Introduction to Multithreading and Multiprocessing in the FreeBSD SMPng Network Stack

Robert N. M. Watson
rwatson@FreeBSD.org

Computer Laboratory
University of Cambridge

Abstract

The FreeBSD SMPng Project has spent the past five years redesigning and reimplementing SMP support for the FreeBSD operating system, moving from a Giant-locked kernel to a fine-grained locking implementation with greater kernel threading and parallelism. This paper introduces the FreeBSD SMPng Project, its architectural goals and implementation approach. It then explores the impact of SMPng on the FreeBSD network stack, including strategies for integrating SMP support into the network stack, locking approaches, optimizations, and challenges.

1 Introduction

The FreeBSD operating system [4] has a long-standing reputation as providing both high performance network facilities and high levels of stability, especially under high load. The FreeBSD kernel has supported multiprocessing systems since FreeBSD 3.x; however, this support was radically changed for FreeBSD 5.x and later revisions as a result of the SMPng Project [1] [6] [7] [8].

This paper provides an introduction to multiprocessing, multiprocessor operating systems, the FreeBSD SMPng Project, and the implications of SMPng on kernel architecture. It then introduces the FreeBSD network stack, and discusses design choices and trade-offs in applying SMPng approaches to the network stack. Collectively, the adaption of the network stack to the new SMP architecture is referred to as the Netperf Project [5].

2 Introduction to Multiprocessors and Multiprocessing Operating Systems

The fundamental goal of multiprocessing is the improvement of performance through the introduction of additional processors. This performance improvement is measured in terms of "speedup", which relates changes in performance on a workload to the number of CPUs available to perform the work. Ideally, speedup is greater than 1, indicating that as CPUs are added to the configuration, performance on the workload improves. However, due to the complexities of concurrency, properties of workload, limitations of software (application and system), and limitations of hardware, accomplishing useful speedup is often challenging despite the availability of additional computational resources. In fact, a significant challenge is to prevent the degradation of performance for workloads that cannot benefit from additional parallelism.

Architectural changes relating to multiprocessing are fraught with trade-offs, which are best illustrated through an example. Figure 1 shows performance results from the Supersmack MySQL benchmark [2] on a quad-processor AMD64 system, showing predicted and measured transactions per second in a variety of kernel configurations:

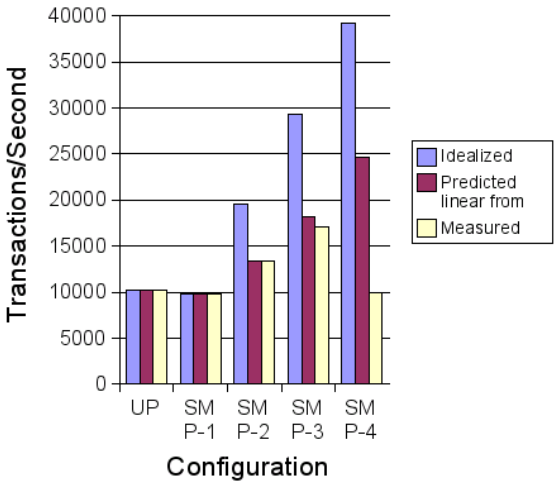

Figure 1: Speedup as Processors Increase for MySQL Select Query Micro-Benchmark

A number of observations can be made from these performance results:

- There is a small but observable decrease in performance in moving from a UP kernel to an SMP

kernel, even with the SMP kernel compiled to run only on a single CPU. This is due to the increased overhead of locked instructions required for SMP operation; the degree to which this is significant depends on the workload.

- An "optimal" performance figure in these results is extrapolated by predicting linear improvement from the single-processor case: i.e., with the addition of each processor, we predict an improvement in performance based on each new CPU accomplishing the amount of work performed in the single processor case. This would require that the hardware and OS support increased parallelism without increased overhead, that the work performed by the application be improved linearly through added parallelism, and that the application itself be implemented to use available parallelism effectively. As suggested by the graph, speedups of less than 1 are quite common.

- The graph also includes a predicted speedup based on linear improvement at the rate measured when going from a one-CPU to a two-CPU configuration. In the graph, the move from two to three processors accomplishes close to predicted; however, when going from three to four processors, a marked decrease in performance occurs. One possible source of reduced performance is the saturation of resources shared by all processors, such as bus or memory resources. Another possible source of reduced performance is in application and operating system structure: that certain costs increase as the number of processors increases, such as TLB invalidation IPIs and data structure sizes, resulting in increased overhead as CPUs are added.

This benchmark illustrates a number of important principles, the most important of which is that multiprocessing is a complex tool that can hurt as much as it helps. Improving performance through parallelism requires awareness and proper utilization of parallelism at all layers of the system and application stack, as well as careful attention to the overheads of introducing parallelism.

2.1 What do we want from MP systems?

Multithreading and multiprocessing often requires significant changes in programming model in order to be used effectively. However, where these changes are exposed is an important consideration: the SMP model selected in earlier FreeBSD releases was selected on the basis of minimal changes to the current kernel model, and minimal complexity. Adopting new structures and programming approaches offers performance benefits with greater software changes. The same design choice applies to the APIs exposed to user applications: in both earlier work on FreeBSD's SMP implementation and the more recent SMPng work, the goal has been to maintain standard UNIX APIs and services for applications, rather than introducing entirely new application programming models.

In particular, the design choice has been made to offer a Single System Image (SSI), in which user processes are offered services consistent with executing on a single UNIX system. This design choice is often weakened in the creation of clustered computing systems with slower interconnects, and requires significant application adaptation. For the purposes of the SMPng Project, the goal has been to minimize the requirement for application modification while offering improved performance. In FreeBSD 5.x and later, multiprocessor parallelism is exposed to applications through the use of multiple processes or multiple threads.

2.2 What is shared in an MP System?

The principle behind current multiprocessing systems is that computations requiring large amounts of CPU resources often have data dependencies that make performing the computation with easy sharing between parts of the computation cost effective. Typical alternatives to multiprocessing in SMP systems include large scale cluster systems, in which computations are performed in parallel under the assumption that a computation can be broken up into many relatively independent parts. As such, multiprocessor computers are about providing facilities for the rapid sharing of data between parts of a computation, and are typically structured around shared memory and I/O channels.

Shared	Not Shared
System memory PCI buses I/O channels ...	CPU (register context, TLB, on-CPU cache, ...) Local APIC timer ...

This model is complicated by several sources of asymmetry. For example, recent Intel systems make use of Hyper-Threading (HTT), in which logical cores share a single physical core, including some computation resources and caches. Another source of asymmetry has to do with CPUs having inconsistent performance in accessing regions of system memory.

2.3 Symmetric Memory Access

The term "symmetric" in Symmetric Multiprocessing (SMP) refers to the uniformity of performance for memory access across CPUs. In SMP systems, all CPUs are able to access all available memory with roughly consistent performance. CPUs may maintain local caches, but when servicing a cache miss, no piece of memory is particularly favorable to access over any other piece of memory. Whether or not memory access

is symmetric is primarily a property of memory bus architecture: memory may be physically or topologically closer to one CPU than another.

Environments in which uniform memory access is not present are referred to as Non-Uniform Memory Access (NUMA) architectures. NUMA architectures become necessary as the number of processors increases beyond the capacity that a simple memory bus, such as a crossbar, can handle, or when the speed of the memory bus becomes a point of significant performance contention due to the increase in CPUs outstepping the performance of the memory that drives them. Strategies for making effective use of NUMA are necessarily more refined, as making appropriate use of memory topology is difficult.

Traditional two, four, and even eight processor Intel-based hardware has been almost entirely SMP-based. Until relatively recently, all low-end server and desktop systems were SMP, and NUMA was largely found in high-end multiprocessing systems, such as supercomputers. However, with the introduction of the AMD64 hardware platform, NUMA multiprocessor systems are now available on the desktop and server.

2.4 Inter-Processor Communication

As suggested earlier, communication between processors in multiprocessing systems is often based on the use of shared memory between those processors. For threaded applications, this may mean memory shared between threads executing on different CPUs; for other applications, it may mean explicitly set up shared memory regions or shared memory used to implement message passing. Issues of memory architecture, and in particular, memory consistency and cache behavior, are key to both correctness and performance in multiprocessing systems. Significant variations exist in how CPU and system architectures handle the ordering of memory write-back and cache consistency.

Also important in multiprocessor systems is the inter-process interrupt (IPI), which allows CPUs to generate notifications to other CPUs, such as to notify another CPU of the need to invalidate TLB entries for a shared region, or to request termination, signalling, or rescheduling of a thread executing on the remote CPU.

3 SMPng

Support for Symmetric Multi-Processing (SMP) has been a compile-time option for the FreeBSD kernel since FreeBSD 3.x. The pre-SMPng implementation is based on a single Giant lock that protects the entire kernel. This approach exposes parallelism to user applications, but does not require significant adaptation of the kernel to the multiprocessor environment as the kernel runs only on a single CPU at a time.

The Giant lock approach offers relative simplicity of implementation, as it maintains (with minimal modification) the synchronization model present in the uniprocessor kernel which is concerned largely with synchronizing between the kernel and interrupts operating on the same CPU. This permits user applications to exploit parallism to improve performance, but only in circumstances where the benefits of application parallelism outweight the costs of multiprocessor overhead, such as cache and lock contention.

The FreeBSD SMPng Project, begun in June, 2000, has been a long-running development project to modify the underlying kernel architecture to support increased threading and substitute a series of more fine-grained data locks for the single Giant lock. The goal of this work is to improve the scalability of the kernel on multiprocessor systems by reducing contention on the Giant lock, resulting in improved performance through kernel parallelism.

The first release of a kernel using the new kernel architecture was FreeBSD 5.0 which offered the removal of the Giant lock from a number of infrastructural components of the kernel as well as some IPC primitives. Successive FreeBSD 5.x releases removed Giant from additional parts of the kernel such as the network stack, device drivers, and the majority of remaining IPC primitives. The recently released FreeBSD 6.0 also removes Giant from the UFS file system and refines the SMPng architecture resulting in significantly improved performance.

SMPng was originally targetted solely at SMP class systems, but with the increased relevance of NUMA systems, investigation of less symmetric memory architectures has become more important for the FreeBSD SMPng Project. Figures 2 and 3 illustrate prototypical Quad Xeon (SMP) and Quad AMD64 (NUMA) hardware layouts relevant to the FreeBSD SMPng Project. Figures 4 and 5 illustrate Graphical Processing Unit (GPU) and cluster architectures not considered as part of this work.

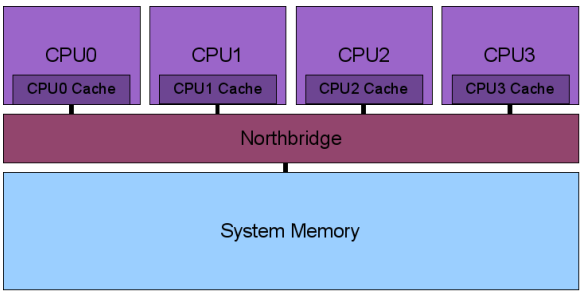

Figure 2: SMP Architecture: Quad-Processor Intel Xeon

3.1 Giant Locked Kernels

Support for multiprocessing in operating systems is either designed in from inception or retrofitted into an existing non-multiprocessing kernel. In the case of

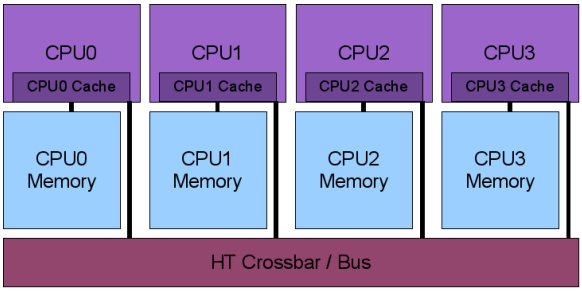

Figure 3: NUMA Architecture: Quad-Processor AMD64

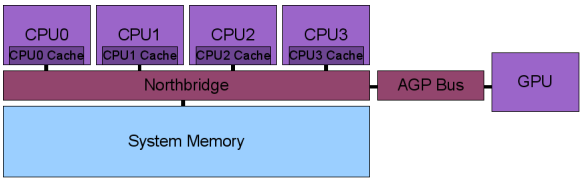

Figure 4: GPU Architecture: External Graphics Processor

most UNIX systems, multiprocessing support has been an after-thought, although the degree of redesign and reimplementation has varied significantly by product and version. The level of change has varied from low levels of change (using a Giant lock to maintain single-CPU synchronization properties and hence single-CPU kernel architecture), to complete reimplementation of the operating system based on a Mach microkernel and/or message passing.

The most straight forward approach to introducing multiprocessing in a uniprocessor operating system without performing a significant rewrite of the system is the Giant lock kernel approach. This approach maintains the property that most kernel synchronization occurs between the “top” and “bottom” halves – i.e., between system call driven operation and device driver interrupt handlers, and can be synchronized using critical sections or interrupt protection levels. In a Giant lock kernel, a single spinlock is placed around the entire kernel, in essence restoring the assumption that the kernel will execute on a single processor.

The FreeBSD 3.x and 4.x kernel series make use of a Giant spinlock which ensures mutual exclusion whenever the kernel is running. While the approach is simple, there are some important details: when a process attempts to enter the kernel, even the process scheduler, it must acquire the Giant lock. This results in lock contention when more than one processor tries to enter the kernel at a time (a common occurence with kernel-intensive workloads, such as network- or storage- centric loads common on FreeBSD). In the FreeBSD 4.x kernel, interrupts are able to preempt running kernel code. However, if an interrupt arrives on a CPU while the kernel is running on another CPU, it must be forwarded to the CPU where the kernel is running using an inter-processor interrupt (IPI).

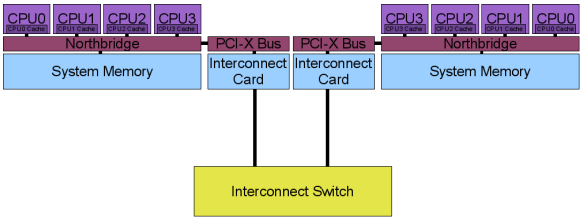

Figure 5: Cluster Architecture: Non-Uniform Memory via Complex Interconnect

3.2 Giant Contention

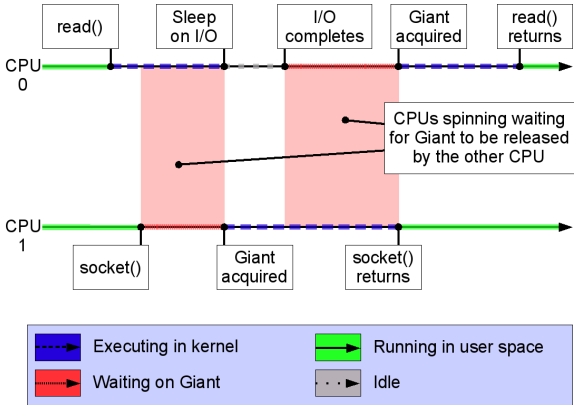

Figure 6: Impact of the Contention of a Giant Lock on Socket IPC

On systems with small numbers of CPUs, Giant Lock kernel contention is primarily visible when the workload includes large volumes of network traffic, inter-process communication (IPC), and file system activity. These are workloads in which the kernel must perform a significant amount of computation, resulting in increased delays and wasted CPU resources as other CPUs wait to enter the kernel, not to mention a failure to use available CPU resources to perform kernel work in parallel. On systems with larger numbers of CPUs, even relatively kernel non-intensive workloads can experience significant contention on the kernel lock, resulting in rapidly reduced scalability as the number of CPUs increases.

3.3 Finer Grained Locking

The primary goal of the SMPng Project has been to improve kernel performance on SMP systems by replacing the single Giant kernel lock with a series of smaller locks, each with more limited scope. This allows the kernel to usefully execute in parallel on multiple CPUs, potentially allowing more effective use of available

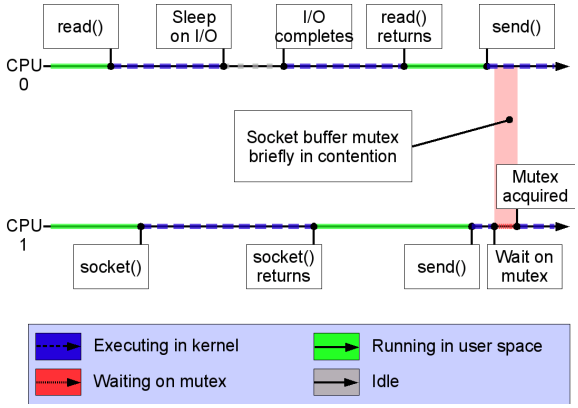

Figure 7: Reduced Lock Contention with Finer Grained Locking

CPUs by the kernel on multiprocessor systems. This also has the added benefit of avoiding wasting CPU as a result of Giant lock contention. This goal requires that the "interrupt level" synchronization model be replaced with one oriented around parallelism, not just preemption, resulting in a number of significant kernel architecture changes. For example, as disabling interrupts on a CPU will no longer prevent interrupt code from executing at the same time as system call code, interrupts must now make use of locks to synchronize with the remainder of the kernel. This in turn leads to a decision to execute interrupt handlers in full thread contexts (interrupt threads or ithreads).

This strategy has a number of serious risks:

- The new synchronization approaches must be more mature than the Giant lock approach, as introducing additional locks increases the risk of deadlocks. They must also address issues relating to concurrency and locking, such as priority inversion.
- Kernel synchronization must take into increased consideration the memory ordering properties of the hardware, as it has become a true multi-threaded program.
- Inter-processor synchronization typically relies on atomic operations and locked bus operations, which are expensive to perform; by adding additional locking requirements, overhead can add up quickly.
- Race conditions previously visible in the kernel only under high memory pressure are now far more likely to occur.

On the other hand, the architectural goals also have a number of significant benefits:

- In adopting synchronization primitives similar to those exposed by user threading libraries, such as mutexes and condition variables, developers familiar with process threading will be able to get started quickly with the kernel synchronization environment.
- By moving from a model with implicit synchronization properties (automatic acquisition and dropping of Giant) in 3.x/4.x to a model of explicit synchronization, the opportunity is provided for introducing much stronger assertions.
- Adopting a more threaded architecture, such as through the use of ithreads, increases the opportunities for parallelism in the kernel, allowing kernel computation to make better use of CPU resources.

The new SMPng kernel architecture facilitates the use of parallelism in the kernel, including the creation of multiprocessor data pipelines. By adopting an iterative approach to development, removing the dependency for Giant gradually over time, the system was left open to other development work continuing as the SMP implementation was improved.

The next few sections document the general implementation strategy followed during the SMPng Project, taking a "First make it work, then make it fast" strategy:

3.4 SMP Primitives

The first step in the SMPng Project was to introduce new locking primitives capable of handling more mature notions of synchronization, such as priority propagation to avoid priority inversion, and advanced lock monitoring and debugging facilities.

3.5 Scheduler Lock

Efforts to decompose the Giant lock typically begin with breaking out the scheduler lock from Giant, so that code executing without Giant will be able to make use of synchronization primitive that interact with the scheduler. The availability of scheduling facilities is fundamental to the implementation of most kernel services, as most kernel services rely on the the `tsleep()` and `wakeup()` mechanisms to manage long-running events.

Simultaneously, scheduler adaptations to improve scheduling on multiprocessor systems can be considered: IPI's between CPUs to allow the scheduler to communicate explicitly with the kernel running on other processors, scheduler affinity, per-CPU scheduler queues, etc. A variety of such techniques have been introduced via modifications to the existing 4BSD scheduler, and in a new MP-oriented scheduler, ULE [13].

3.6 Interrupt Threads

Next, interrupt handlers are moved into ithreads, allowing them to execute as normal kernel threads on various CPUs and use of kernel synchronization facilities. This has the added benefit that interrupt handlers now gain access to many more kernel service APIs, which previously often could not be invoked from interrupt context.

3.7 Infratructure Dependencies

With basic services such as synchronization and scheduling available without the Giant lock, additional important dependencies are then locked down. Among these are the kernel memory allocator and event timers. This includes both the general memory allocator and specific allocators such as the Mbuf allocator. In FreeBSD 6.x, a single Universal Memory Allocator (UMA) is used to allocate most system memory, rather than using a separate memory allocator for the network stack [12]. This allows the network stack to take advantage of the slab allocation and per-CPU cache facilities of UMA, make use of uniform memory statistics, and interact with global notions of kernel memory pressure.

3.8 Data-Based Locking

In most subsystems, data-based locking is used, combining locks and reference counts to protect the integrity of major data structures. Generally, we have started with coarser-grained locking to avoid introducing overhead without first determining that finer granularity helps with parallelism. Typically, the Virtual Memory system will be an early target as there is almost constant interaction between processes and virtual memory due to the need for multiprocessor operation to invalidate TLBs across processors. In this stage, locking will be applied based on data structures in a relatively naive fashion, in order to provide a first cut of Giant-free operation that can then be refined.

3.9 Slide Giant off Gradually

As Giant becomes unnecessary for subsystems or components and all of their dependencies, remove the Giant lock from covering those paths. This has the effect of reducing general contention on Giant, improving the performance of components still under the Giant lock.

3.10 Synchronization Refinment

Drive refinement of locking based on lock contention vs. lock overhead. Make use of facilities such as mutex profiling and hardware performance counters.

When balancing overhead and contention, there are a number of strategies that can be used. For example, replicating data structures across CPUs can prevent contention on locks, if the cost of maintaining replication is lower than the overhead the contention would cause. Statistics structures are a prime starting point, as they are frequently modified, so reducing writing to the same memory lines will avoid cache invalidations. Statistics can then be coalesced for presentation to the user: this approach is used for a variety of memory allocator statistics.

Likewise, synchronization with data structures accessed only from a specific CPU can often be performed using critical sections, which see lower overhead as they need only prevent preemption, not against parallelism. Another example of this approach is used in the UMA memory allocator: in 5.x, per-CPU caches are protected with mutexes due to accesses to the cache from other CPUs during certain operations. In FreeBSD 6.x, per-CPU caches are protected using critical sections, avoiding cross-CPU synchronization for per-CPU access.

Operating system literature documents a broad range of strategies for inter-CPU synchronization and data structure management, including lockless queues and Read-Copy-Update (RCU). As hardware architectures vary in both performance and semantics, optimization approaches may be specific to hardware configurations.

4 FreeBSD Network Stack

Having reviewed the FreeBSD SMPng kernel architecture, we will now explore how this architecture is implemented in the FreeBSD network stack. The FreeBSD network stack is one of the most complex components of the BSD kernel, consisting of over 400,000 lines of code excluding distributed file systems and device drivers, also large subsystems.

The network stack includes a number of service abstractions, such as network interfaces, communications sockets, event dispatch, remote procedure calls (RPCs), a protocol-independent route table, and user event models. Of particular importance is that data flows rapidly and continuously across many layers of abstraction and implementation, requiring careful consideration of the interactions between components. These software layers of abstraction often, but not always, map to layers in protocol construction.

4.1 Introduction to the Network Stack

The network stack contains many large and complex components:

- “mbuf” memory allocator
- Network interface abstraction, including a number of queueing discplines
- Device drivers implementing network interfaces

- Protocol-independent routing and user event model
- Link layer protocols – Ethernet, FDDI, SLIP, PPP, ATM, etc.
- Network layer protocols – UNIX Domain Sockets, IPv4, IPv6, IPSEC, IPX, EtherTalk/AppleTalk, etc.
- Socket and socket buffer IPC primitives
- Netgraph extension framework
- Many netgraph nodes implementing a broad range of services

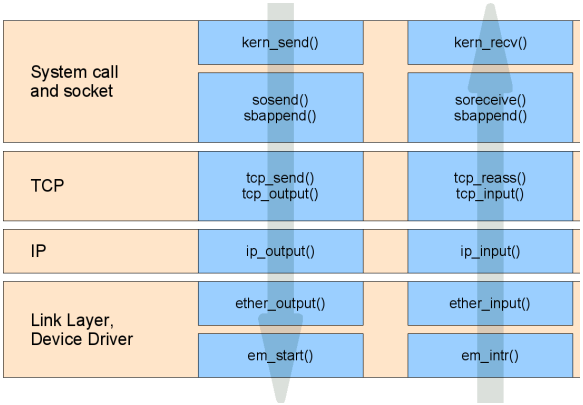

Figure 8: FreeBSD Network Stack: Common Dataflow

4.2 Network Stack Concerns

Introducing parallelism and preemption introduces a number of additional concerns:

- Per-packet costs: network stacks may process millions of packets per second – small costs add up quickly if per-packet.
- Ordering: packet ordering must be maintained with respect to flow, as protocols such as TCP are very sensitive to minor misordering.
- Optimizations may conflict: optimizing for latency may damage throughput, or optimizing for local data transfer may damage routing performance.
- When using locking, ordering is important – lock order prevents deadlock, but passage through layers in the network stack is often bi-directional.
- Some amount of parallelism is available by virtue of the current network stack architecture – introducing new parallelism is necessary in order to improve utilization of MP resources, but depends on introducing additional threads, which can increase overhead.

These concerns are discussed in detail as the locking strategy is described.

4.3 Locking Strategy

The SMPng locking strategy for the network stack generally revolves around data-based locking. Using this strategy involves identifying objects and assigning locks to them; the granularity of locking is important as each lock operation introduces overhead. Useful rules of thumb include:

- Don't use finer-grained locking than is required by the UNIX API: for example, parallel send and receive on the same socket has benefit, but parallel send on a stream socket has poorly defined semantics, so not permitting parallelism can avoid unnecessarily complexity.
- Lock references to in-flight packets, not packets themselves. For example, lock queues of packets used to hand off between threads, but use only simple pointe references within a thread.
- Use optimistic concurrency techniques to avoid additional lock overhead – i.e., where it is safe, test a value that can be read atomically without a lock, then only acquire the lock if work is required that may have stronger consistency requirements.
- Avoid operations that may sleep, which can result in multiple acquires of mutexes, as well as unwinding of locks. In general, the network stack is able to tolerate failures through packet loss under low memory situations, so take advantage of this property to lower overhead.

Also important is consideration of layering: as objects may be represented at different layers in the stack by different data structures, decisions must be made both with respect to whether layers share locks, and if they don't share locks, what order locks may be acquired in. Control flow moves both "up" and "down" the stack, as packets are processed in both input and output paths, meaning that if locks are simply acquired as processing occurs, lock order cycles will be introuced as processing occurs in two directions.

The following general strategies have been adopted in the first pass implementation of fine-grained locking for the network stack:

- Low level facilities, such as network memory allocation, route event notification, packet queues, and dispatch queues, generally have leaf locks so that they can be called from any level of the stack including device drivers.

- Protocol locks generally fall before device driver locks in the lock order, so that device drivers may be invoked without releasing stack locks.

- Protocol locks generally fall before socket locks in the lock order, so that protocols can interact with sockets without releasing protocol locks.

Just as asynchronous packet dispatch to the netisr in earlier BSD network stacks allows avoiding of layer recursion and reentrance, it can also be used to avoid lock order issues with an MPSAFE network stack. This technique is used, for example, to avoid recursing into socket buffer code when a routing event notification occurs as the result of a socket event, and prevents deadlock by eliminating the "hold and wait" part of the deadlock recipe. The netisr will processed queued routing socket events asynchonously, delivering them to waiting sockets.

4.4 Global Locks

Global locks are used in two circumstances: where global data structures are referenced, or where data structures are accessed sufficiently infrequently that coalescing locks does not increase contention. The following global locks are a sampling of those added to the network stack to protect global data structures:

Lock	Description
ifnet_lock	Global network interface list
bpf_mtx	Global BPF descriptor list
bridge_list_mtx	Global bridge configuration
if_cloners_mtx	Cloning network interface data
disc_mtx, faith_mtx gif_mtx, gre_mtx, lo_mtx ppp_softc_list_mtx stf_mtx, tapmtx, tun_mtx, ifv_mtx	Synthetic interface lists
pfil_global_lock	Packet filter registration
rawcb_mtx, ddp_list_mtx, igmp_mtx, tcbinfo_mtx, udbinfo_mtx ipxpcb_list_mtx natm_mtx, rtsock_mtx	Per-protocol control block lists
hch_mtx	TCP host cache
ipqlock, ip6qlock	IPv4 and IPv6 fragment queues
aarptab_mtx, nd6_mtx	Link layer address resolution
in_multi_mtx	IPv4 multicast address lists
mfc_mtx, vif_mt mrouter_mtx	IPv4 multicast routing
sptree_lock, sahtree_lock regtree_lock, acq_lock spacq_lock	IPSEC

The following is a sampling of locks have been added to data structures allocated dynamically in the network stack:

Structure	Field	Description
ifnet	if_addr_mtx	Interface address lists
	if_afdata_mtx	Network protocol data
	if_snd.ifq_mtx	Interface send queue
bpf_d	bd_mtx	BPF descriptor
bpf_if	bif_mtx	BPF interface attachment
ifaddr	ifa_mtx	Interface address
socket	so_rcv.sb_mtx	Socket, socket receive buffer
	so_snd.sb_mtx	Socket send buffer
ng_queue	q_mtx	Netgraph node queue
ddpcb	ddp_mtx	netatalk PCB
inpcb	inp_mtx	netinet PCB
ipxpcb	ipxp_mtx	netipx PCB

4.5 Network Stack Parallelism

Parallelism in the FreeBSD kernel is expressed in terms of threads, as they represent both execution and scheduling contexts. In order for one task to occur

in parallel with another task, it must be performed in a different thread from that task. In order for the FreeBSD kernel to make effective use of multiprocessing, work must therefore occur in multiple threads.

A fair amount of parallelism in the network stack is simply from conversion of the existing BSD network stack model to the SMPng architecture:

- Each user thread has an assigned kernel thread for the duration of a system call or fault, which performs work directly associated with the system call or fault. In the transmit direction, the user thread is responsible for executing socket, protocol, and interface portions of the transmit code, which includes the cost of copying data in and out of user space. In the receive direction, the user thread is responsible for primarily for executing the socket code, along with copying data in and out of user space; under some circumstances, calls into the protocol and interface layers may also occur.

- Each interrupt request (IRQ) is assigned its own ithread, which is used to execute the handlers of interrupt sources signaled by that interrupt. As long as devices are assigned different interrupts, their handlers can execute in parallel. By default, this will include execution of the link layer interface code, but a dispatch to the netisr thread for higher stack layers.

- A number of kernel tasks are performed by shared or assigned worker threads, such as callouts and timers running from a shared callout thread, several task queue processing threads for various subsystems, and the netisr thread in the network stack, which is primarily responsible for the protocol layer processing of in-bound packets.

While multithreading is required in order to experience parallelism, multithreading also comes with significant costs, including:

- Cost of context switching, which may include the cost of cache flushes when a thread migrates from one CPU to another and the cost of entering the scheduler.

- Cost of synchronizing access to data between threads: typically, a locked or otherwise synchronized data structure or work queue.

Figure 9 illustrates the UDP send path, and possible parallelism between the user thread sending down the stack layers, and ithread receiving acknowledgements from the network stack in order to recycle packet memory.

Figure 10 illustrates the UDP receive path, and possible parallelism between the user thread interacting with the socket layer, netisr processing the IP and UDP layers, and the ithread receiving packets from the network interface and processing the link layer.

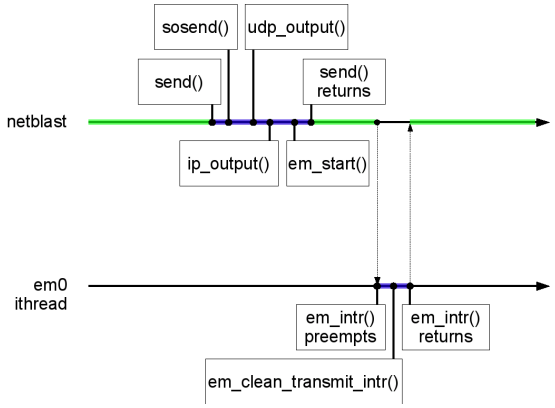

Figure 9: Parallelism in the UDP send path

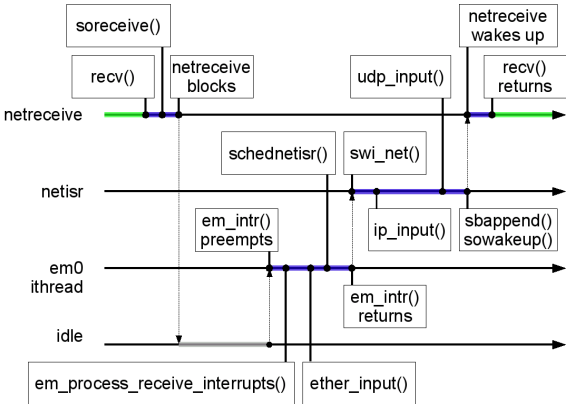

Figure 10: Parallelism in the UDP receive path

5 MP Programming Challenges

Multiprocessing is intended to improve performance by introducing greater CPU resources. However, unlike a number of other hardware-based performance improvements, such as increasing clock speed or cache size, multiprocessor programming requires fundamental changes in programming model. In this section, we consider two important concerns in multiprocessing and multithreading programming and their relationship to the network stack: deadlock, and event serialization.

5.1 Deadlock

Deadlock is a principal concern of systems with synchronous waiting for ownership of locks on objects. Deadlocks occur when two or more simultaneous threads of execution (typically kernel threads) meet the following four conditions:

- Attempt to simultaneously access more than one resource which can be owned, but not shared (mutual exclusion).

- Hold and wait: threads acquire and hold resources in an order.

- No preemption: once acquired, a resource cannot be preempted without agreement of the thread.
- Circular wait: threads acquire and attempt to acquire resources such that a cycle is formed, resulting in indefinite wait.

The above description is intentionally phrased in terms of resources rather than locks, as deadlock can occur in more general circumstances. For example, low memory deadlock is another type of widely experienced deadlock.

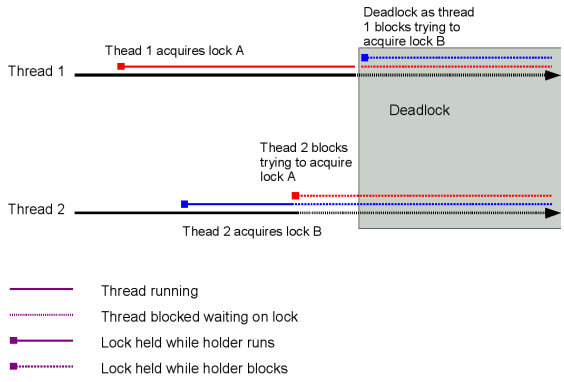

Figure 11: Deadlock: The Deadly Embrace

There is extensive research literature on deadlock avoidance, detection, and management; however, one of the most straight forward and easiest ways to avoid deadlock is simply to follow a strict lock order. Lock orders indicate that, whenever any two locks can be acquired as the same time, they will always be acquired in the same order. This breaks lock order cycles, and thus prevents deadlock, and is a widely used technique.

In order to assist in documenting lock orders and prevent cycles, BSDI created WITNESS, a run-time lock order verifier, which was refined by the FreeBSD Project to support additional lock types and assertion types. WITNESS can be used as both a tool to document a specification for lock interaction through a hard-coded lock order list, and to dynamically discover lock order relationships through run-time monitoring. WITNESS maintains a graph of lock acquisition orders, and provides run-time warnings (along with stack traces and other debugging information), when declared or discovered lock orders are directly or indirectly violated.

WITNESS and other lock-related invariants also detect and report a variety of other lock usage, such as acquiring sleepable locks while holding mutexes or in critical sections, or kernel threads returning to user space while holding locks.

FreeBSD also makes use of other deadlock avoidance techniques, including the use of optimistic concurrency techniques in which attempts are made to acquire locks in the wrong order, and then if this would

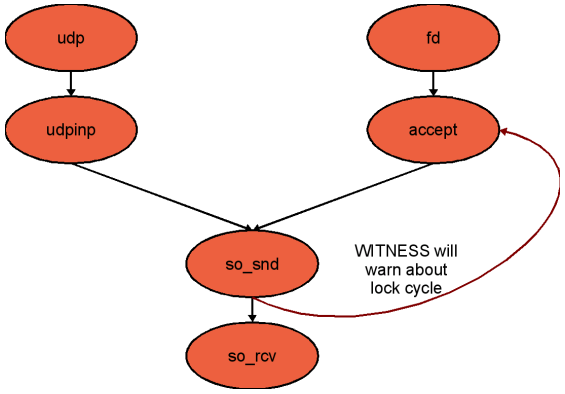

Figure 12: Lock order verification with WITNESS: Cycles in the lock graph are detected and reported using graph algorithms.

result in a deadlock, falling back on the defined order. Another technique used in the kernel is the use of guard locks, acquired before acquiring locks on multiple objects with no defined lock order between them. By serializing attempts at simultaneous acquire behind a lock, the lock order of the objects becomes defined only when they are acquired at once, and no conflicting lock order can be simultenously defined, preventing deadlock.

5.2 Event Serialization

In a singlethreaded programming environment, the order of events is largely a property of programmed order, so maintaining the order associated with a data structure or the processing of data is essentially a problem of ordering of the program. In a multithreaded programming environment, concurrency in code execution means that parallel threads of execution may execute at different rates, and that any ordering of events must occur as a result of planning. If events must occur in a specific order, programmers must either execute them in a single thread (which serialized events into programmed order), or synchronization primitives and communication primitives must be used so that ordering is either maintained during computation, or restored during post-processing after the computation.

This is particularly relevant to the implementation of the network stack, in which discrete units of work, typically represented by packets, are processed in a number of threads. The order of packets can have a significant impact on performance, and so maintaining necessary orders is critically important. For example, out-of-order delivery of TCP packets can result in TCP perceiving packet loss, resulting in a fast (and unnecessary) retransmit of data. Packet ordering must typically be maintained with respect to its flow, where the granularity of the flow might include a stream of packets sourced from a particular network interface, packets between two hosts, or packets in a particular connec-

tion.

In the single-threaded FreeBSD 4.x receive path, ordering is maintained throughout through the use of last-in, first out (LIFO) queues between threads, effectively serializing processing. A single netisr thread processes all inbound packets from the link layer to the network layer. Naively introducing multithreading into a network stack without careful consideration of ordering might be performed by simply introducing additional in-bound packet worker threads (netisrs). Figure 13 illustrates that this might result in misordering of packets in a simple packet forwarding scenario: two packets might be dispatched in one order to different worker threads, and then be forwarded in reversed order due to scheduling of the worker threads.

In FreeBSD 6.x, two modes of operation are documented for packet processing dispatch: queued serialized dispatch with a single netisr thread, or direct dispatch of packet processing from the calling context. In direct dispatch mode, context switches are reduced by performing additional packet processing in the originating thread for a packet, rather than passing all packets to a single worker thread – for example, in the interrupt thread for a network interface driver. This implements a weaker ordering by not committing to an ordered queue, but maintains sufficient ordering. Weakened packet ordering improves the opportunities for parallelism by permitting more concurrency in packet processing, and is an active area of on-going work in the SMPng Project. One downside to direct dispatch in the ithread is reduced opportunity for parallelism, as in-bound processing is now split between two threads: the ithread and the receiving user thread, but not the netisr.

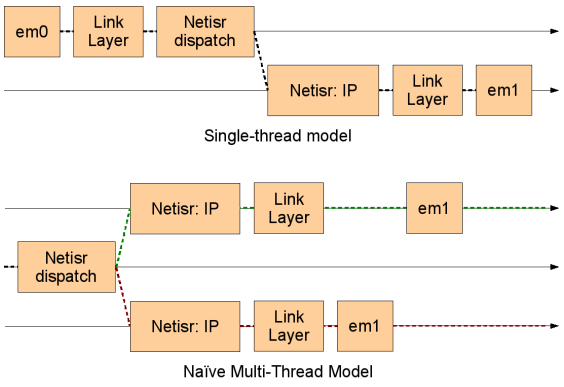

Figure 13: Singlethreaded and naive multithreaded packet processing, in which sufficient ordering is no longer maintained

6 Status of the SMPng Network Stack

As of FreeBSD 6.0, the vast majority of network stack code is run without the Giant lock in the default configuration. This includes most link layer network device drivers and services, such as gigabit ethernet drivers and ethernet bridging, ARP, the routing table, IPv4 input, filtering, and forwarding, FAST_IPSEC, IP multicast, protocol code such as TCP and UDP, and the socket layer. In addition, many non-IP protocols, such as AppleTalk and IPX are also MPSAFE.

Some areas of the network stack continue to require Giant, and can generally be put in two categories:

- Code that requires Giant for correctness (perhaps due to interacting with another part of the kernel that requires Giant), but can be executed with Giant but an otherwise Giant-free network stack.

- Code that requires Giant for correctness, but due to lock orders and construction of the network stack, requires holding Giant over the entire network stack when used.

In the former category lie the KAME IPSEC implementation and ISDN implementation. Giant is required over the entire stack because these code paths can be entered in a variety of situations where other locks (such as socket locks) can already be held, preventing Giant from being acquired when it is discovered the non-MPSAFE code will be entered. Instead, Giant must be acquired in advance unconditionally.

Other areas of the system also continue to require the Giant lock, such as a number of legacy ISA network device drivers and portions of the in-bound IPv6 stack. In both cases, Giant will be conditionally acquired in an asynchronous execution context before invoking the non-MPSAFE code. A number of consumers of the network stack also remain non-MPSAFE, such as the Netware and SMB/CIFS file systems. With the FreeBSD 6.0 VFS now able to support MPSAFE file systems, locking down of these file systems and removal of Giant is now possible; in the mean time, they execute primarily in VFS consumer threads that will already have acquired Giant, and not synchronously from network stack threads that run without Giant, permitting the network stack to operate without Giant.

Components operating with Giant for compatibility continue to see higher lock contention and latency due to asynchronous execution. It is hoped that remaining network stack device drivers and protocols requiring Giant will be made MPSAFE during the 6.x branch lifetime.

7 Related Work

Research and development of multiprocessor systems has been active for over forty years, and has been performed by hundreds of vendors for thousands of products. As such, this section primarily points the reader at a few particularly useful books and sources relevant

to the SMP work on FreeBSD, rather than attempting to capture the scope of prior work in this area.

Curt Schimmel provides a detailed description of multiprocessor synchronization techniques and the application in UNIX system design in *UNIX Systems for Modern Architectures*, including detailed discussion of design trade-offs [14].

Uresh Vahalia provides general discussion of advanced operating system kernel architectures across a number of UNIX systems in *UNIX Internals* [15].

The FreeBSD SMPng architecture has been significantly impacted by the design and implementation strategies of the Solaris operating system, discussed in *Solaris Internals* by Jim Mauro and Richard McDougall [9].

The Design and Implementation of 4.4BSD by Kirk McKusick, et al, describes earlier BSD kernel architecture, and particularly synchronization, in great detail, and makes a useful comparison with *The Design and Implementation of the FreeBSD Operating Systems*, which describes the FreeBSD 5.x architecture [10] [11].

A good general source of information on multiprocessing and multithreading programming techniques, both for userspace and kernel design, are the design and implementation papers relating to the Mach operating system project at Carnegie Mellon [3].

8 Future Work

Remaining work on the SMPng network stack falls primarily into the following areas:

- Complete removal of Giant requirement from all remaining network stack code (device drivers, IPv6 in-bound path, KAME IPSEC).
- Continue to explore improving performance and reducing overhead through refining data structures, lock strategy, and lock granularity, as well as further exploring synchronization models.
- Continue to explore improving performance through analyzing cache footprint, interprocessor cache interactions, and so on.
- Continue to explore how to further introduce useful parallelism into network processing, such as additional parallel execution opportunities in the transmit path and in network interface polling.
- Continue to explore how to reduce latency in processing through reducing queued dispatch, such as via network interface direct dispatch of of the protocol stack.

It is expected that the results of this further work will appear in future FreeBSD 6.x and 7.x releases.

9 Acknowledgments

The SMPng Project has been running for five years now, and has had literally hundreds of contributors, whose contributions to this work have been invaluable. As a result, not all contributors can be acknowledged in the space available, and the list is limited to a subset who have worked actively on the network stack parts of the project.

The author greatfully acknowledges the contributions of BSDI, who contributed prototype reference source code for parts of a finer-grained implementation of the BSD kernel, and specifically, network stack, as well as their early development support for the SMPng Project as a whole. The author also wishes to recognize the significant design, source code development, and testing contributions of the following people without whom the Netperf project would not have been possible: John Baldwin, Antoine Brodin, Jake Burkholder, Brooks Davis, Pawel Dawidek, Julian Elischer, Don Lewis, Brian Feldman, Andrew Gallatin, John-Mark Gurney, Paul Holes, Peter Holm, Jeffrey Hsu, Roman Kurakin, Max Laier, Nate Lawson, Sam Leffler, Jonathan Lemon, Don Lewis, Scott Long, Warner Losh, Rick Macklem, Ed Maste, Bosko Milekic, George Neville-Neil, Andre Oppermann, Alfred Perlstein, Luigi Rizzo, Jeff Roberson, Mike Silberback, Bruce Simpson, Gleb Smirnoff, Mohan Srinivasan, Mike Tancsa, David Xu, Jennifer Yang, and Bjoern Zeeb.

Financial support for portions of the Netperf Project and test hardware was provided by the FreeBSD foundation. The Netperf Cluster, a remotely managed cluster of multiprocessor test systems for use in the Netperf project, has been organized and managed by Sentex Communications, with hardware contributions from FreeBSD Systems, Sentex Communications, IronPort Systems, and George Neville-Neil. Substantial additional testing facilities and assistance have been provided by the Internet Systems Consortium (ISC), Sandvine, Inc., and Yahoo!, Inc. The author particularly wishes to acknowledge Kris Kennaway for his extended hours spent in testing and debugging SMPng and Netperf Project work as part of the FreeBSD package building cluster.

10 Conclusion

The FreeBSD SMPng Project has now been running for five years, and has transformed the architecture of the FreeBSD kernel. The resulting architecture makes extensive use of threading, fine-grained and explicit synchronization, and offers a foundation for a broad range of future work in exploiting new hardware platforms, such as NUMA. The FreeBSD SMPng network stack permits the parallel execution of the network stack on multiple processors, as well as a fully preemptive network stack. In this paper, we've presented

background on MP architectures, an introduction to the SMPng Project and recent work on SMP in FreeBSD, and the design principles and challenges in adapting the network stack to this architecture.

11 Availability

The results of the SMPng Project began appearing in FreeBSD releases beginning with FreeBSD 5.0. The FreeBSD 6.x branch reflects further completion of SMPng tasks, and significant refinement of the work in FreeBSD 5.x. General information on the FreeBSD operating system, as well as releases of FreeBSD may be found on the FreeBSD web page:

`http://www.FreeBSD.org/`

Further information about SMPng may be found at:

`http://www.FreeBSD.org/smp/`

The Netperf project web page may be found at:

`http://www.FreeBSD.org/projects/netperf/`

References

[1] BALDWIN, J. Locking in the Multithreaded FreeBSD Kernel. In *Proceedings of BSDCon'02* (February 2002), USENIX.

[2] BOURKE, T. Super smack. http://vegan.net/tony/supersmack/.

[3] CARNEGIE MELLON UNIVERSITY. The Mach Project Home Page. http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html.

[4] FREEBSD PROJECT. FreeBSD home page. http://www.FreeBSD.org/.

[5] FREEBSD PROJECT. FreeBSD Netperf Project. http://www.FreeBSD.org/project/netperf/.

[6] FREEBSD PROJECT. FreeBSD SMP Project. http://www.freebsd.org/smp/.

[7] HSU, J. Reasoning about SMP in FreeBSD. In *Proceedings of BSDCon'03* (September 2003), USENIX.

[8] LEHEY, G. Improving the FreeBSD SMP Implementation. In *Proceedings of FREENIX Track: 2001 USENIX Annual Technical Conference* (June 2001), USENIX.

[9] MAURO, J., AND MCDOUGALL, R. Solaris Internals: Core Kernel Architecture, 2001.

[10] MCKUSICK, M., BOSTIC, K., KARELS, M., AND QUARTERMAN, J. The Design and Implementation of the 4.4BSD Operating System, 1996.

[11] MCKUSICK, M., AND NEVILLE-NEIL, G. The Design and Implementation of the FreeBSD Operating System, 2005.

[12] MILEKIC, B. Network Buffer Allocation in the FreeBSD Operating System. http://www.bsdcan.org/2004/papers/NetworkBufferAllocation.pdf.

[13] ROBERSON, J. ULE: A Modern Scheduler for FreeBSD. In *Proceedings of BSDCon'03* (September 2003), USENIX.

[14] SCHIMMEL, C. UNIX Systems for Modern Architectures, 1994.

[15] VAHALIA, U. UNIX Internals: The New Frontiers, 1996.