Shared Memory in the Many-Core Age

Stefan Nürnberger, Gabor Drescher, Randolf Rotta,
Jörg Nolte, and Wolfgang Schröder-Preikschat*

Brandenburg University of Technology Cottbus–Senftenberg, Germany
{snuernbe,rrotta,jon}@informatik.tu-cottbus.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany
{drescher,wosch}@cs.fau.de

Abstract. With the evolution toward fast networks of many-core processors, the design assumptions at the basis of software-level distributed shared memory (DSM) systems change considerably. But efficient DSMs are needed because they can significantly simplify the implementation of complex distributed algorithms. This paper discusses implications of the many-core evolution and derives a set of reusable elementary operations for future software DSMs. These elementary operations will help in exploring and evaluating new memory models and consistency protocols.

1 Introduction

Parallel algorithms are based on distributing computation tasks over multiple execution threads. In shared memory programming models, these threads can access the computation's data directly in a logically shared address space. Most parallel algorithm can be expressed easily with respect to correctness because manual data partitioning and transfers are not necessary, c.f. [10]. Just the inter-task data dependencies require explicit synchronisation.

However, attaining optimal performance with shared memory programming is challenging. In fact, multi- and many-core processors are distributed shared memory (DSM) systems that use message passing internally. They implement the illusion of a shared memory by implicit inter-thread communication. For performance optimisation, it is necessary to understand the distributed structure and the behaviour of the employed consistency protocols, see for example [24].

Message passing could be used directly [19] and would provide explicit control over all communication. But this often requires a considerable effort, which distracts from high-level optimisation. For example, optimising data access locality instead of communication locality and balancing the task decomposition is more effective and easier with shared memory [29]. Furthermore, hardware-based DSMs are efficient on a small scale [21]. On larger scales, software-level DSMs can incorporate algorithm-specific knowledge for higher performance [20,7,4].

* This work was supported by the German Research Foundation (DFG) under grant no. NO 625/7-1 and SCHR 603/10-1, by the Transregional Collaborative Research Centre "InvasIC Computing" (SFB/TR 89, Project C1), and the German Federal Ministry of Education and Research (BMBF) grant no. 01IH13003C.

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 351–362, 2014.
© Springer International Publishing Switzerland 2014

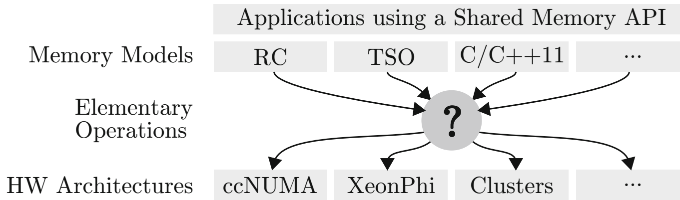

Fig. 1. Elementary operations bridging memory models and hardware

The first software DSMs targeted networks single-threaded computers. With the transition to many-core architectures, the hardware evolved considerably and became more diverse and heterogeneous. Therefore, the many-core age poses a good opportunity to improve upon past DSM research. Examples such as the Quarks DSM for fast networks [8,30] show that rethinking the design of software DSMs is worthwhile. In addition, emerging memory models like in C++11 demand new consistency protocols for software DSMs. A generic infrastructure is needed in order to cope with the many possible combinations of memory models, consistency protocols, and hardware platforms (Fig. 1). This paper presents a set of elementary operations that serve as reusable building blocks for DSMs.

The paper is organised as follows. The design of efficient software-level DSMs depends a lot on the underlying hardware's structure and the interface to the applications on top of the shared memory. Section 2 analyses the implications of the hardware's and software's evolution toward many-core architectures. The section also gives an overview of existing implementation approaches.

Thereafter, Section 3 derives a software architecture of elementary operations from the previous section's analysis. The elementary operations serve as building blocks for DSMs that can be reused in many implementations. They encompass communication mechanisms, memory management operations, and access tracking mechanisms. The final section concludes with a summary and directions of future work.

2 Software DSMs in the Many-Core Age

DSMs provide a shared logical address space across multiple threads, which do not necessarily have global access to all of the memory. The illusion of a shared memory within this address space is created by forwarding access requests and fetching data into local replica. The DSM has to implement mechanisms to detect read and write access, to communicate changes between the replica, and to synchronise concurrent access on behalf of the applications.

Memory models define what applications have to expect about the time and order in which their data changes become visible to other nodes in the worst case. Below these models, consistency protocols define how the DSMs actually achieve a compliant behaviour. Consistency protocols usually provide much stronger guarantees than the memory model they implement. However, programming

against a memory model ensures the portability of the applications and leaves space for the hardware-specific optimisation of consistency protocols.

The next subsection summarises existing DSM implementations with focus on core mechanisms. Then, the second subsection discusses the hardware's evolution toward many-core architectures and its impact on DSM implementations. The last subsection discusses related memory models and consistency protocols.

2.1 Common Software DSM Mechanisms

The most distinctive aspect of DSM systems is the handling of data replication. DSMs that always forward access requests to an owner without replication fall into the family of Partitioned Global Address Spaces (PGAS), see for example [11,9,14,32]. They need only little bookkeeping and their implicit communication patterns are still very easy to comprehend. On the downside, repeated access to remote data is inefficient because it is mapped to repeated data transfers. In contrast, replication-based DSMs manage local copies of recently accessed data, see for example [20,7,4]. Similar to hardware-based caches, they try to exploit spatial and temporal locality in the application's data access patterns. While this can speed up the execution of many algorithms, the necessary bookkeeping can induce considerable overhead [30].

DSMs can be split into three categories with respect to the interface they provide to applications: The first category are systems aimed at unmodified legacy shared memory programs, usually implemented as a wrapper around system libraries or included in the operating system [2,16,15,17,20,12,18]. They usually use hardware-support to track the application's memory accesses. Secondly, library-based DSMs provide an explicit programming interface [7,4,30,32,25]. The applications have to call specific functions in order to prepare for data access, commit changes, and request synchronisation. Finally, language-based DSMs provide explicit sharing constructs [11,9,14]. In addition, the compiler can convert shared data access into respective library calls for a library-based DSM. In all three cases, the employed programming languages need a memory model suitable for shared memory programming. Otherwise, the compiler's optimisation can break the consistency by assuming a too weak model [5].

Deeply related to the application interface are mechanisms that detect and track the application's access to shared data. Explicit application programming interfaces export DSM functions to the application. These have to be called tell about read/write accesses and synchronisation requests. In high-level languages like C++, such calls can be hidden quite well behind standard interfaces [32,1]. Language and compiler extensions can be used to convert access to shared variables into respective library calls. An especially interesting approach is abusing transactional memory extensions because these produce detailed read/write logs and library calls for code sections that are marked as transactions.

Another common approach are memory protection traps [25,18]. Within page-based logical address spaces, read/write access rights can be revoked temporarily for individual pages. Any access to such address ranges raises a trap, which is then handled by the consistency protocol. Data modifications in affected pages

can be reconstructed by comparing against golden copies. Detecting the destination of individual accesses through separate traps is possible but inefficient. An alternative are virtual machines. These apply binary code transformation and just-in-time compilers to insert DSM library calls where necessary.

Finally, high-level knowledge about memory access patterns can be exploited directly. Some programming models, such as data flow based models, expose coarse grained data dependency information [6]. This is used mainly to order the execution of tasks but can be used also to replicate, update, and invalidate data that is shared by tasks.

2.2 From Single-Core to Many Cores

Early DSM systems like Ivy [20] and Munin [7] targeted networks of single-threaded processors. Apart from expensive high performance computing hardware, the typical inter-processor networks used to be weakly coupled with low bandwidth and very high latency relative to local memory accesses. In comparison to the network, the processors were quite fast and designed for high single-thread performance.

The high latency and processing overhead of the networking hardware penalised high numbers of relatively small messages like they are exchanged by simple consistency protocols [7]. In order to communicate with fewer and larger messages, complex memory models allowed to manually state application-level knowledge and the consistency protocols adapted to observed access patterns. Because of the relatively fast processors, the implied bookkeeping overhead was negligible. Nevertheless, manual message passing seemed to be much more straightforward and easier to optimise [8].

The development of many-core architectures is driven by the need for higher energy and space efficiency [3,27]. In order to increase the compute throughput per watt for parallel computations, inefficient features that just increase the single-thread performance are stripped away from the cores. This leads to small efficient cores, which can be integrated in high number on a single chip. Also, networking hardware is integrated tightly into the processors. On-chip networks like [26] provide high throughput communication between a large number of cores and memory controllers. Likewise low latency processor-interconnects such as QPI, PCIe, and Infiniband are widely used now.

Many-core DSMs have to address three major aspects: *Consistency islands*, *mandatory parallelism*, as well as *diversity and heterogeneity*.

Caches often reduce the communication volume between threads and main memory. The consistency of their replicated data is maintained by cache consistency protocols. These can be efficient even with a large number of threads, but most rely on a fixed upper bound of participating threads [21]. Hence, scaling out many-core processors to larger setups, like in the DEEP project [13], does not extend to global cache consistency. This leads to networks of *consistency islands*. Each island contains many threads that can cooperate through hardware-based cache consistency. The network between islands may provide remote memory access and even atomics to enforce ordering. But remote data replicated in local

caches can become inconsistent because no notifications about write accesses are communicated between islands.

In conclusion, software DSMs span multiple consistency islands and the threads inside each island should share their data replica. Otherwise, storing separate replica for each of the many threads would waste memory and cache space. The additional overhead of coordinating the concurrent access to shared replica is hopefully compensated by sharing the costs of replica management between all threads.

Secondly, any bookkeeping overhead of software DSMs is amplified by the slow performance of single threads. For example, remote memory access over Infiniband links can be as fast as $2\mu s$, which corresponds to just 8 cache misses on the Intel Xeon Phi (280ns/miss) [24]. Frequent remote memory accesses might be more efficient than managing local replica. In addition, it is significantly more efficient to use 2MiB instead of 4kiB pages to describe the logical address spaces. Thus, page-based access tracking has to process 512x larger pages.

In consequence, exploiting all types of parallelism in DSM implementations is mandatory to fully utilise the high throughput of whole consistency islands. Just designing simpler protocols like in the Quarks DSM [30] will not be sufficient.

Finally, DSMs have to deal with diversity between and heterogeneity inside many-core platforms, even though they share the same instruction set and data encoding. Depending on the application domain, different design trade offs between network bandwidth, cache size, and micro-architecture features are more efficient. Similarly, mixing cores optimised for single-thread throughput with cores optimised for high parallel throughput is useful for a large class of applications [13]. Hence, abstractions over the platform's structure are needed.

2.3 Memory Models and Consistency Protocols

Memory models define the permitted reordering and elimination of concurrent accesses to shared memory. Applications usually target the memory model of the used programming language or software-level DSM. Compilers and DSMs translate this onto the memory model(s) of the underlying hardware.

The strictest model, called *sequential consistency*, executes all accesses exactly in the order that was expressed by the programmer. However, to improve the performance, modern hardware and compilers employ optimisations that reorder the accesses [22]. For example, the compiler can eliminate any access to memory locations that are considered private by the compiler. Similarly, the hardware does not have to keep modified data in caches consistent with the main memory immediately. This results in a logical reordering of reads and writes from the main memory's point of view. Other common optimisation techniques include store buffers, request queues, out-of-order execution, and speculation techniques such as branch prediction and prefetching.

For single-threaded programs, these optimisations do not result in any observable change of program logic. But surprising effects can arise with multi-threaded programs. Most programming languages do not state their memory model explicitly and the compilers are free to assume a sequential execution of the generated

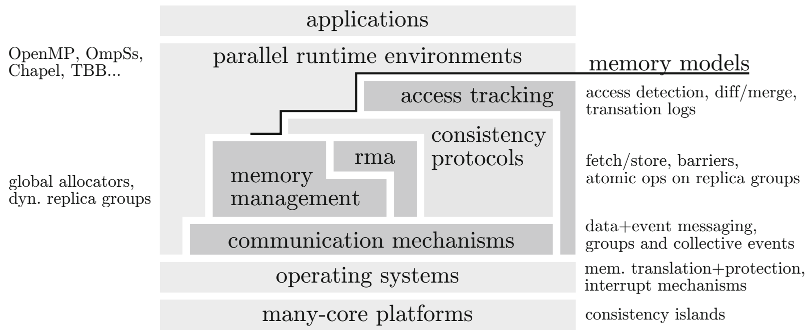

Fig. 2. Architectural overview of memory consistency support for parallel runtimes. The architecture provides basic mechanisms supporting the implementation of a consistency protocol.

instructions. On processors with a *relaxed* memory model, additional synchronisation instructions would be needed to regain the intended behaviour. Newer languages with explicit support for multi-threading, such as C++11, address this issue by defining a memory model based on sequential consistency for data race free programs. There, application programmers have to explicitly resolve data races by applying *atomic operations* instead of normal read/write access.

The compilers, DSM implementations, and low-level system programmers rely on the hardware's memory model. Some architectures provide a formal definition of their model, for example `SPARC-TSO`. Most architectures, for instance `x86`, only provide ambiguous descriptions in prose although their models can be specified formally [28]. These models start with very relaxed semantics and provide *memory barriers* to enforce stricter models.

Another common primitive to regain control over concurrent memory access are *atomic operations*. They are especially useful for the implementation of high-level synchronisation primitives and lock-free data structures. These instructions differ between hardware architectures but usually include variants of test-and-set (TAS), compare-and-swap (CAS/DCAS), fetch-and-increment (FAI), and Load Linked/Store Conditional (LL/SC). Depending on the memory model, these operations do not provoke a full memory barrier but only give guarantees for the affected memory addresses and direct data dependencies.

Memory models are a contract between application and system on a semantic level. Below these, consistency protocols represent concrete strategies that achieve compliant effects. Different consistency protocols can realise the same model. Apparently, a consistency protocol for sequential consistency also satisfies more relaxed models and can simply ignore all memory barriers.

3 Elementary Operations for Many-Core DSMs

Future distributed shared memory programming should look more like current shared memory programming on relaxed memory models. The shared memory

abstraction needs a flexible definition of its memory consistency model. Relaxations in this model are the key point for performance improvements exploiting implementation freedoms. This is analogous to the memory models provided by hardware. However, the semantics of these models must be defined rigorously [31] in order to be useful and to prove their correctness. There is an ongoing process defining the memory models for shared memory architectures. The semantics of the distributed models must be treated equally.

Most programmers should not need to care about the underlying hardware memory model. Instead, a useful abstraction should let them state the needed guarantees in the form of an explicit memory consistency model. This explicit model is provided by the implementation of a consistency protocol. The implementations of custom protocols benefit from reuse of common functionality. Elementary mechanisms map this functionality to fitting operations on the underlying hardware model. In distributed shared memory, such as clusters of many-cores, this hardware model is most likely heterogeneous, forming consistency islands on which more efficient mappings exist. When consistency related events are restricted to such an island, more efficient implementations of these mechanisms can be used.

Whereas past DSM systems provided the programmer with a distinct programming model, we rather treat DSM as an optimizing feature to existing programming models. The envisioned use case is an augmentation of parallel runtime environments through replication.

In combinations like these, consistency guarantees for programmers must be based on data-race-freedom. Providing fixed guarantees without race freedom requires tight control over the whole stack of programming language, compiler optimizations, and hardware architecture while it significantly inhibits performance optimization. Providing a guatanteed consistency model for data-race-free programs is possible in a compiler-agnostic way.

3.1 Communication Mechanisms

The basic shared memory abstraction provided by a given programming model needs to provide a means by which memory can be managed and accessed. A basic communication mechanism is needed for coordination. It should provide the following features:

Data Transfer as a mechanism to read and write memory contents in a distributed memory system. This will be used by the remote memory functionality and to create and manage replicas of memory locations.

Event Notification as very lightweight mechanism to notify hardware threads of consistency related events like invalidations. Also preemptive notifications are needed to interrupt applications when necessary.

Thread Groups are a basic feature to group the propagation of events. They reflect the overall system topology and are used to bootstrap efficient replication across consistency islands. Groups are dynamic and identify partakers in sharing that need to be involved in consistency related action.

Collective Events must be provided that efficiently disseminate events in a group.

3.2 Memory Management, Replication, and Remote Access

With the global coordination in place, shared memory can be managed. There is no strong requirement on the user-visible interface to this memory, but the runtime is expected to provide some notion of a global address space. The common memory management mechanisms are:

Allocators for globally coordinated memory. They provide the mapping mechanism for named entities of shared memory from which all shared memory operation needs to be bootstrapped. Through this call, a sharing participant is registered and its address space has to be adapted accordingly.

Annotation mechanism to configure the semantics of a shared memory range at runtime. This can be used to provide features like the current ownership declaration from MYO. Many shared-memory systems do not employ this feature since they pertain to exactly one set of semantics.

Replica management is the key aspect of performance improvements in a distributed memory system. Each replica is a locally cached version of the shared-memory location. Caching is the single most important feature in reducing access times to shared memory. Giving guarantees on the actuality of cached data is the concern of consistency model semantics. Operations provided in replica management include creation, update, and invalidation of single replicas and groups. An acknowledgement mechanism must be provided to check for the successful invalidation or update of replicas. Replica management is a background task based on asynchronous messages.

Remote Memory Access used for direct access and modification of a remotely available memory location where replication is not beneficial. Remote memory can be implemented through address space manipulation, mapping areas of remote physical memory on the PCIe bus, or hardware provided RDMA in InfiniBand networks. Whenever such hardware support is not available, the remote memory operations must resort to explicit message passing. Also the GASPI abstraction can be employed as an implementation technique here. However, remote memory operations and their interleaving with local operations on the same memory may have semantics that are hard to describe. Coherence may not be available on some hardware architectures when memory is accessed locally through the processor and concurrently through e.g. an InfiniBand controller. These memory semantics will require additional fences for correct operation. Alternatively, implementations can tunnel local access through the remote access channel, thereby forcing a serialization point with the remote events.

Atomic Operations provide *write atomicity* enforcement, a guarantee that the write operation can be seen either by all other threads, or none. They are a special case of remote memory operations. As far as atomic update operations are concerned, LL/SC should be provided because it can be implemented on asynchronous messages (and is allowed to fail), yet it enables implementation of all other atomic operations (FAI, TAS, CAS, DCAS).

3.3 Access Tracking

Consistency protocols share a couple of additional requirements. These can also be provided as basic mechanisms to ease implementation of new protocols. They concern the connection of application behavior and consistency related events. The proposed mechanisms are:

Access Tracking provides a mechanism to track read and write access to replicas. Depending on the shared memory API and desired memory model semantics this can possibly be made explicit, e.g. using object-oriented programming. In the worst case it must be possible to track every single memory access. Ususally only the first write to a valid replica or read access to an invalid replica needs to be detected. Obvious implementation choices include traps through virtual memory mapping protection mechanisms (i.e. Segmentation Fault handlers), low level virtualization, or compiler instrumentation.

Diff/Merge for memory locations is used in order to weaken exclusive write access, and implement multiple writer protocols. This has been implemented in a variety of distributed shared memory systems to avoid overhead through false sharing. It can also be used to offer lightweight updates of larger sharing units in order to decrease communication bandwidth. The mechanism must offer shadows or transparent copies of affected memory locations (e.g. pages) and an efficient coding for generating, storing and applying a difference mask. This is a prime example for work that should be delegated to helper threads on many-core architectures.

(Versioned) Modification Tracking per replica is needed in a basic form to trigger consistency related actions without explicit calls from the API (see access tracking above). Through additional versioning an implementation of restricted transactional memory can enable lock elision techniques like provided in current off-the-shelf multi-core processors.

The described mechanisms are employed to build the semantics of the desired memory consistency model. Depending on the placement of threads that take part in the sharing of a memory location, the implementation details of the single mechanisms can or rather must vary. If sharing is restricted to a single consistency island, e.g. only among threads of a single accelerator card, some consistency requirements may be provided by hardware directly. As soon as sharing stretches across more than one island, implementations must be adapted to the new situation. Strategic placement of tasks will therefore stay a significant tool for optimized performance in a shared memory system, just like it is with today's ccNUMA architectures.

4 Conclusions and Future Directions

In this paper, the benefits and challenges of distributed shared memory systems were examined with respect to networks of many-core processors. The many-core age provides good opportunities to improve upon past DSM research. For

instance, the underlying hardware evolved much from the loose networks of fast single-threaded nodes to the tightly coupled networks of consistency islands with many relatively slow threads. Likewise, the application domains evolved far beyond the first numerical simulation codes. The need for increasingly complex data structures and parallel algorithms pushes toward new parallel languages, programming models, and memory models.

However, implementing efficient DSMs became more challenging. Exploiting effectively, for example, consistency islands and their internal parallelism, raises the effort for basic DSM infrastructure. Fortunately, the memory models and their consistency protocols share many common mechanisms. The paper derived an architecture of elementary operations as building blocks for future DSMs. These help mapping the application's memory model to efficient consistency protocols while reusing common infrastructure.

The *Consistency Kernel* (CoKe) project evaluates the presented elementary operations in detail. This includes efficient hardware abstractions even on hardware without cache coherence like the experimental Intel SCC many-core processor and clusters of Intel Xeon Phi processors. A part of the *OctoPOS* project [23] at the collaborative research center for invasive computing explores memory models for invasive computing. This effort targets processors with multiple consistency islands and reuses the elementary operations. Finally, the *Many Threads Operating System* (MyThOS) project researches minimal operating system components for many-core accelerators. While focusing on lightweight thread management for HPC applications, generic system services can be shared with CoKe.

References

1. Mintomic, `http://mintomic.github.io` (accessed: 2014)
2. Scalemp vsmp foundation, `http://www.scalemp.com` (accessed: 2014)
3. Agarwal, A., Levy, M.: The kill rule for multicore. In: 44th ACM/IEEE Design Automation Conference, DAC 2007, pp. 750–753 (June 2007)
4. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: Treadmarks: Shared memory computing on networks of workstations. Computer 29(2), 18–28 (1996)
5. Boehm, H.: Threads cannot be implemented as a library. ACM Sigplan Notices, 261–268 (2005)
6. Bueno, J., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Implementing ompss support for regions of data in architectures with multiple address spaces. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS 2013, pp. 359–368. ACM, New York (2013)
7. Carter, J.B.: Design of the munin distributed shared memory system. Journal of Parallel and Distributed Computing 29(2), 219–227 (1995)
8. Carter, J.B., Khandekar, D., Kamb, L.: Distributed shared memory: Where we are and where we should be headed. In: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V), pp. 119–122. IEEE (1995)
9. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. International Journal of High Performance Computing Applications 21(3), 291–312 (2007)

10. Chapman, B.: Scalable shared memory parallel programming: Will one size fit all? In: 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2006, p. 3 (February 2006)
11. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. ACM Sigplan Notices 40(10), 519–538 (2005)
12. Cordsen, J., Garnatz, T., Sander, M., Gerischer, A., Gubitoso, M.D., Haack, U., Schröder-Preikschat, W.: Vote for peace: Implementation and performance of a parallel operating system. IEEE Concurrency 5(2), 16–27 (1997)
13. Eicker, N., Lippert, T., Moschny, T., Suarez, E.: The deep project - pursuing cluster-computing in the many-core era. In: 2013 42nd International Conference on Parallel Processing (ICPP), pp. 885–892 (October 2013)
14. El-Ghazawi, T., Smith, L.: Upc: uni ed parallel c. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing (2006), p. 27. ACM (2006)
15. Fleisch, B., Popek, G.: Mirage: A coherent distributed shared memory design. In: Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, SOSP 1989, pp. 211–223. ACM, New York (1989)
16. Göckelmann, R., Schoettner, M., Frenz, S., Schulthess, P.: Plurix, a distributed operating system extending the single system image concept. In: Canadian Conference on Electrical and Computer Engineering, vol. 4, pp. 1985–1988. IEEE (2004)
17. Itzkovitz, A., Schuster, A., Shalev, L.: Thread migration and its appli- cations in distributed shared memory systems. Journal of Systems and Software 42(1), 71–87 (1998)
18. Lankes, S., Reble, P., Sinnen, O., Clauss, C.: Revisiting shared virtual memory systems for non-coherent memory-coupled cores. In: Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM 2012, pp. 45–54 (2012)
19. Lauer, H.C., Needham, R.M.: On the duality of operating system struc- tures. SIGOPS Oper. Syst. Rev. 13(2), 3–19 (1979)
20. Li, K.: Ivy: A shared virtual memory system for parallel computing. In: ICPP (2), pp. 94–101 (1988)
21. Martin, M.M.K., Hill, M.D., Sorin, D.J.: Why on-chip cache coherence is here to stay. Commun. ACM 55(7), 78–89 (2012)
22. McKenney, P.E.: Memory barriers: a hardware view for software hackers. Linux Technology Center, IBM Beaverton (2010)
23. Oechslein, B., Schedel, J., Kleinöder, J., Bauer, L., Henkel, J., Lohmann, D., Schröoder-Preikschat, W.: OctoPOS: A parallel operating system for invasive computing. In: Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA), EuroSys, pp. 9–14 (2011)
24. Ramos, S., Hoefler, T.: Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi. In: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing, pp. 97–108. ACM (June 2013)
25. Saha, B., Mendelson, A., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R.: Programming model for a heterogeneous x86 platform. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI 2009, p. 431 (2009)
26. Salihundam, P., Jain, S., Jacob, T., Kumar, S., Erraguntla, V., Hoskote, Y., Vangal, S., Ruhl, G., Borkar, N.: A 2 Tb/s 6x4 Mesh Network for a Single-Chip Cloud Computer With DVFS in 45 nm CMOS. IEEE Journal of Solid-State Circuits 46(4), 757–766 (2011)

27. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: A many-core x86 architecture for visual computing. In: ACM SIGGRAPH 2008 Papers, SIGGRAPH 2008, pp. 18:1-18:15. ACM, New York (2008)
28. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO. Communications of the ACM 53(7), 89 (2010)
29. Snir, M.: Shared memory programming on distributed memory systems. In: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models, PGAS 2009, pp. 3:1–3:1. ACM, New York (2009)
30. Swanson, M., Stoller, L., Carter, J.: Making distributed shared memory simple, yet efficient. In: Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments, pp. 2–13. IEEE (1998)
31. Zappa Nardelli, F., Sewell, P., Sevcik, J.: Relaxed memory models must be rigorous. In: Exploiting Concurrency Efficiently and Correctly Workshop (2009)
32. Zheng, Y., Kamil, A., Driscoll, M., Shan, H., Yelick, K.: Upc++: A pgas extension for c++. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1105–1114 (May 2014)