Many-Core Chips — A Case for Virtual Shared Memory

Gernot Heiser
NICTA[*] and University of New South Wales and Open Kernel Labs
Sydney, Australia
gernot@nicta.com.au

ABSTRACT

We make the case for virtual shared memory (VSM) for supporting future many-core chips. VSM is a shared memory abstraction implemented over distributed memory by a hypervisor, providing the operating system direct access to all memory in the system. VSM on a distributed-memory system, such as a many-core chip with local memory associated with each core or small group of cores, provides a non-uniform memory model to the operating system. We argue, based on our experience with a prototype called vNUMA (implemented on a cluster), that this model can perform well for NUMA-aware software. The indirection layer provided by the virtualization provides benefits to hardware manufacturers, as it can absorb certain faults, including faulty nodes and packet losses in the interconnect.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Distributed memories, Virtual memory*

General Terms

Algorithms, Design, Management, Performance

Keywords

Many-core processors, hypervisor, virtual machine, scalability, guest operating system

1. INTRODUCTION

Future many-core systems, with thousands of cores on a single chip, will be significantly different from present multi-core chips [1].

Busses scale poorly and will be replaced by on-chip interconnect networks with local subnets linked by routers, and inter-core message-passing latencies varying by maybe an order of magnitude depending on the physical distance. Cores (or groups of cores) will have their own clocks, synchronised via protocols running across the interconnect. Cores (or groups of cores) will have significant amounts of local memory. The only shared memory is likely to be off-chip; even the closest off-chip memory may be distributed, due to three-dimensional integration. This implies that shared memory will be several orders of magnitude more expensive to access than local memory.

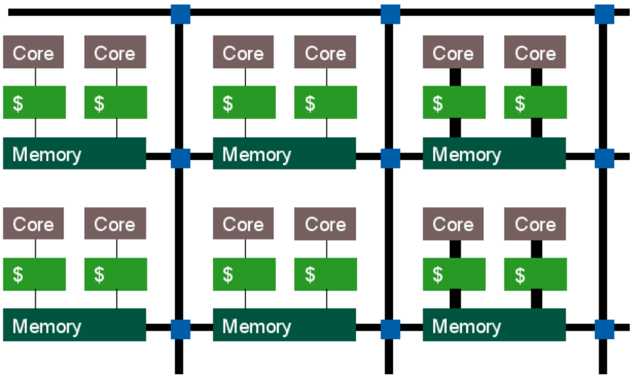

Figure 1: Likely architecture for thousands of cores.

The need to keep yields reasonably high will force manufacturers to ship chips with many dead cores and interconnects. These will be detected and removed from the software-visible hardware during a burn-in phase or even at boot time. The result is that the actual topology of the chip will not be determined by the part number, but will need to be discovered at boot time, and the software must adopt to the topology.

Such a many-core chip will therefore look not unlike a contemporary workstation cluster: a distributed system with a high-speed interconnect. Compared to local memory, the shared off-chip memory will be expensive to access and will appear like some network-attached storage in a cluster, and will be seen as backing store rather than being directly accessed during computation.

This observation makes it natural to explore programming paradigms developed for distributed systems: explicit *message-passing* and *distributed shared memory* (DSM). Message-passing approaches, such as MPI [15], are an obvious approach that maps well to the message-passing nature of the hardware (as it does in distributed systems), and is likely to be the approach of choice for highly-parallel applications. However, shared memory is a more convenient programming paradigm in many circumstances and pro-

[*]NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

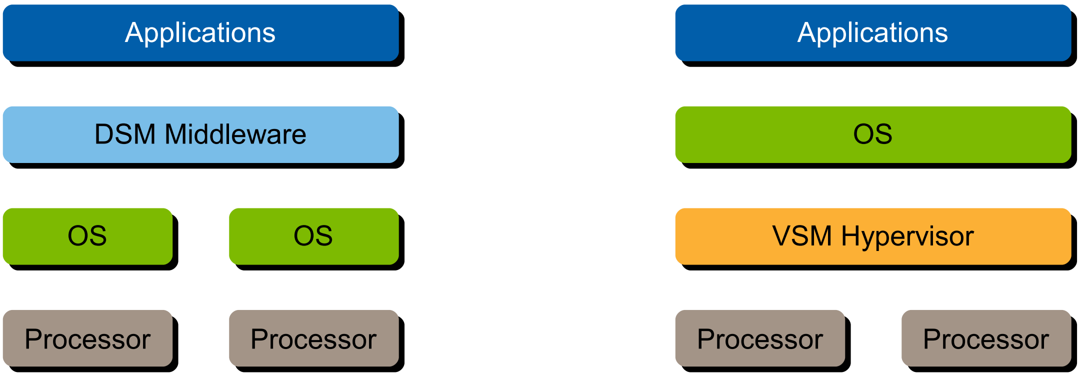

Figure 2: Architecture a classical distributed shared memory (left) and virtual shared memory (right).

vides better support for many legacy programs. For that reason, DSM is still popular in distributed systems today, and like to remain so in the future.

In this paper we make the case for a DSM-like approach as a way to manage many-core systems. We argue that, contrary to classical DSM approaches, such as Ivy [14], Munin [4] or Treadmarks [13], shared memory for many-core systems should not be implemented as middleware (i.e. on top of the OS), but below the OS(es) inside a system virtualization layer. This extends the classical OS notion of virtual memory across the distributed system. To distinguish it from DSM, we refer to this approach as *virtual shared memory* (VSM). Figure 2 contrasts the architectures of DSM and VSM.

We have recently developed a prototype VSM system, called vNUMA [5, 6] ("virtual NUMA"), on a conventional workstation cluster. We argue that the vNUMA approach presents a promising way of managing many-core chips, as it simplifies dealing with the distributed nature of the hardware. It can do so without introducing overheads to high-performance applications, as explicit message-passing still works with full performance.

In the next section we will present a brief overview of vNUMA as it is relevant to this discussion. In Section 3 we will discuss specific advantages VSM provides to many-core chips, and in Section 4 we will discuss the impact on applications. Section 5 presents related work and Section 6 concludes the paper.

2. VNUMA OVERVIEW

vNUMA is a Type-I hypervisor which presents a shared-memory multiprocessor to the (guest) operating system. Specifically, as vNUMA is implemented on distributed-memory hardware, where the cost of accessing remote memory is orders of magnitude higher than for local memory, the virtual hardware provides *cache-coherent non-uniform memory access* (ccNUMA). Hence, NUMA-aware software will perform better on vNUMA than software that assumes uniform memory-access costs.

Implementing shared memory inside the hypervisor has a number of advantages. For one, all the memory in the system becomes part of the VSM, and therefore the OS can access all memory from all nodes. (Of course, a side effect of this virtualization is that the hypervisor can also partition the system, dividing the complete physical memory into several virtual machines, each running a separate, isolated OS. Besides other advantages of virtualization, this supports the deployment of OSes that will not scale to thousands of processors.)

Another advantage is that running in the machine's most privileged mode gives a VSM system access to optimisations that are beyond the reach of DSM middleware such as MPI. These include the efficient emulation of individual instructions, and the use of the performance-monitoring unit (PMU) to track the execution of specific instructions.

Implementing VSM inside the hypervisor also changes some of the trade-offs compared to middleware systems, and as a result requires different protocols. For example, software running on DSM systems is typically aware of this, and specifically of the fact that the unit of migration and coherency is a hardware page. This is not the case for a multiprocessor OS, especially a NUMA-unaware one, which expects data migration and coherency to have a cache-line granularity.

vNUMA therefore includes a number of enhancements to established DSM protocols to support efficient write-sharing within a page [6]. For example, vNUMA supports three different modes of write sharing of pages: *write-invalidate*, *write-update/multiple writer*, and *write-update/single writer*. vNUMA adapts the write-sharing mode based on the observed sharing patterns. In particular, vNUMA detects and efficiently handles atomic instructions (such as compare-exchange) used by the OS to implement locks. For some optimisations (e.g. batching write-update messages), vNUMA makes use of the weak store order provided by modern processors [11].

Evaluation of vNUMA running a multiprocessor version of Linux on a small (8-node) cluster showed that scalability for HPC workloads was generally better than Treadmarks, an example is shown in Figure 3 (see our recent paper [6] for more details). For coarse-grain parallelism represented by compiles matched the performance of middleware solutions (`dist-cc`).

The evaluation also showed that vNUMA performance is highly sensitive to network latency. It furthermore showed that in the case of a database benchmark (PostgresSQL), performance was severely degraded by the design to locking used in that system — fine-grained locking using complex hierarchies of locks and assuming uniform memory access. (Note that running PostgresSQL on a DSM system would lead to worse results, if it could be done at all.) No such performance issues with locks were found in the Linux kernel, showing that this is not an inherent limitation of the VSM approach. Nevertheless, the example showed that VSM will not work well for every application—parallel applications with significant sharing will have to adapt. VSM does not promise a free lunch!

3. VSM ON MANY-CORES

Compared to a cluster, a VSM implementation will benefit from

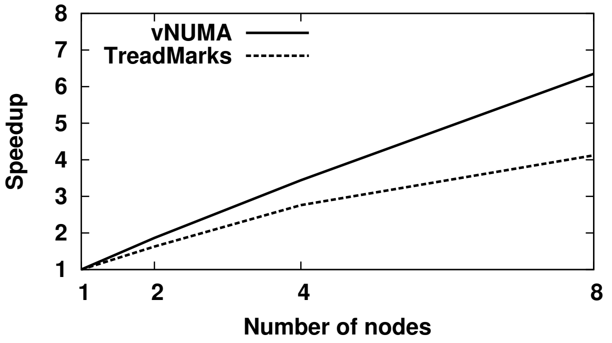

Figure 3: Scalability comparison of vNUMA and TreadMarks using the Splash-2 [19] "water" benchmark.

a number of advantages a multi-core chip has over a traditional cluster. For one, network latencies (measured in CPU-core cycles) are orders of magnitude lower for an on-chip interconnect compared to Ethernet.

More importantly, if the VSM approach is adopted, support for it will be designed into future many-core chips. This has the potential to significantly reduce overheads on a number of operations that we found expensive in vNUMA (on current COTS hardware). For example, vNUMA traps all writes to multiple-writer shared pages in order to determine when updates need to be distributed. While this provides better performance than single-writer protocols in the presence of a limited degree of false sharing, if such writes are frequent, performance will suffer. Architectural support, e.g. in the form of write protection at a cache-line granularity, could reduce this bottleneck.

Furthermore, if the VSM paradigm is widely adopted, then software will adapt to it, for example by changing OS data structures to avoid false sharing. According to our (limited) experience with vNUMA scalability, NUMA-aware software will generally work well, and programs that do not share memory at all will not be affected by VSM in their performance.

The VSM approach can provide some obvious benefits to processor manufacturers:

- The hypervisor can transparently deal with a small amount of message loss in the interconnect. This allows chipmakers to more agressively optimise the network, to the degree where it is no longer fully reliable. In fact, vNUMA, designed for notionally unreliable Ethernet, makes use of the fact that in a cluster environment, Ethernet is in reality "almost" reliable, losing or damaging messages very rarely. vNUMA deals with this by using checksums and timeouts, rather than more sophisticated protocols designed for really unreliable networks.

- Cache coherence does not have to be provided by hardware. With growing number of cores, hardware solutions become more complicated and costly. High-performance applications do not need them, as they deal with distribution explicitly, reducing the benefit of providing coherence in hardware. Shifting coherence protocols into the hypervisor has the added benefit that software can easier adapt protocols to access patterns.

- The hypervisor can transparently re-map memory addresses and core IDs. This not only allows it to deal with unreliable hardware, but also naturally supports turning off cores for power management. The virtual-memory paradigm can be taken to its logical conclusion by transparently swapping out local memory to off-chip backing store.

- Core heterogeneity is easy to support, as individual cores or groups of cores can run their own OS, with the hypervisor simplifying access to remote memory. Heterogeneous OSes are also easy to support, the main requirement is that each core's ISA supports the hypervisor.

It would be possible to implement VSM inside native OSes running on many-cores. However, we expect virtualization to be widely used in future systems anyway, for reasons of resource isolation / quality of service, and for dynamic resource management, in particular saving energy by shutting down unused cores. A single chip will typically run multiple, heterogeneous operating systems, each with varying allocations of physical resources.

Only the hypervisor has access to the whole system, and as such is the ideal place to implement VSM. For example, it could use Capabilities [7] for controlling access to pages or coarser-gain memory regions by guest OSes.

4. VSM AND APPLICATIONS

It would obviously be a fallacy to expect a VSM to scale for parallel applications utilising hundreds or thousands of nodes. On the one hand, the latency of communication cannot be hidden by presenting a shared-memory model. On the other hand, the more nodes access the same data, the more coherency traffic is required, and for n nodes, the cost of this is $O(n^2)$. Communication-intensive applications are best served by explicit message passing as supported by MPI or similar middleware.

The real scalability test of a VSM system is whether it can support a large number of processors in the absence of contention. More precisely, the system should not impose communication overhead for applications that do not communicate [10]. It is precisely designed to achieve this: The coherency protocols ensure that pages which are only written by a single core will be owned exclusively by that core, while read-only pages are shared. Coherency overheads only arise when pages are shared, or change their mode.

As such, the small-cluster scalability results we obtained from our vNUMA prototype should be representative of parallel applications running on a small subset of cores of a large many-core system; the main difference being that the many-core system should be more VSM-friendly than a cluster for the reasons discussed in the introduction.

Furthermore, the coherency protocols ensure that message-passing middleware like MPI on top of VSM should be able to perform as well as without the VSM layer: as it never shares data, but copies it between nodes by explicit messaging, it does not create coherency traffic.

Hence, the VSM stays out of the way of software that does not need it, but is there to support software that benefits from a shared-memory model.

5. RELATED WORK

VSM is based on the ideas of DSM, pioneered by Ivy [14]. Mirage [9] moved DSM into the OS to improve transparency. Munin [4] utilised weaker memory consistency to support simultaneous writers.

Disco [3] carves a NUMA system into multiple virtual SMP nodes for the benefit of existing operating systems that may not support a NUMA architecture. This is, in a way, the opposite of VSM, which combines separate nodes into a single virtual NUMA system, allowing a single operating system instance to span multiple nodes that do not share memory.

Since our initial publication on vNUMA [5], systems using similar ideas have emerged: Virtual Iron's VFe hypervisor [18] the Virtual Multiprocessor from the University of Tokyo [12]. While these systems demonstrate combining virtualization with distributed shared memory, they are limited in scope and performance. Virtual Iron attempted to address some of the performance issues by using high-end hardware, such as InfiniBand rather than Gigabit Ethernet, which effectively makes the network more similar to what we expect from future many-cores. Virtual Iron has since abandoned the product for commercial reasons, which largely seems to stem from its dependence on such high-end hardware. More recently, startup company ScaleMP started to market their vSMP system [16], which seems similar in nature (also uses InfiniBand). This supports our claim that there is on-going interest in SMP as a programming model on distributed-memory hardware.

Catamount [2] partitions shared memory between nodes but makes remote partitions available via virtual-memory mapping. Work on many-core scheduling [8] is orthogonal to the VSM concept. Barrelfish [17] deals with many-core resource heterogeneity by making it explicit. While we agree that this is the best way to achieve best performance, it only benefits applications that are designed to deal with explicit heterogeneity.

6. CONCLUSIONS

We made a case for virtual shared memory, i.e., a virtual-memory abstraction implemented over physically distributed memories by a hypervisor, as an attractive model for managing future many-core chips. Based on our experience with a cluster-based prototype, we argue that VSM provides a shared-memory abstraction for software that needs it, without imposing significant overheads on software that does not share (virtual) memory. We have argued that the approach integrates well with the use of virtualisation for resource management on many-cores, and simplifies dealing with faulty cores, faulty interconnects and heterogeneity. It may allow processor manufacturers to move cache coherence protocols from hardware into software.

7. REFERENCES

[1] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th Design Automation Conference*, pages 746–749, San Diego, CA, USA, June 2007.

[2] R. Brightwell. Lightweight kernel support for direct shared memory access on a multi-core computer. In *Proceedings of the 1st Workshop on Managed Many-Core Systems*, Boston, MA, USA, June 2008.

[3] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15:412–447, 1997.

[4] J. B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29:219–227, 1995.

[5] M. Chapman and G. Heiser. Implementing transparent shared memory on clusters using virtual machines. In *Proceedings of the 2005 USENIX Technical Conference*, pages 383–386, Anaheim, CA, USA, Apr. 2005.

[6] M. Chapman and G. Heiser. vNUMA: A virtual shared-memory multiprocessor. In *Submitted to USENIX'09*, Jan. 2009.

[7] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966.

[8] A. Fedorova, V. Kumar, V. Kazempour, S. Ray, and P. Alagheband. Cypress: A scheduling infrastructure for a many-core hypervisor. In *Proceedings of the 1st Workshop on Managed Many-Core Systems*, Boston, MA, USA, June 2008.

[9] B. D. Fleisch and G. J. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, 1989.

[10] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, New Orleans, LA, USA, Feb. 1999.

[11] Intel Corp. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, Oct. 2002. http://www.intel.com/design/itanium2/documentation.htm.

[12] K. Kaneda. Virtual machine monitor for providing a single system image. http://web.yl.is.s.u-tokyo.ac.jp/~kaneda/dvm/.

[13] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 115–131, 1994.

[14] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7:321–59, 1989.

[15] Message Passing Interface Forum. MPI: A message-passing interface standard, Nov. 2003.

[16] ScaleMP. http://www.scalemp.com. Accessed March 2009.

[17] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the 1st Workshop on Managed Many-Core Systems*, Boston, MA, USA, June 2008.

[18] A. Vasilevsky. Linux virtualization on Virtual Iron VFe. In *Proceedings of the 2005 Ottawa Linux Symposium*, July 2005.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.