive and efficient abortable mutual exclusion" // PODC '03 Proceedings of the twenty-second annual symposium on Principles of distributed computing, pp. 295-304. Also cf. Leonidas I. Kontothanassis, R. Wisniewski, M. Scott, "Scheduler-Conscious Synchronization" // ACM Transactions on Computer Systems, Feb. 1997, pp. 3-40.

[154] B. He, W. Scherer, M. Scott, "Preemption Adaptivity in Time-Published Queue-Based Spin Locks" // High-Performance Computing 12th International Conference, Springer, 2005, pp. 7-18 (also: TR 867, Department of Computer Science, University of Rochester).

The original algorithm proposed and discussed by He, Scherer and Scott's article assumes the availability of clock globally synchronized to sub-microsecond (~ 100 ns) range. In actuality however, not all conceivable variants of time-published locks need global clock synchronization. Instead, lock holder can periodically publish a heartbeat signal (e.g. by incrementing a counter variable in the lock structure, or by publishing the value of its local clock), whereas lock waiters would monitor for this heartbeat signal using their local clocks to detect whether the holder is still running or had been preempted – in that case however an additional aspect that should be handled is a possibility of thread migration that would disrupt a continuity of local clock.

   - Timestamp update code should track all time-published locks held by the current thread, and hence needs to know and keep track of all time-published locks held by the thread.

   - If this general approach is to be followed at all, it is much easier (instead of updating time stamp) to register held lock at acquisition time with the operating system by placing lock address in the array of held locks pointers maintained in user/kernel shared page (for the duration of holding the lock); then as the holding thread is preempted, kernel can mark all locks registered in this segment and thus held by the thread being preempted as “held by preempted thread” in the lock structure – a flag immediately visible to the waiters prompting them to abandon busy-wait and enter blocking wait (perhaps also yielding directly in favor of the lock holder).

   - Code executed with lock held may produce a wide distribution of execution time while in the critical section. Some instances of execution within a critical section can be long-lasting even if the holder is not preempted, for legitimate reason of the amount of processing that needs to be performed. In these cases, it does not make sense for the waiters to keep spinning, despite the fact that the lock holder is not preempted and keeps updating the timestamp. A sound approach should still provide a cut-off threshold for waiters’ transitioning from busy-wait to blocking wait, even though lock holder is not preempted and keeps publishing its timestamp. Thus time-published locks is not a replacement for adaptive spin/block algorithm, but at best is a complementary approach that would still benefit from self-tuning spin/block algorithm – albeit not in cases of preemption, but only in cases of legitimate long execution.

- Consider possible merits of changing VCPU thread priority depending on the priority of VMS process being executed, in particular when current process is holding a mutex (PCB.MTXCNT != 0 and PRIO=16).

- To reduce incidence of thread priority changes, `smp_lock` and `smp_mutex` method `set_criticality(prio)` can be modified to `set_criticality(pmin, pmax)`. If thread acquiring the lock already has priority *pmin* or higher, it is considered to be “good enough” and thread priority is not elevated during thread acquisition. If thread priority is below *pmin*, it is raised all way to *pmax*. This way threads already executing at CRITICAL_OS_HI typically won’t need to elevate to CRITICAL_VM when locking locks such as used for interlocked operations in portable mode and can save the cost of two “set thread priority” calls.

- Another approach to reduction of incidence of “set thread priority” calls would be to reduce the number of VCPU priority bands in use in VAX MP, with some of logical priorities being assigned the same value id.

- Various lesser possible improvements are described in the comments throughout the source code and tagged with “ToDo” tag.

Retrospective on features of host OS desirable for SMP virtualization

- One capability commonly needed by applications relying on fine-grained parallelism is efficient control over thread preemption in order to avoid lock holder preemption. A wide class of applications falls under this category, from multiprocessor simulators like VAX MP to database engines.

  Primitives intended to counteract thread preemption would normally be used to bracket critical sections within the applications, with the purpose to prevent the thread being preempted while holding a resource lock. Preemption of resource holder can lead to a wide range of pathologies, such as other threads piling up waiting for the preempted thread holding the lock. These blocked threads may in their turn hold other locks, which can lead to an avalanche of blocking in the system, resulting in drastically increased processing latencies and waste of CPU resources due to context switching and rescheduling overhead. Furthermore, if resource locks are implemented as spinlocks or hybrid locks with "spin-then-block" behavior, blocked waiters will spin, wastefully consuming CPU resources and hindering preempted lock holder to complete its work and release the lock. Priority inversion is easy to occur in this situation as well.

  Yet, despite this situation being common, mainstream operating systems usually do not offer efficient low-overhead mechanism for preemption control. Usually the best they offer is system call to change thread priority, but this call requires userland/kernel context switch and in-kernel processing, which is fine for applications with coarse-grained parallelism, but is expensive for applications relying on fine-grain parallelism. By juxtaposition, mainstream operating systems do provide efficient locking primitives that normally do not require userland/kernel context switching, such as Linux *futex* or Windows *CRITICAL_SECTION*[155], but they do not provide similarly efficient mechanisms for preemption control to match those locking primitives.

  Windows tries to address lock holder preemption issue by using enlarged scheduling quantum in server edition, that (hopefully) should allow thread enough time to handle the request and release the locks, and also reduces context switching overhead. However large scheduling quantum does not help applications that need to execute for an extended time, most of that time not holding resource locks, but frequently acquiring locks for short intervals. In fact, enlarged quantum may even aggravate the problem by preempting lock holder for longer time.

  > At the other end of the spectrum is extreme counter-example of Linux BFS scheduler misused beyond its intended purposes. A personal communication from a friend doing Oracle scalability support relates the story of a panic call from a company that went production with high-end system based on Linux configured to use trendy at the time BFS scheduler. BFS tries to divide

[155] VAX MP `smp_lock` is similar in essence.

> available CPU time between computable processes in the system in a “fair” way, not imposing by default a minimum quant size. At 15,000 processes in the system representing client sessions, a fraction of them in a runnable state, time slice each process was getting allocated to it was next to zero, resulting in CPUs spending next to 0% running user-mode code and close to 100% in kernel mode rescheduling the processes.

PTHREADS interface defines priority protection associated with locks, whereby a thread acquiring the lock will temporary have thread’s priority elevated for the duration of holding the lock, however this feature has limited usability for a variety of reasons described earlier in section “Supported host platforms”, but in addition to that it would not be *per se* an adequate match to low-overhead locking primitives that complete most of the time in userland space, without switching to kernel, whereas PTHREADS interface at the bottom would have to rely on regular “set thread priority” system calls with all their overhead of userland/kernel context switching and intra-kernel actions.

Although the issue of inopportune preemption had been addressed in research early on[156], the only mainstream production operating systems that eventually came to provide a form of low-overhead preemption control are Solaris and AIX. Solaris *schedctl* facility provides functions *schedctl_start()* and *schedctl_stop()* with very efficient low-overhead implementation. When application wants to defer involuntary thread preemption, this thread calls *schedctl_start* that stores a flag in userland/kernel communication memory page accessible to the kernel but also mapped into user space. If scheduler sees this flag, it will try to honor it and give the thread extra time before it is being preempted. The flag is strictly advisory, kernel is under no obligation to honor it, and indeed if the flag stays on for more than a couple of ticks and the thread does not yield, kernel will stop honoring this flag until the thread yields. If kernel considered thread for a preemption, but let it go on because of the “do not preempt me” flag, kernel will set “preemption pending” flag in the user/kernel shared page. When the thread had released the lock and calls *schedctl_stop*, the latter will reset “do not preempt me” flag and check for

[156] Problem of preemption control in operating systems was originally tackled by J. Elder et. al., “Process Management for Highly Parallel UNIX Systems” // Proceedings of the USENIX Workshop on Unix and Supercomputers, Sept. 1988; J. Zahorjan, E.D. Lazowska, D. Eager, “The effect of scheduling discipline on spin overhead in shared memory parallel systems” // IEEE Transactions on Parallel and Distributed Systems, Apr 1991, pp. 180-198; R. Wisniewski, L. Kontothanassis, M. Scott, “Scalable Spin Locks for Multiprogrammed Systems” // Proceeedings of the 8th International Parallel Processing Symposium, pp. 583-589 (expanded version: TR 545, Computer Science Dept., University of Rochester, April 1993); R. Wisniewski, L. Kontothanassis, M. Scott, “High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors” // PPOPP '95 Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM, 1995, pp. 199-206; Leonidas I. Kontothanassis, R. Wisniewski, M. Scott, “Scheduler-Conscious Synchronization” // ACM Transactions on Computer Systems, Feb. 1997, pp. 3-40. See also Maged M. Michael, Michael L. Scott, “Relative Performance of Preemption-Safe Locking and Non-Blocking Synchronization on Multiprogrammed Shared Memory Multiprocessors” // Proceedings of the 11th International Parallel Processing Symposium, IEEE, 1997, pp. 267-273.

"preemption pending" flag. If the latter is set, the thread will yield voluntary.[157] AIX provides very similar facility.[158]

Solaris *schedctl* is a useful mechanism, but has a number of obvious problems.

First, it does not provide a way to associate priority with the resource whose lock is being held (or, more generally, with thread logical state; see footnote below). Application is likely to have a range of locks with different criticalities and different needs for holder protection.[159] For some locks, holder preemption may be tolerated somewhat, while other locks are highly critical, furthermore for some lock holders preemption by high-priority thread is acceptable but not preemption by low-priority thread. Solaris *schedctl* does not provide such capability for priority ranging relative to the context of the whole application and other processes in the system.

Second, in some cases application may need to perform high-priority processing without knowing in advance how long it will take. In majority of cases processing may be very short (let us say, well under a millisecond), but sometimes may take much longer (such as a fraction of a second). Since *schedctl* would not be effective in the latter case, application would have to resort to system calls for thread priority control in all cases[160], even in majority of "short processing" cases, with all the overhead of this approach.

Finally, *schedctl* is strictly advisory mechanism. Kernel is under no obligation to honor it and the calling thread fundamentally still remains low-priority thread preemptable by other low-priority

[157] See man page for schedctl_start(3C). Compare Richard McDougall, Jim Mauro, "Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture", Prentice Hall/Sun Microsystems, 2007, pp. 217-218, 243. Also see "Inverted schedctl usage in the JVM" by David Dice (https://blogs.oracle.com/dave/entry/inverted_schedctl_usage_in_the).

[158] AIX 5L Differences Guide, Version 5.0 Edition (http://ibm.com/redbooks) describes them in section *Context switch avoidance*: "For application programs that are using their own thread control or locking code, it is helpful to signal the dispatcher that the program is in a critical section and should not to be preempted or stopped. AIX 5L now allows an application to specify the beginning and ending of a critical section. [...] After an initial call of *EnableCriticalSections()*, a call to *BeginCriticalSection()* increments a memory location in the process data structure. The memory location is decremented again by a call to *EndCriticalSection()*. This location is checked by the dispatcher, and if it is positive, the process receives another time slice (up to 10 ms). If the process sleeps, or calls yield(), or is checked by the dispatcher a second time, this behavior is automatically disabled. If the process is preempted by a higher priority process, it is again queued in the priority queue, but at the beginning instead of the end of the queue. If a thread is still in a critical section at the end of the extra time slice, it loses its scheduling benefit for one time slice. At the end of that time slice, it is eligible again for another slice benefit. If a thread never leaves a critical section, it cannot be stopped by a debugger or control-Z from the parent shell."

[159] We refer just to locks hereafter for simplicity, but the need of a thread for preemption control does not reduce to locks held alone, and may result from other intra-application state conditions, such as executing urgent peace of code in response to high-priority event (that may potentially be blocking for other threads) or other code path that can lead to wait chains unless completed promptly.

[160] Or introduce extra complexity by trying to gauge and monitor the duration of the processing.

compute bound threads. Moreover, kernel starts ignoring *schedctl* requests under heavy load, i.e. exactly when the aid of *schedctl* is most needed.

A better version of preemption control would be “deferred set thread priority” call. Instead of storing “do not preempt me” flag in the shared user/kernel page, thread would store instead a “desired priority” value (with some special value such as -1 designated to mean “no change desired”). In majority of cases thread will complete critical section and reset “desired priority” to “no change” before kernel could notice the request for “desired priority”. The implementation is thus very low overhead in this majority of cases. However if kernel considers the thread for descheduling or performs any other function that requires accounting for the thread priority, it would check first for “desired priority” request in the shared page, and if it is set, would perform actual kernel-level promotion of the thread to the requested priority in the scheduler database, as if regular “set thread priority” request had been processed. Kernel will also store a “priority changed” flag in the shared user/kernel page indicating thread priority had been elevated. Kernel will then make scheduling decisions based on new thread priority level. In the minority of cases all this would happen, thread will notice “priority changed” flag at the completion of critical section (or on other downward transition in required logical priority level) and execute system call to downgrade its priority. In predominant number of cases described “deferred set thread priority” mechanism would eliminate the needs for making actual system call into the kernel, while maintaining safe preemption control. If rescheduling is not actually needed, multiple “set priority” calls can be executed and coalesced entirely within user space, without the overhead of kernel/user space switching. Suggested mechanism is also devoid of three Solaris *schedctl* drawbacks discussed above.

Thread priority change is very frequent operation when running virtual SMP on non-dedicated host platform. It would be desirable to implement it without the overhead of user/kernel context switching for thread priority management.

> (Suggested primitive is a part of a potentially wider mechanism that, essentially, may allow the process to participate in fine-grained control over the scheduling of its threads, and to perform kernel/application cooperative scheduling while avoiding the bottleneck of low-bandwidth userland/kernel system call communication channel separating intra-application knowledge of scheduling needs from in-kernel scheduler.)

- It is desirable for a process to be able express its need for short voluntary sleep intervals (and hence small scheduling quanta), in a way similar to provided in Microsoft Windows with function `timeBeginPeriod`.
    - And also be able to obtain an estimate of actual duration of minimum voluntary sleep interval.

- Ability to check during spin-wait whether lock holder had been preempted, so spin-wait can be immediately converted to OS-level blocking wait without wasting CPU resources.

- It is desirable to have system call to boost other thread's priority *to at least P*, rather than *to P*. That is, if target thread is already executing above P, its priority is not downgraded.

- Ability to wait for multiple objects of heterogeneous nature, such as event flags or semaphores, and file handles.

- A form of sleep system call with the following arguments:

  - event or set of event/file/synchronizaton object handles;
  - time to sleep, in microseconds or nanoseconds;
  - priority to set for the thread while it is sleeping, so it get control back ASAP on wakeup;
  - actual duration of sleep time, returned within a single system call, without an overhead of additional system calls to query time[161].

[161] OS X implements gettimeofday call on x64 via using base value stored in shared per-process user/kernel page and adjustments to base value coming from RDTSC instruction counting accrued CPU cycles, thus the call is cheap since it incurs no context switch into the kernel, and has very high resolution.

Appendix A

List of general references about memory-consistency problem and model:

Paul E. McKenney, “Memory Barriers: a Hardware View for Software Hackers”, at http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2009.04.05a.pdf; an earlier version of this article is also published as 3-part series “Memory Ordering in Modern Microprocessors” in Linux Journal

“Linux Kernel Memory Barriers”, by David Howells and Paul E. McKenney, Linux kernel distribution, file Documentation/memory-barriers.txt

Paul E. McKenney, “Is Parallel Programming Hard, And, If So, What Can You Do About It?”, http://www.linuxinsight.com/is-parallel-programming-hard-and-if-so-what-can-you-do-about-it.html or www.rdrop.com/users/paulmck/perfbook/perfbook.2011.09.05a.pdf , chapter 12 and appendix C

Sarita V. Adve, Kourosh Gharachorloo, “Shared Memory Consistency Models: A Tutorial”, WRL Research Report 95/7, at http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf; a significantly abbreviated version is republished in IEEE Computer, Dec 1996

Thomas Rauber, Gudula Rünger, “Parallel Programming for Multicore and Cluster Systems”, Springer, 2010, pp. 82-88.

Alpha Architecture Handbook, version 4 // Compaq 2008, EC–QD2KB–TE, chapter 5, “System Architecture and Programming Implications (I)”; Alpha Architecture Reference Manual, 4th edition // Compaq 2002, chapter 5, “System Architecture and Programming Implications (I)”

“Handling Memory Ordering in Multithreaded Applications with Oracle Solaris Studio 12 Update 2”, Part 2, “Memory Barriers and Memory Fences”, Oracle, 2010, at http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-memory-barriers-fences-176056.pdf

“Shared Memory, Threads, Interprocess Communication” // HP OpenVMS Systems, ask the wizard, at http://www.openvms.compaq.com/wizard/wiz_2637.html

Kourosh Gharachorloo et.al., “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors” // Proceedings of the 17th Annual International Symposium on Computer Architecture, at http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.8112

http://en.wikipedia.org/wiki/Memory_barrier

http://en.wikipedia.org/wiki/Memory_consistency_model

For Intel x86 and x64 memory-consistency model, refer to:

Susmit Sarkar et. al., "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors", at http://www.cl.cam.ac.uk/~pes20/weakmemory/cacm.pdf

Susmit Sarkar et. al., "A Better x86 Memory Model: x86-TSO", at http://www.cl.cam.ac.uk/~pes20/weakmemory/x86tso-paper.tphols.pdf

Susmit Sarkar et. al., "A Better x86 Memory Model: x86-TSO (extended version)", at http://www.cl.cam.ac.uk/~pes20/weakmemory/x86tso-paper.pdf or http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-745.pdf

"Relaxed-Memory Concurrency" page at http://www.cl.cam.ac.uk/~pes20/weakmemory/index.html

"Intel 64 Architecture Memory Ordering White Paper", at http://www.multicoreinfo.com/research/papers/2008/damp08-intel64.pdf now a part of Intel Architecture Software Developer's Manual, Volume 3

Comments by Erich Boleyn of Intel PMD IA32 Architecture group in "spin_unlock optimization(i386)" thread http://www.gossamer-threads.com/lists/engine?do=post_view_flat;post=105365;page=1;mh=-1;list=linux;sb=post_latest_reply;so=ASC

As a footnote, for older x86-CC model (deficient and superseded by x86-TSO model) refer to:

Susmit Sarkar et. al., "The Semantics of x86-CC Multiprocessor Machine Code", at http://www.cl.cam.ac.uk/~pes20/weakmemory/popl09.pdf

Susmit Sarkar et. al., "The Semantics of x86 Multiprocessor Machine Code // Supplementary Examples", at http://www.cl.cam.ac.uk/~pes20/weakmemory/examples.pdf

Also see Intel, AMD and Microsoft documents:

"Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A & 3B): System Programming Guide" (order number: 325384-039US), sections 8.* and 19.34, at http://www.intel.com/Assets/PDF/manual/325384.pdf

"Intel Architecture Software Developer's Manual, Volume 3: System Programming" (order number 243192), sections 7.1.2 and 7.2.2 at http://communities.intel.com/servlet/JiveServlet/downloadBody/5061-102-1-8118/Pentium_SW_Developers_Manual_Vol3_SystemProgramming.pdf

"AMD x86-64 Architecture Programmer's Manual, Volume 2: System Programming" (publication 24593, revision 3.18), chapter 7, at http://support.amd.com/us/Processor_TechDocs/24593.pdf

“Multiprocessor Considerations for Kernel-Mode Drivers”, Microsoft, 2004, at http://msdn.microsoft.com/en-us/windows/hardware/gg487433

For overview of cache coherence protocols see:

Ulrich Drepper, “What Every Programmer Should Know About Memory”, at http://people.redhat.com/drepper/cpumemory.pdf or http://www.akkadia.org/drepper/cpumemory.pdf

http://en.wikipedia.org/wiki/Cache_coherence

http://en.wikipedia.org/wiki/MESI_protocol

http://en.wikipedia.org/wiki/MSI_protocol

http://en.wikipedia.org/wiki/MOSI_protocol

http://en.wikipedia.org/wiki/MOESI_protocol