

# Peter J. Keleher

Research Summary - July, 2000

## 1. Overview

My research is primarily in the field of “distributed systems”. Loosely defined, this encompasses work in operating systems, runtime systems, and distributed object and database systems. In all cases, my approach is to identify problems, propose solutions in the form of specific policies and mechanisms, and then empirically evaluate them. I am a systems builder: very few projects end in analysis or simulation.

The systems that I study often operate in fluid, dynamic environments. Simply put, I try to make systems operate better in such circumstances. Central to my approach is the notion of building systems that transparently adapt applications to changing conditions.

In the CVM (Coherent Virtual Machine) and Deno (Decentralized Network Objects) projects, the barriers to such adaptation derive from consistency constraints of replicated state. Strong consistency requirements force all copies of shared state to be maintained in identical, or nearly identical, versions. These requirements are relaxed by letting the underlying system temporarily leave copies inconsistent, thereby allowing expensive network communication operations to be delayed, aggregated, and possibly even omitted altogether.

In the Active Harmony project, the barriers to adaptation are rigid execution models that statically define all parameters of application execution. We break these barriers down by allowing applications to export detailed application alternatives to the system, which can then adapt the application to available resources either at application startup or during execution.

The Sections 2-4 discuss each of my three major projects, and Section 5 concludes with current and future work. Note that references to papers written by other researchers are included only when absolutely necessary to establish context. This is in no way an attempt to construe my work as the only relevant work in these areas. The purpose of this document is to discuss the work that I have done, not to review entire fields.

This research is supported by grants from Microsoft, the National Security Agency, a National Science Foundation Faculty (NSF) Early Career Development Award, and two additional NSF awards.

## 2. Software Distributed Shared Memory (TreadMarks and CVM)

A Software Distributed Shared Memory (SDSM) system [27] is a software system that runs parallel shared-memory applications on clusters of workstations. In the most general sense, such a system can host parallel applications running on a cluster of standard workstations (e.g. a single-processor Dell PC) attached via a general-purpose interconnect (e.g. Fast Ethernet). Performance is critical, as such systems essentially use software to replace functionality more typically performed in hardware. The rest of the section describes our research in SDSM consistency models, implementations, multi-threading, tools, and abstractions.

### 2.1 Relaxed consistency models and implementations

Maintaining memory consistency is usually implemented in hardware, and implementing hardware-like functionality in software causes large overheads. Researchers in SDSM have addressed this by allowing distinct copies of a given data item to remain inconsistent for strictly delimited periods of time. This extra flexibility allows much more efficient software implementations.

Early work in this area included the *release consistent* Munin [5] and the *causally consistent* Clouds [1]. My PhD work defined *lazy release consistency* (LRC) [24], the most common relaxed consistency model used in SDSM today. LRC essentially allows SDSM's to delay enforcing consistency until forced by potential causality, with causality defined by synchronization. During the course of my dissertation research, I designed and built the TreadMarks [2, 25] system. TreadMarks was the first implementation of the LRC memory model, and is still the primary system to which new SDSM systems are compared. Although the performance

advantages enabled by LRC vary from application to application, it is not unusual for relatively fine-grained applications to perform more than twice as well with LRC than with earlier consistency models. One measure of the impact of this work is that `citeseer.nj.nec.com`, an online citation index, shows that the original memory model and implementation papers have each been cited on the order of two hundred times. TreadMarks has since been commercialized, and is being incorporated into the standard OpenMP toolkit of a major software tools supplier.

Upon arriving at Maryland, I built a new, extensible, LRC-based SDSM system called CVM<sup>1</sup>. I used empirical studies of CVM to show that the performance advantages of LRC implementations derive from two distinct differences from previous systems. First, LRC is more relaxed than previous consistency models, and therefore allows underlying systems to better aggregate messages and handle false sharing. Second, LRC allows practical multiple-writer protocols to be implemented. A multiple-writer protocol is one that allows multiple processes to modify the same data item (usually a shared page) concurrently. I showed that the majority of LRC's benefit came from the relaxation of the consistency model, with multi-writer protocols being important in only a minority of the cases [16]. This work also described and evaluated the first single-writer LRC protocol. Single-writer protocols can easily have an order of magnitude less memory overhead than corresponding multi-writer protocols. This work preceded later work on dynamic protocols at Rice [4], and the *home-based* LRC implementations at Princeton [45].

I later studied home-based and standard LRC implementations [18], showing that current optimized implementations performed very similarly across a broad suite of applications. However, after augmenting both types of protocols with support for automatic updates, I was able to show that home-based protocols are particularly well suited for applications based on global barrier synchronization. The final update protocol performed an average of 53% better than the best “standard” protocol across a suite of seven barrier-based applications [15, 21].

The studies in the previous two paragraphs were performed solo, resulting in three single-author conference papers [16, 18, 21] and two single-author journal papers [15, 20].

## 2.2 Per-node multi-threading

Despite the advantages of relaxed consistency models, remote operations are expensive to perform on top of general-purpose networks and protocol stacks. Multi-threading is a common means of hiding latency in the hardware arena. The basic idea is to switch to a second thread when the current thread blocks on a remote operation, allowing the second thread to perform useful work while the first waits.

My group at Maryland was the first to investigate the use of multi-threading as a means of hiding the latency of remote operations in SDSM's, where remote latencies are much larger. Our techniques were able to improve the performance of three of the seven applications studied by at least 17%, with all the applications benefiting to some degree. Note that we concentrated our research on steady-state application behavior. Subsequent researchers reported even larger improvements by including the effects of startup costs on short-running applications [29]. We characterized a number of effects of the multi-threading technique, some obvious (cache pollution), and some not so obvious (restrictions on the memory and programming models). The initial conference version of this work [39] was forwarded to a “Best Papers” issue of IEEE TOCS [40].

Maintaining multiple threads on each node has other benefits. Each thread provides a handle to a specific amount of work. Moving the thread also moves the work, allowing thread migration to be used for load balancing [41], dynamic parallelism adaptation [41], and communication minimization [42]. Load balancing is particularly important when the system's constituent machines are either loaded [44], or heterogeneous [43].

## 2.3 Race detection

SDSM's are an ideal platform for *race detection* tools. Loosely defined, a race is a pair of unsynchronized accesses by different processes, such that at least one is a write. The outcome of a race is undefined, and usually constitutes a bug. SDSM's are good platforms for race detectors because much of the information

---

<sup>1</sup> Downloaded by over 500 distinct sites.

needed to find races are already maintained by the system. In particular, the system maintains enough consistency information to determine if any two program points are synchronized, and to determine the set of modified shared data during any interval. My group built a practical online race detection system that is guaranteed to catch all races that actually occurred during an execution, with an order of magnitude less overhead than previous systems [30, 31]. This work preceded, and helped to spark, a small flurry of race-detection papers in the operating system community by researchers at Wisconsin [33], Technion [12], and Washington [36].

## 2.4 Tapes

Finally, I defined the *tape* mechanism [14, 19], a high-level abstraction that unifies the expression and implementation of a number of techniques for improving the performance of SDSM protocols. A tape is essentially an object that encapsulates an arbitrary number of updates to shared data. Tapes are created through calls to the tape library that start and stop recording of updates to shared data made by the local process. Once created, a tape provides a convenient way to manipulate the updates. The data referenced by a tape can be sent to another process. Tapes can be reshaped by changing the set of data to which they refer. Tapes can also be added, subtracted, and projected, allowing a single tape to describe any arbitrary set of updates.

Tape-based synchronization libraries are layered on top of existing consistency protocols and synchronization interfaces, and are used to direct data movement. Tapes allow shared accesses to be recorded, grouped, and manipulated at a very high level. These tapes can be used to predict future data accesses and to eliminate subsequent misses by moving data before it is needed.

I used the tape mechanism to build Tapeworm, a synchronization library that uses information gathered at runtime to reduce access misses. Tapeworm's interface consists of auto-locks, producer-consumer regions, and record/reply barriers. Auto-locks pre-validate data that is accessed while locks are held. Producer-consumer regions use the first access to a region as a hint to request the rest of the region before it is needed. Record/replay barriers allow accesses to be recorded during one iteration and then played back during future iterations. The combination of these mechanisms allows Tapeworm to significantly improve overall performance of iterative applications. More significantly, Tapeworm allowed the expression of a variety of different optimizations with a simple, elegant abstraction.

Tapes can also be used to mirror the data movement in recent update-based protocols, including home-based LRC, entry consistency, and scope consistency. These protocols differ from generic LRC in terms of the programming model, the memory model, and the flow of data. Tapes-based implementation can be used to separate the performance effects of the latter factor from the effects of the former two factors. Such implementations can also be used to provide "front-ends" to a SDSM system, somewhat analogously to the front end of a compiler.

In general, Tapes have at least two major advantages over optimizations of specific protocols. First, tapes provide a high-level abstraction of shared accesses, and are protocol-independent. Tapes make few requirements on the underlying protocol, providing a terse, powerful approach to managing data movement. Second, tape mechanisms can be implemented and used incrementally. Applications can be completely debugged before any tape mechanisms are added. One by one, tape mechanisms can be used to improve data movement at inefficient points in application executions.

The tapes work was described in a single-author workshop paper [17], a single-author paper at OSDI (one of the top two operating systems conferences) [19], and a single-author paper in ACM TOCS (the top journal in the field) [14].

## 3. Grid Computing (Active Harmony)

The second major thrust of my research (with Dr. Hollingsworth at Maryland) has centered on Active Harmony, a resource management system intended for dynamic environments [11]. Our overall goal is to efficiently run large, parallel, potentially long-lived jobs on networks of workstations. The allocation must be dynamic in that the machines might have non-Harmony loads, the applications may be dynamic, and initial

estimates of resource requirements might be wildly incorrect. Hence, Harmony allows applications to be adjusted during execution, not just at application startup.

There are three, relatively distinct parts to this research. The first is Active Harmony itself, i.e. the infrastructure that allows applications and the system to communicate about resource allocation. The second is an investigation of operating system mechanisms needed to allow “Harmonized” jobs to run on non-dedicated nodes without affecting the performance of local processes. The last is a set of investigations into one specific scheduler policy: a backfilling scheduler for batch systems.

### 3.1 Active Harmony

Grid computing can be defined as the simultaneous and coordinated use of semi-autonomous computing resources in physically separate locations. By using a collection of specialized computational and data resources located at different facilities around the world, work can be done more efficiently than if only local resources were used. Both grid-computing environments and the applications that run on them can be characterized by distribution, heterogeneity, and changing resource requirements and capacities. These attributes make static approaches to resource allocation unsuitable. Systems need to dynamically adapt to changing resource capacities and application requirements in order to achieve high performance in such environments.

Active Harmony is a middleware system that manages distributed execution of computational objects in such environments. Such systems must be able to tolerate changing network and resource capacities because their execution environments are large, and often non-dedicated. The latter means that additional non-Harmony load may come and go during the execution of Harmony jobs.

Such a system should also be able to efficiently handle differing offered load. For example, the system should assign all of the system’s resources to a single application if there are no others in the system, but should “fairly” distribute resources among multiple applications.

The middleware must have fine-grained control over each application’s execution if it is to handle the above requirements well. Harmony’s main innovation is an expressive interface that allows applications to export *tuning options* to the middleware. By exposing different parameters that can be changed at runtime, applications can be made to adapt to changes in their execution environment due to other programs, the addition or deletion of nodes, communication links etc. For this to work, applications must not only expose their options, but also the resource utilization of each option, and the effect that choosing a given option will have on the application’s performance. We have designed and built an expressive and powerful resource description language for this purpose [26].

Tuning options allow us to move policy into a global manager, concentrating both information gathering and application control into a single (possibly hierarchical) location. This has two effects. First, accumulating global information into one place provides better information to resource allocators. Second, grouping control of all applications into a centralized manager allows adaptations to be made in situations where competing application-specific schedulers would not cooperate for the global good.

For example, consider a parallel application whose speedup improves rapidly up to six nodes, but improves only marginally up to eight nodes. A resource allocator might give this application all eight nodes in the absence of any other applications. After all, the last two nodes were not being used for any other purpose, and do marginally improve the application’s response time. However, a new job entering the system could probably make more efficient use of those two nodes. If the decision about whether to reconfigure the first application is made by the application itself, no reconfiguration will occur. A centralized decision-maker should be able to infer that reconfiguring the first application to only six nodes will improve overall efficiency and throughput, and would be in a position to make the reconfiguration happen.

### 3.2 Operating System Mechanisms

The second part of this work is a set of mechanisms and policies that allow *guest jobs* to use resources on non-dedicated, idle workstations. This task is complicated by the fact that “idle” workstations are often only temporarily idle. Past systems [28] have dealt with such situations by immediately migrating guest jobs off

such nodes as soon as any activity is detected. The reason for this conservative behavior is that such nodes are usually owned by entities other than those that are running the guest jobs. Hence, the impact of guest jobs on the performance of local jobs must be minimized in order to ensure that the owners of such machines continue to allow guest jobs to execute.

We believe that such policies waste many opportunities to exploit idle cycles because of overly conservative estimates of resource contention. We attempt to use *fine-grained cycle stealing* to maximize overall throughput by leaving guest jobs on machines even when resource-intensive host jobs start up. However, the host job will be adversely affected unless the guest job's resource use is strictly limited. We have introduced three new mechanisms that can be used to limit the impact of the guest job's resource use.

First, we modified the Linux scheduler in order to prevent guest processes from running when runnable host processes are present [34]. The change essentially establishes guest processes as a different class, such that guest processes are not chosen if any runnable host processes exist. This is true even if the host processes have lower runtime priorities than the guest process.

Second, we limit the guest job's consumption of memory resources [34]. Unfortunately, memory is more difficult to deal with than the CPU. Reclaiming the processor from a running guest job only requires saving processor state, and restoring the cache state. The cost of reclaiming page frames from a guest job is negligible for clean pages, but quite large for modified pages because they need to be flushed to disk before being reclaimed. The simple solution to this problem is to permanently reserve physical memory for the host processes, but many guest jobs are quite large. Simulations and graphics rendering applications can often fill all available memory. Hence, not allowing guest processes to use the majority of physical memory would prevent a large class of applications from taking advantage of idle cycles.

Instead, we modified the page replacement algorithms to include a system of preferences and thresholds. The resulting system is flexible, but still provides relatively hard bounds on the amount of memory contention that can be caused by guest jobs. This approach reduced the delay of host jobs by almost a factor of three in some cases, with little effect on guest jobs.

Third, we limit the guest job's bandwidth to the network and secondary storage by using *rate windows* [35]. A rate window is a low-overhead facility that gives us the ability to set hard per-process bounds on I/O and network usage. Such management is useful both for managing resource allocations of competing users (such as virtual hosting of web servers), and for rate-based clocking of network protocols as a means of improving the utilization of networks with high bandwidth-delay products.

By using all of these techniques together, we are able to leave even CPU- or I/O-bound guest jobs on workstations even after the workstations become "active", with almost no degradation of host job performance. The performance of guest jobs can easily double because of the resultant reduction in costly guest job migrations.

### 3.3 Backfilling Schedulers

This work is not strictly part of Harmony, but has similar goals. A batch scheduler is one that accepts non-interactive jobs into a dedicated system and assigns them to nodes as they become available. *Backfilling* is a simple, common, and effective way of improving the utilization of batch schedulers. Simple first-come-first-served approaches are ineffective because large jobs can fragment the available resources. Backfilling schedulers address this problem by allowing small jobs to move ahead in the queue, provided that they do not delay subsequent jobs.

Previous research showed that inaccurate estimates of execution times can lead to better backfilling schedules. My group characterized this effect on several real workloads, and showed that such overestimations lead to a less tightly-packed queue, allowing the backfilling scheduler more freedom to re-arrange jobs. Further, we identified the *shape* of jobs that limit this effect through queue congestion, and described simple, effective ways to prevent severe performance problems [46].

We used this work as a starting point to investigate a variety of other techniques for reducing average slowdown, including job queue randomization, queue sorting by job length, several varieties of speculation execution, job adaptation, weakening queue guarantees, and combinations of the above [32]. By combining several of these techniques, we were able to reduce average slowdowns by nearly an order of magnitude.

#### 4. Mobile Replication and Databases (Deno)

I have spent much of the last two years working on Deno [23], an asynchronous data replication system with database-like functionality specifically designed for use in mobile and weakly-connected environments. Asynchronous solutions for managing replicated data have a number of advantages over traditional synchronous replication protocols in such environments. They can operate with less than full connectivity, easily adapt to frequent changes in group membership, and make few demands on the underlying network topology. However, this functionality comes at a price. Asynchronous solutions are generally either slow and require reconciliations, or have lower availability because they rely on primary-copy schemes.

I designed a decentralized, asynchronous replica management protocol (single-author PODC [13]) that addresses these concerns. The protocol retains the advantages of current asynchronous protocols, but generally performs better, has fewer connectivity requirements, and higher availability. No server ever needs to have complete knowledge of group membership, and a given server only needs to be in intermittent contact with at least one other server to take full part in the voting and commitment process. As such, the protocol is highly suited for environments with weak or intermittent connectivity.

The protocol's strengths result from a novel combination of weighted voting and epidemic information flow. The latter is a process where information flows through a system like a virus passing from one host to the next. Each server periodically chooses a random communication partner and *pushes* information to it. Despite the randomness, a piece of data created at one server flows to all other servers exponentially quickly. The protocol is completely decentralized. There is no designated primary site for any data item, i.e., no single site "owns" an item and serializes the updates to that item. Any site can create new object replicas, and sites need only be able to communicate with a minimum of one other site at a time. Instead of *synchronously* assembling quorums, votes are cast and disseminated among system sites *asynchronously* through pair-wise, epidemic-style propagation. Any site can either commit or abort any transaction unilaterally, and all sites eventually reach the same decisions.

Deno's protocols use voting to arbitrate among competing updates to a given object. A server wishing to update an object creates the update record, propagates it to other servers, and waits for the update to *commit* before making the changes visible to the local applications. Simplifying, an update commits when a plurality of the servers vote for it as the next update for that object. Since I assume ACID semantics, the set of updates that commit on a given object are totally ordered.

The use of voting allows the system to have higher availability than primary-copy protocols [37, 38]. The use of *weighted* voting allows implementations to improve performance by adapting currency distributions to site availabilities, update activity, or other relevant characteristics. Each site has a specific amount of currency, and the total currency in the system is fixed at a known value. The advantage of a static total is that sites can determine when a plurality or majority of the votes have been accumulated *without complete knowledge of group membership*. This last attribute is key in dynamic, wide-area environments because it allows the protocol to operate in a completely decentralized fashion, eliminating performance bottlenecks and single points of failure.

The use of epidemic protocols divorces protocol requirements from communication requirements. Essentially, an epidemic algorithm only requires protocol information to move throughout the system *eventually*. The lack of hard deadlines and connectivity requirements is ideally suited to mobile environments, where individual nodes are routinely disconnected. Second, epidemic protocols remove reliance on network topology. Synchronization partners in epidemic protocols are usually chosen randomly, eliminating the single point of failures that occur with more structured communication patterns, such as spanning trees.

One of the benefits of Deno’s decentralized structure is that important operations, such as changes in group membership, can be performed without global consensus. My group’s description of the manner in which currency management is used to implement these changes [6] won a “Best Paper” award and was forwarded to the Journal of Distributed and Parallel Databases [7].

## 5. Current and Future Work

### Transactional Semantics

My current work, all based loosely on Deno, is moving in three new directions. First, in a collaboration with Dr. Franklin at Berkeley, we are extending our basic, single-object protocol [8] to multiple-object (transactional) updates [9]. Adding transactional support to epidemic protocols requires a number of tradeoffs to get good performance. We have implemented and evaluated several different approaches on a network of Linux computers, focusing on the performance implications of providing different levels of consistency. Unsurprisingly, we have so far found that strongly consistent protocols can be significantly slower than weakly consistent alternatives. However, the gap nearly disappears when updates are allowed to propagate across the system speculatively.

Independent of our research on transactional voting protocols, Holliday *et al.* also proposed a quorum-based epidemic approach that provides strong serializability and transactional semantics [10]. Holliday’s work assumes a more traditional replicated database environment, and static, globally-known currencies, whereas our work is geared more towards environments with weak-connectivity and incomplete system information.

### Wide-Area Security

The second new direction is in addressing security. In collaboration with Dr. Bhattacharjee at Maryland, we are building an infrastructure for providing secure transactional support for mobile databases [22]. This is clearly crucial for mobile systems with decentralized control and information. Our infrastructure protects highly-available, decentralized replication systems against both external and internal attacks. External attacks refer to those by entities that have not been authenticated into the system. This class of attack is prevented by using traditional public-key techniques.

Internal attacks are much more interesting. An internal attack refers to one in which an entity has been authenticated into the system, and then tries to subvert the protocol by misrepresenting others’ votes or lying about its own. The former case can also be handled by public-key signatures. The latter case, however, can only be addressed by modifying the transaction commit criteria to take the “trust” level of votes into account.

Our early results show that this modification of the commit criteria, while potentially expensive, still results in much more efficient transaction commits than the best competing approach from the literature. Our results also suggest that, far from hurting system scalability, the effect of the security modifications on system performance actually diminishes with increasing system size.

Additionally, we show that a technique originally proposed for the secure multicast domain can be used to replace use of digital signatures with message hashes throughout our infrastructure. This replacement reduces the overhead of digital signatures in our system by at least a factor of 1000.

### Ubiquitous Computing

Finally, my students and I are extending the Deno work in the direction of ubiquitous computing. We are investigating the support of aggregate, global attributes. For example, we are designing an asynchronous error-bounding technique that allows a single global bound to be partitioned/shared across all of the servers in a system. We have defined protocols that bound the total deviation, summed across all replicas in the system, from a “master” copy of a shared datum.

We are also designing decentralized replication protocols that scale to very large numbers of servers and objects. The key to making this work is in finding ways to efficiently summarize information about object cach-

ing and update propagation. We are currently working on a multi-tiered approach that uses a hierarchical name-space, and Bloom filters [3] to summarize the portions of the tree that a server caches.

## 6. References

- [1] M. Ahamad, P. W. Hutto, and R. John, "Implementing and Programming Causal Distributed Shared Memory," in *Proceedings of the 11<sup>th</sup> International Conference on Distributed Computing Systems*, May 1991.
- [2] C. Amza, A. L. Cox, S. Dwarkadas, **P. Keleher**, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, pp. 18--28, February 1996.
- [3] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422-426, July 1970.
- [4] A. L. C. C. Amza, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel, "Adaptive Protocols for Software Distributed Shared Memory," *Proceedings of the IEEE*, vol. 87, pp. 467-475, 1999.
- [5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," in *Proceedings of the 13<sup>th</sup> ACM Symposium on Operating Systems Principles*, October 1991.
- [6] U. Cetintemel and **P. J. Keleher**, "Light-Weight Currency Management Mechanisms in Deno," in *The 10<sup>th</sup> IEEE Workshop on Research Issues in Data Engineering (RIDE2000)*, February 2000.
- [7] U. Cetintemel and **P. J. Keleher**, "Light-Weight Currency Management Mechanisms in Mobile and Weakly-Connected Environments," invited paper in *The Journal of Distributed and Parallel Databases*.
- [8] U. Cetintemel and **P. J. Keleher**, "Performance of Mobile, Single-Object, Replication Protocols," in *The 19<sup>th</sup> Symposium on Reliable Distributed Systems*, October 2000.
- [9] U. Cetintemel and **P. J. Keleher**, "Support for Speculative Update Propagation and Mobility in Deno," University of Maryland UMIACS-TR-99-70, October 29 1999.
- [10] J. Holliday, R. Steinke, D. Agrawa, and A. E. Abbadi, "Epidemic Quorums for Managing Replicated Data," in *Proceedings of the 19<sup>th</sup> IEEE International Performance, Computing, and Communications Conference (IPCCC 2000)*, February 2000.
- [11] J. K. Hollingsworth, **P. J. Keleher**, and K. D. Ryu, "Resource-Aware Meta-Computing," in *Advances in Computing*, vol. 53: Academic Press, 2000, pp. 110-171.
- [12] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordechai, "Towards Integration of Data Race Detection in DSM System," accepted for publication in *The Journal of Parallel and Distributed Computing (JPDC)*.
- [13] **P. J. Keleher**, "Decentralized Replicated-Object Protocols," in *The 18<sup>th</sup> Annual Symposium on Principles of Distributed Computing (PODC '99)*, May 1999.
- [14] **P. J. Keleher**, "A High-Level Abstraction of Shared Accesses," *The ACM Transactions on Computer Systems (TOCS)*, January 2000.
- [15] **P. J. Keleher**, "The Impact of Symmetry on Software Distributed Shared Memory," accepted for publication in *The Journal of Parallel and Distributed Computing (JPDC)*.
- [16] **P. J. Keleher**, "The Relative Importance of Concurrent Writers and Weak Consistency Models," in *Proceedings of the 16<sup>th</sup> International Conference on Distributed Computing Systems*, 1996.
- [17] **P. J. Keleher**, "Sparks: Coherence as an Abstract Type," in *The 5<sup>th</sup> IEEE International Workshop on Object-Oriented Systems in Operating Systems*, 1996.
- [18] **P. J. Keleher**, "Symmetry and Performance in Consistency Protocols," in *The 13<sup>th</sup> International Conference on Supercomputing*, June 1999.
- [19] **P. J. Keleher**, "Tapeworm: High-Level Abstractions of Shared Accesses," in *Proceedings of the 3<sup>rd</sup> Symposium on Operating Systems Design and Implementation*, February 1999.
- [20] **P. J. Keleher**, "Update Protocols and Cluster-based Shared Memory," *Computer Communications*, vol. 22, pp. 1045-1055, July 1999.

- [21] **P. J. Keleher**, "Update Protocols and Iterative Scientific Applications," in *The 12<sup>th</sup> International Parallel Processing Symposium*, March 1998.
- [22] **P. J. Keleher**, B. Bhattacharjee, K.-T. Kuo, and U. Cetintemel, "A Security Infrastructure for Mobile Transactional Systems," University of Maryland UMIACS-TR-2000-19, July 5 2000.
- [23] **P. J. Keleher** and U. Cetintemel, "Consistency Management in Deno," accepted for publication in *The Journal on Special Topics in Mobile Networking and Applications (MONET)*.
- [24] **P. J. Keleher**, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proceedings of the 19<sup>th</sup> Annual International Symposium on Computer Architecture*, May 1992.
- [25] **P. J. Keleher**, S. Dwarkadas, A. Cox, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proceedings of the 1994 Winter Usenix Conference*, January 1994.
- [26] **P. J. Keleher**, J. K. Hollingsworth, and D. Perkovic, "Exploiting Application Alternatives," in *The 19<sup>th</sup> International Conference on Distributed Computing Systems*, June 1999.
- [27] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," in *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.
- [28] M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," in *International Conference on Distributed Computing Systems*, 1988.
- [29] T. C. Mowry, C. Q. C. Chan, and A. K. W. Lo, "Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.
- [30] D. Perkovic and **P. Keleher**, "Online Data-Race Detection via Coherency Guarantees," in *Proceedings of the 2<sup>nd</sup> Symposium on Operating Systems Design and Implementation*, 1996.
- [31] D. Perkovic and **P. Keleher**, "A Protocol-Centric Approach to On-The-Fly Race Detection," accepted for publication in *IEEE Transactions on Parallel and Distributed Systems*.
- [32] D. Perkovic and **P. J. Keleher**, "Randomization, Speculation, and Adaptation in Batch Schedulers," in *Supercomputing '00*, September 2000.
- [33] B. Richards and J. Larus, "Protocol-Based Race Detection," in *The 2<sup>nd</sup> SIGMETRICS Symposium on parallel and Distributed Tools (SPDT)*, August 1998.
- [34] K. D. Ryu, J. K. Hollingsworth, and **P. J. Keleher**, "Mechanisms and Policies for Supporting Fine-Grained Cycle Stealing," in *The 13<sup>th</sup> International Conference on Supercomputing*, June 1999.
- [35] K. D. Ryu, J. K. Hollingsworth, and **P. J. Keleher**, "Rate Windows for Efficient Network and I/O Throttling," University of Maryland UMIACS-TR-2000-53, July 2000.
- [36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs," in *Proceedings of the 16<sup>th</sup> Symposium on Operating Systems Principles*, 1997.
- [37] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRESS," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 188-194, May 1979.
- [38] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing Update Conflicts in a Weakly Connected Replicated Storage System," in *Proc. of the ACM Symposium on Operating Systems Principles*, 1995.
- [39] K. Thitikamol and **P. Keleher**, "Multi-Threading and Remote Latency in Software DSMs," in *The 17<sup>th</sup> International Conference on Distributed Computing Systems*, May 1997.
- [40] K. Thitikamol and **P. Keleher**, "Per-Node Multithreading and Remote Latency," *IEEE Transactions on Computers*, vol. 47, pp. 414-426, April 1998.
- [41] K. Thitikamol and **P. Keleher**, "Thread Migration and Communication Minimization in DSM Systems," *The Proceedings of the IEEE*, vol. 87, pp. 487-497, 1998.
- [42] K. Thitikamol and **P. J. Keleher**, "Active Correlation Tracking," in *The 19<sup>th</sup> International Conference on Distributed Computing Systems*, June 1999.

- [43] K. Thitikamol and **P. J. Keleher**, “Thread Migration and Load Balancing in Heterogeneous Environments,” in *The 5<sup>th</sup> ACM Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR)*, May 2000.
- [44] K. Thitikamol and **P. J. Keleher**, “Thread Migration, Load Balancing, and Heterogeneity in Non-Dedicated Environments,” in *The 2000 International Parallel and Distributed Processing Symposium*, May 2000.
- [45] Y. Zhou, L. Iftode, and K. Li, “Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems,” in *Proceedings of the 2<sup>nd</sup> Symposium on Operating Systems Design and Implementation*, October, 1996.
- [46] D. Zotkin and **P. J. Keleher**, “Job-Length Estimation and Performance in Backfilling Schedulers,” in *The 8<sup>th</sup> High Performance Distributed Computing Conference*, August 1999.