Abstract—We present core elements of Samhita, a new user-level software distributed shared memory (DSM) system. Our work is motivated by two observations. First, the rise of many-core architectures is producing a growing emphasis on threaded codes to achieve performance. Second, architectural trends, especially in high performance interconnects, suggest a new look at overcoming the bottlenecks that have hindered DSM performance. Samhita leverages the capabilities of remote direct memory access (RDMA) interconnects, and views the problem of providing a shared global address space as a cache management problem. Performance results on two 256 processor clusters demonstrate scalability on microbenchmarks and two real applications. The results are the largest scale tests and achieve the highest performance of any DSM system reported to date.

Keywords—distributed shared memory; distributed run-time systems; distributed systems

I. INTRODUCTION

The emergence of many-core architectures and the need to exploit parallelism is perhaps the most important challenge in computing today. Programmers are striving to exploit concurrency across virtually all platforms and application areas. The need for better programming models is widely discussed. On platforms with the largest market share (e.g., portable devices, laptops, desktops, servers) the traditional shared memory programming model dominates. Consequently, there is a growing ecosystem of shared memory parallel programmers, tools and design practices. However, this community of parallel programmers needs to grow even faster.

Meanwhile, on high-end platforms distributed memory architectures still have their place. They provide a cost-effective approach to meeting the extreme demands for processing and memory of critical problem domains such as scientific simulation and data-intensive computing. Unfortunately, the dominant programming model for distributed memory is message-passing, a programming model that is widely seen as more difficult than a shared memory model, especially for inexperienced programmers.

Hence, we have a challenge. On the one hand, we want to leverage and encourage the shared memory parallel programming community, e.g., pthreads programmers, existing shared memory codes. On the other hand, we want to leverage the cost and scalability advantages of distributed-memory systems. One possible solution is distributed shared memory (DSM). For more than 15 years, researchers have attempted to build DSM systems, which provide shared memory semantics over physically distributed memory. The goal is to hide the necessary underlying data distribution from the programmer, thereby providing a simpler programming model than a typical distributed memory model such as message passing. A critical component of any DSM system is its consistency model, which defines the semantics of memory accesses. To improve performance, consistency models are typically relaxed to enable greater parallelism in data accesses. DSM systems have been implemented as hardware [1]–[3], software [4]–[15], and hybrid systems [16]–[19]. Unfortunately, although much of this work yielded valuable insight, the lasting impact of these systems has not been high.

Historically, DSM systems were doomed by unacceptable performance due to relatively slow interconnection networks. However, we believe that the data in Table I suggests it is time to give DSM systems a new look. The data shows two major trends that have occurred since the early work on DSM systems in the late eighties and nineties. First, network performance has increased significantly, with network latencies falling by nearly three orders of magnitude. More importantly, network bandwidth is almost comparable to main memory today, as compared to the early nineties, when it was more than two orders magnitude slower than main memory. Second, memory latencies have remained largely flat, while processor clock speeds have grown by over two orders of magnitude. This increasing gap (the memory wall) is largely hidden by careful cache management and accurate hardware prefetching. These two trends fundamentally change the design challenge of a DSM system by casting it as a cache management problem. In effect, if we can design sophisticated cache management, particularly prefetching, we can exploit the similarity in bandwidth between network and memory today and hide the higher latency of remote memory access to provide shared memory performance comparable to current hardware solutions.

One final observation that motivates a new look at DSM is the emergence of Remote Direct Memory Access (RDMA), the technology underlying the impressive interconnect performance numbers in Table I. RDMA provides a mechanism
for moving data from the memory of one computer to another without involving either computer’s operating system. It is perhaps ironic that RDMA is a more natural fit for the shared memory programming model than the message passing model. While sequential, two-sided message passing paradigms such as MPI have been optimized for execution over RDMA using buffer affinities, large message transfer still involves additional higher layer messaging and added software latency. RDMA requires only one side to have the information to complete a communication, which corresponds nicely to a shared memory programming model, where one thread needs only a pointer in the shared global address space to access any data from any other thread.

Motivated by these observations, we have designed and implemented Samhita, a prototype DSM system. The goals of our work are to explore the design and implementation challenges of DSM over modern interconnects, to investigate appropriate memory consistency models, and to evaluate the performance of various classes of applications over a DSM system that leverages emerging architectural trends. Samhita is a portable, user-level system. It is architected with memory servers and compute threads that may be co-located on the memory servers, enabling its use both for multi-threaded applications and single-threaded applications that require a large address space. Our current implementation reflects a new regional consistency (RegC) model, which is based on the common POSIX threads model. Existing Pthreads codes can be easily ported to run over Samhita. RegC is a superset of scope consistency (ScC) [20], with significant data transfer improvement over release consistency (RC) [21] or its modified form, lazy release consistency (LRC) [22]. Details of the RegC model are beyond the scope of this paper.

The rest of the paper is organized into four sections. In Section II we describe the architecture of Samhita. The details of our prototype implementation are presented in Section III. Results of a performance evaluation are given in Section IV, and the last section summarizes our contributions.

II. ARCHITECTURE OF SAMHITA

In DSM systems the physical memories of the computers on which the application is running are combined together to form the shared address space. In early DSM systems such as Ivy [4], Mirage [5], Mermaid [6], Munin [7] and TreadMarks [8], each node maintains an entire copy of the shared address space. This presents a significant limitation since the shared address space is limited to the physical memory available on a single cluster node. In later DSM systems such as JIAJIA [12] and Cashmere-LVM [11], the shared address space is distributed across the physical memories of the nodes running the application. However, this design couples the shared memory footprint with the number of threads needed to run the application, preventing a single threaded application running on one node from accessing a large pool of shared memory. In contrast, Samhita separates memory servers from compute resources, enabling cluster designs that use a dedicated bank of memory servers to serve applications independent of the number of threads. Furthermore, earlier DSM implementations were implemented on relatively slow interconnection networks such as ATM and Fast Ethernet, which had significantly lower bandwidth than main memory subsystems of their time. The earliest attempt to use memory-mapped communication (similar to RDMA) to implement DSM systems is Virtual Interface Architecture [13]. Later DSM systems like NGDSM [14] and ViSMI [15] are implemented specifically using the InfiniBand switched fabric, but continued to maintain complete copies of the address space similar to early DSM systems, significantly limiting the size of the shared address space.

The main motivation for the architecture of Samhita is to (a) provide multi-threaded applications with a consistent view of memory and (b) provide single-threaded applications access to a large address space. This is achieved by a novel approach wherein we separate the notion of serving memory from the notion of consuming memory for computation. Samhita associates every thread of the application with a local cache to access the global address space. This cache can be viewed as another level in the memory hierarchy. The problem then reduces to efficiently managing this new level of cache by detecting memory access patterns and intelligently prefetching and discarding data to hide the latency of remote memory access. This approach is orthogonal to the use of relaxed consistency models and hence can be used in conjunction with such models to achieve cumulative performance benefits. Furthermore, Samhita is designed to support multiple interconnection networks directly over their native protocols, which bypass the operating system. To
achieve portability across interconnects, Samhita uses a layered approach, with an underlying communication layer efficiently abstracting the physical communication semantics.

Samhita’s architecture consists of (a) compute servers, (b) memory servers, and (c) resource managers. These components run on the nodes of a cluster, which are interconnected using a high-performance interconnection network such as InfiniBand, Myrinet or 10 Gigabit Ethernet. The role of each component is as follows.

**Compute servers** execute one or more threads of control from one or more applications and access memory served by the memory servers. Multiple application threads may execute on different compute servers and have access to a consistent global address space—from the application’s perspective processing cores on a physically remote compute server appear as individual cores of a shared memory system. Samhita exposes a *fork-join* execution model similar to POSIX threads. An application starts with one thread of execution, which then spawns other threads. Placement of threads on to individual compute servers is handled by a resource manager. Note that individual compute threads of the application actually correspond to processes in Samhita (see Section III-C). The Samhita runtime system transparently shares the global data across the different processes, effectively these processes appear as threads sharing the global address space of the process.

**Memory servers** act as a pool to provide the global address space. In Samhita, each page of the global address space is served by a single memory server, i.e., a “home based” protocol. The maximum size of the global address space is equal to the amount of memory exported by the pool of servers. Samhita’s global address space is partitioned among the memory servers. The resource manager is responsible for co-ordinating memory allocation requests from compute servers and allocating a set of memory servers to satisfy each allocation. To mitigate the impact of hot spots, each allocation is strided across multiple memory servers.

**Resource managers** co-ordinate the actions of compute servers and memory servers and provide core functionality for implementing synchronization primitives. More specifically, the three tasks handled by the resource managers are:

**Job startup.** A well known resource manager coordinates the startup of a new application and allocates an available compute server. The application starter first contacts the resource manager to retrieve a reference to a compute server and then contacts the process starter component running on each compute server to start the application.

**Thread placement.** When an application starts a new thread, the run-time system contacts the resource manager to get the compute server on which to start the new thread and uses the process starter component to instantiate a new copy of the application. The runtime system maps the data section of the new loaded binary into the global address space to make it a thread of the running application instead of a complete new instantiation.

**Synchronization operations.** These include barrier synchronization, condition variable signaling and lock acquisition/release. Since these operations are relatively frequent and can easily become a bottleneck, Samhita uses a scale-out architecture with multiple resource managers to improve performance of synchronization operations. During the initialization of a synchronization variable, a well-known resource manager returns a reference to a particular resource manager that is responsible for handling all synchronization operations on the variable.

### III. IMPLEMENTATION OF SAMHITA

The implementation of Samhita includes three components: the memory server, the resource manager and a runtime system that runs on each compute node and is linked to the application. Concern for portability and low overhead drives the implementation. For a system to be portable it must require no modifications to the operating system kernel and must depend only on standard system libraries. The current Samhita prototype runs on the GNU/Linux family of operating systems. Samhita depends only on standard system libraries, namely *glibc* (GNU C library implementation of the C standard library) and *libelf* (part of elfutils). Samhita requires that *glibc* support the Native POSIX Thread library implementation of POSIX Threads. To reduce overheads, Samhita interfaces directly with the interconnection network hardware.

Samhita presents a minimalistic API to the programmer that mimics the POSIX threads API (we are working on a transparent wrapper to enable existing POSIX threaded codes to run over Samhita). To maintain consistency of the global address space, Samhita provides barrier synchronization, condition variable signaling and mutual exclusion locks synchronization primitives. The current prototype implementation of Samhita implements a barrier synchronization protocol, which is described in Section III-D.

#### A. Samhita Communication Layer (SCL)

The Samhita Communication Layer (SCL) provides a minimal set of APIs required for Samhita to interact with any interconnection network. The current implementation of SCL supports the Quad Data Rate (QDR) InfiniBand hardware. In principle SCL is similar to the ADI layer of MPICH. The primary design decision in SCL is the fundamental abstraction for communication—either a serial protocol similar to MPI or a new direct memory access based protocol. Implementing a serial line abstraction over memory based hardware such as RDMA imposes additional protocol overhead in exchanging information on registered memory.
In contrast, a memory based communication layer can be implemented efficiently over physical serial line hardware such as TCP/IP over 10 Gigabit Ethernet. This reasoning lead us to choose a communication abstraction based on direct memory access for SCL.

The APIs provided by SCL can be categorized as connection management, send/receive primitives (for small message transfer), get/put primitives (for arbitrary length data transfer using direct memory transfer), test/wait primitives (for non-blocking operations) and memory allocation/registration primitives.

B. Samhita cache management

The global address space in Samhita is divided into pages. The page size is currently set to the operating system virtual memory page size of 4K. Each page of the global address space is served by a single memory server. Since Samhita operates at the granularity of pages, the impact of false sharing is exaggerated. To reduce the impact of false sharing Samhita utilizes a multiple-writer protocol. Whenever an application accesses a memory location in the global address space, the access is through its local cache. If the page corresponding to the memory location is not present in the cache the run-time system gets a copy of the page from the corresponding memory server and stores it in the local cache. To reduce the number of misses for applications that exhibit spatial locality, we use cache lines consisting of multiple pages. In essence the Samhita run-time uses the concept of demand paging to populate the local cache.

We also use anticipatory paging, or prefetching, to exploit spatial locality. On encountering a cache miss, Samhita places a request for the missing cache line and an asynchronous request for the adjacent cache line. In principle, this design is similar to modern microprocessors, where the missing cache word is brought in synchronously, with the rest of the cache line being populated asynchronously by the cache controller.

It is important to minimize the cost of data retrieval on a cache miss. If the missing cache line is fetched into a buffer and then memcpy'd into the application address space, an unacceptable overhead results, e.g., ~38us for a 64K cache line. To avoid the extra copy we can allocate the missing page in the application address space, register it with SCL and fetch the data from the memory server directly into the application address space. However, the cost of a memory registration and de-registration operation is still ~30us.

This level of overhead is unacceptable, given that the actual data transfer takes almost the same time. The solution is to avoid the cost of both the extra copy and the memory registration by changing the mapping between virtual addresses and the underlying physical memory address using the mremap system call. In this approach, the Samhita run-time uses a pre-allocated cache region, which is registered with SCL. On a cache miss we fetch a copy of the page into the pre-allocated region. We then use the mremap system call to modify the mapping of the physical pages from the cache region into the faulting address in the application’s virtual address space. This reduces the overhead of fetching pages from the memory server from tens of microseconds to about 4 us/cache line. When we evict a page due to a capacity miss, the run-time re-maps the physical pages from the application’s virtual address space back into the cache region managed by the Samhita run-time system.

In Samhita every page in the local cache can exist in one of two modes, read-only or read-write. The granularity of the local cache in Samhita is a cache line. Similar to a mode associated with any page, every cache line belongs either to a read list or a write list. A cache line belongs to the read list if every page of the cache line is cached in read-only mode. A cache line belongs to the write list if one or more pages of the cache line are cached in read-write mode.

In Samhita the cache eviction policy is biased against pages that exist in read-only mode in the cache. Normally, the victim for page eviction is a cache line from the read list, selected using the first-in, first-out (FIFO) algorithm. However, when the number of cache lines in the read list falls below a threshold the victim is then selected from the write list, again using the FIFO algorithm, i.e., the cache line that was added first to the write list is the victim chosen for eviction.

C. Samhita global address space management

Management of the shared address space in Samhita involves detecting cache misses and transitions from read-only to read-write modes. This is achieved by using the mprotect system call in conjunction with a SIGSEGV handler. The SIGSEGV signal is raised by the operating system for any invalid memory access. We install a signal handler for the SIGSEGV signal, which is called every time the signal is generated. We describe the following cause-effect scenarios for a SIGSEGV signal to be generated in our run-time system.

1) The application tries to read a memory location in the Samhita global address space, but the page associated with this location is not present in the local cache. When the signal handler is called the Samhita run-time system requests a copy of the page from its corresponding memory server and a copy is cached in read-only mode. The faulting instruction is restarted and the second access is successful.

2) The application tries to write to a memory location in the Samhita global address space, but the page associated with this location is not present in the local cache. When the signal handler is called the run-time system requests a copy of the page from its corresponding memory server and a copy is cached in read-only mode. The faulting instruction is restarted.
3) The application tries to write to a memory location in the global address space, and the page associated with the location exists in the local cache but in read-only mode. The signal handler creates a twin of the page and the copy is upgraded to read-write mode. The faulting instruction is restarted and the next access is successful.

If a page in the local cache is in read-write mode, updates made to the page need to be propagated to the corresponding memory server when the page is evicted from the cache. Recall that Samhita uses a multiple-writer protocol to reduce the impact of false sharing, which requires a mechanism to accurately detect the write set within a page. To compute the write set of a page we use the exclusive disjunction (XOR) difference between the current copy of the page and the copy of the page on the memory server. This XOR difference is called a page diff of a page. To compute a page diff we use an approach based on page twins similar to the technique described in [7].

Each compute thread in Samhita is a process. During application start the run-time system maps the data segment of the application binary to the shared global address space transparently. When a process corresponding to a thread sends the changes made to the pages in its local cache to the appropriate memory servers to update their copies. Given this information from each compute node, the memory servers construct and send lists of pages to be invalidated at the respective compute nodes, which in turn send page diffs for the invalidated pages to the corresponding memory servers to update their copies. The case where only one compute thread holds a copy of a page in read-write mode is special, in that the corresponding memory server does not invalidate its copy. Instead, the memory server marks its copy of that page as invalid. In this way, at the end of the stage every memory server either has a current copy of the page or knows the compute thread that has the current copy. A similar exchange of information between compute threads and memory servers identifies cached pages in read-only mode that must be invalidated.

When a compute thread encounters a cache miss after the barrier synchronization operation and requests a copy of a page that has been marked invalid at the memory server, the memory server sends a request to the compute thread holding the current copy of that page for a page diff to update the memory server copy. The compute server on receiving this request stops the execution of the compute thread and creates a similar exchange of information between compute threads and memory servers to update their copies. The barrier synchronization protocol does not invalidate its copy of the page or downgrade the protection of the page from read-write to read-only when it receives this request for a page diff. This barrier synchronization protocol reduces the set of invalidated pages and associated traffic to only those pages that have been modified on more than one compute node, which is the lower bound for communication overhead.

IV. PERFORMANCE EVALUATION

In this section we describe several performance studies that evaluate the performance of Samhita and of applications run over its shared memory programming model. The experiments were carried out on two different clusters. The first, System G, is a 2600 core cluster (326 node), with each node containing two quad-core 2.8GHz Intel Xeon (Penryn Harpertown) processors and 8GB of main memory. The second platform is Ithaca, a 672 core cluster, with each node containing two quad-core 2.26GHz Intel Xeon (Nehalem Gainestown) processors with 24GB of RAM. Both the clusters are interconnected over quad data rate (QDR).
Section IV-A presents the evaluation using synthetic benchmarks, while Section IV-B presents the evaluation of Samhita using LU factorization. We choose to focus on LU, since there is considerable work on efficient MPI implementations of this benchmark. Furthermore, LU has significant data transfer requirements, which stresses DSM systems. Lastly in Section IV-C we present an evaluation of the Blackscholes application from the PARSEC benchmark [23], which illustrates the minimal effort required to port existing threaded applications to Samhita.

A. Synthetic benchmarks

In our synthetic benchmarks we measure the memory bandwidth achievable in Samhita in terms of read and write operations. The read memory bandwidth is measured by performing `memcpy` from the Samhita global address space to a locally allocated memory region of the same size. The write memory bandwidth is measured by performing a `memset` operation on to a memory location in Samhita’s global address space. For both the read and write memory bandwidth tests we use an allocation of 1GB and measure the bandwidth obtained by varying the local cache size used by the compute threads. The cache sizes used in the benchmarks are 256MB, 512MB, 768MB, 1GB and 2GB.

The performance results in Figure 1 show that Samhita achieves nearly 2.2 gigabytes per second on Ithaca when reading from remote memory and writing to local memory. This compares well with the memcpy bandwidth within a single Nehalem node of ~3.9 GB/sec. This performance is also close to the achievable maximum of 2.6GB/sec on quad data rate InfiniBand on this platform. The performance of memcpy on System G is bound by the memory bandwidth of the shared bus design in the Penryn processor. Internal timing results show that pages are brought in from the remote memory server at nearly 2.6 GB/sec, but the lower memory bandwidth on the processor increases the time taken to copy data from the global address space to the locally allocated memory. Since both Intel and AMD have moved to cc-NUMA architectures, we expect Samhita’s performance to track the results on Ithaca and be bound by the interconnect’s bandwidth.

The performance results in Figure 2 show the overhead of transitioning a page from read only mode to read-write mode, creating the twin page and calculating and transmitting the XOR difference. Since the size of the memory in the global address space (1 GB) being memset is larger than the cache (except for the cache size of 2048MB), the cost of memset includes the cost of handling capacity misses and associated page evictions. Even in this scenario Samhita achieves a bandwidth of 790 MB/sec, which is 81% of the achievable bandwidth on 10 Gigabit Ethernet using TCP Offload Engine (TOE) and exceeds the achievable bandwidth by 23% on the non-TOE stack as reported in [24].

The largest overhead in Samhita is associated with SIGSEVs. There are two kinds of SIGSEGV costs, one associated with a cache miss and one on a protection fault. The costs presented are per cache line (64K) for the memset microbenchmark with a cache size of 512MB on SystemG. The average cache miss cost is ~40us including cache line fetch (64K), computing the page diff and sending it to the memory server, which translates to less than 3us/page. The data transmission cost is 28us for transmitting 64K on QDR InfiniBand. The average cost for a protection error is ~3.2us, which includes twin creation time. Achieving these performance levels is a challenge. For example, a processor cache miss on System G costs 180ns, whereas the interconnect can send a small message in ~1.2us. Our implementation is highly optimized to operate under these performance constraints. In contrast, the optimized MVAPICH implementation of MPI over InfiniBand takes ~5us to transmit a 4K page, without having to perform any diff computations or SEGV handling.
While we use memset and memcpy to measure read and write performance, in the future, we will implement these operations directly on the memory servers, without fetching data to/from compute nodes.

**B. LU factorization**

A standard HPC benchmark is the LU factorization of a dense $n \times n$ matrix. Using the Samhita API, we implemented a simple shared-memory parallel version of the right-looking Level 3 BLAS version of the LU algorithm, patterned after the ScaLAPACK routine `dgetrf`. Our algorithm is parallelized over block-columns of the matrix, with block size 128. We measured performance for various problem sizes and numbers of threads. For comparison purposes, we also measured the performance of the equivalent MPI based ScaLAPACK routine (`pdgetrf`) run with a $1 \times p$ processor topology, where $p$ is the number of processor cores.

Figure 3 shows weak scaling results for a typical case, run on Ithaca. Performance on smaller and larger problem sizes, and on System G is very similar. By ‘weak scaling’ we mean that the data set size $n^2$ scales with the number of cores. For all experiments on Ithaca we assigned 8 processes per compute node, and used 16 memory servers, with each memory server serving up to 15GB of memory. We see that for modest core count, LU over Samhita trails ScaLAPACK by only a few percent. At 64 threads, Samhita is slightly faster than ScaLAPACK; and the advantage grows as ScaLAPACK incurs greater and greater communication costs for large $p$. This is not unexpected since this version of the parallel LU algorithm (parallelized over block columns) is known to hit a scalability wall due to the fact that the number of block columns per core shrinks as $p$ grows. The Samhita version is able to scale reasonably well up to 256 cores.

Strong scaling results from Ithaca are given in Figure 4, where the number of threads grows but problem size is fixed at $n = 30000$. Again we see that the Samhita implementation tracks very well the performance of the MPI based ScaLAPACK implementation, and surpasses it at $p = 64$.

Samhita also provides a large address space for single threaded applications. To illustrate this scenario we ran a single threaded LU benchmark, with increasingly larger problem sizes. When the cache size boundary is crossed a performance penalty of less than a factor or two occurs. Beyond that the performance degrades very slowly as the problem size grows. Since the LU algorithm makes repeated passes through essentially the entire data set, the cost of capacity evictions in this case is high. In contrast, the only alternative today for single threaded applications needing a large address space is to swap to disk, which incurs a performance penalty of over five orders of magnitude.

**C. Black-Scholes application from PARSEC**

This application calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE). We used the large input data set from the PARSEC suite, which had 10 million entries. To run this benchmark, we trivially ported the original pthread implementation to Samhita. The benchmark is a scalable master-worker algorithm, and shows excellent strong scaling up to 32 cores (speedup of $\sim 22$ for 32 cores). There is a subsequent drop-off due to the limited problem size used in the benchmark.

**V. CONCLUSION**

Recent and future trends in interconnect performance, relative to processor and main memory performance, warrant a new look at distributed shared memory. We have presented the Samhita distributed shared memory system implemented over quad data rate InfiniBand networks. Samhita uses a cache based architecture, and performance results from two different 256 core clusters show the scalability of Samhita for real applications. To our knowledge, the results reported are both the largest scale tests by an order of magnitude
and achieve the highest performance of any DSM system reported to date.

We are extending Samhita in several directions. First, we are exploring pre-eviction policies that complement prefetching. Pre-eviction operates in the background and automatically evicts least recently used pages when the cache is almost full, thereby removing page diff from the critical path on a capacity miss. Second, we are developing load balancing strategies that can migrate an application thread across compute nodes in response to varying computational load.

REFERENCES


