Rhymes: A Shared Virtual Memory System for Non-Coherent Tiled Many-Core Architectures

King Tin Lam*, Jinghao Shi*, Dominic Hung*, Cho-Li Wang*, Zhiquan Lai*, Wangbin Zhu†, Youliang Yan†
*Department of Computer Science, The University of Hong Kong, Hong Kong, China
{ktlam, jhshi, chdhung, clwang, zqlai}@cs.hku.hk
†Huawei Technologies Co., Ltd., Shenzhen, China
{zhuwangbin, yanyouliang}@huawei.com

***Abstract*—The rising core count per processor is pushing chip complexity to a level that hardware-based cache coherency protocols become too hard and costly to scale someday. We need new designs of many-core hardware and software other than traditional technologies to keep up with the ever-increasing scalability demands. A cluster-on-chip architecture, as exemplified by the Intel Single-chip Cloud Computer (SCC), promotes a software-oriented approach instead of hardware support to implementing shared memory coherence. This paper presents a shared virtual memory (SVM) system, dubbed Rhymes, tailored to new processor kinds of non-coherent and hybrid memory architectures. Rhymes features a two-way cache coherence protocol to enforce release consistency for pages allocated in shared physical memory (SPM) and scope consistency for pages in per-core private memory. It also supports page remapping on a per-core basis to boost data locality. We implement and test Rhymes on the SCC port of the Barrelfish OS. Experimental results show that our SVM outperforms the pure SPM approach used by Intel's software managed coherence (SMC) library by up to 12 times through improved cache utilization for applications with strong data reuse patterns.**

Keywords*—*Cache coherence; Software managed coherence; Non-coherent many-core architectures;

I. Introduction

Processor architectures have undergone seismic changes in recent years with a bifurcated roadmap to manycore: one goes for coprocessors like GPGPUs; the other keeps increasing core counts of general-purpose CPUs in a Moore's Law pattern. With conformity to accustomed instruction sets, the latter is naturally more welcomed by software architects. If it goes on, 2020 and beyond would be the time that 1000-core CPUs become commonplace. It is however a great challenge to software developers marching to such many cores. Operating systems, parallel programming tools and support infrastructures for efficiently driving so many cores are hard to make. Harnessing parallelism of 1,000 cores in a chip does require a radical rethink to address at least two fundamental problems.

The long-existing "memory wall" problem, first described by Wulf and McKee [1], will get intensified on a many-core CPU. Current memory architectures do not keep up with the scale of hundreds, let alone a thousand, of cores. Entering the 1000-core era, imbalanced growth rates of core count and off-chip memory bandwidth will narrow the external bandwidth available to each core. The problem is compounded by big data workloads that impose high stress on the off-chip DRAM. The second roadblock is the "coherency wall" beyond which the protocol overhead of enforcing hardware cache coherency exceeds the value of adding cores. These problems are rooted in the limitations of current hardware designs. Consequently, researchers are exploring *tiled* architectures to incorporate more cores by two game-changing ideas: replacing the bus/crossbar interconnect with a fast on-chip network and forgoing hardware cache coherency.

The Intel Single-chip Cloud Computer (SCC) is a 48-core research processor epitomizing this architectural shift from traditional ccNUMAs to a non-coherent, "cluster-on-chip" architecture. Each core is assigned its own private memory and runs a separate OS instance. As a perfect match with the SCC, a more scalable OS design approach—*multikernel*—has been proposed and realized into the Barrelfish OS [2]. All cores (OS instances) can access shared memory as usual despite the fact that cache coherence is then ensured by software. The SCC exposes programmable on-chip memory, called *message passing buffer (MPB)*, to software for fast inter-core communication such as coherence maintenance. The research problem is then about how to exploit the complex memory hierarchy—on-chip vs. off-chip memory and private vs. shared memory—efficiently and easily.

This paper describes our research effort to design and implement *Rhymes (Runtime with HYbrid MEmory Sharing)*, a shared virtual memory (SVM) system, on top of Barrelfish to achieve the goal. The emergence of recent non-coherent many-core architectures such as Intel SCC has revived the topic of SVM, reassessing what the right memory consistency model and software optimizations (affinity, profiling, etc) are suited to many-core CPUs. Despite a myriad of SVMs built for clusters of workstations, many old design principles have outlived their usefulness for a many-core SVM. In the past, due to long network latencies, the chief protocol design goal was to minimize the number of message roundtrips between the loosely-coupled cluster nodes, say by caching more data in node-local memory. But now, today's experimental tiled chips can already attain inter-core latency as low as tens of clock cycles. This implies (on-chip) message passing between two processes is even faster than local DRAM access. The off-chip memory bandwidth wall is likely to widen the gap. Thus, the entire SVM design philosophy for many-core is somewhat the reverse for the traditional. Instead of message round-trips, it is now of higher priority to minimize DRAM accesses (for performance) and next the total on-chip network traffic (for power saving). Our protocol design and implementation, detailed in Section II and Section III-C respectively, strive

to maintain maximal on-chip data locality, minimal metadata footprint and coherence traffic accordingly.

To the best of our knowledge, this is the first work to build SVM support on Barrelfish. The novelty of our work lies in a *two-way* cache coherence protocol (Section III) implementing a hybrid memory model which combines the architectural benefits of software *distributed shared memory (DSM)* and *shared physical memory (SPM)* approaches to share virtual memory pages efficiently. Each page can be mapped to DSM mode, SPM mode or MPB mode (cached in on-tile MPB for fast shared access) to meet its own consistency and locality needs. In other words, pages can be placed in different types of physical memory (private DRAM, shared DRAM or on-chip MPB), cached differently and have their coherence maintained at selectable granularity (cache lines vs. pages) using different protocols (scope- vs. release-consistent) that run in harmony. Pages can be mode-switched, or remapped, dynamically to respect memory access patterns on a per-page-per-core basis. In effect, Rhymes boosts data locality by maximizing the utilization of on-chip caches and programmable buffers. Currently, the SVM prototype relies on user annotations to trigger page mode switching. But our ultimate goal is to help users gain performance transparently through automatic page remapping guided by a memory access profiler which tracks shared access frequency through regular memory protections. We evaluate the system with a range of benchmarks, including Graph 500 [3] and Malstone [4]. Experimental results (Section IV) show that our SVM, with frequently accessed shared data marked in the application code (as if they were detected by a profiler), can win over the pure SPM approach used by Intel's SMC by up to 12 times due to cache effects from page remapping—a speedup substantial enough to compensate all online profiling costs (generally about 10% of the runtime).

Although SCC and Barrelfish, where our SVM is built atop, are just research prototypes that are presently no match for another x86-based many-core architecture—Intel's MIC (Xeon Phi running embedded Linux OS)—in popularity and many aspects, we hold an original view that the use of a non-coherent architecture with programmable on-chip memory plus a multikernel OS is a big step forward in designing a highly scalable many-core system of a large scale per chip. Besides saving the ever-increasing protocol verification complexity and cost, saving of energy (wasted in cache snooping) is an secondary advantage. Previous work [5] has shown that only about 10% of the application memory references actually require cache coherence tracking. Applications can have most data RO-shared and few RW-shared; hardware coherence thus could overkill and also lead to waste of energy (possibly up to 40% of the total cache power [6], [7]). A further merit, which we consider the biggest, of an SCC-like architecture is its flexibility offered to the upper (software) level. With *software-defined networking (SDN)* being a new approach to designing, building and managing networks, we have the sense that future networks-on-chip might also have some SDN functions that can be used for implementing *software-defined coherence (SDC)*—a new concept. Processor chips that eternally bypass the hardware coherence wall and allow flexible SDC-based protocols defined on a per-application basis may become popular one day. In terms of generic contributions of our work that are independent of SCC or Barrelfish, our proposed software techniques—hybridized memory model, two-way protocol and page mode switching (or remapping)—are particularly useful to enhance the system performance when it comes to the need of a chip with a massive scale of cores and of an SDC-based approach in the future.

II. System Design of Rhymes SVM

Figure 1 depicts the entire system hierarchy (Rhymes atop Barrelfish on SCC). Summarized in one sentence, Rhymes is a page-based, scope-consistent[1] SVM system implemented in user space, using locks and a home-based[2], invalidate-based, multiple-writer protocol to synchronize shared memory updates. The use of relaxed memory consistency models and "traffic-thrifty" coherence protocols is a critical factor affecting the network communication cost and hence the system performance. To design a high-speed, scalable SVM system atop the Barrelfish-SCC stack, we propose a novel hybrid memory model to maintain the coherence of shared data at page level. When shared variables are allocated, they can be mapped into different virtual memory page types.

The SCC features a new memory type known as the *message-passing buffer type (MPBT)* for handling data allocated in the on-tile MPB (16KB SRAM on each tile), together with an additional instruction—CL1INVMB—to invalidate all the L1 cache lines that are tagged with MPBT[3]. The memory address space covered by all the MPBs (a total of 386KB SRAM) are accessible by all cores, and are cacheable in L1 but not L2. The memory model used for synchronizing a page depends on the page type (MPBT vs. non-MPBT). Logically, each page is given a sharing mode:

- *Shared Physical Memory (SPM) mode*: In SPM mode, cache means "cache lines" in L1. For each virtual memory page, there is only one data copy allocated in the shared DRAM. This is a centralized approach bearing much similarity to traditional shared-memory systems (SMP or multi-core) except that the cache coherence of the shared memory now becomes software-managed. SPM-mapped shared pages are located in the shared DRAM of SCC and synchronized based on release consistency.
- *Distributed Shared Memory (DSM) mode*: In DSM mode, cache means "cached pages" in private memory. For each virtual memory page, there exist multiple cached copies of the page. Each cached copy is allocated in the private DRAM of each core accessing the page. The cached copy is replicated to private memory from a golden copy allocated in the shared DRAM, which is referred to as the *home copy* of the page. The memory coherence of DSM-mapped pages is maintained using a scope consistency protocol.

The two-mode protocol semantics are detailed in Section III. Here, we illustrate the global virtual address space provided

[1]The data consistency model guaranteed for the programmer is scope consistency (ScC), but the underlying mechanism to realize a scope-consistent view of the virtual memory may be stronger than ScC when the pages are mapped to SPM mode

[2]SCC provides shared DRAM, so we can simply store shared pages' home copies there, and every core maps each home at the same virtual address.

[3]Data not allocated in MPB can also be tagged with MPBT by setting a bit in the page table; the purpose is to bypass L2 cache and exploit the write-combine buffer.

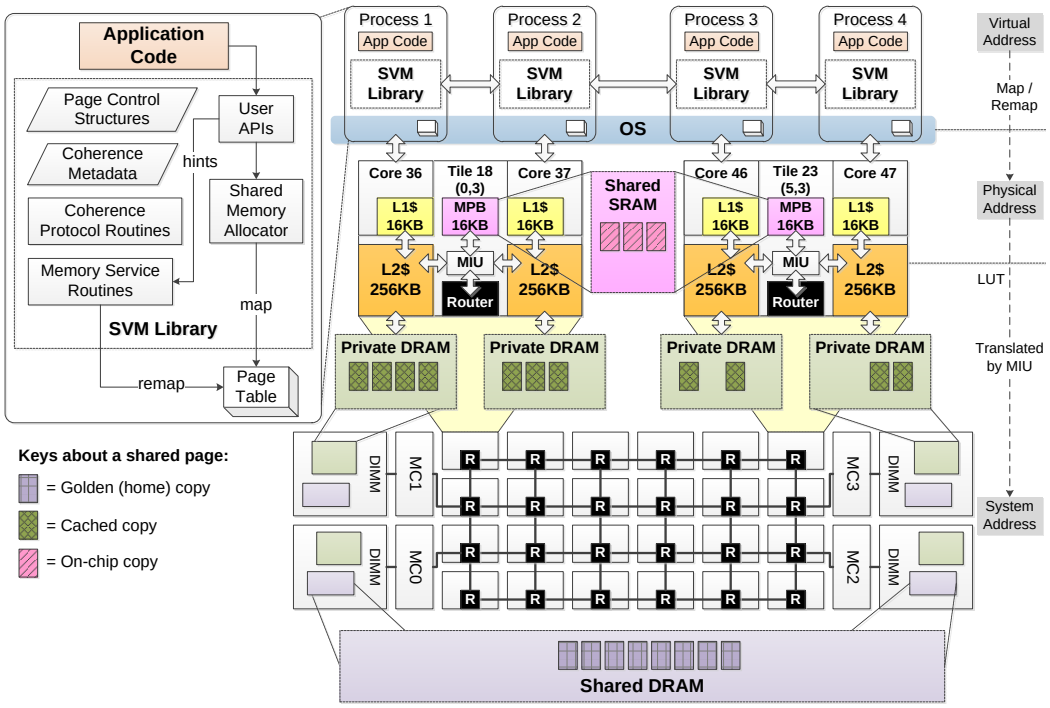


Fig. 1. System architecture of Rhymes SVM on the Intel SCC

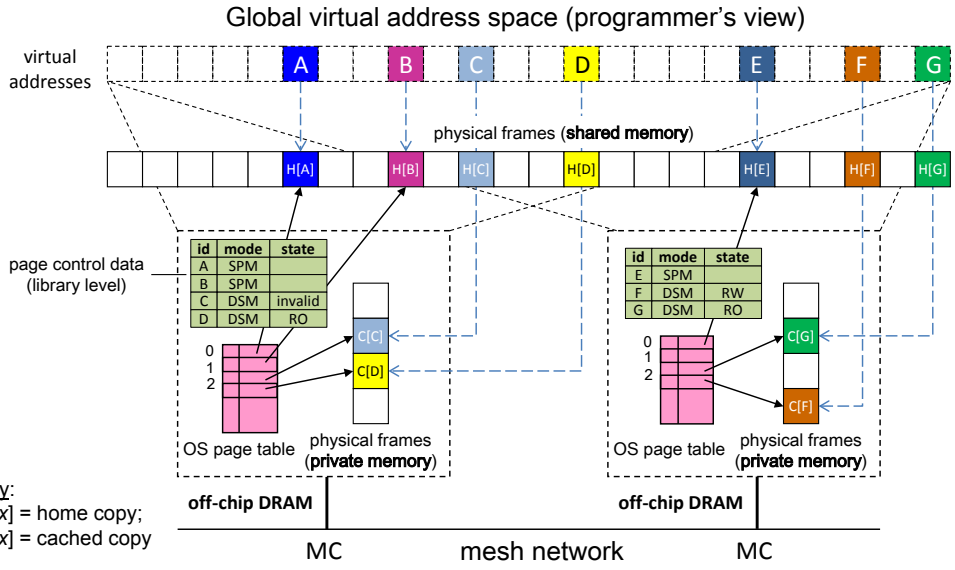


Fig. 2. A global address space virtualized by Rhymes

by our SVM in Fig. 2. Private memory of each core acts like a cache of the shared memory. Programmer's view of a single memory space is seamlessly provisioned through the page-fault mechanism. Coherence metadata such as page mode and state (page control data) are maintained internally in the SVM.

III. Two-Way Cache Coherence Protocol

Based on SCC-supported invalidate and flush operations, the SPM mode basically follows an eager release consistency model. The merit of adopting scope consistency for the DSM mode is to keep the number of write notices and invalidations minimal, taking advantage of the association between the lock id and the shared variables. As we mentioned, the system supports dynamic page remapping from one memory type to another. But this can cause consistency problems unless the system has protocol support for both DSM- and SPM-mapped pages running in harmony. The handling of two pages mapped to DSM mode and SPM mode respectively will be different and never redundant. Figure 3 illustrates the semantics and overheads of the two-way protocol. In the code snippet, the array a of 1024 integers (exactly one page denoted by A) is being read and written in the three loops. Below we detail the protocol actions the system applies to each cache line/page, based on the page mode assigned.

Code	SPM Mode	DSM Mode
rhy_lock(1);	invalidate MPBT cache lines (CL1INVMB)	invalidate cached pages per write notices received
for (i=0; i<1024; i++) sum +=a[i];	*overhead-free* *(but all cold misses)*	read fault trapped on 1st read: memcpy **H[A]** to **C[A]**; remap VA(A) to C[A]; set C[A] to read-only; resume access; (but all cold misses)

for (i=0; i<1024; i++)
for (j=0; j<1024; j++)
c[i] += a[j];
	overhead-free
	(L1 cache hits for a)
		overhead-free
		(L1 and L2 cache hits for a)

for (i=0; i<1024; i++)
for (j=0; j<1024; j++)
a[i] += b[j];
	write through (to WCB) 1024^2 times;
	flush to DRAM 131,072 times
	($1024^2 \times 4 / 32$)
		write fault trapped on 1st write:
		memcpy **C[A]** to **T[A]**; add A to dirty list;
		set C[A] to read-write; resume access
		overhead-free for all the (1024^2 -1) writes to a

rhy_unlock(1);	flush WCB	flush diffs of each dirty page: for each trunk *k* in A, if (**C[A]**[k] xor **T[A]**[k]) **H[A]**[k] = C[A][k]
	#DRAM access (due to a): **128 (cold misses) + 131,072**	**#DRAM access (due to a): 128 (cold misses) +** **128 x 7 = 896 (worst case) or 128 x 4 = 512 (best case)**

keys: H[] = home copy; C[] = cached copy; T[] = twin copy; VA() = virtual address of; assume sum, b, c are local variables;
bold underlined (red-colored) to represent probable off-chip access to DRAM

Fig. 3. Illustration and cost analysis of the two-way coherence protocol

A. SPM Mode

Our coherence mechanism for the SPM Mode is the same as that in Intel SMC and MetalSVM [8]. The PMB bit (i.e. MPBT) of the page table entry is set to bypass L2 cache, i.e. the shared data are cacheable in L1 only. Recall that SCC L1 cache can be made write-back or write-through. For write-back cache, update will be lost when multiple writers write different parts of the same cache line. To avoid this, we follow Intel SMC's way to make L1 cache write-through by setting the PWT flag in the page table for each page frame.

For writes on MPBT-tagged cache lines, the writing through will not go to the main memory directly but to the write combining buffer (WCB) first. So what we need to flush is the WCB[4] for ensuring that the modifications are written down to the off-die shared memory. We add a dummy write to the last line of the MPB across synchronization points such as lock acquire to flush the WCB. When entering a critical section (no matter which lock id), the lock acquire routine will invalidate the cache entries in L1 by executing the CL1INVMB instruction. This invalidation affects all MPBT-tagged cache lines for all the page frames which are allocated in SPM page mode. When leaving a critical section, the SVM lock release routine will flush the WCB to reflect the updates to main memory. In contrast to the original LRC model [9], this mechanism will write down also modifications which are outside the current critical section. This will invalidate cached data more than necessary, lowering cache hits. But the advantage is that almost no software coherence maintenance overheads (e.g. write detection) are generated. Obvious disadvantages of this protocol are that L2 cache is bypassed for shared data, and the writing through to DRAM may put heavy stress on the off-chip memory bus, despite the help of the WCB, as we analyzed in Fig. 3.

[4]WCB is flushed in 3 cases: 1) 32 bytes filled in; 2) next write to a non-consecutive MPBT addr. (not in the same cache line); 3) next write to non-MPBT addr.

B. DSM Mode

To address the drawbacks of the SPM protocol, the DSM protocol works in a reverse philosophy. It traps writes to shared memory using memory protection and synchronizes per-core private cache copies of shared pages across synchronization points. The associated page-level false sharing issues are resolved by using the classical twinning and diffing technique [9]. We illustrate this protocol using Fig. 3. In our system, every page begins with SPM mode. Some pages switch to DSM mode, assumed a profiler (or programmer via API) varies their page mappings system-wide. Suppose the virtual page A [virtual address = VA(A)] in the example was originally mapped to a physical frame H[A] in the shared DRAM. This page frame is the golden (home) copy. After A has switched to DSM mode, the original SPM-mapped frame will be unmapped. The unmap system call is provided by Barrelfish and its internal implementation has included invalidating TLB using the INVLPG instruction. So the OS page table or TLB no longer has the VA $\rightarrow$ PA mapping for page A. However, its page control record stored at the SVM library level won't be deleted, and the record has stored the frame address (or capability reference, capref, in Barrelfish) of the golden copy. Hence, the SVM still knows where to locate the golden-copy physical frame for memory copying when necessary. The page control record is updated to set page state to read-only and set mode to DSM (a.k.a. private mode).

Later, when a core tries to read/write using VA(A), a page fault will be generated since VA(A) is no longer mapped to any frame. The faulting core traps this exception and call the SVM's page fault handler to locate the golden copy of A and replicate it from shared DRAM to a newly allocated frame that we call the *cached copy*, C[A], in private DRAM of the core. Then VA(A) is remapped to C[A]; access is resumed on C[A] instead of on H[A]. Access to C[A] has advantages over H[A] because this cached copy can exploit the L2 cache by tagging it as non-MPBT. So concerning the second for-loop in the example, DSM mode always performs better than SPM mode. At the cost of page faulting, replication

and remapping upon the first read/write in a critical section, we gain a better guarantee of on-chip access speed for probably a lot of accesses. DSM mode is suitable for pages that have strong reuse patterns.

In a critical section, upon the first write on a shared page (i.e. write fault), the faulting core will perform the following:

1) Make a page replication from the golden copy (shared DRAM) to a cache copy (private DRAM) if the page is of read-only state and of private mode but the faulting core does not have a cached copy mapped in its private memory, as the page control record indicates.
2) Modify the page access control: change read-only to read-write state.
3) Create a twin copy from the local copy of the page in its private memory.
4) Add the page id of the page being written to a dirty page list.

One important bridge between the SPM- and DSM-mode protocols is the introduction of memory protection of the golden copy of a shared page when it begins being remapped to a private copy by any core. Our system is indeed able to support the flexibility that different cores use different mappings for the same page, provided that some conflicting combinations are avoided by the design of the library APIs. One combination our system allows is this: core 1 maps page A to DSM mode (non-MPBT, write-back cache, private memory) whilst core 2 maps A to SPM mode (MPBT, write-through, shared memory). So in the runtime, they are accessing different page replicas. To support this functionality seamlessly through our coherence protocol, a remapping of the golden page to read-only MPBT is required. With this read-only protection, any writer of the page in SPM mode will be trapped by the write fault handler when it updates the golden page. The faulting core will add this page to the dirty list. Upon unlock, it will generate a write notice of this page into the write notice list of the lock id used to guard this update. Next time, when a core holding a private copy of this page acquires this lock, it will get the write notice and know that the golden copy has been changed by someone else. So it must invalidate the local cached copy. This is the only channel for SPM-mode updates to propagate to the cores running the DSM-mode protocol.

C. Implementation

Our SVM implementation is based on the 2012-06-06 release of Barrelfish. We modified the Barrelfish rck code to hijack the whole MPB space which was originally managed by the OS. In other words, it is then up to our library's decision about how to use all the 8 KB per-core MPB space. The virtual memory subsystem of Barrelfish is very different from Linux. Barrelfish manages resources, including physical page frames, using capabilities. In user space, Barrelfish borrows the BSD concept of keeping the hardware-dependent bits of memory management (e.g. inserting mappings) in a physical map (pmap) and providing a more general interface built on virtual memory regions (vregions) and virtual address spaces (vspaces). Having this platform-independent API keeps user-level code and new hardware-specific optimizations portable. We call `pmap→f.modify_flags` to change the various bits of a page table entry (PTE). On SCC, a new status bit—PMB (bit position 7)—has been added to the TLB entries of the data cache. The PMB bit together with the PCD and PWT bits (cache-disable and write-through bits) determine the memory type of the data. For example, encountering `VREGION_FLAGS_MPB` (in Barrelfish speak) on the SCC port, the Barrelfish kernel will set both the PMB and PWT bits of the PTE to true, and the memory region is mapped as MPBT type. We provide custom malloc, lock/unlock and barrier routines for programmers as if they were programming on a cache-coherent shared-memory machine using POSIX mutex and barrier functions.

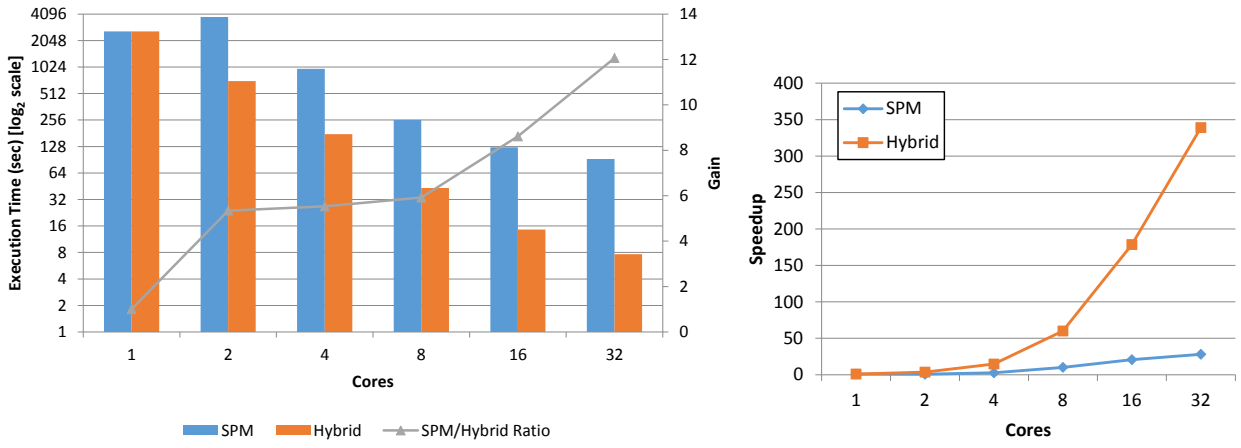
IV. Performance Evaluation

Our performance evaluations are conducted on an Intel SCC machine with 32GB RAM and Barrelfish installed. We run all experiments using a static frequency setting: 533MHz (tile) and 800MHz (mesh). For convenience, we use the terminology of *Hybrid* mode to mean the system runs with both SPM and DSM page modes while the *SPM* mode means all pages are mapped to SPM page mode.

A. Bucket Sort

We ported a bucket sort kernel similar to that in Terasort whose program logic is essentially a parallel bucket sort with three major steps: 1) spawn the data generator in parallel on each compute node; 2) perform sort locally on the data assigned to each node; 3) merge results onto global file system. Our experiment uses a problem size of 8,388,608 keys, 256 buckets per core and applies bubble sort as the per-bucket local sorting algorithm. In step 1, the program only performs sequential access. So we can set both the original array and bucket array as SPM mode. Before step 2, we set only the bucket array as DSM mode, since we would sort on the bucket array (intense access). In step 3, we can either set the bucket array back to SPM, or just leave it alone, since we only read the bucket array in step 3 (we chose the latter). Then hybrid mode has no twin and diff overhead in step 1 and step 3, while still having page fault overhead, which is almost negligible. The benchmarking results are plotted as Fig. 4(a) and Fig. 4(b).

The program running in Hybrid mode exhibits superlinear speedup and greatly outstrips the SPM counterpart when scaling the number of cores, thanks to augmented cache effect due to L2 enabling when every core is intensively sorting its portion of the shared data. Superlinear speedup phenomena were also observed by others like MetalSVM [8]. To study the performance in more depth, we have taken the hardware performance counter statistics as shown in Fig. 5 to analyze the performance. We observe that the items (2) "bus utilization", and (7) "pipeline stalled waiting for data memory read" are many times higher in the 32-core SPM execution mode. The bus utilization cycles in SPM mode are about 100 times more that in Hybrid mode, and this is the key reason for why Hybrid mode outstrips SPM mode that much. A significant portion of CPU time is wasted on waiting for the bus—the major performance killer of the SPM mode. This can also be reflected in the number of cycles that the pipeline stalls on read (item (7) in Fig. 5). We can infer from here that the SPM-mode execution has involved many more off-chip DRAM accesses than the Hybrid-mode. This proves that having a hybrid memory model to cater for the diversity of memory



(a) Execution time

(b) Speedup

Fig. 4. Benchmarking results of BucketSort (For 1-core case, Hybrid mode skips all DSM mappings and falls back to pure SPM handling.)

access patterns that exist in many programs is an important move to make scalable many-core SVM systems.

B. Malstone

Malstone [4] is a benchmark that emulates detecting "drive-by exploits" from log files. We ported Malstone (version A) to our SVM library. The core function of the program is to compute the ratio of compromised (a.k.a. flagged or marked) visits to total visits for each website. See Fig. 6(a) and 6(b) for the execution time and speedup measurements respectively. The run in hybrid mode has attained 6x to 8x performance gain over the run in SPM mode. The best case of this application happens on 16 cores (8x better speedup resulted in Hybrid mode). Again, superlinear speedup is observed due to cache effect and slightly due to the relative increase of the number of memory controllers when scaling up the number of cores to run the parallel program. The superlinear speedup in the case of using hybrid mode is even much higher than that observed with SPM mode. The reason is that with the hybrid memory model, the pages mapped as private page mode (MPBT bit off) can exploit L2 cache and hence have augmented the cache effect enormously.

C. Graph 500

We ported the Graph 500 benchmark to our SVM. One thing is that the original Graph 500 BFS kernel is somewhat unrealistic—real-world applications won't just traverse each node without doing anything on it. Therefore, we slightly modify it to model a more real-life application scenario like the case of Malstone—finding the scope of vulnerable users on a social network graph. Each node in the graph represents a user profile. Each edge in the graph represents some relationship with another user. The process begins with a black list of suspicious users who are posing security threats on the social network. For a particular user (source node), we check its relations against the black list of users. We want to traverse all users (nodes) connected to this user to see if any of them belong to the black list. The benchmark uses a shared array of 64-bit long integers to implement blacklist of user ids, and outputs a count of users linking to the blacklisted members at program end. We test with a blacklist of size 128KB and mark it as "read-only private" using the SVM API so that the array can be cached in L2. The benchmark was run in both SPM and Hybrid modes. The benchmarking results of the ported Graph 500 benchmark are shown in Fig. 7(a) and 7(b). We get a 5x performance gain from Hybrid over SPM in running the BFS kernel (step 3) on 48 cores. In terms of total time (steps 1+2+3), Hybrid mode still obtains 2.15 times better speedup than SPM mode on 48 cores. We also observe that the gain from hybrid increases with the number of cores. This is a direct result of exploiting a larger aggregate L2 cache capacity. Or in other words, with more cores to share the workload, the size of per-core partition of the graph data set gets closer to the L2 cache size of each core, resulting in fewer cache misses that go to the off-chip DRAM.

V. Related Work

Currently, SVM research efforts have all been made on SCC Linux. Our work is the first SVM on Barrelfish. Software Managed Coherence (SMC) [10] is an open-source package shipped by Intel to provide an SVM space for the SCC. As SMC is runnable on SCC Linux only, we could not make a direct experimental comparison with it. However, we have studied its synchronization mechanism. SMC directly maps the SVM space to the cacheable shared physical memory (SPM) or shared DRAM. Shared data can only be cached in L1. Mapping all pages to SPM mode for execution on our system is equivalent to SMC on Barrelfish. On the other hand, we augment this baseline system with DSM mode and page remapping to enhance data locality.

MetalSVM [8] is a page-based SVM built into a bare-metal hypervisor that emulates NUMA architecture for guest Linux operating systems on SCC. MetalSVM currently supports two memory models: lazy release consistency and strong consistency. They claimed to provide lazy release consistency (LRC) as in Treadmarks [9], but their implementation is the same as Intel SMC (using CL1INVMB and write-through L1 caches). There is no lock id associated with each critical section, so the scope of invalidation affects all MPBT-tagged cache lines including those that were not updated in or before the current

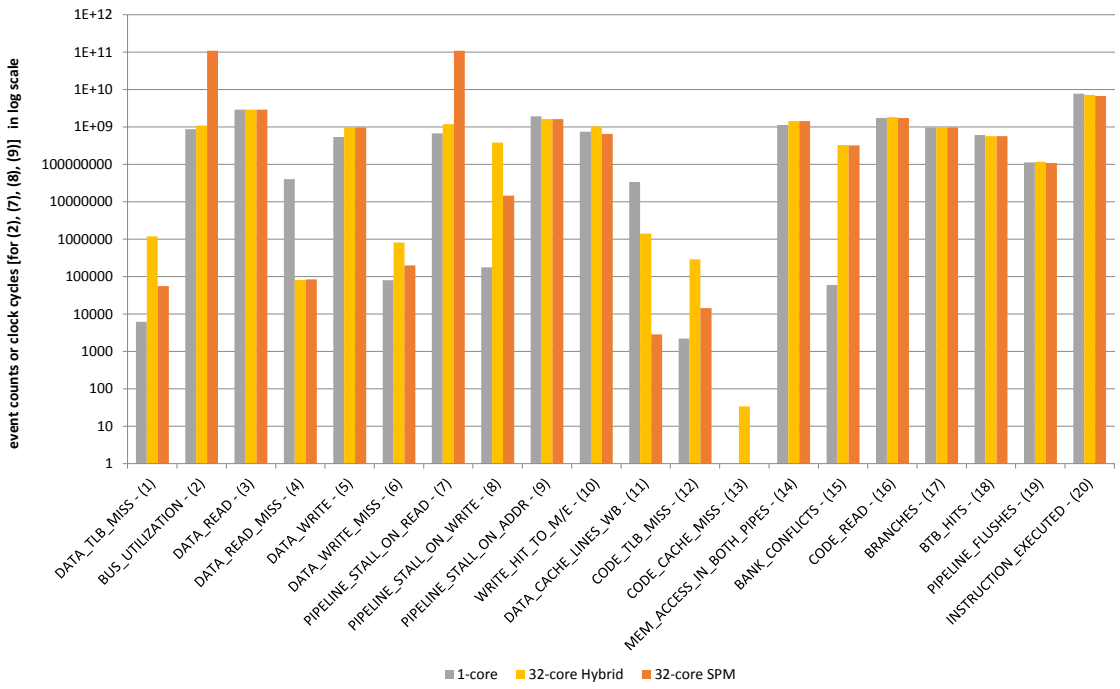
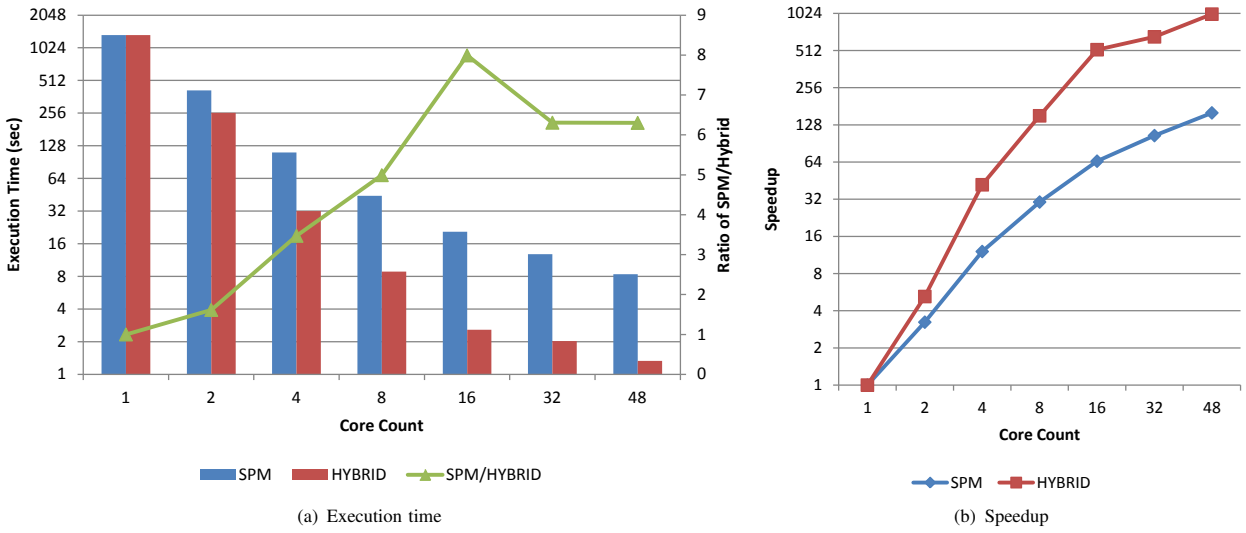


Fig. 5. Performance counter analysis on BucketSort



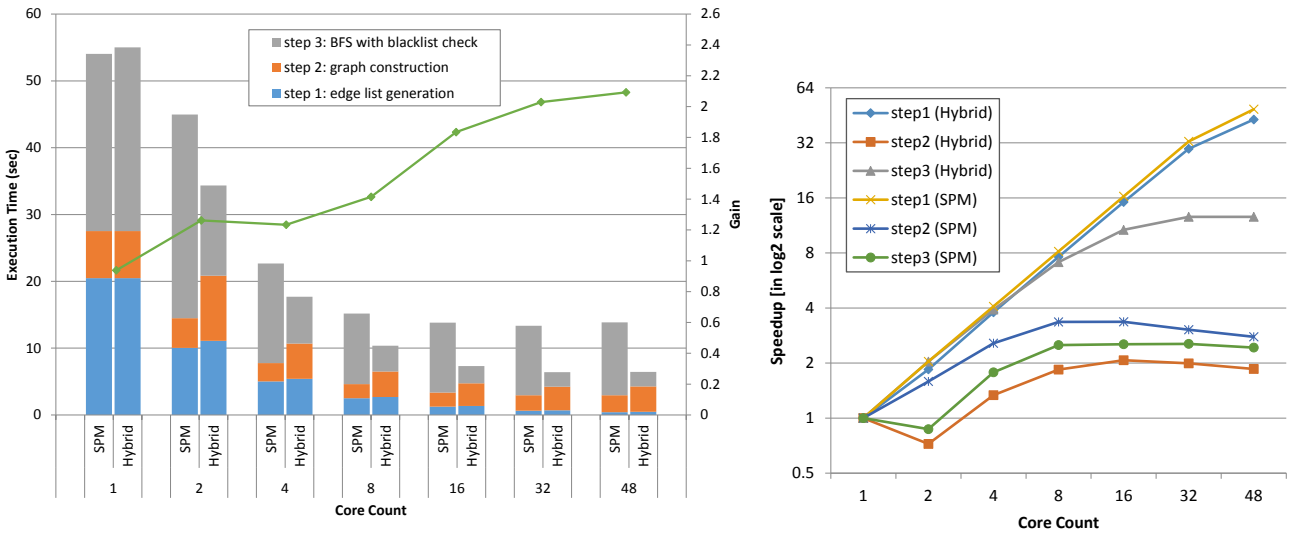
(a) Execution time

(b) Speedup

Fig. 6. Benchmarking results of Malstone

lock interval. So some updates are propagated eagerly, not lazily. In contrast, our DSM mode (scope consistency) using lock ids to differentiate scopes of updates is a truly lazy and fine-grained approach. Regarding their strong memory model, their motivation is for legacy code. At each moment, only one core, the page owner, is allowed to read/write the page. This serialized behavior may result in poor scalability for write-intensive programs.

Kim et al. [11] developed a page-based release-consistent software SVM for SCC Linux based on the commit-reconcile and fence (CRF) memory model. Our SVM is similar to theirs in that cores keep private copies of shared data. But our way of doing this is selective on a per-page basis (guided by the programmer or profiler). In their SVM, all paged data must go through a private memory region called "sache" before they can be accessed. Bringing updates of a page from one core to another entails two memcpy's between saches and shared DRAM. Our SVM supports direct update to the golden copy in shared memory. As another difference, the coherence unit of their SVM is the variable's length. While this design avoids false sharing, they require the user or compiler framework to reconcile and commit every defined shared variable. Their inserted code for per-variable reconcile and commit may penalize the overall runtime, induce more memory transactions and state bookkeeping overheads while page-based reloads, in our SVM, play an aggregate effect and fit data-intensive workloads better.



(a) Execution time

(b) Speedup

Fig. 7. Benchmarking results of Graph 500

VI. Conclusion

Efficient system software support for many-core processors is an inexorable next step in the evolution of computers. This has revived the topic of software managed coherency in recent years following the release of Intel's SCC. We propose a scope-consistent page-based shared virtual memory (SVM) system to support POSIX-like lock-based parallel programming. As an innovation, we devise a hybrid memory model to enforce release consistency for pages allocated in shared physical memory (SPM) and scope consistency for page copies distributed in per-core private memory spaces (DSM). Our SVM system supports a per-core specific choice of page remapping. Embracing such diversity allows different memory access patterns to exploit enhanced cache effects. The memory model is implemented into a two-way cache coherence protocol which synchronizes SPM-mapped pages by partial hardware support (SCC's CL1INVMB) and synchronizes DSM-mapped pages via conventional SVM techniques (page faulting, twinning, diffing, etc). A hybrid execution mode is to mix the handling of the two kinds of virtual memory pages in a harmonious way. This win-win design reaps the benefits of the two protocols. Our experimental results collected from a series of benchmarks, including Malstone and Graph 500, have proved the effectiveness of our coherence protocol design and dynamic page remapping technique to boost cache locality.

Acknowledgment

This research is supported by Hong Kong RGC grant HKU 716712E. Special thanks go to Intel China Center of Parallel Computing (ICCPC) and Beijing Soft Tech Technologies Co., Ltd. for their kind support of the SCC platform in their Wuxi data centers for this work.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. ACM, 2009, pp. 29–44. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629579

[3] "The Graph 500 benchmark," http://www.graph500.org.

[4] C. Bennett, R. L. Grossman, D. Locke, J. Seidman, and S. Vejcik, "Malstone: towards a benchmark for analytics on large data clouds," in *Proceedings of the 16th ACM International Conference on Knowledge Discovery and Data Mining*, 2010.

[5] J. Nilsson, A. Landin, and P. Stenström, "The coherence predictor cache: A resource-efficient and accurate coherence prediction infrastructure," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 10.1–. [Online]. Available: http://dl.acm.org/citation.cfm?id=838237.838517

[6] M. Ekman, F. Dahlgren, and P. Stenstrom, "TLB and snoop energy-reduction using virtual caches in low-power chip-microprocessors," in *Proceedings of the International Symposium on Low Power Electronics and Design*, ser. ISLPED '02, 2002, pp. 243–246.

[7] M. Loghi, M. Letis, L. Benini, and M. Poncino, "Exploring the energy efficiency of cache coherence protocols in single-chip multi-processors," in *Proceedings of the 15th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '05. New York, NY, USA: ACM, 2005, pp. 276–281. [Online]. Available: http://doi.acm.org/10.1145/1057661.1057728

[8] S. Lankes, P. Reble, C. Clauss, and O. Sinnen, "The path to MetalSVM: Shared virtual memory for the SCC," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*, 2011.

[9] 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*, ser. WTEC '94, 1994, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267074.1267084

[10] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha, "A case for software managed coherence in many-core processors," in *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar '10)*, 2010. [Online]. Available: https://www.usenix.org/legacy/events/hotpar10/final_posters/Zhou.pdf

[11] J. Kim, S. Seo, and J. Lee, "An efficient software shared virtual memory for the single-chip cloud computer," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, ser. APSys '11. ACM, 2011, pp. 4:1–4:5. [Online]. Available: http://doi.acm.org/10.1145/2103799.2103804