Chip multiprocessors have introduced a new dimension in scaling for application developers, operating system designers, and deployment specialists.

EXTREME

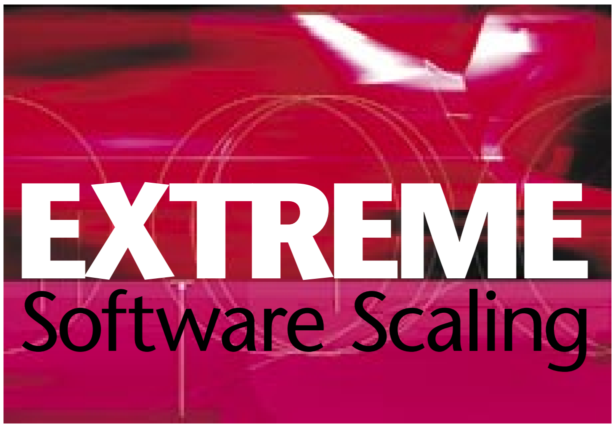
Software Scaling

The advent of SMP (symmetric multiprocessing) added a new degree of scalability to computer systems. Rather than deriving additional performance from an incrementally faster microprocessor, an SMP system leverages multiple processors to obtain large gains in total system performance. Parallelism in software allows multiple jobs to execute concurrently on the system, increasing system throughput accordingly. Given sufficient software parallelism, these systems have proved to scale to several hundred processors.

More recently, a similar phenomenon is occurring at the chip level. Rather than pursue diminishing returns by increasing individual processor performance, manufacturers are producing chips with multiple processor cores on a single die. (See "The Future of Microprocessors," by Kunle Olukotun and Lance Hammond, in this issue.) For example, the AMD Opteron[1] processor now uses two entire processor cores per die, providing almost double the performance of a single core chip. The Sun Niagara[2] processor, shown in figure

RICHARD MCDOUGALL, SUN MICROSYSTEMS

EXTREME Software Scaling



1, uses eight cores per die, where each core is further multiplexed with four hardware threads each.

These new CMPs (chip multiprocessors) are bringing what was once a large multiprocessor system down to the chip level. A low-end four-chip dual-core Opteron machine presents itself to software as an eight-processor system, and in the case of the Sun Niagara processor with eight cores and four threads per core, a single chip presents itself to software as a 32-processor system. As a result, the ability of system and application software to exploit multiple processors or threads simultaneously is becoming more important than ever. As CMP hardware progresses, software is required to scale accordingly to fully exploit the parallelism of the chip.

Thus, bringing this degree of parallelism down to the chip level represents a significant change to the way we think about scaling. Since the cost of a CMP system is close to that of recent low-end uniprocessor systems, it's inevitable that even the cheapest desktops and servers will be highly threaded. Techniques used to scale application and system software on large enterprise-level SMP systems will now frequently be leveraged to provide scalability even for single-chip systems. We need to consider the effects of the change in the degree of scaling on the way we architect applications, on which operating system we choose, and on the techniques we use to deploy applications—even at the low end.

CMP: JUST A COST-EFFECTIVE SMP?

A simplistic view of a CMP system is that it appears to software as an SMP system with the number of processors equal to the number of threads in the chip, each with slightly reduced processing capability. Since each hardware thread is sharing the resources of a single processor core, each thread has some fraction of the core's overall performance. Thus, an eight-core chip with 32 hardware threads running at 1 GHz may be somewhat crudely approximated as an SMP system with thirty-two 250-MHz processors. The effect on software is often a subtle trade-off in per-thread latency for a significant increase of throughput. For a throughput-oriented workload with many concurrent requests (such as a Web server), the marginal increase in response time is virtually negligible, but the increase in system throughput is an order of magnitude over a non-CMP processor of the same clock speed.

There are, however, more subtle differences between a CMP system and an SMP system. If threads or cores within a CMP pro-

A Chip Multiprocessor

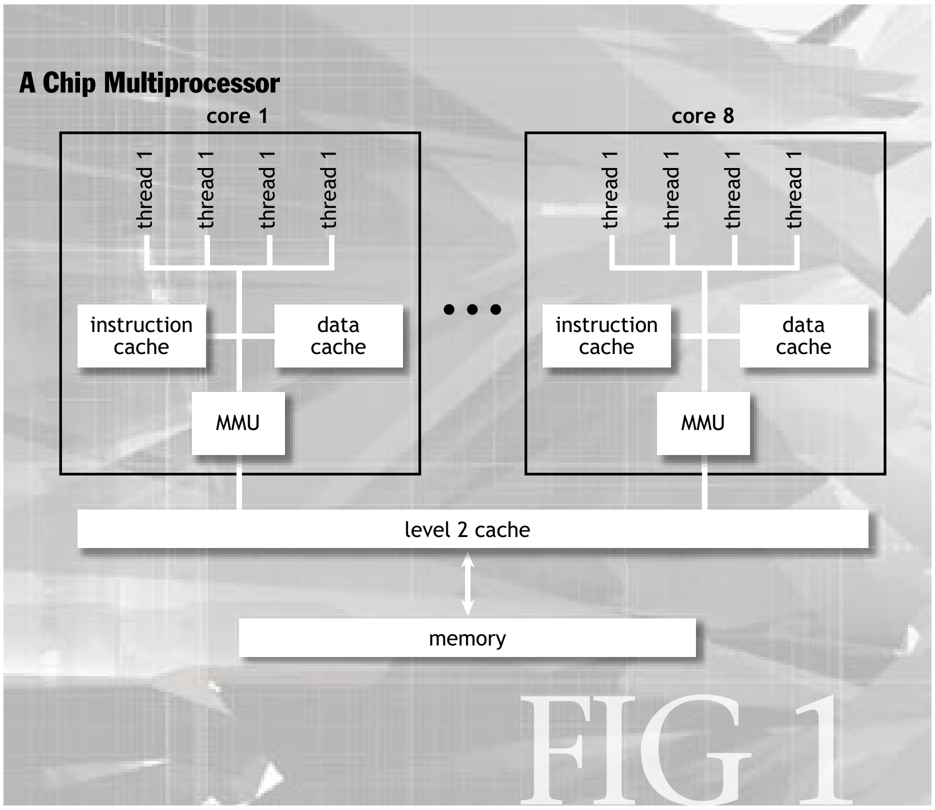


FIG 1

cessor share important resources, then some threads may impact the performance of other threads. For example, when multiple threads share a single core and therefore share first-level memory caches, the performance of a given thread may vary depending on what the other threads, of the same core, are doing with the first thread's data in the cache. Yet, in another similar case, a thread may actually gain if the other threads are constructively sharing the cache, since useful data may be brought into the cache by threads other than the first. This is covered in more detail later as we explore some of the potential operating system optimizations.

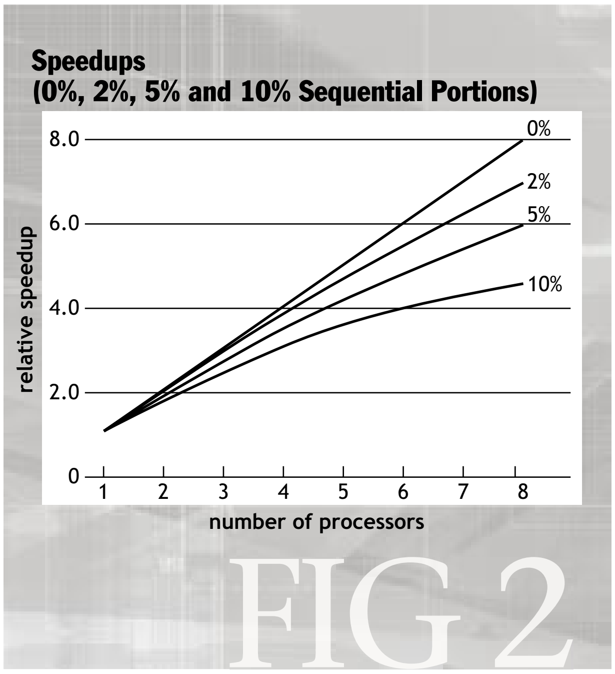


FIG 2

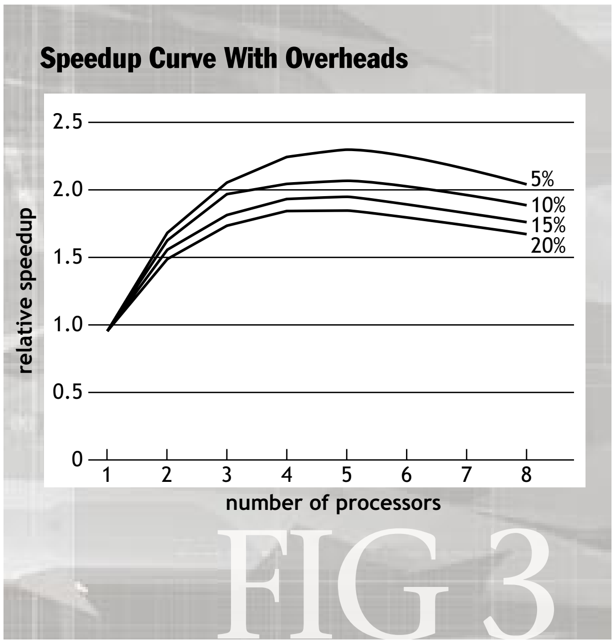


FIG 3

SCALING THE SOFTWARE

The performance of system software ideally scales proportionally with the number of processors in the system. There are, however, factors that limit the speedup.

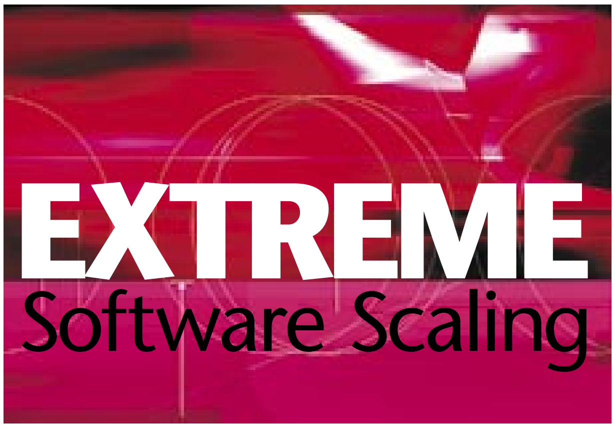
Amdahl's law[3] defines scalability as the speedup of a parallel algorithm, effectively limited by the number of operations that must be performed sequentially (i.e., its *serial fraction),* as shown in figure 2. If 10 percent of a parallel program involves serial code, the maximum speedup that can be attained is three, using four processors (75 percent of linear), reducing to only 4.75 when the processor count increases to eight (only 59 percent of linear). Amdahl's law tells us that the serial fraction places a severe constraint on the speedup as the number of processors increase.

In addition, software typically incurs overhead as a result of communication and distribution of work to multiple processors. This results in a scaling curve where the performance peaks and then begins to degrade (see figure 3).

Since most operating systems and applications contain a certain amount of sequential code, a possible conclusion of Amdahl's law is that it is not cost effective to build systems with large numbers of processors because sufficient speedup will never be produced. Over the past decade, however, the focus has been on reducing the serial fraction within hardware architectures, operating systems, middleware, and application software. Today, it is possible to scale system software and applications on the order of 100 processors on an SMP system. Figure 4 shows the results for a series of scaling benchmarks that were performed using database workloads on a large SMP configuration. These application benchmarks were performed on a single-system image by measuring throughput as the number of processors was increased.

INTRA- OR INTER-MACHINE SCALE?

Software scalability for these large SMP machines has historically been obtained through rigorous focus on intra-machine *scalability* within one large instance of the application within a single operating system. A good example is a one-tier enterprise application such as SAP. The original version of SAP used a single and large monolithic application server. The application instance



obtains its parallelism from the many concurrent requests from users. Providing there are no major serialization points between the users, the application will naturally scale. The focus on scaling these applications has been to remove these serialization points within the applications.

More recently, because of the economics of low-end systems, the focus has been on leveraging inter-machine scaling, using low-cost commodity one- to two-processor servers. Some applications can be made to scale without requiring large, expensive SMP systems by running multiple instances in parallel on separate one- to two-processor systems, resulting in good overall throughput. Applications can be designed to scale this way by moving all shared state to a shared back-end service, like a database. Many one- to two-processor systems are configured as mid-tier application servers, communicating to an intra-machine scaled database system. The shift in focus to one- to two-processor hardware has removed much of the pressure to design intra-machine scalability into the software.

The compelling features of CMP—low power, extreme density, and high throughput—match this space well, mandating a revised focus on intra-machine scalability.

IMPACT OF CMP ON APPLICATION DEVELOPERS

The most significant impact for application developers is the requirement to scale. The minimum scaling requirement has been raised from 1-4 processors to 32 today, and will likely increase again in the near future.

BUILDING SCALABLE APPLICATIONS

Engineering scalable code is challenging, but the performance wins are huge. The data in the scaling curves for Oracle and DB2 in figure 4 show the rewards, from a great deal of performance tuning to optimization for scaling. According to Amdahl's law, scaling software requires minimization of the serial fraction of the workload. In many commercial systems, natural parallelism comes from the many concurrent users of the system.

The simple first-order scaling bottlenecks (those with a large serial fraction) typically come from contention for shared resources, such as:

- **Networks or interconnects.** Bandwidth limitations on interconnects between portions of the system—for example, an ingress network on the Web servers, tier-1 and -2 networks for SQL traffic, or a SAN (storage area network).
- **CPU/Memory.** Queuing for CPU or waiting for page faults as a result of resource starvation.

Scaling of Throughput-Oriented Workloads on SMP Hardware

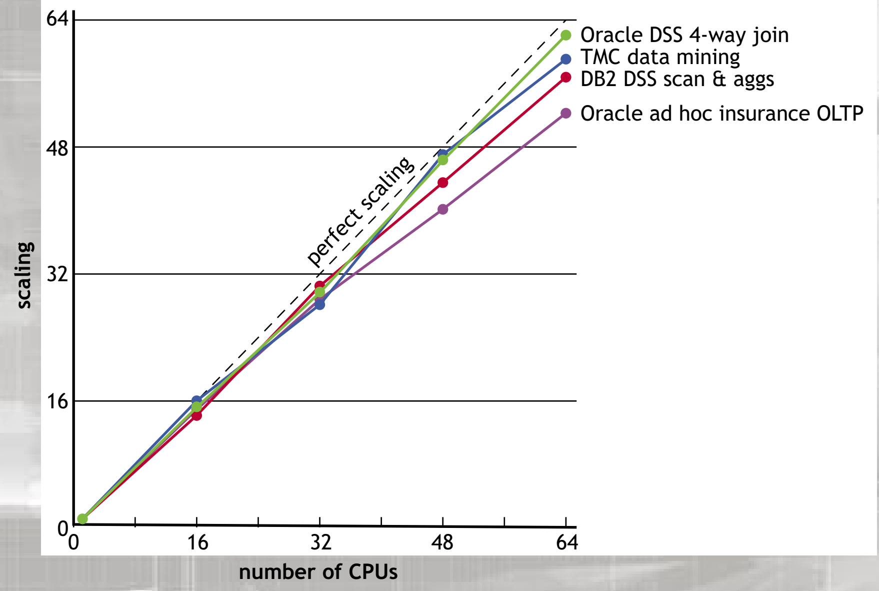


FIG 4

- **I/O throughput.** Insufficient capacity for disk I/O operations or bandwidth.

The more interesting problems result from intrinsic application design. These problems manifest from serial operations within the application or the operating environment. They are often much harder to identify without good observation tools, because rather than showing up as an easy-to-detect overloaded resource (such as out of CPU), they often exhibit growing amounts of idle resource as load is increased.

Here's a common example. We were recently asked to help with a scaling problem on a large online e-commerce system. The application consisted of thousands of users performing payment transactions from a Web application. As load increased, the latency became unacceptable. The application was running on a large SMP system and database, both of which were known to scale well. There was no clear indicator of where in the system the problem occurred. As load was increased, the system CPU resources became *more* idle. It turned out that there was a single table at the center of all the updates, and the locking strategy for the table became the significant serial fraction of the workload. User transactions were simply waiting for updates to the table. The solution was to break up the table so that concurrent inserts could occur, thus reducing the serial fraction and increasing scalability.

For CMP, we need to pay attention to what might limit scaling within one application instance, since we now need to scale in the order of tens of threads, increasing to the order of 100 in the near future.

WRITING SCALABLE LOW-LEVEL CODE

Many middleware applications (such as databases, application servers, or transaction systems) require special attention to scale. Here are a few of the common techniques that may serve as a general guideline.

Scalable algorithms. Many algorithms become less efficient as the size of the problem set increases. For example, an algorithm that searches for an object using a linear list will increase the amount of CPU required as the size of the list increases, potentially at a super-linear rate. Selecting good algorithms that optimize for the common case is of key importance.

Locking. Locking strategies have significant impact on scalability. As concurrency increases, the number of threads attempting to lock an object or region increases, resulting in compounding contention as the lock becomes "hotter." In modern systems, an optimal approach is to provide fine-grained locking using a lock per object where possible. There are also several approaches to making the reader side of code lock-free at the expense of some memory waste or increased writer-side cost.

Cache line sharing. Multiprocessor and CMP systems use hardware coherency algorithms to keep data consistent between different pipelines. This can have a significant effect on scaling. For example, a latency penalty may result if one processor updates a memory object within its cache, which is also accessed from another processor. The cache location will be invalidated because of the cache coherency hardware protocol, which ensures only one version of the data exists. In a CMP system, multiple threads typically access a single first-level cache; thus, colocating data that will be accessed within a single core may be appropriate.

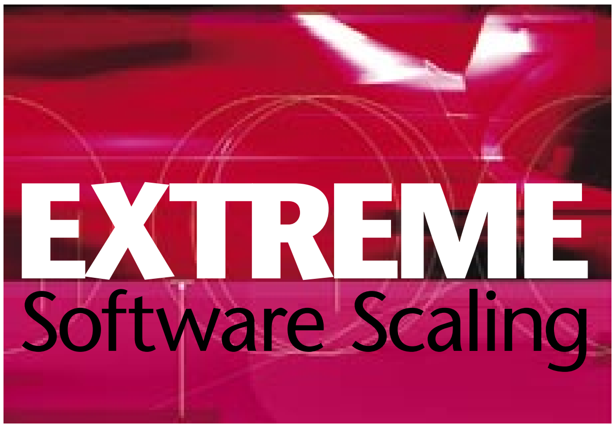
Pools of worker threads. A good approach for concurrency is to use a pool of worker threads; a general-purpose, multithreaded engine can be used to process an aggregate set of work events. Using this model, an application gives discrete units of work to the engine and lets the engine process them in parallel. The worker pool provides a flexible mechanism to balance the work events across multiple processors or hardware threads. The operating system can automatically tune the concurrency of the application to meet the topology of the underlying hardware architecture.

Memory allocators. Memory allocators pose a significant problem to scaling. Almost every code needs to allocate and free data structures, and typically does so via a central system-provided memory allocator. Unfortunately, very few memory allocators scale well. The few that do include the open source Hoard, Solaris 10's libumem slab allocator, and MicroQuill's SmartHeap. It's worth paying attention to more than one dimension of scalability: different allocators have different properties in light of the nature of allocation/deallocation requests.

CONDUCT SCALABILITY EXPERIMENTS EARLY AND OFTEN

Time has shown that the most efficient way of driving out scaling issues from an application is to perform scaling studies. Given the infinite space in which optimizations can be made, it is important to follow a methodology to prioritize the most important issues.

Modeling techniques can be used to mathematically predict response times and potential scaling bottlenecks in complex systems. They are often used for predicting the performance of hardware, to assist with design trade-off analysis. Modeling software, however, requires intimate knowledge of the software algorithms, code paths, and system service latencies. The time taken to construct



a model and validate all assumptions is often at odds with running scaling tests.

A well-designed set of scaling experiments is key to understanding the performance characteristics of an application, and with proper observation instrumentation, it is easy to pinpoint key issues. Scalability prediction and analysis should be done as early as possible in the development cycle. It's often much harder to retrofit scalability improvements to an existing architecture. Consider scalability as part of the application architecture and design.

Key items to include in scalability experiments are:

- **Throughput versus number of threads/processors.** Does the throughput scale close to linearly as the amount of resource applied increases?
- **Throughput versus resource consumed (i.e., CPU, network I/O, and disk I/O) per transaction.** Does the amount of resource consumed per unit of work increase as scale increases?
- **Latency versus throughput.** Does the latency of a transaction increase as the throughput of a system increases? A system that provides linear throughput scalability might not be useful in the real world if the transaction response times are too long.
- **Statistics.** Measure code path length in both number of instructions and cycles.

OBSERVATION TOOLS ARE THE PRIMARY MEANS TO SCALABLE SOFTWARE

Effective tools are the most significant factor in improving application scalability. Being able to quickly identify a root cause of a scaling issue is paramount. The objective of looking for scaling issues is to easily pinpoint the most significant sources of serialization.

The tools should help identify what type of issue is causing the serialization—the two classic cases being starvation resulting from escalating resource requirements as load increases, and increasing idle time as load increases. Ideally, the tools should help identify the source of the scaling issue rather than merely pointing to the object of contention. This helps with identifying not only what the contention point is, but also perhaps some offending code that may be overutilizing a resource. Often, once the source is identified, many obvious optimizations become apparent.

Consider tools that can do the following:

- **Locate key sources of wait time.** What are the contended resources, which one is causing the resource utilization, and how much effect is the contention having on overall performance?
- **Identify hot synchronization locks.** How much wall clock and CPU time is serialized in locking objects, and which code is responsible?
- **Identify nonscalable algorithms.** Which functions or classes become more expensive as the scale of the application increases?
- **Make it clear where the problem lies.** This is done either in the application code, which you can affect, or by pointing to a contention point in a vendor-supplied middleware or operating system. Even though the contention point may lie in a vendor code, it may result from how that code is being called, which can be affected by optimizing the higher-level code.

CMT AND SOFTWARE LICENSING

Another impact of the hardware architecture's scaling characteristics is on software licensing. Application developers often use the number of processors in the system to determine the price of the software. The number of processors has been a convenient measure for software licensing, primarily because of the close correlation between the costs of the hardware platform and the number of processors. By using a license fee indexed by the number of processors, the software vendor can charge a roughly proportional fee for software.

This is, however, based on old assumptions that are no longer true. First of all, an operating system on a CMT platform reports one virtual processor for every thread in the chip, resulting in a very expensive software license for a low-end system. Software vendors have been scrambling to adjust for the latest two-core CMT systems, some opting for one license fee per core, and others for each physical chip. Licensing by core unfairly increases software licenses per dollar unit of hardware.

In the short term, operating system vendors are providing enhancements to report the number of cores and physical processors in the system, but there is an urgent need for a more appropriate (and fair) solution. It is likely that a throughput-based license fee that uses standard benchmarks will be pursued. This would allow license fees to be charged in accordance with the actual processing power of the platform. Such a scheme would allow software licenses to scale when more advanced virtualization schemes, which divide up processors into subprocessor portions, are used (such as priority-based resource partitioning). These schemes are becoming more commonplace as utility computing and server consolidation become more popular. The opportunity for operating system vendors is to choose a uniform metric that can be measured and reported, based on the actual use by an application.

IMPACT OF CMP FOR OPERATING SYSTEMS

The challenge for the operating system is twofold: providing scalable system services to the applications it hosts, and providing a scalable programming environment that facilitates easy development of parallel programs.

CMP ENHANCEMENTS FOR OPERATING SYSTEMS

An SMP-capable operating system kernel works quite well on CMP hardware. Since each core or hardware thread in a chip has an entire set of registers, they appear to software as individual CPUs. An unchanged operating system will simply implement one logical processor for every hardware thread in the chip. Software threads will be scheduled onto each hardware thread just as in an SMP system, with equal weighting according to the operating system kernel's scheduling policy (see figure 5).

Basic changes to optimize for CMT processors will include elimination of any busy wait loops. For example, the idle loop is typically implemented as a busy spin that checks a run queue looking for more work to do. When multiple hardware threads share a single core, the idle loop running on one thread will have a detrimental effect on other threads sharing the core's pipeline. In this example, leveraging the hardware's ability to park a thread when there is no work to do would be more effective.

Further operating system enhancements will likely be pursued to optimize for the subtle differences of CMPs. For example, with knowledge of the processor architecture and some information about the behavior of the software, the scheduler may be able to optimize the placement of software threads onto specific hardware threads. In the case of a CMP architecture with multiple hardware threads sharing a core, first-level cache, and TLB (translation look-aside buffer), there may be a benefit if software threads with similar memory access patterns (constructive) are colocated on the same core, and those with destructive patterns are separated onto different cores.

OPERATING SYSTEM SCALING

The challenge with scaling operating system services has historically been the shared state between instances of the services. For example, consider a global process table that

Software Threads Scheduling on CMP Cores/Threads

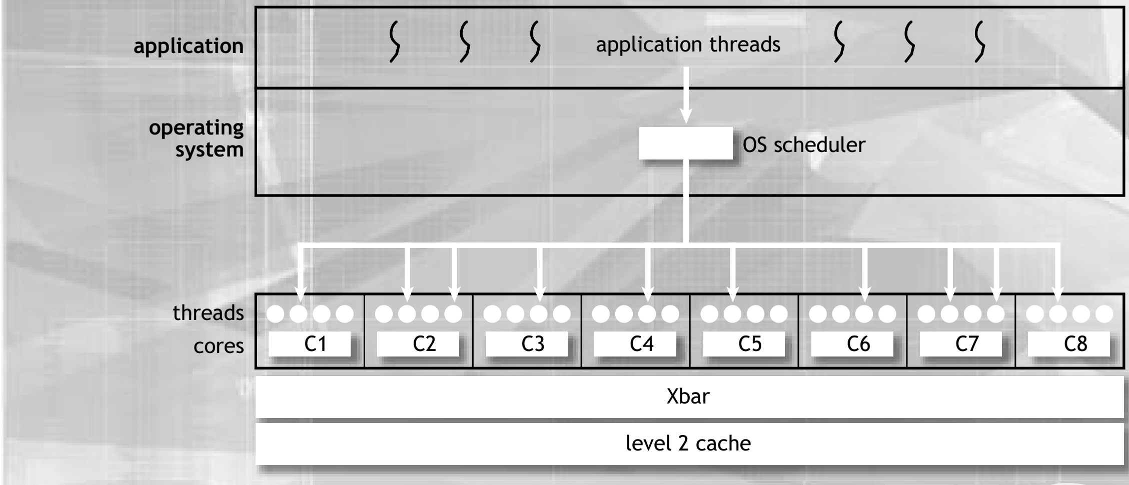


FIG 5

EXTREME Software Scaling

needs to be accessed and updated by any program wanting to start a new process. In a multiprocessor system, synchronization techniques must be used to mitigate race conditions when two or more threads attempt to update the process table at the same time.

The common techniques require serialization around either the code that accesses these structures or the data structures themselves. Early attempts to port Unix to SMP hardware were crude—they were typically retrofits of existing operating system codes with simple, coarse-grained serialization. For example, the first SMP Unix systems used a slightly modified implementation with a single global lock around the operating system kernel to serialize all requests to its data structures. Early versions of SunOS (1.x), Linux (2.2), and FreeBSD (4.x) kernels used this approach. Although easy to implement, this approach helps scalability only for applications that seldom use operating system services. Applications that were entirely compute-intensive showed good scalability, but those that used a significant amount of operating system services saw serialization yielding little or no scalability beyond one processor.

In contrast, successful operating system scaling is achieved by minimizing contention, restricting serialization to only fine-grained portions of data structures. In this way, the operating system can execute code within the same region concurrently on multiple processors, serializing only momentarily while accessing shared data structures. This approach does, however, require substantial architectural change to the operating system and in some cases a ground-up redesign focused on scalability.

A well-designed operating system allows high levels of concurrency through its operating system services. In particular, applications invoking system services through libraries, memory allocators, and other system services must be able to execute in parallel even if they access shared facilities. For example, multiple programs should be able to allocate memory concurrently without serializing. Other areas that are critical to scalability include parallel access to shared hardware (e.g., I/O) and the networking subsystem.

SCALING ENHANCEMENTS IN FREEBSD

FreeBSD has seen a significant amount of scaling effort, starting with 5.x kernels.[4] Architectural changes include new kernel memory allocators, synchronization routines, the move to ithreads, and the removal of the global kernel lock from activities such as process scheduling, virtual memory, the virtual file system, the UFS (Unix file system), the networking stack, and several common forms of inter-process communication. The scaling work in FreeBSD has successfully improved scaling (estimates suggest to the order of 12 processors).

SCALING ENHANCEMENTS IN LINUX

Scaling was greatly improved in Linux 2.2 kernels by breaking up the global kernel lock. It is said to scale on the order of two to four processors. Linux 2.4 scaling was improved to eight to 16 by introducing much finer-grained locking in the scheduler and I/O subsystem. This improved the scaling of many items, including interrupts and I/O. Later efforts in Linux kernels focused on scaling the scheduler for larger numbers of processes and improving concurrency through the networking subsystem.

SCALING ENHANCEMENTS IN SOLARIS

The Solaris operating system is built around the concept of concurrency, and serialization is restricted to very small and critical parts of data structures. The operating system is designed around the notion that execution contexts are individual software threads, which are scheduled and executed in parallel where possible.

Replacing the original Unix memory allocators with the Slab[5] and Vmem[6] allocators led to significant scalability gains. These provide consistent in-time allocations as the object set sizes grow, and they pay special attention to avoid locking by providing per-processor pools of memory that allow allocations and deallocations to occur without having to access global structures.

Scalable I/O is achieved by allowing requesting threads to execute concurrently even within the same device driver, and further by processing interrupts from hardware devices as separate threads, allowing scaling of interrupt handling.[7]

In some cases, there are requirements for high levels of concurrent access to data structures. For example, performance statistics for I/O devices require updates from potentially thousands of concurrent operations. To mitigate contention around these types of structures, statistics are kept on a per-processor basis and then aggregated when required. This allows concurrent access to updates, requiring serialization only when the statistics are read.

The Solaris networking code was rearchitected to eliminate the majority of the global data structures by introducing a per-connection vertical perimeter.[8] This allows the TCP/IP implementation to operate in near-lockless mode within a single connection, requiring locking only when global events such as routing changes occur.

Integrated observation tools are key to optimizing scaling issues. Facilities for observing sources of locking contention on systems with live workloads have been critical to making improvements in important areas. More recently, Dtrace, perhaps one of the more revolutionary approaches to performance optimization, allows dynamic instrumentation of C and Java code.[9] It can quickly pinpoint sources of contention from the top of the application stack through the operating system.

These types of techniques allow the Solaris kernel to scale to thousands of threads, up to 1 million I/Os per second, and several hundred physical processors. Conveniently, this scaling work can be leveraged for CMP systems. Techniques such as those described here, which are vital for large SMP scaling, are now required even for entry-level CMP systems. Within the next five years, expect to see CMP hardware scaling to as many as 512 processor threads per system, pushing the requirements of operating system scaling past the extreme end of that realized today.

OPERATING SYSTEM UTILIZATION METRICS

The reporting of processor utilization on systems with multithreaded cores poses a challenge. In a single-core chip, throughput often increases proportionally with processor utilization. In a multithreaded chip, there is much greater opportunity for sharing of resources between hardware threads, and therefore a nonlinear relationship exists between throughput and the actual utilization of a processor. As a result, calculation of "headroom" based on reported processor utilization may no longer be accurate.

For example, a processor core with two threads (such as an Intel Xeon) presents itself to the operating system as two separate processors. If a software thread fully uses one of the threads and the other is completely idle, the processor will appear 50 percent busy and be reported as such by the operating system. Running two of these threads on the processor may often yield only a 10 percent throughput increase on Xeon architecture, but since both threads are utilized, it will report as 100 percent busy. So this system now reports 50 percent utilization when it's at 90 percent of its maximum throughput.

This effect will vary depending on how many of the resources are shared by hardware threads within the processor, and ultimately will need some redefinition of the meaning of system utilization metrics, together with some new facilities for reporting. The impact on capacity planning methodology will also need to be considered.

LEVERAGING VIRTUALIZATION FOR PARALLELISM

So far we have examined how to find ways to use the many hardware threads available with CMTs by scaling individual applications or operating systems. Another

Different Types of Virtualization

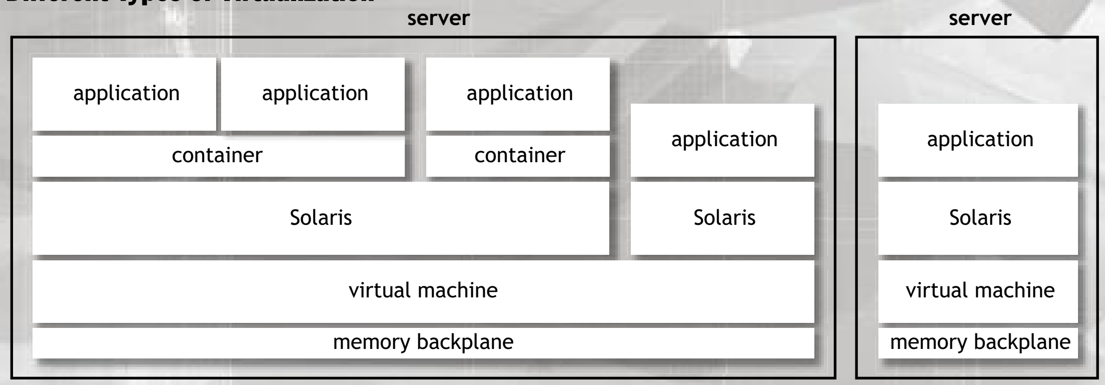
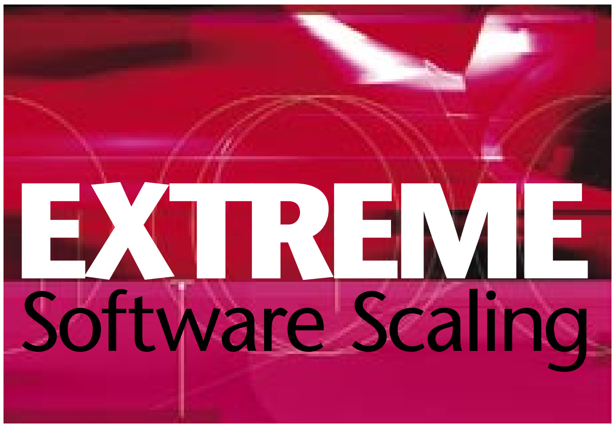


FIG 6



way to use these resources effectively is to run multiple nonscalable applications or even several unoptimized operating systems at once, using techniques such as operating system or server virtualization.

These facilities typically allow multiple instances of an application to be consolidated onto a single server (see figure 6).

For example, the Solaris Container facility allows multiple applications to reside within a single operating system instance. In such an environment, you can leverage the cumulative concurrency as applications are added. By adding two Web servers, each of which has concurrency of 16 threads, you can potentially increase the system-wide concurrency to 32 threads. This side effect presents a useful mechanism that allows you to deploy applications with limited scalability in a manner that can exploit the full concurrency of a CMP system.

Another relevant virtualization technology is the virtual machine environment, which allows multiple operating system instances to run on a single hardware platform. Examples of virtual machine technologies are VMware and Xen. These environments allow consolidation of applications and operating systems on a single system, which provides a mechanism to deploy even nonscalable operating systems on CMP architectures, albeit with a little more complexity.

CMP REQUIRES A RETHINKING BY DEVELOPERS

The introduction of CMP systems represents a significant opportunity to scale systems in a new dimension. The most significant impact of CMP systems is that the degree of scaling is being increased by an order of magnitude: what was a low-end one- to two-processor entry-level system should now be viewed as a 16- to 32-way system, and soon even midrange systems will be scaling to several hundred ways.

For application developers, this represents a new or revised focus on intra-machine scalability within applications and a rethinking of how software license fees are calculated. For operating system developers, scalability to hundreds of ways is going to be a requirement. For deployment practitioners, CMP represents a new way to scale applications and will require consideration in the systems we architect, the way we tune, and the techniques we use for capacity planning. Q

REFERENCES

1. AMD Opteron Processor; http://www.amd.com.
2. Kongetira, P., Aingaran, K., and Olukotun, K. 2005. Niagara: a 32-way multithreaded SPARC processor. *IEEE Micro* 25 (2): 21–29.
3. Amdahl, G. M. 1967. Validity of the single-processor approach to achieving large-scale computing capabilities. *Proceedings of AFIPS Conference:* 483-485.
4. The FreeBSD SMP Project; http://www.freebsd.org/smp/.
5. Bonwick, J. 1994. The Slab allocator: an object-caching kernel memory allocator. Sun Microsystems.
6. Bonwick, J., and Adams, J. 2001. Magazines and Vmem: extending the Slab allocator to many CPUs and arbitrary resources. Sun Microsystems and California Institute of Technology.
7. Kleiman, S., and Eykholt, J. 1995. Interrupts as threads. *ACM Sigops Operating Systems Review* 29 (2): 21-26.
8. Tripathi, S. 2005. Solaris OS network performance. Sun White Paper (February).
9. Cantrill, B. M., Shapiro, M. W., Leventhal, A.H. 2004. Dynamic instrumentation of production systems. *Usenix Proceedings*.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

RICHARD McDOUGALL, had he lived 100 years ago, would have had the hood open on the first four-stroke internal combustion gasoline-powered vehicle, exploring new techniques for making improvements. He would be looking for simple ways to solve complex problems and helping pioneering owners understand how the technology worked to get the most from their new experience. These days, McDougall uses technology to satisfy his curiosity. He is a Distinguished Engineer at Sun Microsystems, specializing in operating systems technology and system performance. McDougall is the author of *Solaris Internals* (Prentice Hall, 2000; second edition, 2005), and *Resource Management* (Prentice Hall, 1999).