OS design for non-cache-coherent systems

Simon Peter (*Student, presenter*) Jana Giceva Pravin Shinde Gustavo Alonso
Timothy Roscoe

Systems Group, ETH Zurich
{speter,gicevaj,shindep,alonso,troscoe}@inf.ethz.ch
No demo.

1 Introduction

Which operating system structures are appropriate for machines with either no cache coherence, or else a number of distinct "coherence islands"? While a clear consensus on the architecture of future multicore processors has yet to emerge, it seems that machines without system-wide cache coherence are likely. Some research chips, like the Intel Single-Chip Cloud computer [3] and the Beehive computer [5], have already emerged. This poster presents early work exploring the tradeoffs which characterize the OS design space for such machines which share system memory, but do not provide cache coherence across the whole machine.

The trend driving the re-emergence of non-cache-coherent systems in general-purpose computer architecture is the quest for greater scalability from multi-core processors, and the bottlenecks inherent in a globally-coherent shared-memory model [4,7]. Now is the time to consider what the system software stack must look like on machines like this. As Baumann et al. [2] (among others) have pointed out, modern OSes like Windows and Linux are essentially large, multithreaded programs which assume coherent shared memory, and then apply performance optimizations based on heuristics of the underlying memory system. This model by itself is not going to work on a non-coherent machine.

There are several alternative OS models, which range from carefully rewriting a shared-memory kernel to include explicit cache-flush operations, through refactoring the OS to treat all shared memory with release semantics, eschewing sharing among coherence islands and relying on message passing, all the way to running a separate instance of a conventional OS on each core and programming the machine as if it was a cluster (cluster-on-chip).

Simply refactoring a shared-memory OS to explicitly flush caches will not provide any scalability benefits over hardware-coherent systems without replacing critical data structures with ones that induce less cache traffic.

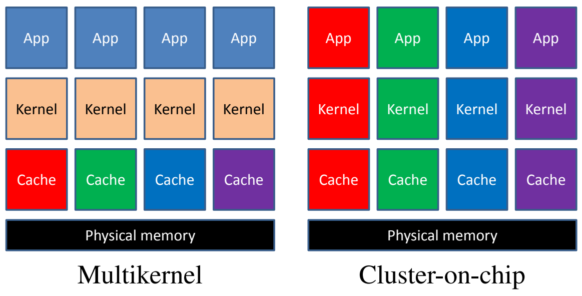

Figure 1: Multikernel vs. cluster-on-chip. Both employ message passing. The Multikernel presents a single OS image to applications, allowing to leverage shared resources, while the cluster-on-chip approach runs an OS instance per core, requiring OS resources to be partitioned at system start.

However, how to design such data structures is an ongoing research topic in itself. The remaining design models use message passing. Figure 1 shows a comparison.

To gain a better understanding of the issues involved in OS support for non-cache-coherent machines, we took the Intel Single-Chip Cloud Computer as an example platform, and ported the open-source Barrelfish multikernel to it. The multikernel model [1] allows us to explore different design options in the space of distributed operating systems.

In particular, we use the resulting system to explore the benefits and costs of managing resources across the whole machine rather than within a single coherence island, by comparing against a cluster-on-chip approach. We show three examples where applications such as databases and parallel applications can obtain tangible benefits from system-wide resource management: dynamically reallocating physical memory among cores or coherence islands, coordinated scheduling of some parallel workloads, and the use of a shared buffer cache among coherence islands.

2 Implementation

We have implemented several OS-level services that leverage the fact that resources, such as RAM and clocks, are shared to improve system performance in several ways.

- A set of per-core memory managers is able to move memory with low latency among cores by transferring only ownership rights.
- A filesystem buffer cache shares all cached buffers among cores to improve overall cache memory efficiency and scalability for applications that can exploit cache synergies.
- Coordinated CPU scheduling allows tight coupling of thread dispatch to cores, improving the scalability of parallel applications that require fine-grain synchronization, by ensuring that a peer thread is always running when synchronized with. We utilize the shared system timer and deterministic per-core schedulers to reduce the amount of communication that has to occur to synchronize dispatch.

Akin to the multikernel model, all services are implemented in a message-passing based, distributed fashion, making all inter-core communication explicit and avoiding excess sharing.

3 Evaluation

We are in progress of porting several applications to Barrelfish and hope to show better elasticity and scalability for their usage domains. The ported applications are:

The postgres database engine. We execute one standalone database server per core in order to simulate a multi-tenant database scenario. By being able to manage memory globally and transfer ownership rights with low latency between cores, we can adapt to changing memory requirements quickly. Figure 2 shows achievable query throughput with varying client load. In a partitioned scenario, memory is statically constrained and query throughput drops after 4 concurrent clients, as the database has to rely on on-disk structures to continue carrying out query processing without dropping connections. Using global memory management, we are able to allocate memory from other, idle database instances and can sustain query throughput.

A parallel application compile trace. We replay the trace to emulate a parallel compilation workload. We hope to show that a shared filesystem buffer cache ultimately provides better scalability for such applications that can leverage multi-core cache synergies than a partitioned cache that has to be kept coherent via other means, such as a central file server.

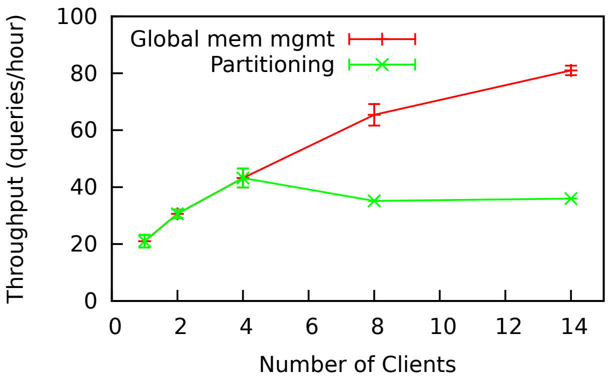

Figure 2: TPC-H query throughput of a Postgres database under partitioning and global memory management with varying number of clients.

The NASA NPB benchmark suite [6], representing the parallel application domain. We run an instance of the benchmark alongside a number of CPU stressor processes to emulate a multi-tasking scenario. We hope to show that coordinated scheduling with a shared system timer in this case can provide better performance and responsiveness for the entire application mix, by inflicting less overhead on the system for schedule synchronization that would be needed using scheduling Middleware in a cluster-on-chip setting.

References

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, Oct. 2009.

[2] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, May 2009.

[3] Intel Corporation. Single-chip cloud computer. http://techresearch.intel.com/articles/Tera-Scale/1826.htm, Dec. 2009.

[4] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. pages 1–11, 2008.

[5] C. Thacker. *Beehive: A many-core computer for FPGAs (v5)*. MSR Silicon Valley, Jan. 2010. http://projects.csail.mit.edu/beehive/BeehiveV5.pdf.

[6] R. F. Van der Wijngaart. NAS parallel benchmarks version 3.3. Technical Report NAS-02-007, NASA Advanced Supercomputing Division, Moffett Field, CA, USA, Oct. 2002.

[7] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 43(2):76–85, 2009.