Scalability : The Software Problem

Jonathan Appavoo, Volkmar Uhlig, and Dilma da Silva

IBM T. J. Watson Research Center
{jappavoo,vuhlig,dilmasilva}@us.ibm.com

1 Introduction

For the past several years we have studied how to scale general-purpose system software. System software is unique, from a software perspective, in its requirement to support and enable parallelism rather than exploiting it to improve its own performance. System software must ensure good overall system utilization and high degree of parallelism for those applications that demand it. To do this, system software must: (*i*) utilize the characteristics of the hardware to exploit parallelism in general-purpose workloads, and (*ii*) facilitate concurrent applications, which includes providing models and mechanisms for applications to exploit the parallelism available in the hardware.

Based on previous experience in constructing scalable hardware and associated operating systems [1, 5, 6, 9–12] we designed a software structure aimed at better suiting the underlying structure of a scalable SMMP (Shared-Memory Multi-Processor) architecture: when additional workload is presented to the system, additional hardware resources can be efficiently utilized to service the load increase. Controlling and managing communication is ultimately the key to scalability therefore we focused on locality — *avoiding inter-processor communication* — by carefully limiting shared data access.

We adopted four key design principles:

1. Avoid off-processor communication in the implementation of communication and scheduling facilities.

2. Use a runtime memory structure that scales with hardware resources and workload demands in a way to to ensure efficient and controlled memory consumption and communication.

3. Avoids communication overheads when there is no logical sharing in the workload — avoiding shared meta-structures and false sharing.

4. Despite targeting SMMPs, use distributed data structures to control communication when there is a demand for shared-resource access.

"Sharing" lies at the heart of the problem. By its very nature, sharing introduces barriers to scalability and it can come from three main sources: First, sharing can be intrinsic to the workload. For example, a workload may utilize a single shared file to log all activity of multiple processes, or a multi-threaded application may use a shared data array across multiple processors. Second, sharing also arises from the data structures and algorithms employed in the design and implementation of the system software. For example, an operating system may be designed to manage all physical memory as a single shared pool and implement this management using shared data structures. Third, sharing can occur in the physical design and protocols utilized by the hardware. For example, a system utilizing a single memory bus and snooping-based cache coherence protocol requires shared-bus arbitration and global communication for memory access, even for data located in independent memory modules.

Achieving scalable performance requires minimizing all forms of sharing, which is also referred to as maximizing locality. On SMMPs, locality refers to the degree to which locks are contended and data — including the locks themselves — are shared amongst threads running on different processors. The less contention on locks and the less data is shared, the higher the locality. Maximizing locality on SMMPs is critical, because even minute amounts of (possibly false) sharing can have a profound negative impact on performance and scalability [3, 5].

2 Concurrency $\neq$ Scalability

A common conception is to assume that software that is implemented to be concurrent is scalable. This, however, is not true. Traditionally, software is considered to be concurrent if it uses fine-grain synchronization. The intent is to construct software which is correct in the face of concurrent execution, but utilizes mutual exclusion in a manner that critical sections have small lock-hold times. This approach ignores the inherently non-scalable runtime structure that such software has with respect to memory access and, hence, communication.

Scalable End-to-End Performance In order not hinder overall system or individual application performance, it is critical for system software to reflect the parallelism of the workloads and individual applications. This point is often overlooked. Smith alludes to the requirements of individual applications, noting that the tension be-

tween protection and performance is particularly salient and difficult in a parallel system and that the parallelism in one protection domain must be reflected in another [8]. In other words, to ensure that a parallel application within one protection domain can realize its potential performance, all services in the system domains that the concurrent application depends on must be provided in an equally parallel fashion. It is worth noting that this is true not only for individual applications, but also for all applications forming the current workload on a parallel system: the demands of all concurrently executing applications must be satisfied with equal parallelism in order to ensure good overall system performance.

3 Virtualization and Scalability

As stated above, to achieve good scalability it is necessary that all software executed to satisfy a desired task scale with respect to the underlying communication capacity and resources of the system. As such, the locality of resource access must be reflected across all protection boundaries and software layers. If any of the layers in the software stack does not scale, the overall system is likely to not scale.

Each protection layer poses a scalability challenge: strict separation by a protection boundary introduces a semantic boundary about dependence and independence of software components. The protection boundary restricts the flow of semantic information on the degree of concurrency, frequency of access, locality, and structuring of refined resources provided in bulk by the lower layers. While a hypervisor mitigates management of low-level resources such as memory pages and processor time slices, an operating system refines such resources into buffer caches, short- and long-lived code and data pages, thread-control blocks, linked lists, inode entries, etc. Each of these higher-level constructs have specific usage scenarios and thus access and communication patterns [4].

Only by careful design of the interfaces we can bridge the semantic gap between the software layers. When managing bulk resources such as memory pages, we explicitly maintain locality information — degree of parallelism and processor sets — with the resource [9]. Additionally, we allow for dynamic adaptation of the synchronization primitives at runtime. For hypervisors, dynamic adaptability is a key requirement since virtual machines are typically long-lived objects and resources managed through the hypervisor get re-assigned for other purposes.

4 Operating System and General-Purpose System Software Scalability

For scalable software design we employed three concepts as our core building blocks:

Events: All activity in the system software can be viewed as requests that induce short-lived events. The event is created at the beginning of a system request and does not block for IO. The event satisfies the request by traversing and potentially manipulating system data structures. All long-lived work is programmed by continuations.

Locality Domains: Virtual processors and memory pools are associated with physical processors and memory modules. Events are bound to virtual processors and memory pools.

Object Decomposition: We have observed that there is a similarity between the runtime structure resulting from an object-oriented design and the runtime structure of scalable system software. Object orientation can result in a runtime where each unique software resource instance has a unique identifier and separate memory allocation, and is dynamically created. These properties can be leveraged to construct software that has a natural model for scaling with increases in workload and hardware resources.

These concepts are further accompanied by the following four mechanisms. These mechanisms were implemented and evaluated in the K42 research operating sytem [2, 7].

Protected Procedure Call : A client-server model is used in K42 to represent and service the demand or load on the system. A system service is placed within an appropriate protection domain (address space) and clients make requests to the service via a Protected Procedure Call (*PPC*). Like the Tornado PPC [5], a call from a client to a server acts like a procedure call that crosses from one address space to another and then back with the following properties:

1. client requests are always serviced on their local processor,
2. clients and servers shared the processor in a manner similar to handoff scheduling, and
3. there are as many threads of control in the server as the client request.

Locality Aware Dynamic Memory Allocation : Dynamic memory allocation is extensively used in the construction of K42 services to ensure that memory resources grow with demand both in size and

location. The K42 memory allocators employ per-processor pools to ensure that memory allocations can be localized to the processors on which the allocations will be accessed to service the requests of those processors.

Object Decomposition : K42 uses an object-based software architecture to construct it services. The model encourages the developers to construct services that scale with demand by design. A service is structured as a set of dynamic interconnected object instances that are lazily constructed to represent the resources that a unique set of requests require. For example, the mapping of a portion of a processes address space to a file would result in the construction of several objects unique to that process, file and mapping. As such page faults by that process to the address range would result in execution of methods only on the objects constructed for that process, mapping and file. Other mappings would result in other independent objects being constructed. The object architecture acts as a natural way of leveraging the locality aware dynamic memory allocation. Objects are created on demand on the processors on which the requests they are required for occur and as such consume memory local to those processors and are accessed with good locality.

Clustered Objects : A unique SMMP object model is used in K42, specifically designed to further encourage and enable the construction of scalable software services. Each object a developer constructs can be optimized with respect to the concurrent access patterns expected. For example, a multi-thread application such as a web server or a parallel scientific application will cause concurrent access to the Process object instance, representing the application, as its threads on different processors page fault. As such the Clustered Object model provides a standard way for implementing the Process Object using distribution, replication and partition in its data structures and algorithms, in a manner that is on demand in nature. The Clustered Object model and infrastructure also incorporates standard object oriented features such as inheritance, polymorphism, and specialization. Additionally it incorporates runtime features such as multi-threaded safe hot-swapping and semi-automatic garbage collection.

Software designed for locality on an SMMP uses partitioning, distribution, and replication of data to control concurrent access. Doing so provides fine-grain control over memory accesses and control over the associated communication. For example, using a distributed implementation of a performance counter, where each processor is assigned its own local padded sub-counter, ensures that updates to the counter result only in local communication.

It is worth noting that there are two forms of communication associated with a memory access: (*i*) utilization of the interconnect fabric to bring the data from storage into the processor's caching system, and (*ii*) utilization of the interconnect for the associated cache coherency messages. On large-scale machines, with complex distributed interconnection networks, non-uniform memory access (NUMA) effects increase the benefit of localizing memory accesses by avoiding remote delays. Additionally, localizing memory accesses restricts communication to local portions of a complex hierarchical interconnect, and thereby avoids the use (and thus congestion) of the global portions of the memory interconnect.

Even small machines, with flat, high-capacity interconnects that do not suffer congestion delays, benefit from localizing memory accesses. The use of large multi-layer caches to mitigate the disparities between processor frequencies and main memory response times results in NUMA-like behavior. Considerable delays result when a cache miss occurs. Partitioning, distribution, and replication help avoid cache misses associated with shared data access by reducing the use of shared data on critical paths. For example, a distributed performance counter with all sub-counters residing on separate cache lines eliminates cache coherency actions (invalidations to ensure consistency) on updates, thus making communication delays associated with cache misses less likely. There is a tradeoff, however: when the value of the counter is needed, the values of the individual sub-counters must be gathered and additional cache misses must be suffered.

Implementations utilizing partitioning, distribution, and replication require programmers to explicitly control and express how the data structures will grow with respect to memory consumption as additional load is observed. For example, consider a distributed table used to record processes, with an implementation that utilizes a table per processor that records just the processes created on that processor. In such an approach the OS programmer explicitly manages the required memory on a per-processor basis. This implementation will naturally adapt to the size of the machine on which it is running, and on large-scale machines with distributed memory banks, data growth can occur in a balanced fashion.

5 Summary

Scaling software to a large number of processor contexts requires *managing* and *minimizing* inter-processor communication. While this is a well-known wisdom for distributed systems with high latency and high overhead

communication fabrics, the same is true for shared memory multiprocessor systems. With the drastically increasing number of processor cores, software needs to explicitly manage and minimize sharing in order to scale. Over the last years and a number of projects we investigated structuring methods within and across software layers and formalized a number of design principles and methodologies that enabled us to scale legacy software.

References

[1] Jonathan Appavoo. *Clustered Objects*. PhD thesis, University of Toronto, 2005.

[2] Jonathan Appavoo, Marc Auslander, Maria Butrico, Dilma da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42: an open-source linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[3] Ray Bryant, John Hawkes, and Jack Steiner. Scaling linux to the extreme: from 64 to 512 processors. In *Ottawa Linux Symposium*. Linux Symposium, 2004.

[4] Eliseu M. Chaves, Jr., Thomas J. LeBlanc, Brian D. Marsh, and Michael L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. In *The Symposium on Experiences with Distributed and Multiprocessor Systems*, Atlanta, GA, March 1991.

[5] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[6] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, et al. K42: Building a complete operating system. In *EuroSys*, Leuven, Belgium, April 2006.

[7] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a real operating system. In *Proceedings of EuroSys'2006*, pages 133–145. ACM SIGOPS, April 2006.

[8] Burton Smith. The Quest for General-Purpose Parallel Computing, 1994. www.cray.com/products/systems/mta/psdocs/nsf-agenda.pdf.

[9] Volkmar Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, Germany, May 2005.

[10] R.C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.

[11] Zvonko Vranesic, Stephen Brown, Michael Stumm, Steve Caranci, Alex Grbic, Robin Grindley, Mitch Gusat, Orran Krieger, Guy Lemieux, Kevin Loveless, Naraig Manjikian, Zeljko Zilic, T. Abdelrahman, Benjamin Gamsa, Peter Pereira, Ken Sevcik, A. Elkateeb, and Sinisa Srbljic. The NUMAchine multiprocessor. Technical Report 324, University of Toronto, April 1995.

[12] Zvonko G. Vranesic, Michael Stumm, David M. Lewis, and Ron White. Hector: A hierarchically structured shared-memory multiprocessor. *IEEE Computer*, 24(1):72–80, January 1991.