Portability Events

A Programming Model for Scalable System Infrastructures

Chris Matthews Yvonne Coady
University of Victoria
{cmatthew, ycoady}@cs.uvic.ca

Jonathan Appavoo
IBM Research
jappavoo@us.ibm.com

Abstract

Clustered Objects (COs) [1] have been proven to be an effective abstraction for improving scalability of systems software [2, 3]. But can we devise a programming model that would allow COs to live outside the specialized environments of these operating systems and still provide benefit? This paper presents an overview of SCOPE (Scalable COs with Portability Events), a prototype user-level library derived from the original implementation of COs in the K42 OS. Our initial results indicate that not only do the benefits of COs hold, but that the notion of a "portability event", responsible for maintaining the runtime environment of COs, provides a powerful programming model that will enable COs to be seamlessly transplanted into systems beyond K42. This paper overviews this programming model, provides details and preliminary results from its prototype implementation in SCOPE, and provides motivation to consider simple language mechanisms to further support portability events and this programming model in general.

Categories and Subject Descriptors D.4.8 [*Operating Systems*]: Performance; D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Clustered Objects, SCOPE, K42, scalability

1. Introduction

Symmetric MultiProcessors (SMPs) – specifically those where more than one processor operates in one shared memory space – offer a programming environment that is a natural extension of a typical uniprocessor. But OS development for SMPs that embraced this natural extension fell short of yielding high performance SMP OSes [1–7]. Part of the problem is that scalability has proven to be an elusive goal due to the negative impact of sharing in SMP systems. Sharing is a phenomenon that naturally results from caching on SMP systems. Figure 1 shows an example of how sharing takes its toll on the standard benchmark SDET [8]. The SDET benchmark simulates a representative workload at increasing levels of concurrency in order to generate a graph of throughput vs. offered load [8].

On Linux, the benchmark suffers from the effects of sharing. As more processors are added to satisfy the load, per processor utilization eventually lessens and the toll of sharing ultimately becomes so great that throughput actually drops. The other line in Figure 1 represents SDET on the K42 OS. K42 is a research OS designed with scalability in mind. K42 was designed to mitigate the effects of sharing by exploiting locality, and per processor data. The scalability characteristics of K42 are a vast improvement over Linux – as processors are added, throughput improves, though not quite linearly.

One strategy for reducing sharing is structuring systems in an object-oriented manner, so that individual requests on the system are serviced by different objects. Despite the fact that K42 is explicitly structured in an object-oriented manner with independent resources represented by independent object instances, when running with four processors throughput is only 3.3 times that of one processor. Running with 24 processors, throughput is 12.5 times that of one processor. Ideally, throughput should increase linearly with the number of processors. Closer inspection revealed that the SDET workload induces sharing on OS resources, thus limiting scalability. In K42, the remaining sharing in common code paths has been further mitigated by leveraging distribution, partitioning and replication in the form of a CO facility. As depicted in Figure 1, with this CO facility in place, the same workload yields a 3.9 times throughput at four processors and a 21.1 times throughput at 24 processors. Thus, with COs it is indeed possible to reduce sharing and produce highly scalable systems.

Mitigating the effects of sharing has proven itself as one way to increase scalability of SMP operating systems. But the effects of sharing are not just felt at the OS level. User-level programs can experience the same sharing related slowdowns regardless of the OS. This is the motivation for SCOPE, to provide some of the mechanisms used in K42 that reduce sharing to the user-level of other systems. However, the question remains as to whether or not the intrinsic dependencies on K42 specific features can be eliminated from the implementation of SCOPE, allowing these benefits to be realized in other systems. This paper details our extension to COs called portability events. Then details how portability events allow the use of clustered objects outside of the K42 kernel.

2. Background: K42 and the Clustered Objects Model

A CO presents the interface of a single object to the client, but in actuality is composed of several component objects. Each component handles calls from a specific subset of the machines processors, maximizing locality and mitigating the negative effects of sharing. Inside every CO, the notion of global information and distributed information is made explicit. Each type of data is separated out into different classes, of which the distributed data classes may have per processor instances. When a request is made, the CO infrastructure allows the programmer to customize where the request will be directed, and how to ultimately satisfy the request. Which data is

PLOS 2006, Oct. 22, 2006, San Jose, California, United States
Copyright © 2006 ACM 1-59593-577-0/10/2006…$5.00

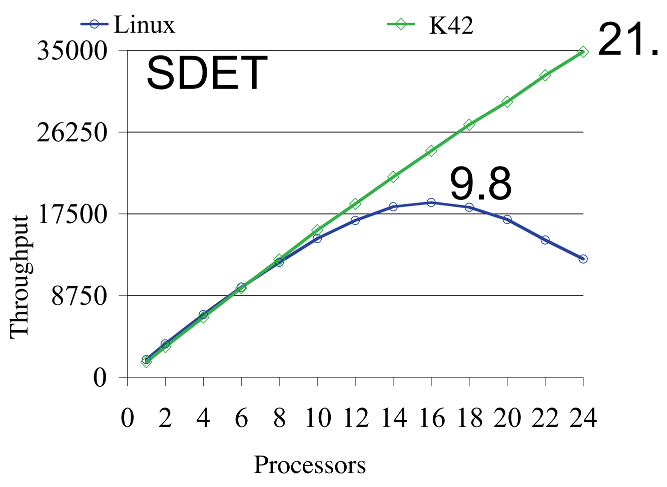

Figure 1. Throughput of a standard benchmark on two different OSes taken by the K42 team [9]. The effects of sharing limit scalability on Linux, K42 is not affected.

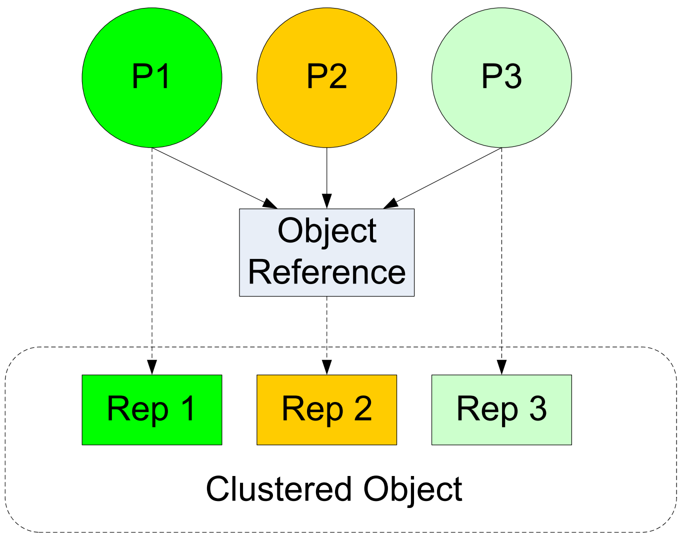

Figure 2. Processors P1,P2 and P3 access a CO through its common reference. The request made to the global reference redirects the call to a different rep for each processor, so that each processor is using a different rep. The root is not shown in this figure.

global, which data is distributed, and how the distribution and aggregation of the data occurs is defined by the creator of a CO and is transparent to the client. This customization is what helps make COs easier for programmers to build scalable objects and therefore services that will scale better.

COs are referenced by a global CO reference that logically refers to the whole CO. Each access to this common reference is automatically directed to a local representative(rep). Figure 2 shows a simple case with three processors marked P1, P2 and P3. Each processor accesses the CO through a global reference, then the CO system redirects the call to a local representative assigned to that processor. Every CO is made up of a root and one or more representatives. These components correspond directly with global and distributed data. Roots contain global data, reps contain instances of distributed data and the methods that control and aggregate the distributed data. A root is not directly accessible except through its representatives; so, in this respect representatives are responsible for providing local access to global data. Roots themselves are responsible for dictating which reps are assigned to handle requests in any given locality domain. In this context we define a locality domain as the memory used by a particular processor.

3. Why Events?

In order to separate the details of the runtime requirements from the use of COs, we introduce a programming model that includes the notion of a portability event. Conceptually, the event is a period in the current thread's execution in which a CO's runtime environment must be available. Once such an event starts, the current environment is determined and made available for the CO's access, and when the event ends, the environment is no longer available, and any appropriate tear-down activity is performed. This model provides a means of dynamically capturing precisely what COs need to execute, and thereby enables their subsequent decoupling from K42 and extraction into a user-level library in Linux and beyond, potentially even into the internals of other operating systems and virtual machines.

One fundamental difference between the K42 kernel environment and the Linux user-level that portability events allow us to deal with is K42's ability to map real addresses to different physical address on a per processor basis. This custom mapping is the backbone of the mechanism used to access a CO. Specifically, this mapping allows the local representative lookup table to be located at the same virtual address for each processor, but be backed by different physical memory and therefore a different table on each processor. The use of this local table means the dereference of a CO only takes two regular pointer dereferences. In K42, a reference to a CO is a pointer into this local table. Thus, to the programmer, an access of a CO c with method m looks like `**c->m();`. In order to abstract this dereference and ensure programmers do not leverage implementation dependant features of the CO infrastructure, K42 developers created a preprocessor macro called *DREF* which performs a CO dereference. A typical CO access in K42 actually looks like: `DREF(c)->m();`. Of course, in K42, the *DREF* is just defined to prepend the **. Beyond K42 however, *DREF* is a more substantial operation. For example, redefining the *DREF* can give us a way to make up for the lack of per processor memory. In a newly defined *DREF* we can look up which processor we are on, then access the appropriate local table.

Before a CO can be dereferenced, the availability of the runtime environment must be ensured. Similarly, after the dereference has taken place, there may be some tear-down logistics required. In the programming model proposed here, the portability event maps to the period of time in which this dynamic environment is available. Starting an event has to be simple, and in our current prototype implementation of this model requires client involvement. In order to ensure that the client is shielded from most of the details, the client program calls a simple macro. For example in out SCOPE implementation for Linux user-level, this allows the CO infrastructure to resolve which CPU that the current thread is running on, and to acquire a base pointer to the current CPUs local translation table. Figure 3 is an example of how a CO is accessed in K42 compared with how a CO is accessed in SCOPE.

The START_EVENT macro is the macro that sets up the current thread for a CO dereference. In our pthreads implementation, the macro checks a pthread key value to ensure the thread is initialized. In future versions, the macro might check if a representative migration to another processor has been triggered then act accordingly, and record garbage collection information. END_EVENT macro

```
CORef ref;
ClusteredObject::Create(ref);
//In K42
DREF(ref)->someMethod();
//vs. in SCOPE
START_EVENT;
DREF(ref)->someMethod();
END_EVENT;
```

Figure 3. An example of how a CO is called.

marks the end of an event. In future implementations of SCOPE it will record garbage collection information, and if necessary trigger a CO deallocation.

To port this event model to other systems, these macros must be appropriately modified, but everything else from the client's perspective remains stable. For instance in K42, START_EVENT and END_EVENT could be empty because in K42 all of the features required do not need to be triggered by the client program. But in Windows, a JVM, or a simple application program, the appropriate runtime environment associated with CO dereferencing requires further support. The ultimate goal of this programming model is to increase the portability of this effective approach for scalable systems. The model makes it simple for a CO written on one system, for example the pre-existing set of K42 COs, to run on other system, such as the Linux application library provided by SCOPE.

All of that said, if the client programmer knows which system the CO is intended for, and how that system works, they can opt to only call the event macros when they are actually needed, and not according to the model in general. In the case of the current pthread version provided by SCOPE, it would suffice to call the START_ EVENT only once when a new thread is created; however, doing so is not portable to other systems, or possible later versions of the pthreaded system. We believe this introduces a dangerous amount of flexibility in the implementation, and consequently argue that the systematic application of those linguistic mechanisms would enforce safe and predictable management for COs, without compromising any of the benefits.

4. Scope: Implementation Overview

In SCOPE, START_EVENT is responsible for preparing a thread with a virtual processor (VP) assignment. In K42's COs, the VP abstraction is the level at which per processor information is maintained. VPs are thus the logical entities to which representatives are assigned in SCOPE. Each VP has its own local lookup table. In K42, VPs are assumed to map closely to the physical processors of the machine.

Once a thread is assigned to a VP, a local lookup table can be created for that thread. That table will be used to store the cache of local representatives for the VP. Once a VP for the thread is resolved, the base pointer to the VP's local translation table has to be acquired and stored in per thread memory[1]. This value, and an offset into the table are how a we produce a different representative for each CO and VP.

Once START_EVENT has ensured that these values are set up, a simple dereference can take place. But START_EVENT can be further leveraged to provide more functionality! A pointer to the current generation record can be established (if garbage collection is active), allowing garbage collection information to be kept.

Once START_EVENT has happened, DREF can be used just as in K42; however, what actually happens in DREF is quite different.

[1] We use the pthread_key functionality of the pthread system to store thread specific data.

```
if(pthread_getspecific( *getVpKey()) == NULL ){
  VPNum *vp = assignVP ();
  pthread_setspecific( *getVpKey (),(void*)vp);
  unsigned long *offset=(unsigned long*)malloc(
                          sizeof (unsigned long));
  *offset=(*vp * sizeof(LTransEntry)
                 * NUMBER_OF_CLUSTERED_OBJECTS)
                          / sizeof(unsigned long);
  pthread_setspecific( *getLTransKey(),
                                      (void*)offset);
}
pthread_setspecific ( *getGenRecKey(),
                              (void*) activate());
```

Figure 4. The SCOPE START_EVENT macro.

```
#define PTHREAD_DREF(ref)                     \\
              (*(ref + *PTHREAD_LTRANS_OFFSET))
#define PTHREAD_LTRANS_OFFSET                 \\
((unsigned long*)pthread_getspecific(         \\
                        *COSMgr::getLTransKey()))
```

Figure 5. The SCOPE DREF macro. A thread specific key is used to get to the local table, then an offset (which is the reference passed into DREF) is added to find the table entry.

In the K42 DREF, all that happened was a double dereference. In SCOPE, the current VP's offset must be used, in combination with the CO's reference to find the local translation table entry that will redirect the call to the appropriate local representative. If no representative is assigned, a miss handling object will delegate a new or preexisting representative to the task. It is important to note that the offset and ref are added together. Out of all the operations possible, addition was used because it makes the DREF call type safe.

Finally, END_EVENT triggers any necessary cleanup. This entails checking if any dynamic operations are needed like a migration or hot swap, and triggering any necessary garbage collection statistics to be updated.

5. Preliminary Evaluation: Performance and Benefits

One of the simplest examples used to evaluate other CO systems and other concurrent systems has been an integer counter [10, 11]. An integer counter is a class with a single integer field. The class has an interface which consists of three methods:

inc add one to the this counter

dec subtract one from this counter

value get the current value of this counter

One can imagine a class like this being used to count frequent events. On systems with many processors there are more potential processors to cause sharing. The machine that this test was run on was a dual processor X86 based machine.

To test the SCOPE event mechanism we introduce 6 simple test cases. These test cases will be run on a dual processor machine, first on both CPUs, then on a single CPU.

Case 1 is a simple integer counter. This counter uses a primitive int field to store its value. In this case we expect to see sharing, and therefore the inherent penalty: reduced CPU utilization. This is how a naive programmer who is not thinking about sharing might implement the counter.

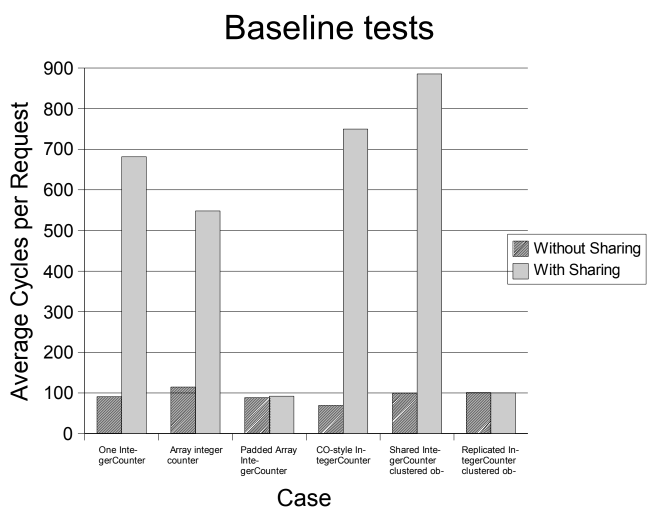

Figure 6. Average runtime results of the six integer counter test cases respectively, each running on one (without sharing) and two processors (with sharing).

Case 2 recognizes that sharing might occur, and breaks up the primitive int field into an array. Inc calls will increment the current processor's element of the array. When the value method is called, we must return the sum of the array elements. We expect to see no real sharing in this case, but since the array elements might be on the same cache line, we expect to see false sharing in some cases, and therefore poor performance.

Case 3 takes cache line sharing into account, and adds buffer space between elements of the array. We expect to see no sharing in this case as there is no shared data, and therefore better performance than the first two cases.

Case 4 is a modified CO that does not use the dereference mechanism. This class is designed to help us find the overhead caused by the dereference mechanism. We expect this case to perform poorly as it will have the effects from sharing, and the virtual dispatch.

Case 5 is a CO that uses the dereference mechanism, but has no data replication. We expect this to be the slowest of all the counters, as it has the sharing effects and the overhead from the SCOPE dereference.

Case 6 is a CO that uses the dereference mechanism, and uses replication to control sharing. In this CO there is one rep per processor. The *int* fields that we are incrementing are in these reps. When an `inc()` operation takes place the *int* on the local rep is incremented. When a value operation is requested the CO's root provides a linked list of all the replicas that we can traverse to find the summation of each processor's *int* values. We expect the replicated counter to perform very well as there is no sharing in the common case.

The results in Figure 6 show that the first three test cases perform as expected. The sharing and false sharing in Case 1 and 2 cause large slowdowns. Case 3 mitigates the sharing and hence does not experience the same slowdowns. As expected, a restructuring of the data can control sharing. Case 4 exhibits the sharing seen in Case 1 and 2, but still is 15% faster than a CO using the dereference mechanism (Case 5); however, Case 4 is 10% slower than Case 1 where we used the simpler class. Case 5 exhibits the same sharing as Case 1, 2 and 3, but also on top of the slowdown caused by sharing, Case 5 is slowed further by the more complex dereference mechanism of SCOPE. Case 6 exhibits no sharing, as expected for a fully replicated CO. The performance is only slightly worse (9%) than that of the padded array counter. The difference is due to the different dereference system and differences in the implementations.

The second set of tests shown in Figure 6 reveals only a small difference between each of the methods of organizing the integer counter when there was no sharing present (run on a single processor). The difference between the CO style counter and replicated CO is relatively small, 20 cycles. We attribute this difference to our SCOPE lookup mechanism.

In terms of performance, we believe this shows that a user-level library can indeed increase the scalability by controlling sharing in this simple counter example. But, can SCOPE provide the same programming benefits as previous implementations?

Previous work has demonstrated the numerous benefits of COs from many perspectives [1,3,4], and this list includes many benefits from the realm of software engineering, including reducing the need for ad hoc data access mechanisms, introducing evolution points to deal with changes in workloads and extending type safety to COs. The COs model inherently reduces the need for ad hoc systems like data structures that are indexed on a per processor basis, by imposing a flexible structuring of data that inherently helps the programmer control sharing. Although the interfaces of a CO do not change, the backing implementation can easily be changed (even dynamically) from a simple implementation for a small workload, to a more complex implementation that can deal with a heavy workload and still maintain throughput. Finally, a very desirable property of language extensions is type safety. Without any compiler modification, COs (most importantly the DREF macro) are type safe.

It is important to point out that it is precisely through leveraging this model of portability events that these desirable properties were maintained in SCOPE, just at they were in previous implementations of COs.

6. Continuing and Future Work

SCOPE still lacks many of the features that are present in K42's COs facility. Most of these features were motivated by K42's pragmatic approach to systems design. One such feature is what they call garbage collection. This is not garbage collection in the traditional sense of an automatic reclamation of unused objects, but rather a semi automatic cleanup of objects that have been manually signaled for destruction. K42's garbage collection is what allows them to make an existence guarantee on COs. Any reference to a CO is guaranteed to point to an active CO. Even after the destruction of a CO is signaled, that CO is kept active until all threads in the system are no longer able to access it.

In the context of garbage collection, SCOPE has to deal with one further fundamental difference between K42 and other environments: the longevity of threads. To make the existence guarantee, K42 uses a technique similar to Read Copy Update (RCU). RCU makes its guarantees about writing data with quiescent periods: it waits until all threads in the system are no longer able to use their current reference. In K42, they have a policy of having no *long living* threads. This means that all threads should terminate in a reasonable amount of time. Since operating systems are request driven, this plays out as a single short lived thread per-request. So, after a CO's destruction is signaled, the system just waits until all the requests that were active at the time finish, then destroys the CO. Because COs are always guaranteed to exist, they have the desirable property of not needing existence locking. This is considered one of the nicer benefits of COs.

We hope portability events will be a useful tool when implementing garbage collection; the expectation of user-level programs not having any long living threads is not reasonable for some programs. So, the techniques used in SCOPE will have to vary from those used in K42.

We also hope to show how the portability events will allow SCOPE to work in other systems. The SCOPE current implementation works at Linux user-level, using pthreads. It could prove interesting to try moving SCOPE to the Microsoft Windows platform or into a JVM, and then revalidate the performance and portability characteristics of SCOPE.

The use of COs in K42 is not completely transparent for the programmers of K42. The DREF must be used to access a CO. In the systems domain, this is considered a good calling convention because the extra overhead caused by the *DREF* macro is made explicit. The virtual dispatch used in object oriented programming has a similar per call overhead; however, its use is transparent to the caller. The transparent virtual dispatch has been accepted in many user-level applications, so it seems reasonable that if there were a transparent CO dereference mechanism, it may be accepted at user-level as well.

Aspect-Oriented Programming (*AOP*) provides mechanisms for concretely implementing cross-cutting concerns in a modular fashion. One of the controversial properties of AOP which may be useful to us is *obliviousness*. That is, when an aspect acts on some part of the system, the original code is not affected. One drawback of COs is that without compiler support, a CO access has to be surrounded in the *DREF* macro. AOP could be used to make this more transparent by applying the *DREF* macro automatically to CO calls, therefore making them appear like regular object accesses to the programmer. Making COs even easier to call could help them become more pervasive in programs, which is one of the original goals of COs and this work.

Automatically applying the *DREF* macro may have even more advantages, in that the added *DREF* calls could be customized for the calling CO. If, for example, there was a long lived CO like the K42 system manager which will never need to be garbage collected, or if a CO didn't use a RCU facility, the necessary tracking information could not be collected. This may have the effect of creating faster access times for COs that opt out of advanced features.

One problem with this (and AOP in general) is that the client program would have to first be compiled with an AOP compiler before being regularly compiled; however, it may be possible to prepackage an AOP compiler that is setup to just process the CO aspects like a simple preprocessor.

These are just a few of the directions which could be explored further to help improve SCOPE's language mechanisms.

References

[1] J. Appavoo, "Clustered Objects," Ph.D. dissertation, University of Toronto, 2005.

[2] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis, "K42 overview," IBM TJ Watson Research, Tech. Rep., 2002.

[3] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system," in *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 1999, pp. 87–100.

[4] E. Parsons, B. Gamsa, O. Krieger, and M. Stumm, "(de-)clustering objects for multiprocessor system software," in *Fourth International Workshop on Object Orientation in Operating Systems 95 (IWOOO'95)*, 1995, pp. 72–81.

[5] R. Bryant, J. Hawkes, and J. Steiner, "Scaling Linux to the extreme: from 64 to 512 processors," in *Ottawa Linux Symposium*, 2004.

[6] P. E. McKenney, J. Slingwine, and P. Krueger, "Experience with an efficient parallel kernel memory allocator," *Softw. Pract. Exper.*, vol. 31, no. 3, pp. 235–257, 2001.

[7] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The performance implications of thread management alternatives for shared-memory multiprocessors," in *SIGMETRICS '89: Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 1989, pp. 49–60.

[8] S. L. Gaede, "Perspectives on the SPEC SDET benchmarks," January 1999.

[9] J. Appavoo, "Personal communications with Jonathan Appavoo," September-December 2005.

[10] ——, "Clustered Objects: Initial design, implementation and evaluation," Master's thesis, University of Toronto, 1998.

[11] N. Shavit and D. Touitou, "Software Transactional Memory," in *Symposium on Principles of Distributed Computing*, 1995, pp. 204–213.

Acknowledgments

This work was partially supported by DARPA under contract NBCH30390004.