# An Efficient Software Shared Virtual Memory for the Single-chip Cloud Computer*

Junghyun Kim, Sangmin Seo, and Jaejin Lee
School of Computer Science and Engineering
Seoul National University, Seoul 151-744, Korea
{junghyun, sangmin}@aces.snu.ac.kr, jlee@cse.snu.ac.kr

## ABSTRACT

The Single-chip Cloud Computer (SCC) is an experimental processor created by Intel Labs. The SCC is based on a message passing architecture and does not provide any hardware cache coherence mechanism. Software or programmers should take care of coherence and consistency of a shared region between different cores. In this paper, we propose an efficient software shared virtual memory (SVM) for the SCC as an alternative to the cache coherence mechanism and report some preliminary results. Our software SVM is based on the commit-reconcile and fence (CRF) memory model and does not require a complicated SVM protocol between cores. We evaluate the effectiveness of our approach by comparing the software SVM with a cache-coherent NUMA machine using three synthetic micro-benchmark applications and five applications from SPLASH-2. Evaluation result indicates that our approach is promising.

## 1. INTRODUCTION

As the number of cores increases in a chip multiprocessor, the on-chip interconnect becomes a major performance and power bottleneck. It takes up a significant amount of the total power budget. A complicated hardware cache coherence protocol worsens the situation by increasing the amount of messages that go through the interconnect. For this reason, there have been many studies on the relationship between on-chip interconnects and cache coherence protocols for manycores. But, most of them increase the chip design complexity and result in introducing high hardware validation costs. Some recent manycores, such as the Intel Single-chip Cloud Computer (SCC)[4], choose a message passing architecture that does not require hardware cache coherence mechanism although this sacrifices ease of programming.

The SCC experimental processor[4] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. Figure 1 describes the overall organization of the SCC. It has 24 core tiles arranged in a 6x4 array and four DDR3 memory controllers. Each tile consists of two P54C IA cores and 16KB message passing buffer (MPB). Each core has 16KB L1 instruction and data caches and a 256KB L2 cache. MPB is designed for fast communication between cores. Intel provides a customized Linux running on a core and an MPI-style communication library called RCCE[14].

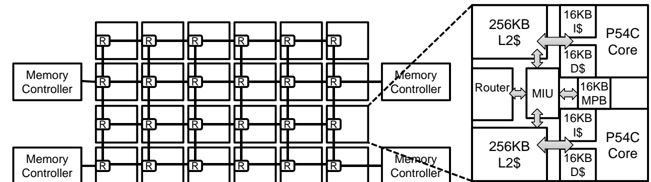


**Figure 1: The organization of the SCC.**

Each memory controller supports up to 16GB DDR3 memory. While this implies that a maximum capacity of 64GB memory is provided by the SCC, each core is able to access only 4GB of memory because it is based on the IA-32 architecture. To overcome this limitation, the SCC allows each core to alter its own memory map. Each core has a lookup table, called LUT, which is a set of configuration registers that map the core's 32-bit physical addresses to the 64GB system memory. Each LUT has 256 entries, and each entry handles a 16MB segment of the core's 4GB physical address space. An LUT entry can point to a 16MB system memory segment, an MPB, or a memory-mapped configuration register. Each core's LUT is initialized during the bootstrap process, but the memory map can be altered by any core at any time. Consequently, the software can freely map any memory region as a shared or private region. The SCC does not provide any cache coherence mechanism. This implies that software or programmers should take care of coherence and consistency of a shared region between different cores.

A software shared virtual memory (SVM) system[1, 5, 7, 8, 9, 10, 11, 13] can be one of the alternatives to the hardware cache coherence mechanism. Software SVMs were originally developed for distributed memory multiprocessors to provide an illusion of a single, globally shared address space for the user. In this paper, we propose an efficient software SVM for the SCC and report some preliminary results. Our software SVM exploits the commit-reconcile and fence (CRF) memory model[12] to guarantee release consistency[3]. Optimizing compiler techniques make our software SVM easy to use. Since the memory consistency model is implemented by software, any memory consistency model can be implemented in our SVM without any hardware sup-

*This work was supported by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) from the National Research Foundation of Korea funded by the Korean government (Ministry of Education, Science and Technology). ICT at Seoul National University provided research facilities for this study.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
APSys 2011, July 11-12, 2011, Shanghai, China
Copyright 2011 ACM X-XXXXX-XX-X/XX/XX ...$10.00.

port if it can be specified by the CRF memory model. We evaluate the effectiveness of our approach by comparing the software SVM with a cache-coherent NUMA machine using three synthetic micro-benchmark applications and five applications from SPLASH-2[15].

## 2. RELATED WORK

There have been many studies done on software SVM systems for distributed memory multiprocessors[1, 5, 7, 8, 9, 10, 11, 13]. Most of them guarantee memory consistency and coherence at the page level. They are based on a page fault handling mechanism to detect pages that are accessed by a processor. They also support multiple writers protocols to avoid page pingponging between processors due to false sharing. These protocols are typically based on maintaining a copy of the original page (e.g., creating a twin) and comparing the modified page to the copy (e.g., making a diff). There are two major differences between our proposal and theirs. First, we do not maintain twins and we do not have any process for making diffs. Based on the CRF memory model, a compiler or programmer identifies the data that should be updated to the main memory or that should be brought from the main memory. Second, our proposal does not require a complicated protocol between processors because the main memory can be configured to be physically shared between different SCC cores by modifying the LUT (however, note that the cores in the SCC do not have a single, globally shared address space). What we need to do is just copying the data between a private memory region and the shared memory region in the SCC according to a much simpler protocol.

Inagaki *et al.* [6] propose a compiler technique to improve the performance of a page-based software SVM system. The compiler generates *write commitments* that are sent to all other nodes to immediately update or invalidate the modified memory region when a synchronization operation is performed. Similar to their approach, our approach is based on a compiler technique. However, thanks to the reconcile operation before a read operation in our approach, the updates do not need to be immediately reflected to other local pages at a synchronization point. Moreover, our approach does not need to update or invalidate the entire page.

## 3. MEMORY CONSISTENCY

In this section, we briefly describe the technique that is used to guarantee memory consistency and coherence in our software SVM.

The CRF memory model is a mechanism-oriented memory model[12]. In addition to a semantic cache, called a *sache*, it has three consistency operations: commit, reconcile, and fence. All load and store operations are executed only on saches locally. These operations cannot access the main memory directly. To update the main memory, programmers or compilers should insert commit or reconcile operations appropriately. The model assumes memory accesses can be reordered as long as data dependences are preserved.

A commit operation updates the local data to the main memory if the address is cached in the sache and its state is dirty. A reconcile operation purges the data in the sache if the address is cached and its state is clean. Fence operations are used to enforce ordering between memory accesses. These commit, reconcile, and fence operations are realized in system calls in our software SVM. Our software SVM defines release consistency (RC)[3] for its memory consistency model.

```
for(i = start; i < end; ++i) {
    a[i] = a[i] + b[i];
    priva += a[i];
}

lock(s);
for(i = 0; i < n; ++i) {
    c[i] += priva;
}
unlock(s);
```

(a)

```
shared(&a, &b, &c);

for(i = start; i < end; ++i) {
    reconcile(&a[i],1,0,sizeof(a[i]));
    reconcile(&b[i],1,0,sizeof(b[i]));
    a[i] = a[i] + b[i];
    priva += a[i];
    commit(&a[i],1,0,sizeof(a[i]));
}

lock(s);
for(i = 0; i < n; ++i) {
    reconcile(&c[i],1,0,sizeof(c[i]));
    c[i] += priva;
    commit(&c[i],1,0,sizeof(c[i]));
}
unlock(s);
```

(b)

```
shared(&a, &b, &c);

reconcile(&a[start],1,0,sizeof(a[start])*(end-start));
reconcile(&b[start],1,0,sizeof(b[start])*(end-start));
for(i = start; i < end; ++i) {
    a[i] = a[i] + b[i];
    priva += a[i];
}

lock(s);
reconcile(&c[0],1,0, sizeof(c[0])*n);
for(i = 0; i < n; ++i) {
    c[i] += priva;
}
commit(&a[start],1,0,sizeof(a[start])*(end-start));
commit(&c[0],1,0,sizeof(c[0])*n);
unlock(s);
```

(c)

**Figure 2: Code translation based on the CRF model to guarantee RC. (a) Original code. (b) After inserting commits and reconciles. (c) The final code.**

To implement RC, the CRF model defines acquire and release operations in RC as follows:

```
release(s)  ≡  commit(*);preFenceW(s);unlock(s);
acquire(s)  ≡  lock(s);postFenceR(s);reconcile(*);
```

To define a release operation in RC, all dirty data in the sache must be updated to the main memory by `commit(*)` before `unlock(s)` is performed. `preFenceW(s)` makes any memory access preceding it to be completed before writing to location `s`. To define an acquire operation in RC, after `lock(s)` has been performed, `postFenceR(s)` makes all load operations to location `s` to be completed before any memory access following it is performed. Then, `reconcile(*)` purges all the clean data in the sache. In our software SVM, `preFenceW(s)` and `postFenceR(s)` are combined with `unlock(s)` and `lock(s)`, respectively. That is, `unlock(s)` and `lock(s)` perform these fence operations in addition to their original function.

We add commit and reconcile system calls to the SCC Linux kernel. When a commit or reconcile operation occurs in the application, the request is sent to the kernel with the following arguments:

- Start address: the address of the first data item to commit or reconcile.

- Number of data items: how many data items to commit or reconcile.
- Stride: the distance between two consecutive data items in bytes.
- Size: the size of a data item in bytes to commit or reconcile.

Consider the code in Figure 2 (a) and assume arrays `a`, `b`, and `c` are shared. To run the code on our software SVM, the compiler or programmer declares the arrays as shared arrays (Figure 2 (b)). Then, a reconcile operation is inserted before every read reference to the shared arrays, and a commit operation is inserted after every write reference to the shared arrays. As an optimization, reconcile and commit operations can be reordered and coalesced according to the rules specified in the CRF model. Since we assume RC, commit operations are moved as close as possible to `unlock(s)` and reconcile operations are moved as close as possible to `lock(s)` (Figure 2 (c)). Then, those operations are coalesced together. Moving commit and reconcile operations as close as possible to synchronization operations increases the chance of coalescing those operations. This significantly reduces the system call overhead.

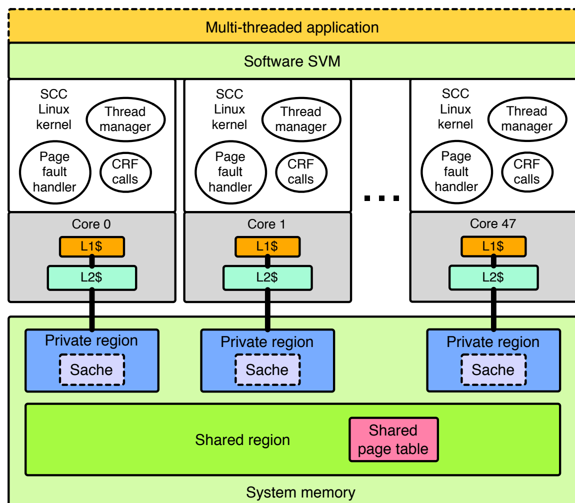


**Figure 3: The organization of the runtime system.**

## 4. RUNTIME SYSTEM

Figure 3 shows the organization of our runtime. SCC Linux is running on each core. The system memory space allocated to each core is divided into two regions: private and shared. The private region is private to the core, and a space in the private region is dedicated to the Linux OS. The remaining space in the private region is allocated to a sache. When a multi-threaded application is running, its shared data is placed in the shared region. The shared region is configured to be uncacheable to L1 and L2 caches by the OS. We modify the SCC linux kernel and add some features (e.g., CRF system calls) to support executing multi-threaded applications. The kernel on each core maintains two page tables: a private page table (PPT) and a shared page table (SPT). The PPT is private to each core and is similar to the page table found in a conventional virtual memory system. It maps each core's virtual address to a physical address in the sache. The SPT is shared between

| | LUT # | Physical address | Contents | |
|---|---|---|---|---|
| | | ⋮ | | |
| System configuration registers (24 segments) | 247 | 0xF7000000 | Configure register - tile 23 | |
| | | ⋮ | | |
| | 224 | 0xE0000000 | Configure register - tile 00 | |
| | | ⋮ | | |
| MPBs (24 segments) | 215 | 0xD7000000 | MPB in tile 23 | |
| | | ⋮ | | |
| | 192 | 0xC0000000 | MPB in tile 00 | |
| | | ⋮ | | |
| Shared region (68 segments, 1088MB) | 107 | 0x7F000000 | System memory address | |
| | | ⋮ | | |
| | 40 | 0x28000000 | System memory address | |
| Private region (40 segments, 640MB) | | ⋮ | | Sache (320MB) |
| | 20 | 0x14000000 | System memory address | |
| | | ⋮ | | For OS kernel (320MB) |
| | 0 | 0x00000000 | System memory address | |

**Figure 4: The LUT for an SCC core.**

all cores and maps each core's virtual address to a physical address in the shared region. A space in the shared region is allocated to the SPT itself.

**LUT setting.** Figure 4 shows the LUT used by each SCC core in our software SVM. The gray area in the LUT shows the mapping between each core's physical address space to the system memory address space for the private and shared regions. As a result of the LUT setting, each core sees the same system memory segment with the same LUT index in the shared region and a different system memory segment with the same LUT index in the private region. The set of system memory segments allocated to the private region of a core is disjoint from that of another core.

**Thread manager.** When we run a multi-threaded application on top of our software SVM, we make each core run the same copy of the application. When the system function `exec` is invoked, the kernel checks whether an environment variable `SCC_SVM` is set. If so, this indicates that the application will be running on our software SVM. Then, only core 0 (say, master) continues its execution. Other cores are waiting for a wake-up event from the master. When the master creates a thread (e.g., by calling `pthread_create`), it sends a wake-up message to a waiting core in a round-robin manner. The message contains information about the address of the function to be executed, arguments of the function, and the stack pointer, to run the created thread on the target core. Then, the woken-up core starts executing the designated function. This process is managed by the thread manager added to the SCC Linux kernel. The thread manager provides POSIX thread library routines.

**Page fault handler.** We also modify the page-fault handler in the SCC Linux kernel. To explain the function of the modified page-fault handler, consider the scenario shown in Figure 5. Assume that core i is the first core that accesses a shared page $p$ with the virtual address $VA_p$ among all cores. In turn, core j accesses $p$ for its first time with $VA_p$. Since the software SVM provides a single, shared address space between all the cores, each core has the same virtual address for the same shared page.

When core i accesses the shared page $p$, a page fault occurs. The first part of the page fault handling process is similar to that of the conventional process and allocates a page frame $p_i$ in its sache (❶). Then, the handler initializes $p_i$ (e.g., $p$ is a page in .bss section). Following this, the handler looks up the SPT with $VA_p$ after obtaining the page lock for the SPT entry of $VA_p$. The SPT entry for $VA_p$ is

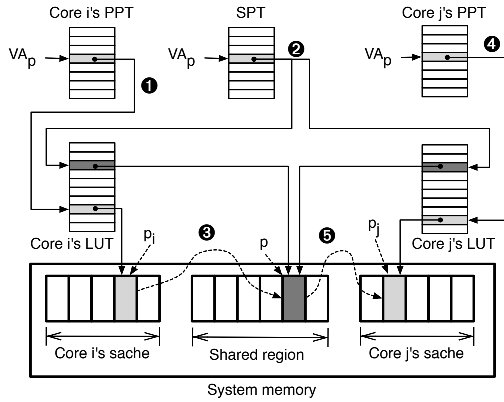


**Figure 5: Page allocation.**

invalid because $p$ is accessed for the first time. The handler sets up the SPT entry to point to the LUT index of $p$ (❷). Then, it copies $p_i$ to $p$ (❸) and releases the page lock. Core i starts to access the private copy $p_i$ of $p$ in its sache.

Now, the page-fault handler of core j acquires the page lock. Since the SPT entry of $VA_p$ already points to the LUT index that is associated with $p$, after allocating the page frame $p_j$ in core j's sache (❹), the handler copies $p$ to $p_j$ (❺). Then, core j starts to access the private copy $p_j$ of $p$ in its sache.

**Table 1: Applications Used**

| Code | Problem Size |
|---|---|
| FFT | 4M points |
| LU/C | 4096x4096 matrix, 16x16 blocks |
| LU/N | 4096x4096 matrix, 16x16 blocks |
| RADIX | 16M integers, radix 1024 |
| OCEAN/C | 1026x1026 ocean |

**Table 2: System Configurations**

| Cache-coherent NUMA Machine (cc-NUMA) | |
|---|---|
| CPU | AMD Opteron 6174 |
| Core frequency | 2.2GHz |
| # of sockets | 4 |
| Cores | 12 per Socket, total 48 cores |
| L1 I-cache | 64KB per core |
| L1 D-cache | 64KB per core |
| L2 Cache | 512KB per core |
| L3 Cache | 12MB per socket |
| Main memory | 96GB |
| Compiler | GCC 4.1.2 |
| Single-Chip Cloud Computer (SCC) | |
| Core type | P54C IA core |
| Core frequency | 533Mhz |
| # of Cores | 48 |
| L1 I-cache | 16KB |
| L1 D-ache | 16KB |
| L2 Cache | 256KB |
| Main memory | 32GB |
| Compiler | GCC 3.4.5 |

# 5. EVALUATION

In this section, we evaluate our software SVM by comparing its performance with a cache-coherent NUMA machine (cc-NUMA).

## 5.1 Methodology

**Machines.** The architecture configurations of the SCC and the cc-NUMA machine are described in Table 2. The cc-NUMA machine has four 12-core AMD Opteron processors in a single system. It is supported by a broadcast-based hardware cache coherence protocol[2].

**Benchmark applications.** In addition to three synthetic micro-benchmark applications: FalseSharing, NoFalseSharing, and LocalDataAccess, we use five applications from the SPLASH-2 benchmark suite[15] for the evaluation. The problem size of each SPLASH-2 application is shown in Table 1. FalseSharing is a program that increments each element of an array in a loop with many iterations. The size of an array element is a single byte. The array is equally divided into $c{\cdot}N$ chunks, where $N$ is the number of threads and $c$ is an integer greater than or equal to 1. Each thread is in charge of $c$ chunks and the chunks are distributed cyclicly to each thread. The chunk size is smaller than the cc-NUMA's L2 cache line size (64 bytes). Thus, a significant amount of false sharing occurs in FalseSharing. NoFalseSharing is a program that is similar to FalseSharing, but the chunk size is exactly the same as the cc-NUMA's L2 cache line size and each cache block is aligned to a cache line boundary. LocalDataAccess is a program that does not cause any L1 cache misses but cold misses. Its data access always hits in the L1 cache. We would like to observe maximal scalability with LocalDataAccess.

**Software SVM for the SCC.** We implement the runtime for the software SVM by modifying the SCC Linux kernel. We insert commit and reconcile operations manually into the applications and optimize them by hand according to the rules specified in the CRF memory model.

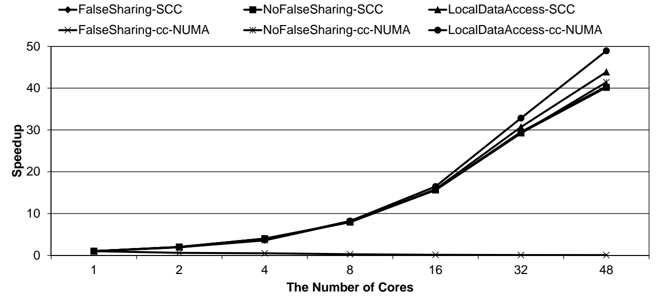


**Figure 6: Speedup of the micro-benchmark applications.**

## 5.2 Result

Figure 6 shows the evaluation result with the three micro-benchmark applications. The speedup is obtained over a single core. For FalseSharing, our software SVM does not suffer from false sharing as the number of cores increases while cc-NUMA suffers from heavy false sharing. Even though pages are falsely shared in our software SVM for FalseSharing and NoFalseSharing, after an SCC core obtains a page in its sache, all accesses to the page occur in the sache, and the updates are committed altogether later to the system memory without any diff process. Interestingly, false sharing is not a source of the scalability bottleneck in our approach.

When we see the speedup of LocalDataAccess, both of the SCC and cc-NUMA scale almost linearly. When the number of cores increases (e.g., 32 and 48), the speedup of the SCC is a little bit worse than that of the cc-NUMA. This is due to the overhead of looking up the SPT (e.g., page lock contention and uncached system memory access). In addition, the overhead of copying pages from the system memory to the sache (e.g., uncached system memory access) takes more portion in the total execution time as the number of cores increases.

Figure 7 shows the evaluation result with the five SPLASH-2 benchmark applications. FFT, RADIX, and OCEAN/C

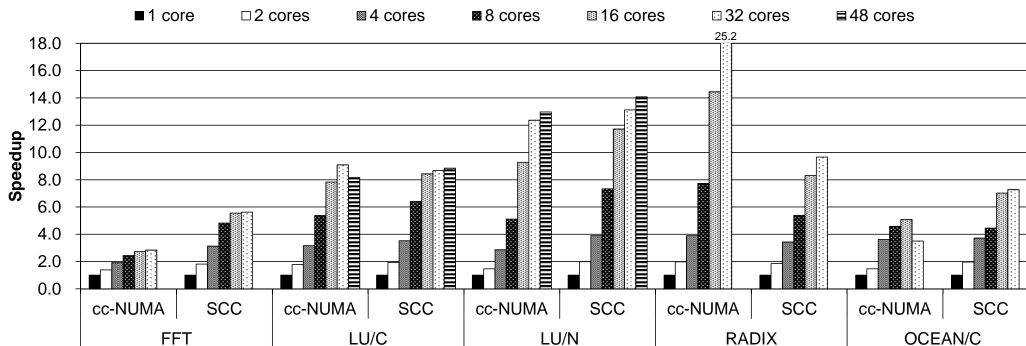


**Figure 7: Speedup of the SPLASH-2 benchmark applications.**

do not have the result for 48 cores because these applications require the number of cores to be a power of two. For all applications but RADIX, the SCC with our software SVM shows better scalability than the cc-NUMA machine. As the number of cores increases, the speedup also increases for the SCC. However, the speedup with 48 cores in cc-NUMA is often worse than that with 32 cores. This is due to false sharing and manifests in the speedup of LU/C and LU/N with 48 cores because LU/N's data accesses are optimized to avoid false sharing.

RADIX executes many spin-wait loops. A flag variable is read repeatedly in the spin-wait loop. In our software SVM, implementing a spin-wait loop requires a reconcile operation to be placed in the loop for the flag variable. Consequently, frequent updates to the flag variable in the sache through the reconcile system call incur a significant overhead. This can be solved with either hardware support or by modifying the application with a post-wait synchronization mechanism.

# 6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose an efficient software SVM for the SCC. It is based on the CRF memory model and guarantees release consistency. Legacy multi-threaded applications can be run on top of the software SVM without any modification. Due to the CRF memory model and compiler support, we do not need to implement a complicated multiple writer protocol that manages the process of creating twins and making diffs. This makes the runtime much simpler. Since the memory consistency model is implemented by software, any memory consistency can be implemented in our software SVM if it can be specified by the CRF model. Based on the preliminary evaluation result that compares our software SVM with a cc-NUMA machine, we foresee such a software SVM can be an alternative to a hardware cache coherence protocol for the SCC-like manycore architectures. As future work, we plan to develop a compiler and its techniques for optimizing commit and reconcile operations based on their properties. In addition, we plan to evaluate our software SVM with more multi-threaded applications.

# 7. REFERENCES

[1] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *SOSP '91: Proceedings of the thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[2] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2):16–29, 2010.

[3] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, 1990.

[4] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom1, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam2, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *ISSCC' 10: Proceedings of the 57th International Solid-State Circuits Conference, February 2010*, 2010.

[5] L. Iftode and J. P. Singh. Shared virtual memory: progress and challenges. *Proceedings of the IEEE*, 87(3):498–507, March 1999.

[6] T. Inagaki, J. Niwa, T. Matsumoto, and K. Hiraki. Supporting software distributed shared memory with an optimizing compiler. In *Proceedings of the 1998 International Conference on Parallel Processing*, ICPP '98, 1998.

[7] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, January 1994.

[8] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA'92: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[9] J. Lee, J. Lee, S. Seo, J. Kim, S. Kim, and Z. Sura. COMIC++: A Software SVM System for Heterogeneous Multicore Accelerator Clusters. In *HPCA'10: Proceedings of the 15th International Symposium on High Performance Computer Architecture*, January 2010.

[10] 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 *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 303–314, 2008.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[12] X. Shen, Arvind, and L. Rudolph. Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 150–161, 1999.

[13] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-write Network. In *SOSP '97: Proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, pages 170–183, October 1997.

[14] R. F. van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Operating Systems Review*, 45(1):73–83, February 2011.

[15] 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 Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, 1995.