2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems

Software Coherence Management on Non-Coherent Cache Multi-cores

Jian Cai, Aviral Shrivastava
Arizona State University
Compiler Microarchitecture Laboratory
Tempe, Arizona 85287 USA
{*jian.cai, aviral.shrivastava*}*@asu.edu*

***Abstract*—The design complexity and power consumption of hardware cache coherence logic increase considerably with the increase in number of cores. Although skipping coherence can simplify hardware and make it more power-efficient, programming becomes more challenging as programmers have to manually insert DMA instructions to ensure that there is coherence of shared data between cores. To reduce the burden of parallel programming, we propose program transformations and a runtime library that will enable correct execution of data-race-free multi-threaded programs. Our scheme manages coherence at byte granularity rather than conventional page-granularity. We further optimize the performance by introducing the concept of private write notice for each core and combining write notices in our coherence implementation. Experimental results of running multi-threaded signal processing benchmarks on the 8-core non-cache coherent Texas Instruments processor TMS320C6678 demonstrates that our technique achieves $12X$ performance improvement over naive scheme of disabling caches, and $2X$ performance improvement over the state-of-art technique.**

***Keywords*-Software Coherence Management, Scratchpad Memory, Multi-core Processor, Software Managed Multicores**

I. Introduction

Multi-core processors are now common in embedded systems, as single-cores cannot deliver the power-efficiency (MIPS per watt) required for embedded applications. However, it is becoming harder to scale the memory subsystem, since the area and power overhead of implementing cache coherence increases dramatically with the number of cores [1]–[4]. As a result, embedded processors (and also general purpose processors) at larger core count are looking to avoid implementing coherence in hardware. For example the latest 8-core TI TMS320C6678 processor features a non-coherent cache memory architecture, or in the more general purpose domain, the 48-core Intel SCC [5] has non-coherent cache memory architecture. Although a Non-Coherent Cache (NCC) architecture simplifies hardware design and improves power-efficiency and scalability of the architecture, it requires explicit DMA instructions to ensure the coherence of shared data when developing parallel programs. This problem is illustrated in Figure 1. Without explicit communication, modifications of shared data will not be propagated properly and cause unexpected problems.

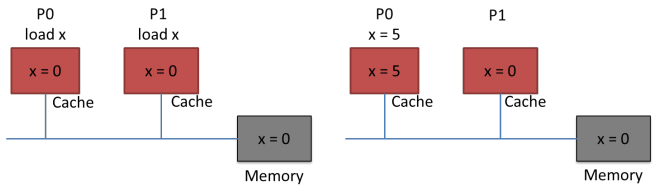


Figure 1: *Coherence of data has to be managed explicitly on a Non-coherent Cache (NCC) multi-core architecture. Suppose physical memory location x initially contains value* 0 *and both cores P0 and P1 caches it. If P0 writes a new value* 5 *at address x, P1 will still stay uninformed and access the stale value.*

There has been quite a lot of research on coherence management of shared data in multiprocessing environment [6]–[8]. Most previous work maintains coherence of shared data are fixed granularity, either at block level [9] or page level [10], [11]. Typical approaches either rely on hardware such as Memory Management Unit's page fault handler to identify writes to shared pages and invalidate its copies in all other cores [7], [10], [11], or develop software to manage the modifications on each core and do compute-intensive comparisons between the original and the modified copy to apply the change [12]. We do not consider the hardware approaches in this paper since modern multi-cores are usually designed without complex hardware to save power. But even with the software approach, a fixed granularity approach may not be the best option for NCC architectures—such approaches usually require much computation power. For example, in the presence of multiple writers to the same shared page, each writing core needs to create and modify its local copy, and compare the modified copy to the original copy in order to apply changes to the original copy [12]. In traditional multi-processor systems with computationally strong processors and relatively slow inter-process communications, such approaches that increase computation to avoid more expensive communication makes sense. However, in modern multi-core systems, each core is designed with relatively weak computational power to preserve power efficiency, yet with much faster communication speed. For example, the theoretical peak performance of an accelerator core on the Cell processor is around 20 GFLOPS [13], while that of a conventional Xeon processor reaches up to 600 GFLOPS [14]. Therefore, we need to come up with a customized solution to adapt the changes.

In this paper, we propose a pure software approach at byte granularity on NCC architectures to manage coherence of shared data among cores while reducing the computational overhead caused by the management. Our approach achieves almost 12X performance improvement compared to a naive way of running parallel programs by disabling caches, and more than 2X performance improvement compared to the state-of-art software approach, on non-coherent cache architectures [12] on the new 8-core non-cache coherent TI TMS320C6678 processor [15]. Experiments also show that the performance overhead of managing coherency via our approach is more manageable than [12].

II. Related Work

Maintaining coherence of shared data has been a heavily studied topic for multi-processor systems, e.g., cluster computing, grid computing, and distributed computing, in an effort to provide a Distributed Shared Memory (DSM) to applications [16]. A DSM creates an illusion of shared memory over a distributed memory system, e.g., a multi-processor system. DSM is in general a software-based approach to manage data coherence between processors. A compendium of many important approaches to provide coherence among the processors of a multi-processor system can be found in [17]. DSMs are closely related to software cache coherence as both try to provide a single image of memory to all processors. Most software DSMs use page-grain coherence management [18]. When the page-grain DSM is implemented by modifying page-fault handler, it is also called Software Virtual Memory, or SVM. All these coarse-grain approaches aim to reduce the communication between processors, even if it results in a slight increase in the computation required.

Although page-grain coherence management reduces communication, it is prone to false sharing. To avoid the adverse effects of false sharing on the performance, some researches propose to manage coherence at variable granularity. Carter et al. [10] introduced a method of managing coherence at size of the data items, through user-defined association between synchronization objects and data items. Their approach relies on Memory Management Unit (MMU) trapping page faults, Scales et al. [9] transparently rewrite the application executable to intercept loads and stores in compiler. Sandhu et. al. [19] introduced a program-level abstraction called shared region, and users explicitly call a function that binds a shared region to a set of memory locations with variable sizes, and access shared regions via some provided functions which will guarantee the synchronization of different processors. Bershad et. al. [11] also provides variable granularity coherence management by asking users to explicitly associate data items and synchronization objects.

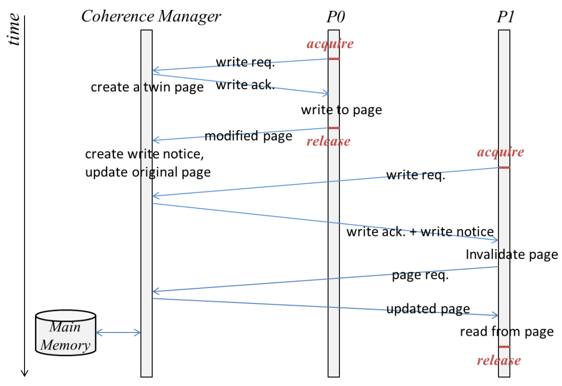


Figure 2: *The way COMIC works.*

More recently, providing coherence and consistency in multi-cores has become a more important subject of research. Several hardware based schemes have been proposed to assist in coherence implementation, and make it more efficient. A recent proposal from Zhao et al. [20] proposes two adaptive cache coherence schemes. The first scheme supports adaptive granularity of storage but fixed cache line size, while the other supports both adaptive granularity of storage and cache line size. Ophelders et al. [21] proposed a hardware-software hybrid scheme which places private data in write-back cacheable memory regions and shared data in write-through cacheable memory regions. While these approaches attempt fine-grain coherence management, they are implemented by introducing new hardware, while we are looking for a software based solution for architectures that do not implement coherence through hardware.

Among the software approaches, Kim et. al. [22] proposed a software shared virtual memory for the Intel SCC [5]—a non-coherent cache architecture. Their approach requires modification of page-fault handler and manages coherence at page granularity. However, several NCC multi-core architectures including the TI TMS320C6678 do not have a MMU for each core, therefore page-fault handler based schemes are not applicable for several embedded NCC architectures. The work most closely related to our work is COMIC [12]. It is a pure software approach designed for multi-core processors without hardware cache coherence [23] and MMU in each core. In the experiments section, we will compare the performance of COMIC with our approach. We describe COMIC in more detail in the next section.

III. Previous Approach

COMIC [12] proposes a software approach which implements Release Consistency model [24] at page granularity. Release Consistency consists of two special operations: *acquire* and *release*. The program execution between acquire and release is called *interval*. All memory accesses after an acquire operation should be performed only after the acquire operation has been performed while all memory accesses before a release operation must have been performed by the

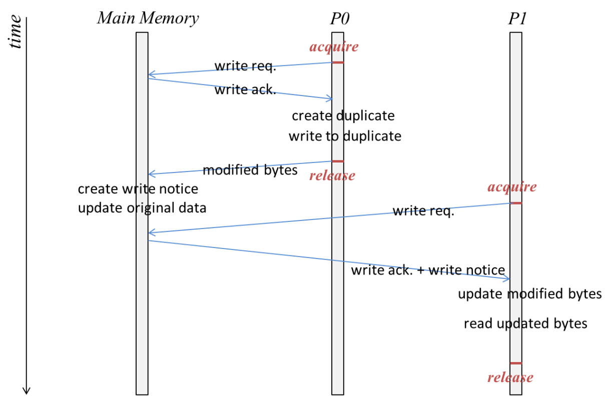


Figure 3: *The way our approach works.*

time when the release operation is performed. Acquire and release operation are performed in program order.

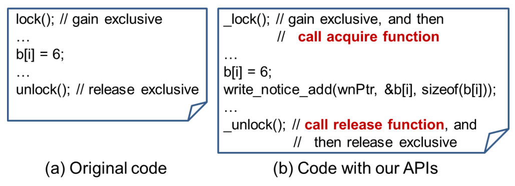
Figure 2 illustrates how COMIC works. It requires one core to act as coherence manager, which is the only processing element that can directly access the main memory. All the other cores have to make memory requests via coherence manager. In addition, whenever write requests to a shared page arrive, the coherence manager creates a copy of the page (termed *twin*), and sends the page to all requesting cores. When the cores are done changing the page, they send back the modified pages, and coherence manager finds out the changes of each core by comparing the unmodified twin to the modified pages, then only applies the changes to the original page in the main memory. Both the centralized management and compute-intensive page comparison contribute to high computational overhead on coherence manager and throttle the overall performance of system.

IV. Our Approach

Our approach implements Release Consistency model at byte granularity. Figure 3 shows an overview of our approach. Whenever a core wants to modified shared data, it first performs an acquire operation. Upon its success, the core creates a private duplicate in main memory and all the subsequent writes go to the duplicate. All writes during the interval are recorded in write notices. Each write notice is a record that saves the the memory location and the bytes modified. On the release operation, the core makes all the write notices visible to other cores by pushing them back to the main memory, together with modification recorded. The subsequent acquiring core can then read the write notices from the main memory and either update or invalidate its local copy of modified data.

A. Code Transformation

We implement our own synchronization primitives. Figure 4 shows an example of how the code will be changed with our management functions. The *_lock* performs an *acquire* operation once the exclusiveness to the critical region is



```
lock(); // gain exclusive
…
b[i] = 6;
…
unlock(); // release exclusive
```

(a) Original code

```
_lock(); // gain exclusive, and then
         //   call acquire function
…
b[i] = 6;
write_notice_add(wnPtr, &b[i], sizeof(b[i]));
…
_unlock(); // call release function, and
         //   then release exclusive
```

(b) Code with our APIs

Figure 4: *Code transformation with our management functions. Figure (a) shows the snippet of original code. Figure (b) shows the transformed code.*

obtained, while the *_unlock* function performs a *release* operation and releases the lock. The *write_notice_add* function records the start address and size of the modification.

B. Fine-Grain Coherence via Write-Notice

Write notice is the key component in implementing release consistency model, and the content of a write notice determines coherence granularity. For example, in traditional page-grain coherence scheme, e.g., COMIC, a write notice will mark which page is updated but not where exactly the page is modified. So whenever any coherence operation (e.g., acquire and release) happens, it either invalidates or updates the entire page, which usually causes unnecessary data transfer. Our coherence scheme works at byte granularity, assuming that no more than one processing elements should access different bits on the same byte simultaneously. In our approach, a write notice records the beginning address and the exact size of the memory locations. By doing so, a core only needs to write back the exact modified part of shared data at the release operation. For example, if a core modified only one word in a cache block, it will write only that word instead of flushing the whole cache line.

C. Reduced Computation Overhead

Compared to communication pattern of COMIC (Figure 2), our approach (Figure 3) writes back the exact modified bytes to main memory instead of pages. As a result, when the modified data is written back to main memory, no comparison is needed to figure out the modification. Also, the duplicate and write notices are created by each acquiring core, but not by the coherence manager as in COMIC. By doing so, we avoid compute-intensive tasks of creating twins, comparing different copies of the same page and applying the difference. Moreover, it also distributes the load to multiple cores and mitigates the throttle of a centralized manager.

D. Further Optimizations

To further reduce the runtime overhead of our approach, we propose two more ways of optimization. The first optimization aims to reduce the contention to memory locations if multiple cores are trying to create write notices at the same time, while the second one tries to reduce the number of write notices.

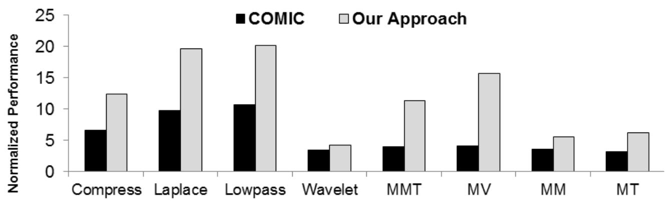


Figure 5: *Comparison of the performance of our approach and COMIC.*

1) Private Write Notices: Two types of write notices are used in our design - private write notices and global write notices. Private write notices record modifications by a specific core during an interval, while global write notices keep records of all modifications from all cores. When a core works on an interval, it works on its private write notices. Private write notices will be merged with global write notices at the release operation. Global write notices can be accessed by any core. During acquire operation, modifications done by other cores can thereby be propagated to the subsequent core.[1] With private write notice, we can avoid creating performance bottleneck caused by centralization.

2) Merging Write Notices: Less number of write notices should reduce the number of inquiries and DMA transfers on acquire operation, while a core goes through all the write notices and apply modifications on shared data. Therefore, merging and reducing number of write notices will reduce its overhead and improve performance. To do so, before creating a write notice, we first check if the range of new write completely or partly overlaps with or is adjacent to any existing write notices. If so, we change the existing notice of interest to include the new write instead of creating a new write notice. In particular, at release stage, we also merge private write notices with the existing global write notices.

V. EXPERIMENTAL RESULTS

A. Experimental Setup

Our experiment platform is TI TMS320C6678 evaluation board [15]. The board has a single C6678 processor and a 2GB DDR3 memory. It is based on TI's KeyStone multi-core architecture, and has eight cores on chip, each of which can run at up to 1.25 GHz. Cores are connected by on-chip teraNet with a bandwidth of 2 Tbps. Each core has its private L1 cache and shares an L2 cache.

[1]Unless noted, write notices mentioned in the rest of the paper refers to the global write notices.

Table I: Benchmarks

Benchmark	Input Size	numIters
Compress	512x512	2048
Laplace	512x512	2048
Lowpass	512x512	2048
Wavelet	4096x4096	8192
MMT	M=4, N=16384, P=64	-
MV	M=4, N=16384, P=1	-
MM	M=4, N=16384, P=64	-
MT	M=512, N=512	8192

We took several commonly-used routines in many numerical or multimedia applications. The benchmarks and their input sizes are described in the first two columns of Table I. MM, MV, and MMT, are matrix-matrix, matrix-vector, and matrix-transposed matrix multiplication, respectively. MT stands for matrix-transpose. All benchmarks are implemented as multi-threaded programs. We divide the computation equally by dividing the output array into equal sub-arrays and making each core be responsible to compute on one of the sub-array, and put a barrier at the end to ensure all cores have finished the computation on its own portion before the final result can be provided as an output.

B. Better Performance than COMIC

Our metric of performance is the reciprocal of execution time. In this experiment, we show performance comparison of our coherence scheme and COMIC, which are normalized to our base line, namely, by disabling cache. As shown in Figure 5, we achieve over 2X performance improvement compared to the COMIC for all benchmarks. Moreover, our approach can achieve on average 12X and up to 20X performance improvement compared to the baseline.

To better understand the individual performance results of the benchmarks, we analyzed the access patterns of benchmarks and categorized their temporal locality and spatial locality as *strong or weak*. We say that a benchmark shows *strong spatial locality*, if more than one contiguous memory locations are accessed in each iteration of the innermost loop. Otherwise the benchmarks are said to show *weak spatial locality*—since it only accesses discontinuous memory locations. We say that a benchmark shows strong temporal locality if any memory location is accessed more than once over time. Otherwise the benchmark shows weak temporal locality, since it accesses memory locations only once. *Compress*, *Laplace* and *Lowpass*, show both strong temporal and spatial locality, since to compute an element in the target array, the program needs to access some surrounding neighbors in the source array, e.g., to calculate b[i][j], the program needs to access a[i-1][j-1], a[i-1][j], a[i-1][j+1], a[i][j-1], a[i][j], a[i][j+1], a[i+1][j-1], a[i+1][j], a[i+1][j+1], and over the time a[i][j] needs to be accessed for the calculation of b[i-1][j-1], b[i-1][j], b[i-1][j+1], b[i][j-1], b[i][j], b[i][j+1], b[i+1][j-1], b[i+1][j], b[i+1][j+1]. Wavelet shows strong temporal locality, but it writes to two discontinuous locations in each iteration, which impairs the spatial locality, and hurts performance as well. Both MV and MMT access a large region of contiguous memory for many times in a loop, showing both relatively strong temporal and spatial locality. Although MM is very similar to MMT in terms of functionality on multiplying two matrices, MM has relatively weak spatial locality. This is because it accesses a matrix column-wise, while the data for the matrix is laid out in memory in row-major order. As a result, MM does not show as much performance improvement as MMT does.

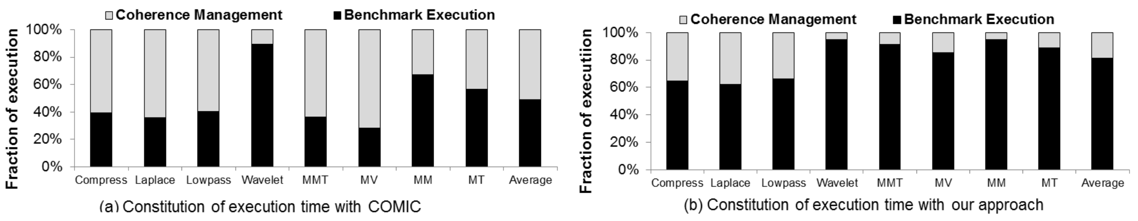


Figure 6: *The comparison of runtime overhead of our approach and COMIC.*

MT shows limited locality of reference. Array elements are accessed in a column-wise manner in the transposed matrix, and each element in both the original and transposed matrix is accessed only once and never again. Arrays are stored in row-major order. However, the result will be the same if arrays are stored in column-major order. We can simply transpose the arrays from row-major order to column-major order in all the benchmarks, and we will get the same result.

Note that the performance of our approach over MMT, MV, MM and MT vary significantly, while the performance of COMIC does not show much variation. This is because for every shared data access, COMIC has to check a dirty bit to find out if the page has been invalidated, whereas our approach does not. As a result, the performance gained from locality of references is compromised in COMIC because of the extra memory accesses introduced may poison the cache.

C. Reduced Runtime Overhead than COMIC

We show other factors that could affect performance in this section. First, note that COMIC dedicates one core to coherence management, so only seven of all the cores were used for actually executing threads. Note that this limitation of being able to execute only one thread on a core is quite a common limitation in several real multi-core architectures, e.g., the IBM Cell processor [23], the 48-core Intel SCC [5], and the TI Keystone architecture [15]—our experimental platform. This is to minimize the overhead of operating system on the cores and to allow extremely power-efficient bare-metal execution. Using less number of threads affects the performance of COMIC. However, that is not the only reason for the worse performance of COMIC. Figure 6 shows the fractions of coherence management and the actual execution of benchmarks in total execution time for both approaches. Overhead comprises all the actions related to coherence management, e.g., comparison of different copies of the same page. As shown in the figure, COMIC takes up to 51% of overall execution time, while our approach takes only 19% on average. By using our write-notice based approach, we successfully avoid expensive twin-page comparison, and eliminate the checking of dirty-bits of pages for every shared data access.

D. More Scalable Overheads with Increasing Computation-to-Communication Ratio

We compare the scalability of our approach to COMIC as the inter-core computation-to-communication ratio increases. To show that, we increase the workload assigned to each core before the synchronization by increasing loop counters in this experiment. By repeating the same work before the barrier, the workload of each core is increased before it needs to communicate with other cores at the barrier, while the amount of time for inter-core communication at the barrier remains the same, since each core modifies the same shared memory locations and thus only needs to propagate the same amount of information to subsequent accessing cores to these locations. Therefore, it reduces the time spent on inter-core communication in the overall running time. The third column of Table I, i.e., *numIters*, shows the number of iterations the workload will be repeated. The bigger $numIters$ is, the lower the weight of inter-core communication in the overall running time. Figure 7 shows the impact of reducing the weight of inter-core communication on the two different approaches. The Y axis shows the performance normalized to the baseline, which disables caches. The figure clearly shows that benchmarks using our approach gets near-linear performance improvement as the number of iterations increases, while their performance suffers from the increasing overhead of COMIC. The overhead of COMIC increases because dirty-bit checking has to be done for every shared data access. Although that bit checking can be done by a cheap modular operation, it accumulates to a large overhead as the number of iterations increases.

Another important observation is that, when the iteration count is small, the overhead of our coherence management stands out more, and our approach performs worse than COMIC. This is because our approach uses write-update protocol. Under such protocol, each core needs to fetch more data than they actually need to. When there is very little computation in each interval (for example, when each thread mostly updates some elements of array with some constants without any calculation), this drawback will become more prominent. However, the overhead of our coherence management gets amortized off as the amount of workload increases, which should be acceptable as the communication should not be dominant most of the time.

VI. Summary

Hardware cache coherence does not scale well as the number of cores. Although processors with non coherent caches are more power-efficient and scalable, they become hard to program. On such processors, manual parallel programming requires significant effort. In this paper, we present a software approach which provides coherence management functions for non-coherent cache multi-core processors to

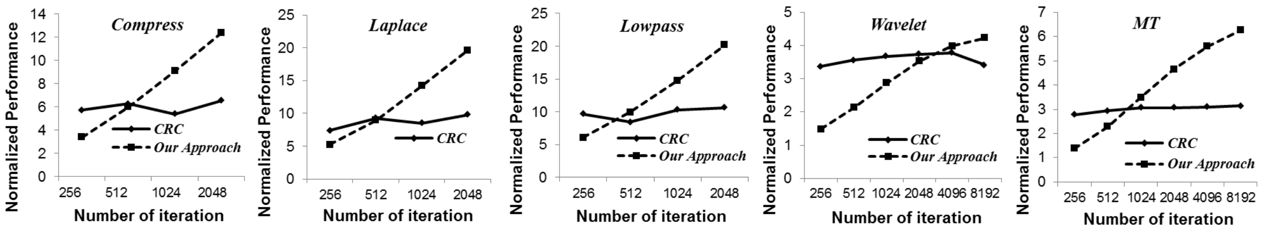


Figure 7: *Compute-intensity is varied by changing the numIters parameter.*

improve programmability. We demonstrate that the benefit of byte-level coherence management on non-coherent cache multi-cores on the avoidance expensive the computation overhead. Experimental results show that our approach performs better than the previous approach due to the abated coherence management overhead. On average, our approach improves performance $12X$ compared to the naive way of disabling cache, and more than $2X$ compared to the most related work.

REFERENCES

[1] G. Bournoutian and A. Orailoglu, "Dynamic, Multi-core Cache Coherence Architecture for Power-sensitive Mobile Processors," in *Proc. of CODES+ISSS*, 2011, pp. 89–98.

[2] B. Choi *et al.*, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *Proc. of PACT*, 2011, pp. 155–166.

[3] Y. Xu, Y. Du, Y. Zhang, and J. Yang, "A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs," in *Proc. of ICS*, 2011, pp. 285–294.

[4] A. Garcia-Guirado, R. Fernandez-Pascual, A. Ros, and J. Garcia, "Energy-Efficient Cache Coherence Protocols in Chip-Multiprocessors for Server Consolidation," in *Proc. of ICPP*, 2011, pp. 51–62.

[5] J. Howard *et al.*, "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, 2011.

[6] L. Kontothanassis and M. Scott, "Software Cache Coherence for Large Scale Multiprocessors," in *Proc. of HPCA*, 1995, pp. 286–295.

[7] P. Stenstrom, "A Survey of Cache Coherence Schemes for Multiprocessors," *Computer*, vol. 23, no. 6, pp. 12–24, 1990.

[8] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-level Shared Memory," *SIGARCH Comput. Archit. News*, vol. 22, no. 2, pp. 325–336, Apr. 1994.

[9] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, "Shasta: a Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory," *SIGPLAN Not.*, vol. 31, no. 9, pp. 174–185, Sep. 1996.

[10] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," in *Proc. of SOSP*, 1991, pp. 152–164.

[11] B. Bershad, M. Zekauskas, and W. Sawdon, "The Midway Distributed Shared Memory System," in *Proc. of Compcon Spring*, feb 1993, pp. 528–537.

[12] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han, "COMIC: a Coherent Shared Memory Interface for Cell BE," in *Proc. of PACT*, 2008, pp. 303–314.

[13] IBM Technical Library, "Cell Broadband Engine Architecture and its First Implementation." [Online]. Available: http://www.ibm.com/developerworks/power/library/pa-cellperf/

[14] AMD, "HPC Processor Comparison," July 2012. [Online]. Available: http://sites.amd.com/us/Documents/49747D_HPC_Processor_Comparison_v3_July2012.pdf

[15] Texas Instruments, "TMS320C6678." [Online]. Available: http://www.ti.com/product/tms320c6678

[16] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *Computer*, vol. 24, no. 8, pp. 52–60, Aug. 1991. [Online]. Available: http://dx.doi.org/10.1109/2.84877

[17] I. Tartalja, *The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions*, V. Milutinovic, Ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997.

[18] J. Protic, M. Tomasevic, and V. Milutinovic, "A survey of distributed shared memory systems," in *Proc. of the Hawaii International Conference on System Sciences*, vol. 1, Jan 1995, pp. 74–84 vol.1.

[19] H. Sandhu, B. Gamsa, and S. Zhou, "The Shared Regions Approach to Software Cache Coherence on Multiprocessors," in *Proc. of PPoPP*, 1993, pp. 229–238.

[20] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas, "Protozoa: Adaptive Granularity Cache Coherence," in *Proc. of ISCA*, 2013, pp. 547–558.

[21] F. Ophelders, M. J. Bekooij, and H. Corporaal, "A Tuneable Software Cache Coherence Protocol for Heterogeneous MP-SoCs," in *Proc. of CODES+ISSS*, 2009, pp. 383–392.

[22] J. Kim, S. Seo, and J. Lee, "An Efficient Software Shared Virtual Memory for the Single-chip Cloud Computer," in *Proc. of APSys*, 2011, pp. 4:1–4:5.

[23] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[24] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors," in *Proc. of ISCA*, 1990, pp. 15–26.