A Case for Software Managed Coherence in Many-core Processors

Xiaocheng Zhou, Hu Chen, Sai Luo, Ying Gao, Shoumeng Yan, Wei Liu, Brian Lewis, Bratin Saha

Intel Corporation

[xiaocheng.zhou, hu.tiger.chen, sai.luo, ying.gao, shoumeng.yan, wei.w.liu, brian.t.lewis, bratin.saha]@intel.com

Abstract

Processor vendors are integrating more and more cores into their chip. These many-core processors usually implement hardware coherence mechanisms, but when the core count goes to hundreds or more, it becomes prohibitively difficult to design and verify efficient hardware coherence support. Despite this, many parallel applications, for example RMS applications [9], show little data sharing, which suggests that providing a complex hardware coherence implementation wastes hardware budget and design effort. Moreover, in some increasingly important domains, such as server and cloud computing, multiple applications may run on a single many-core chip. Those applications require coherence support among the cores that they are running on, but not between different applications. This indicates a strong requirement for dynamically reconfigurable coherence domains, which is extremely hard to support with hardware-only mechanisms. In addition, hardware coherence is believed to be too complex to support heterogeneous platforms such as combined CPU-GPU systems, regardless whether the GPU is integrated or discrete.

In this paper, we argue that software managed coherence is a better choice than hardware coherence for many-core processors. We believe that software managed coherence can make better use of silicon, efficiently support emerging applications, dynamically reconfigure coherence domain, and most importantly, still be able to provide performance comparable to hardware coherence. We implemented a prototype system with software managed coherence over a partially-shared address space and show promising results.

1. Introduction

When single-CPU designs hit the power wall, processor architecture entered the multi-core era [7]. Today, multi-core processors dominate the market from client to server platforms, and it is believed that general purpose many-core chips will emerge soon that have several tens to hundreds of cores on a single chip. Moreover, accelerator processors such as GPUs are being integrated into heterogeneous architectures as well.

Cache coherence is a fundamental issue that must be addressed during many-core design. With a small number of cores (e.g., less than ten), a hardware coherency implementation has proven effective. However, when the core count goes to several tens or hundreds, it becomes prohibitively expensive and difficult to design and verify efficient hardware coherence support [3][4].

Even when hardware coherence in a many-core processor is feasible, it still shows a couple of limitations compared to a software implementation. First, hardware coherence is not as flexible as its software counterpart. Given the increasing importance of server and cloud computing, many applications will run concurrently on a single chip. Dynamically forming a coherent domain for each application is challenging for a hardware coherence implementation, but not for software managed coherence. Second, emerging applications show characteristics like limited data sharing that suggest full hardware coherence support may be unnecessary.

In this paper, we propose not implementing complex hardware coherence mechanisms. Instead, we argue that software managed coherence is a better choice for many-core processors because it can utilize silicon more efficiently, support more flexible usage, and provide comparable performance.

This paper makes the following contributions:

- It advocates software managed coherence for many-core architecture. Software managed coherence can be flexible and performance competitive.
- It describes our prototype implementation that runs on both an Intel 32-core server and a machine containing an experimental Intel microprocessor called

the "Single-chip Cloud Computer" (SCC) [14]. The results demonstrate that performance of software managed coherence can be comparable to hardware coherence.

The rest of the paper is organized as follows. Section 2 argues why software managed coherence is a better choice to meet the demands of many-core applications. Section 3 describes our implementation of software managed coherence. We show performance results in section 4, describe related work in section 5 and conclude in section 6.

2. Why software managed coherence is a better choice

We advocate using software managed coherence in future many-core processors, instead of relying on hardware coherence across the full chip. In this section, we explain why software managed coherence is a better choice for many-core processors given emerging architectural trends and application characteristics.

Dynamically reconfigure coherence domains

We expect that in the near future, single-chip cloud computers such as SCC [14] will become popular. These will typically have multiple applications running at the same time on the chip. While these applications may require coherence support among the cores they run on, they will not need coherence support between applications. Static partitioning cannot solve the problem totally, since the applications may request more cores or free cores during execution. Therefore, supporting dynamically-reconfigurable coherence domains will be important. Unfortunately, current hardware coherence protocols are not flexible enough to do so. It is not too difficult to add extensions to statically partition cores into coherence domains. However, it is nontrivial to support dynamic reconfiguration.

Software managed coherence is a more natural way to support dynamic partitioning than hardware coherence. It maintains the metadata of coherence domain information in runtime data and flexibly reconfigures domains based on that runtime information. It is much easier to add or remove a core from a coherence domain compared to hardware coherence. Moreover, higher-level information such as load balance, data location, and other data can be leveraged to decide the optimal policy for a particular coherence domain.

Support emerging applications

Emerging many-core applications may not require full hardware coherence at all. We found that there are two important characteristics for server and cloud workloads: one is little data sharing and few accesses to shared data, and the other is coarse-grained data synchronization. For example, one important application domain for many-core processors is RMS applications [9]. However, we observed that the parallel threads of these applications share little data both statically and dynamically [10]. This implies that the vast majority of executions only operate on their non-shared data. Thus these applications do not need any coherence enforcement, which means a complex hardware coherency implementation would be an overdesign for them. Better use could have been made for those transistors.

Moreover, even in the presence of data sharing, coarse-grained data synchronization dominates. For example, map-reduce [16] and BSP [17] models both need only coarse-grained data synchronization. This also suggests that full hardware coherence is unnecessary. Furthermore, software managed coherence can achieve comparable performance by taking advantage of coarse-grained data synchronization. We believe a release consistency model is a perfect match for such many-core workloads. In addition, release consistency models can generally be implemented in software efficiently.

Reduce hardware cost

As we mentioned, when processors integrate several tens or hundreds of cores, it becomes increasingly difficult to design and verify efficient hardware coherence support. As [3] and [4] point out, the difficulties come from two aspects: 1) The conventional interconnect is expected to become the system bottleneck with so many cores competing for the communication channels if existing coherence protocols are used; 2) The new scalable interconnects not only make the implementation of hardware coherence protocols extremely complicated, but they make them notoriously hard to verify. This leads to a dilemma between interconnect scalability and coherence protocol complexity. We view that scalable interconnects are the future for many-core processors and believe that coherence enforcement can be best achieved by a software-only approach.

Previous software managed cache coherence proposals have demonstrated that comparable or better performance can be achieved with zero or a few hardware additions such as new instructions, modified cache structures, and new replacement policies [5][6]. How-

ever, additional optimizations can be done to further improve the performance of software approaches. For example, instead of managing coherence at the cache/physical memory level, software managed coherence can maintain coherence of virtual memory space. This results in hardware cache resources like hardware directories and complex circuitries being saved. As another example, release consistency models keep memory updates locally until a release point, and transfer the updates at an acquire point. Thus, many individual coherence messages are effectively combined into a bulk communication that reduces pressure on the interconnection and lowers power consumption.

Support heterogeneous computing

Heterogeneity is expected to proliferate in future computing platforms. This trend can be seen in the TOP500 machine list [15], where many systems with both CPUs and GPUs appear. We believe heterogeneity will be common in future many-core systems. Since their cores can have different architectures and instruction sets, software managed coherency will be a better solution than hardware mechanisms [8].

In summary, we believe that software managed coherence is a better choice for future both homogeneous and heterogeneous many-core platforms. In fact, some experimental microprocessors like SCC have already begun to remove hardware coherence support, and leave coherency handled by software in order to get better scalability and lower power consumption.

3. Implementation of software managed coherence

We implemented a system that uses a software-only approach to enforce coherence. Our system runs on both multi-core servers and SCC. Figure 1 describes our system architecture. There are two key components in our runtime. One is a domain management module that manages dynamically partitioned coherence domains. It also manages any related resources. Another is a coherence policy module that implements a set of software managed coherence policies. For a certain domain, the user can specify a policy managed by the policy module.

Coherence domain management

The coherence domain management module exports APIs to the application to allocate/free and expand/shrink coherence domains. After the application is bound to a domain, our system provides resource virtualization for it that intercepts system calls related to the domain resources. For example, the Windows system call *GetSystemInfo* will only return core information in its domain, and the following *SetProcessAffinityMask* call only sets affinity in this domain.

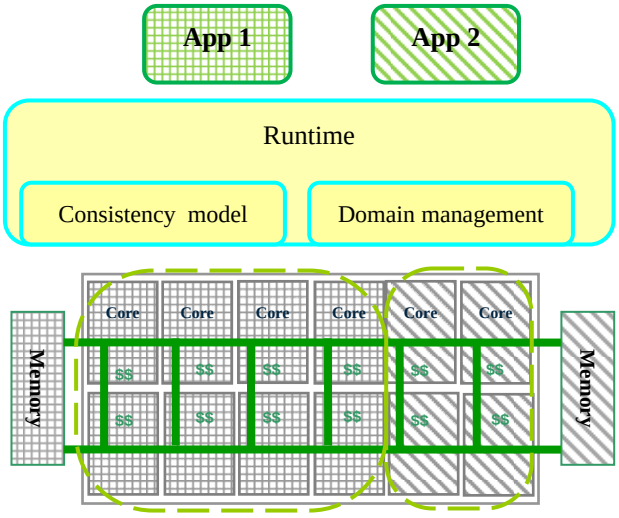

Figure 1 Architecture of the system with software managed coherence

We implemented this module in two systems. In a system with a single operating system, we don't change the OS scheduler, but instead wrap all system calls related to CPU information such as *GetSystemInfo* and *fork*. When the program forks a task, we intercept the request and set its affinity to a core in the domain before it returns. In a system with multiple operating systems such as SCC, we have to build an abstract software layer above the operating system to manage the coherence domain.

Consistency model

Each coherence domain can choose its own consistency model from the consistency control module. This allows different applications running on the same chip to use different consistency models. For example, some applications (e.g. a simple web server) do not require consistency; while other applications need release consistency; and still others may need mixed consistency models.

In our implementation, we choose to support release consistency [19] since it has proved very useful at reducing communication overhead in coherency management. It also matches the coarse-grained data synchronization model of many parallel applications.

Partially-shared virtual memory

Our system adopts the shared virtual memory approach of [13] to provide a coherent, shared virtual address space among all threads at page granularity. For each shared virtual page, there is a single home thread maintaining the golden copy of the page, and all the local copies maintained in other threads need to synchronize with that golden copy at synchronization points.

When one thread performs an acquire operation, all subsequent writes are recorded. At a release point, the changes are merged into the golden copy. The bulk communication at release points reduces the number of communication messages and so reduces the communication overhead.

We use the virtual memory protection mechanism to detect accesses to these shared pages. Based on access modes, each shared page can be *Invalid*, *Read-only*, or *Read-Write* for each thread. We use twin pages for each shared virtual page to compute page differences to update the golden copy. We maintain the metadata (e.g., status, home info) for the page in a local directory structure.

We also choose to only provide a *partially*-shared virtual memory space. In our system, the virtual address space of each thread is divided into two parts. One is shared virtual address space that is shared and accessible by all threads of the same application; and the other is private virtual address space that is private to each thread and only it can access. The shared space contains just shared data, which cuts down the amount of memory that needs to be kept coherent. We introduce one new type qualifier "*shared*" to the standard C/C++ languages. By default, all data are private and allocated in the private space. It is the programmer's responsibility to mark shared data by using the new qualifier or by calling a special memory allocation API.

4. Experimental evaluation

We used both a SCC machine and a 32-core server to evaluate our software managed coherence implementation. The research SCC microprocessor contains a total of 48 cores on a single chip connected with a fast and scalable on-chip mesh network. It has no hardware cache coherence support in order to simplify the design and reduce power consumption. The 32-core server we used is a SMP machine that contains four 8-core Intel® Core™ i7 processors. We measured two workloads, Black Scholes and Art, which are typical kernel benchmarks for many core servers. Black Scholes is a financial workload from MCBench [18] that prices options using the Black Scholes method. Art is a workload from SpecOMP that performs image recognition. Standard input sets are used for both workloads.

We used two versions of each workload. One is the original Pthreads/OpenMP version of the workload. We ran this version to determine the performance when there is hardware coherence support. We cannot run the original version on SCC due to its lack of hardware coherency. We also ported the two workloads to our software coherence system by inserting explicit coherence API calls to manage coherence. We ran the ported versions to get performance data with software managed coherence on both SCC and the 32-core server.

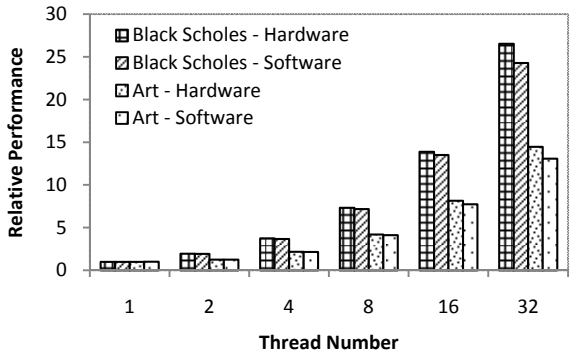

Figure 2 Performance comparison of software managed coherence and hardware coherence

Figure 2 compares the performance on the 32-core system between software managed coherence and hardware coherence. Note that both performance numbers for software and hardware coherence are relative to a sequential version on single core. This figure shows that software managed coherence produces comparable performance to hardware coherence. It also shows good scalability for both software managed coherence and hardware coherence.

We believe that we can get better performance for software coherence by further optimizing the implementation. For example, a large part of the overhead is from changing the protection of the shared virtual pages and recording dirty pages when they are modified. This overhead can be reduced if we can modify the OS to support these operations more efficiently.

We show the performance of the two workloads running on SCC in Figure 3. Scalability for these two workloads is good even though coherence is managed

by software. These results show that the SCC scalability compares well with the 32-core server. As we mentioned, there is no hardware coherence support, so we have no way to directly compare the performance of software managed coherence with hardware coherence on SCC. We believe that performance of software managed coherence would be close to that for hardware coherence on SCC if that were implemented.

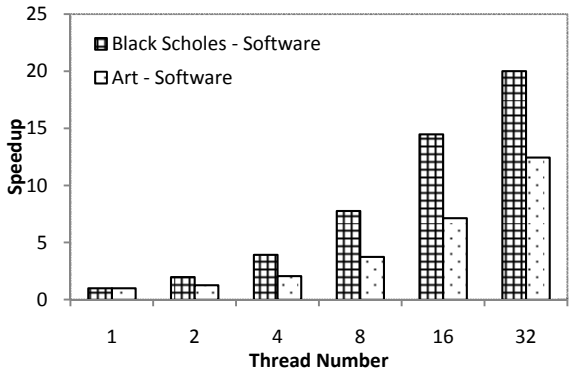

Figure 3 Performance data on SCC with software managed coherence

5. Related work

In the past, several authors advocated using software coherence for multi-chip systems [1][2] but expressed concern about uncertain performance. Today, however, computer technology is entering the many-core era. The on-chip interconnects of many-core systems are much faster than the network in SMP or cluster systems. Also, emerging many-core applications tend to show coarse-grained data synchronization and little data sharing. These trends suggest revisiting the idea of using software coherence. Our belief is that software-managed coherence will help solve the power and complexity issues of many-core chips, and will better match the requirements of emerging applications.

Fensch et al. [5] propose using software to map lines to physical caches and hardware to support remote cache access. With these changes, there is no need for hardware-coherence support. They extend the TLB structure to enable an OS-based mapping mechanism. Compared to our approach, their proposal involves nontrivial hardware logic and state storage that makes it difficult to use current designs as building blocks for many-core processors. In addition, their OS-based scheme requires significant kernel modifications that may not be possible without OS sources.

Rigel [6] is an architecture using software to manage cache coherence. All local stores are invisible to other cores until an eviction occurs or the data is explicitly flushed by software. Compared to our work, Rigel has a different architecture (cluster cache and global cache) and different algorithms (local operation and global operation). On another hand, it maintains the cluster's cache coherence by software with additional hardware support, while our implementation is a pure software-managed coherence approach.

6. Conclusion and future work

In this paper, we advocate supporting software managed coherence for many-core processors instead of expensive hardware coherence. Our software managed coherence mechanism can dynamically support multiple coherence domains, efficiently supports emerging applications, reduces hardware cost for coherence management, and more easily supports heterogeneous computing platforms. We implemented a system using our software-only approach to manage the coherence of a partially-shared virtual address space. The experimental results show that in both SCC and a 32-core server, the performance of software managed coherence is comparable to hardware coherence.

In the future, we may quantify the hardware savings with our software managed approach, and explore opportunities to further improve software managed coherence performance with simple hardware extensions.

7. Reference

[1] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon. Comparison of Hardware and Software Cache Coherent Schemes. International Symposium on Computer Architecture, 1991.

[2] R. N. Zucker and J-L. Baer. Software versus Hardware Coherence: Performance versus Cost. International Conference on System Science, 1994.

[3] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-core Architecture: Understanding Mechanisms, Overheads and Scaling. International Symposium on Computer Architecture, 2005.

[4] D. Abts, S. Scott, and D. J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. Internal Parallel and Distributed Processing Symposium, 2003.

[5] C. Fensch, and M. Cintra. An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs. High Performance Computer Architecture, 2008.

[6] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. International Symposium on Computer Architecture, 2009.

[7] J. Rattner. Tera-Scale Research Program. Intel Development Forum. Spring 2006.

[8] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming model for a heterogeneous x86 platform. Programming Language Design and Implementation, 2009.

[9] Dubey P. Recognition, Mining and Synthesis moves computers to the era of tera, Technology @ Intel, Feb 2005.

[10] S. Yan, X. Zhou, Y. Gao, H. Chen, S. Luo, P. Zhang Peinan, C. Naveen, R. Ronny, and S. Bratin. Terascale Chip Multiprocessor Memory Hierarchy and Programming Model. International Conference on High Performance Computing, 2009.

[11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Ractive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. International Symposium on Computer Architecture, 2009.

[12] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. RajaMony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. IEEE Computer, Feb 1996.

[13] L. Kontothanasis, R. Stets, G. Hunt, U. Rencuzogullari, G. Altekar, S. Dwarkadas, and M. L. Scott. Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks. ACM Transactions on Computer Systems, August 2005.

[14] Intel Corp. Single-chip Cloud Computer. At http://techresearch.intel.com/articles/Tera-Scale/1826.htm

[15] http://www.top500.org

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Symposium on Operating System Design and Implementation. December, 2004.

[17] L. G. Valiant. A Bridging Model for Parallel Computation. Communications of the ACM 33, 8 (Aug. 1990), 103-111.

[18] T. Mattson, Y. Chen. MCBench: The Many Core Benchmark Suite, Intel Tracing and Simulation Summit 2007, Hillsboro, OR, US, May 14-16, 2007.

[19] S.V. Adve, A.L. Cox, S. Dwarkadas, R. Rajamony, W. Zwaenepoel, A comparison of entry consistency and lazy release consistency implementations, Proceedings of Second International Symposium on High-Performance Computer Architecture, 1996.