Distributed State Sharing and Predicate Detection over RDMA

Paper #75, 11 pages

ABSTRACT

We present SST (Shared State Table), a framework for nodes in a single rack connected by RDMA to share local states and detect system events. An SST over a group of nodes is a table consisting of a row for each member of the group and columns representing state variables. We provide a mechanism for defining system events (called predicates) and callbacks (triggers) over the entries of the state table. The triggers are executed when the predicates are detected to be true. SST focuses on minimizing the time to detect predicates with the secondary concern of judicious use of RDMA resources. SST optimizes for RDMA operations and abstracts them from the programmer, thus providing a convenient interface to code applications to run over RDMA networks. We carefully examine the characteristics of one-sided RDMA reads and writes to find an optimal design for the SST, and provide new insights into system design using RDMA for peer-to-peer architectures. SST adds minimal overhead over raw RDMA primitives and scales linearly with the number of nodes. Our experimental study measures the delay required for detecting different categories of predicates, and illustrate the use of SST in scenario based on OSPF routing.

1. INTRODUCTION

A fundamental component of any distributed system is the mechanism employed for sharing of state between its components. Replicas of a service running in a rack of a data center might share information about load, computation progress, parameters, and the servers might adaptively modify their behaviors as the shared state evolves. Closely coupled distributed systems share application state and user-defined data relevant to their distributed functionality. System monitoring and management tools have a strong dependence upon state sharing and tracking. For example, load balancers, replica managers, and other fault-tolerance tools, typically exchange short messages frequently, approximating a shared global view of the system by tracking the evolution of local states.

Although there has been prior work on tracking or data-mining state to monitor global system status, for example in the Astrolabe [13] system, most existing systems were designed to run over the TCP/IP network stack, which introduces substantial delays due to the need to process packets in the OS kernel. Such systems have often been deployed at genuinely massive scale, and hence are designed to tolerate relatively high latencies and low data sharing rates. Our premise is that these decisions result in latencies are far higher than necessary. For example, state sharing systems that run at rack scale often exchange just a few parameters: load, queue length, etc. The control programs that coordinate network routers or switches at the top of the datacenter routing hierarchy similarly require relatively small amounts of shared state, with which they optimize route selection and avoid congesting links or the routers themselves. Were a solution available that could support this functionality at sharply lower latency, the door would be opened to adaptive behaviors well beyond what classical management infrastructures currently achieve.

With this objective, we focus upon RDMA networking technology. RDMA eliminates the need for CPU processing at the end host by providing direct memory access through an RDMA-capable NIC. RDMA is dramatically faster than kernel-mediated TCP/IP and is finding widespread adoption in modern datacenters and HPC clusters, to the extent that the technology is quickly becoming a de-facto standard.

Given an RDMA infrastructure, it would of course be possible to revisit classical distributed monitoring and management tools, recoding them to replace their TCP message-passing layers replaced by equivalent RDMA messaging. However, such an approach would be unlikely to achieve the highest possible performance because the entire architecture of existing tools is ill-fitted to a zero-copy data sharing model. To realize the full benefit of RDMA, it is far preferable to design new solutions, working from the bottom up to fully leverage RDMA. Such an approach can yield dramatic performance gains, as seen in recent work on key-value stores

[6, 10]. Here we do something similar for state sharing. Note that our target use case is poorly matched to the key-value model, and hence that these recent results do not solve our problem.

In this paper, we present **SST** (Shared State Table), a framework to share local states and detect system events for servers in a single rack connected by RDMA. SST consists of two main components, the **State Table** and the **Predicate Detection** subsystem. The state table consists of rows for each node with columns of the table denoting state variables, and the predicate detection system allows applications to register (predicate, trigger) pairs to react to system events. A predicate defines a set of conditions on the state variables, and triggers define actions that need to be taken when their corresponding predicate is true.

As an example of how SST might be used, consider an application running on a group of nodes that involves sending messages from a master node to all the other nodes with the condition that the new message be sent only when the previous one has been delivered to all the nodes. Suppose the nodes have identifiers $\{0, 1, ..., n\}$ with 0 being the ID of the master node. Let the messages have integer identifiers in an increasing sequence. If `statetable` is an instance of the SST state table, we can define `statetable[i].msg_num` to be the ID of the most recent message received by node i. The condition that node 0 should send message k if all nodes have received message $k-1$ can be expressed as `statetable[i].msg_num`$= k-1$ for all $i \in \{1, 2, ..., n\}$. In our terminology, this condition is the *predicate,* and sending the next message is the *trigger.*

SST maintains an updated copy of the state table in the local memory of every node by posting RDMA operations (reads and writes), and it continuously evaluates predicates on the local copy. It focuses on minimizing the time to detect predicates and is targeted at low-level network applications where performance is critical. SST makes optimal use of RDMA operations and abstracts them from the programmer, thus providing a convenient interface for programming distributed applications to run on RDMA networks.

We have implemented and compared two versions of SST, **SST-reads** and **SST-writes**, based on one-sided RDMA reads and RDMA writes, respectively. Our paper makes the following contributions:

1. We propose a predicate-driven event model and show that it represents a useful abstraction for important classes of computations in distributed systems.

2. We present a high-performance implementation of this new model, and we evaluate the solution in a variety of RDMA settings.

3. We explore various implementation choices, notably SST-reads (based on on-sided RDMA reads) and SST-writes (based on one-sided RDMA writes). We show that in most cases, SST-writes represents a better choice.

4. We offer insights into the achievable RDMA performance for tightly coupled groups of nodes in a datacenter network design.

We have evaluated the system on the Susitna cluster of Emulab, which is connected with 40 Gbps Infiniband, and in the Texas Stampede cluster, which has a 56-Gbps Infiniband system that includes routing over one or two levels of top-of rack switches: the former lets us explore a rack-scale computing case, while the latter is more similar to what might be seen in a datacenter routing infrastructure. We show that SST adds minimal overhead over raw RDMA primitives, scales linearly with the number of nodes, and is tolerant of RDMA routing delays. We discuss and evaluate different types of predicates varying in complexity and structure and demonstrate a use case of OSPF routing.

The paper is organized as follows: In section 2 we provide some background on RDMA. In section 3 we discuss our high-level system model, and in section 4 we describe the design of SST in detail. Section 5 describes our experimental evaluations and results. Finally, section 6 discusses related work on state management systems and RDMA based systems, and we conclude with a discussion of future work.

2. BACKGROUND ON RDMA

For reasons of brevity, we assume that the reader is familiar with RDMA. We only provide a short summary here with a focus on setting some of the notations right and stating the technologies we use.

SST runs on the Verbs API which exposes RDMA functionality available on Infiniband hardware, but should also be compatible with RoCE (Ethernet). RDMA is a connection-based protocol, hence there is a connection establishment step, which occurs before our experiments run. The protocol supports two modes of operations : a) synchronous, consisting of **send** and **receive**, which require the cooperation of both the sender and the receiver and b) asynchronous, also known as one-sided operations, consisting of **read** and **write** in which a process P, respectively, reads from or writes to memory region of process Q, with no action by Q. We use the Mellanox hardware for development and testing, however, the code should run on any hardware compliant with the relevant standards.

We use the RDMA one-sided operations, which require *posting* the read/write request to the NIC and busy *polling* for completion notification of the operation at the initiator node.

3. SYSTEM MODEL

3.1 State Table

A row is a user defined struct containing some number of entries. By an entry, we refer to a cell of the table i.e. a state variable of some node. Each entry is a boolean, integer, floating point value, or any other "plain old data" type that will fit within a single cache line. Every node has a single local row. Taken together, the rows from all nodes in the group form a State Table. All rows in the table are readable by any host, but only the local row may be modified. SST internally makes sure that modifications to a local row quickly become visible in all copies of the state table.

3.2 Predicate Model

Predicates are boolean-valued pure functions that operate on a state table. Users may register any number of predicates to detect specific events, along with lists of triggers to be run when each predicate first evaluates to true. Triggers are void-valued functions that can change the local row as well as add additional predicates and triggers. All operations involving predicates occur only locally on the host where they are registered.

It is very convenient to program in this model. As an example, a k-step synchronous algorithm can be implemented by registering predicates for each step that check whether the entries in a step_num column indicate that all nodes have reached that step. The associated trigger could actually perform the work for the step and then increment the local step_num entry.

4. DESIGN

4.1 State table

The row format is provided by a template parameter at initialization of SST and must be a struct in C++. An example is given in Listing 1 in which every row contains a step number, an array of timestamps and percentage CPU load. As a consequence of the fact that a row represents the local state of its owner node, only the owner of a row has write access to it. A node can read the local states of other rows and therefore, has read access to all the rows. SST maintains an in-memory copy of the table at every node.

For RDMA operations to succeed, it is very important that the memory layout of the rows be predictable. We require that the struct specifying the format of the row must be a POD (plain old datatype) and should not contain pointers to the actual state members. The memory layout of member variables of non-POD structures can be in non-contiguous regions and similarly, the actual data pointed to by a member pointer can be somewhere else in memory. An example of a non-POD struct is given in Listing 2. All its members are either non-POD types or pointers. For instance, a vector's size is not fixed at compile time, so the vector object stores a pointer to the actual member array. There are two major issues associated with such structures. First, it is not clear how to determine the actual addresses of the state variables from a general struct template provided by the user. Second, since the state variables are stored in non-contiguous addresses, we have to pin all the different memory regions to the NIC and exchange all addresses at SST creation time, which complicates the initialization process. Worse, this means that reading a row of a remote node or writing the local row to a remote node requires multiple RDMA operations, one for each contiguous memory region of the struct, leading to a severe degradation in performance. In this context, it is important to recall that RDMA offers very basic primitives; in particular, there is no pointer indirection in remote memory. By avoiding these complex scenarios and allowing only POD structs without pointers, we are guaranteed that all the state variables are stored in one contiguous memory region. This makes it simple to pin memory and exchange addresses during initialization, and reading/writing a remote row takes only one RDMA operation.

```
struct Row {
  int step_num;
  struct timespec timestamps[2];
  double cpu_load;
};
```

Listing 1: An example of an SST row structure

```
struct Row {
  vector <struct timespec> timestamps;
  list <int> costs;
 map <int, string> ip_addrs;
  double *cpu_load;
  int *step_num;
};
```

Listing 2: non-POD structs are not allowed

We fix the structure of the table (number of rows and their owner nodes as well as the row format) at SST creation because any changes to row structure or membership would require reallocating the table at each node, leading to repinning of memory with the NIC. Since this is such a costly operation, it would effectively take no less time than destroying the SST object and creating a new one, so we explicitly require the SST object to be re-created in order to change the structure of the table.

4.2 Operations on the table

Our goal is to support lock-free, fully asynchronous state reads and updates, i.e. a node should be able to update its local row and read other rows without being delayed by other nodes, or depending upon any form of cooperation by other nodes. Subject to this constraint, SST should guarantee atomicity at the level of updates and reads to objects that reside fully within a single cache line and are written or read by some single instruction, analogous to lock-free updates and reads to volatile variables in C++. We are left with two design choices, one based on RDMA one-sided reads and the other based on RDMA one-sided writes. We describe both of these options.

4.2.1 SST-reads

In SST-reads, every pair of nodes exchange the address of their local row with each other at initialization. A local state update is implemented by a DMA write to the local row, whereas reading other rows is achieved by posting RDMA reads to their owners and storing the result in the local table.

Our first design decision is to determine how a node should use RDMA read operations to keep its local in-memory table updated. We first measure the latency of RDMA reads for two nodes. Figure 1(a) plots the time it takes one of the nodes from posting the operation to successful completion for varying data sizes. We break up the time in two parts : a) time to post the request to the NIC and b) remaining time until the end of successful polling. The time to post the read is constant (since it only involves notifying the NIC of the request), whereas time to poll completion increasing linearly with data size as the data transfer time increases. As is clear from the graph, the majority of the time is spent busy polling for completion of the RDMA read.

To better utilize this idle time, we post all the requests to all the rows first and only then wait for them to complete. The pseudo-code is given in Listing 3. This allows for the operations to be carried out in parallel, and reduces the overall latency by a significant amount as the time for polling completion is amortized over all the operations. We verify this experimentally by varying the number of rows and measuring the average time taken for the parallel reads to finish. The graph is plotted for different data sizes in Figure 3(a). From 1 to 7 parallel operations the total time increases by a factor of less than 2 for all data sizes, which is much less than 7. A schematic of the RDMA operations performed by a node in SST-reads is given in Figure 2(a).

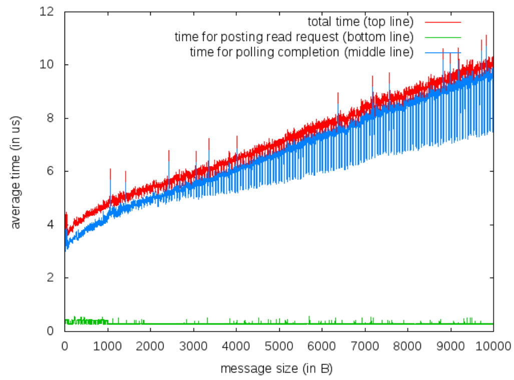
4.2.2 SST-writes

In SST-writes, every pair of nodes exchange the address of each other's row in their table at initialization. A local state update involves posting RDMA writes to all the other nodes while reading other rows then, is simply reading from the local memory.

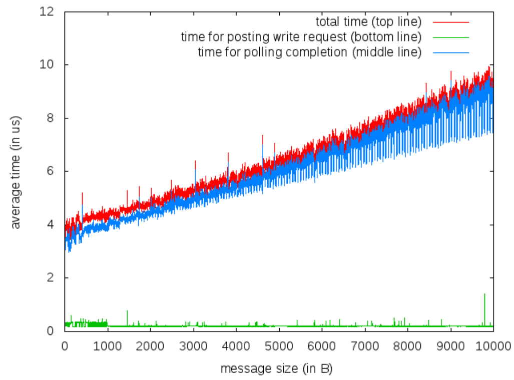
```
for every other node k:
  post read of local row of node k

for num_rows−1 times:
  poll completion of one entry
```

Listing 3: Table refresh in SST-reads

RDMA write operations have similar behavior to RDMA reads as shown in Figure 1(b). Therefore, by posting write requests in parallel, we can improve performance compared to posting each write sequentially (Figure 3(b)). The pseudo-code for the parallel writes is given in 4. Figure 2(b) shows the pattern of RDMA operations performed by a node in SST writes.



(a) Reads



(b) Writes

Figure 1: Latency of RDMA read and write operations as message size increases

4.3 Predicates and Triggers

Predicates are C++ functions that take the SST object as a parameter and return a bool and are local to every node. Each predicate has a list of triggers associated with it that are executed in order, when the evaluation of the predicate returns true. Triggers are C++ functions that take the SST object and return

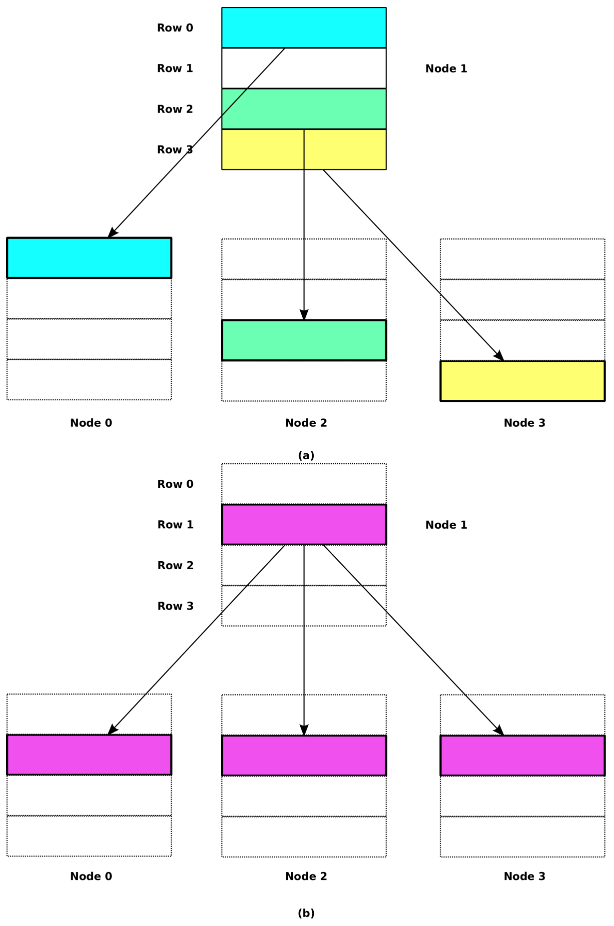


Figure 2: Illustration of SST-reads vs SST-writes. Node i owns row i of the table for i = 0,1,2,3. In (a), Node 1 reads the rows 0,2,3 from their respective owners. In (b), Node 1 writes its local row at all the nodes.

void.

We consider three types of predicates : a) Predicates that always remain in the system and fire associated triggers each time they are evaluated to be true, b) Predicates that remain in the system until the first time they are true, and therefore, fire only once and c) Predicates that fire only when their evaluation changes from false to true. This provides additional choice to the user who can use an appropriate type based on the use case.

For SST reads, if for every evaluation of predicates, we needed to read the remote rows, detection would be very slow. Accordingly, SST instead performs predicate detection and table refreshes (RDMA reads) in two separate threads. The predicate detection thread continuously evaluates the predicate on the local copy of table while it is simultaneously refreshed by the reader thread. Our belief is that SST use cases will require the highest possible performance, and that given this goal,

```
for every other node k:
  post write of local row to node k

for num_rows-1 times:
  poll completion of one entry
```

Listing 4: Local row update in SST-writes

dedicating a thread (indeed, a core) to play the role of refreshing the table is an acceptable level of overhead. However, the frequency of refreshes could be ramped down, if necessary.

In the writes version, in contrast, the local node controls when to push the local state changes to other nodes, and no separate thread is needed. In most systems, the number of state updates are far less than the number of reads, and therefore, the writes version in most cases would involve fewer RDMA operations. Threads and their functions for both SST reads and writes is listed in Table 4.

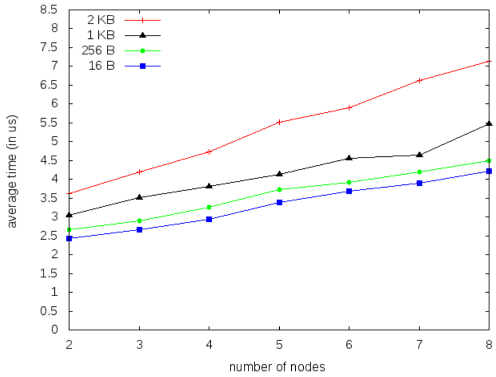
The predicates and triggers are provided the original SST object so that they can change the local row and add additional predicates. This means that the table entries can change while the predicate or trigger is being run which might not be desired by all. For example,. Therefore, SST provides a function get_snapshot that returns a constant copy of the table for evaluation. It is worth noting that since, the triggers associated to a single predicate are executed in order, the final state of the SST depends on this order of execution as well as the order in which the predicates are detected to be true.

4.4 Race Conditions

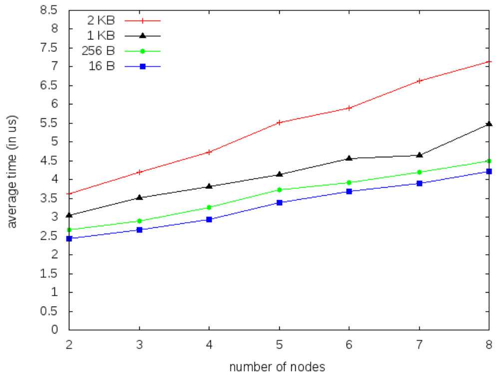
Read-write races arise in our design. In addition to race conditions caused by local threads, local reads/writes to the table can conflict with remote operations. There are two such cases, one in each version of the SST:

1. In SST Reads, a write to the local row can conflict with the RDMA read operation of the row by a remote node
2. In SST Writes, an update of a local row at a remote node can conflict with the remote node's local read of it

In the context in which we expect SST to be used, introducing locks or other concurrency controls into the SST itself would create an undesirable latency overhead for the common, non-conflicting case. Instead, we rely on a feature of the Intel processor architecture: DMA reads and writes of single cache lines are atomic [8]. Thus, if each state variable is of size less than the cache line size (64 B typically), RDMA reads and writes of



(a) Reads



(b) Writes

Figure 3: Latency of parallel RDMA reads and write as the number of nodes increases

such variables are atomic. This provides a basic guarantee of read/write atomicity for each state variable separately. Some of the prior work we cited, notably the FaRM key-value store, makes this same atomicity assumption but then goes beyond it, implementing a transactional layer within FaRM by adding a timestamp to every cache-line. We considered this, but concluded that transactions are best viewed as an end-user abstraction. Accordingly, SST limits itself to the most basic atomicity property. Any higher form of atomic transaction, such as reading a consistent snapshot of a row, should be implemented on top of SST using explicit concurrency control.

5. EVALUATION

We report some microbenchmarks, discuss the performance of SST for different types of predicates and table rows, show how SST scales to larger numbers of nodes, then describe and evaluate a sample application of SST.

5.1 Experimental Setup

All the experiments in this section were conducted

Thread	Work performed
Reader	Refreshes local table in SST-reads
Detector	evaluates predicates and fires triggers
Main	establishes connections, creates SST object , updates local row

Figure 4: Work performed by various threads in SST

on the the Emulab Susitna cluster, which has 35 nodes using 40 Gb/s Mellanox NICs. Each node has four 16-core Opteron 6272 Processors and 128 GB of DDR3 memory. The cluster is running Ubuntu 14.04.1 LTS.

5.2 Performance Metrics

5.2.1 Latency

SST is aimed at high-performance, latency-sensitive applications. The single most important property that affects application latency is the time SST takes to detect a predicate becoming true. A higher rate of predicate detection allows triggers to be fired more promptly and hence allows the application to respond quickly to state changes.

We define the time to detect a predicate as follows. When a predicate changes from false to true, this must correspond to an update of some local state variables by a node. Since writes of a state variable to memory are atomic, we can order the writes and identify the last write to some state variable by a node that caused the predicate to be true. So, we are interested in measuring the time from this last write to the time it is detected at the node where the predicate is registered. In SST-reads, this start time is when the state variable was updated in local memory, whereas in SST-writes, it is when the node initiated the remote write to all the nodes.

Since, in practice, if the predicate turns true by near simultaneous writes by multiple nodes, the start of time is difficult to measure. Instead, in our experiments, there is a clear single node that updates its row that turns the predicate true. Also, any sensible predicate for our interest should be such that this update and the detection happen at different nodes and therefore, the time elapsed is difficult to measure. The latencies are of the order of microseconds and it is not possible to synchronize time to nanoseconds precision. Therefore, the technique we employ is sort of a round trip of detection. Node A updates an entry of its row that fires predicate P at node B which in its trigger, sets some entry of its own row that fires the predicate Q at node A. Node A measures the whole round trip time and then, if the time to detect for P is already known, using the linearity of expectation (as we will see the time is probabilistic), we get an estimate (upper-bound) for time to detect Q.

5.2.2 Throughput

Another measure of performance is how many times can we evaluate a predicate and fire its associated trigger. To measure this, we make the conditions right for a predicate to evaluate (periodically or permananently) to true and then, count how many times the associated trigger is fired.

When the predicate is true for an extended period of time, throughput mostly depends on processor speed and not on any network characteristics. This is because predicate evaluation is performed in a separate thread. However, if the predicate is periodically true, then latency improves throughput as it reduces the period of consecutive detections.

5.3 Parameters

We identify different parameters that affect the core performance of SST.

5.3.1 Predicate complexity

Increase in the number of rows a predicate depends on, increases the time of detection for two reasons :

a) it increases the evaluation time of the predicate and

b) time to update the local copy of the table to reflect simultaneous changes to the rows goes up. Specially, in SST-reads, if half of the updated values are not detected in one table refresh, then they are not visible until the end of the next refresh cycle. This effect is somewhat less pronounced in SST-writes, because all the values are updated as soon as the write of the last update in local memory completes.

It is worth noting that the actual nature of the predicate is not a major factor (whether it finds the sum or the maximum of a column etc. is irrelevant), since the time is dependent on how fast can we refresh the table which is entirely determined by the RDMA operation (read or write), table size and the number of nodes.

5.3.2 Table size

Increase in the size of a row increases the time to refresh a table in SST-reads because data transfer time increases. In SST-writes, state variables can be selectively updated, so this is not much of a factor.

Increase in the number of rows increases the time to complete a table refresh in SST-reads and increases the time to update local row in SST writes.

5.3.3 Number of Nodes

Increasing the number of nodes increases the number of rows, but additionally, it means that the rate of RDMA operations will be higher. However, the small row size of typical uses and highly-capable modern NICs ensure that this is not a major factor.

5.4 Micro-benchmarks

5.4.1 Latency of Simplest Predicate

The simplest predicate is the one that depends on a single remote row in a 2 node system. The SST consists of the two rows and a single column **a** of type int, initially 0 for both rows. Node 0 at a random moment in time, sets its own entry to 1. We are interested in measuring the time to detect this change at node 1. To measure time, we do a round trip of detection. On detecting predicate at node 1, node 1 sets its own entry to 1 in the attached trigger, which is in turn detected by node 0. Node 0 measures the time taken from setting $SST[0].a = 1$ to detecting $SST[1].a = 1$ and divides it in half to get an upper bound for detecting the predicate. The predicates and the triggers are illustrated in Table 5 and the sequence of operations of the two nodes is given in Table 6. It is important to appreciate that the time taken depends on the actual order of local and remote operations on the memory and is thus probabilistic. This has been illustrated in Figure 7 (a) and (b). In Figure 7 (a), S1, E1 and S2, E2 denote the two RDMA reads by the remote process. R1 and R2 are their respective reads of the local memory. If the local write shown as W1 takes place just before R1, the remote process detects it at E1. If the local write instead takes place just after R1 (W2), the remote process misses it at E1 and detects it at E2. Time to detect in case 1 is length of W1E1, while it is length of W2E2 in case 2. Similarly, for the writes case in Figure 7 (b), S,E denote the write by the remote process with W denoting the time of DMA write to local memory and R1, R2 denote the points of local probe of memory. The time to detect is the length of SR2 which depends on the position of W between the interval [R1, R2]. Notice, that there is a definite order among the operations on memory becaues of the atomicity of the reads and writes of a state variable as observed in Section 4.

Due to the variation in the time of detection as well as in the RDMA operations themselves, we run a lot of tests and calculate the mean and standard deviation of the time. Notice that the standard deviation values will be higher for the reads version as compared to the writes version because local memory access takes less than one-fifth the time of an RDMA operation [9] (this can be seen in Figure 7, the length of the intervals for reads and writes is different). To give a sense of the performance overhead incurred by SST, we implement this predicate detection using directly the RDMA read and write operations i.e. without the formalisms of a table and predicate detection subsystem. The results are provided in Table 8. As is clear from the data, the overhead incurred by SST is negligible.

P0 : SST[1].a = 1	P1 : SST[0].a = 1
T0 : end timer	T1 : set SST[1].a = 1

Figure 5: P0 and P1 are the predicates for nodes 0 and 1 resp. and T0 and T1 are their associated triggers

Node 0	Node 1
create SST	create SST
set SST[0].a=0	set SST[1].a=0
register (P0, T0)	register (P1, T1)
wait for random time	
start timer	
set SST[0].a = 1	

Figure 6: Sequence of operations for the two nodes

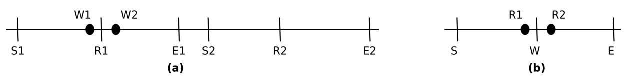


Figure 7: Time of detection is different depending on the order between DMA writes and DMA reads by local and remote memory operations. (a) and (b) illustrate this for SST-reads and writes respectively

5.4.2 *Latency of Most Complex Predicate*

We now measure the time to detect a predicate that depends on the entire column of the state table. We consider the detection of the abstract predicate

$$SST[1].a \wedge SST[2].a \wedge ... \wedge SST[n-1].a$$

by node 0 for n=25 nodes. All predicates involving aggregates over a column like average, maximum, minimum etc. fall in this category. Since, we want all the nodes except 0 to set their state variable a to 1 at the same time, we make them wait till they detect $SST[0].a$ to 1. Node 0 starts the timer when it sets its entry to 1 and ends it when it detects that all the other entries are 1. To get an estimate of the time it takes to detect this predicate, we subtract the time to detect the simple predicate found in the previous experiment with n=25. This works because of the linearity of expectation. The averages time to detect the predicate is found to be 12.04 microseconds for SST-reads and 11.6 microseconds for SST-writes, up by less than 6 times.

5.4.3 *Throughput of a Basic Predicate and Trigger*

	SST Reads	RDMA Reads	SST Writes	RDMA Writes
mean	2.22	4.60	4.50	5.81
std. dev.	2.47	3.21	2.17	1.67

Figure 8: Results for the simplest predicate detection

The predicate remains the same as the previous case, but it is registered only at node 0 and the **a** entries are already set to 1. So, as soon as the predicate is registered it starts firing. The trigger simply increments the counter. This captures the basic function call overheads of predicate detection. We find that there are 800 thousand number of evaluations and trigger executions per second which confirm that the evaluation speed depends is determined by the cpu speed.

5.5 Effects of Predicate and Row Complexity

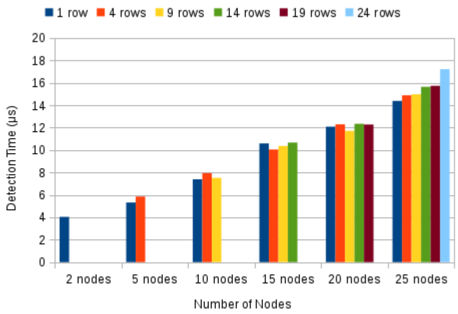
5.5.1 *Latency with predicate complexity and the number of nodes*

We vary the number of rows a node depends on, **r** and the number of nodes **n**. For a given $r < n$, the predicate at node 0 is

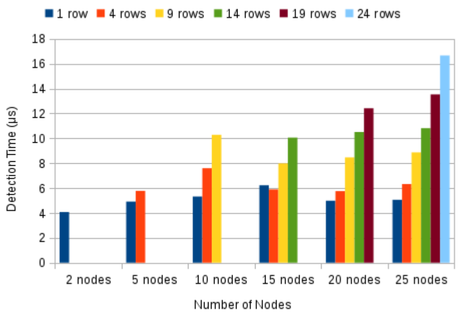
$$SST[1].a \wedge SST[2].a \wedge ... \wedge SST[r].a$$

At initialization, we ensure that $SST[i].a$ is 1 for all nodes upto $r - 1$, and then, node r sets it to 1 after a random wait. We again measure the time using a round trip of detection with r setting its entry after detecting $SST[0].a = 1$ and find the detection time by a similar technique as in Section 5.4.2. The results are shown in Figure 9(a) for SST-reads and 9(b) for SST-writes.

We see that the SST-reads and SST-writes have different characteristics overall. With SST-reads, we see that for a given number of nodes, the latency of detection is very high even with the simple predicates, but remains roughly constant with the variation in predicate complexity. With SST-writes, we see that for a given number of nodes, the latency of detection is low for simple predicates and increases gradually with the predicate complexity converging to about the same value corresponding to the SST-reads data. This is because, SST-reads continuously updates the table by posting read operations to all the nodes, so if the updated value by node r that validates the predicate is missed by node 0 in a refresh cycle, it won't become available until the next cycle. However, in SST-writes node r simply writes the data to all the nodes and therefore, when it completes on node 0, node 0 detects the predicate. Hence, SST-writes has a better performance with predicates operating on fewer rows of the table for large number of table rows, while otherwise, the performance is comparable. As a corollary, for the same predicate complexity, time of detection increases with increasing number of nodes for SST-reads, while it remains roughly constant for SST-writes.



(a) SST-reads



(b) SST-writes

Figure 9: Latency of predicate detection with the complexity of predicates for different number of nodes

5.5.2 *Latency with row size and the number of nodes*

The row in this experiment consists of a single array **a** of integers, size of which is varied from 1 (4 Bytes) to 1024 (4 KB). The predicate depends on all the rows and all the columns and is as follows :

$$\forall_{0 \leq i < n} \forall_{0 \leq j < s} SST[i].a[j]$$

where, s is the size of the array. All the entries of all the rows are 1 except SST[n-1].a[s-1] which is set to 1 at a random time. We detect at node 0.

The results are shown for SST-reads in Figure 10. We find SST-reads and SST-writes to perform equivalently in this case, since, the size of the row increases the completion time of RDMA read and write operations identically. As is clear from the graph, the time for detection increases with the number of nodes as well as the data size.

5.5.3 *Throughput variation with predicate complexity*

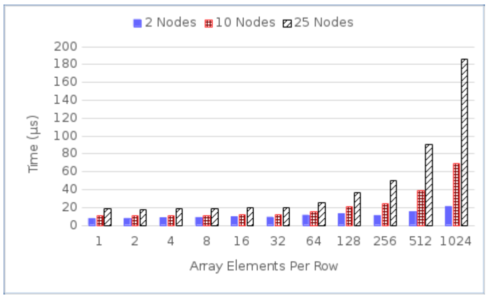


Figure 10: Latency of predicate detection with number of nodes for different number of row elements

To see how many a times can a predicate evaluate and fire the trigger continuously, we vary its complexity by varying r. The predicate remains the same as in Section 5.5.1 and the trigger increments the counter. The predicate is set to true before registering the predicate. The results are shown in Figure 11. The graph shows that the number of evaluations decreases slightly with the predicate complexity as the time to evaluate it increases. However, the throughput is pretty good even for large rows, the number of evaluations for the predicate accessing 29 rows is about 500 thousand per second.

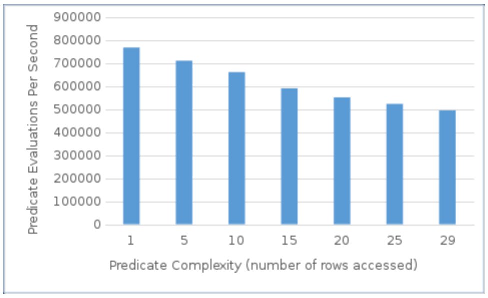


Figure 11: Variation of throughput with the predicate complexity

The number of times a predicate evaluates periodically depending on remote actions is explored in the next section along with scalability.

5.6 Scalability

For scalability, we describe the experiment distributed counting, where all nodes start counting from 0 in synchrony i.e. all nodes increment their counter, wait for everyone to increment it, then increment again and so on. The predicates this is

$$\forall_i SST[i].count \geq SST[k].count$$

for a node k registered at all the nodes. The trigger simply increments the counter. All nodes measure the time it takes to count up to a million and we take the average (the individual times are very close anyway since, the counting is synchronous). Figure 12 shows the plot of time with the number of nodes for read and write. Throughput in this experiment for SST-reads varies from about 400 thousand evaluations per second for 3 nodes to 40 thousand/s for 28 nodes. For SST-writes, it varies from 325 thousand/s for 3 nodes to 48 thousand/second for 28 nodes. From the graph, it is clear that SST scales well, the time varies linearly with the number of nodes. It is interesting to note that writes version scales better than the reads version. This is because of a similar reason as in 5.5.1. If a node in SST-reads does not see the updated counter values in an iteration, it will have to wait until the next cycle of table refresh, but a node in SST-writes will see the updated counter values of another node as soon as that node writes it into its memory. Thus, SST-writes scales better and performs better than SST-reads, even if the updates are fairly high.

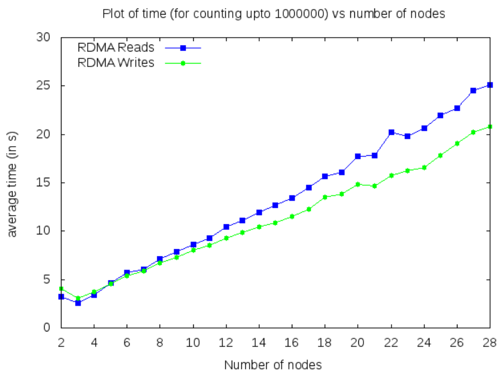


Figure 12: Time to count up to 1 million for different numbers of nodes

5.7 Routing Application

To demonstrate how SST could be used to support network applications that share state, we implemented a basic OSPF-like routing application using SST Writes. As in the OSPF protocol [11], each "router" node in the network forwards messages to other routers based on a forwarding table, which it constructs based on a link-state table that describes how each router is connected to other routers and networks and the cost associated with each link. However, instead of using gossip messages to exchange link state information, the link state table is stored in an SST instance. The SST has a column for each router (or other network endpoint) in the network, and the values in each column indicate the cost of the link to that router. Thus, `sst[src].link_state[dest]` is the cost (i.e. OSPF metric) of the link from `src` to `dest`, or a special value such as -1 if there is no link from `src` to `dest`. When a router gains or loses a link, or changes the cost of an existing link (e.g. due to a congestion control policy), it simply updates a value in its local row, and the SST ensures that this change in the link state table is propagated to all the other routers.

Instead of recomputing their routing table every time the link state changes, as in standard OSPF routing, our routers use a more sophisticated predicate to detect changes that could potentially cause a shortest path to change. After computing all shortest paths and creating its routing table, a router registers a predicate that evaluates to true if any of the links used in a shortest path increase their cost, or if any of the links left out of a shortest path decrease their cost. The trigger for this predicate is to recompute the routing table and construct a new predicate based on the new shortest paths. This demonstrates how SST's predicate system can be used to detect system events with more advanced logic.

To measure the performance of SST in this application, we measured the time it took for all the routers in a system to install new routing tables after a link state change. In order to ensure that a single link change would cause all routers to recompute their routing tables, the nodes in the test system were configured to use a virtual network topology in which one node was a central "hub" connecting to all other nodes with low-cost links, and all the other nodes were connected to each other in a ring of higher-cost links. Figure 13 shows the total time for all nodes to finish updating their routing tables, as well as the time it took for the first remote node (other than the node making the link change) to react. The gap between these times represents the variation in network latency between RDMA writes propagating the updated SST row to each remote node. Figure 13 also shows the amount of time a single node spent recomputing all shortest paths from the link state table, for each network size, in order to show the portion of the response time that did not depend on SST latency.

5.8 SST in other settings

5.8.1 Nodes on different racks

SST is built with one important assumption, that of rack scale of nodes. More importantly, this assumes that all the nodes are connected to the same switch, so each pair of nodes has full bisection bandwidth and identical RDMA latencies. This is reflected in the design of refresh table in SST-reads and table update in SST-writes. We post all of the operations and then wait for them to complete. If the connection of a node A to some other node B is comparatively slow (because they are in different racks, for instance), the time it takes for a refresh or update to complete at A will increase, in which case it might make more sense to post read-

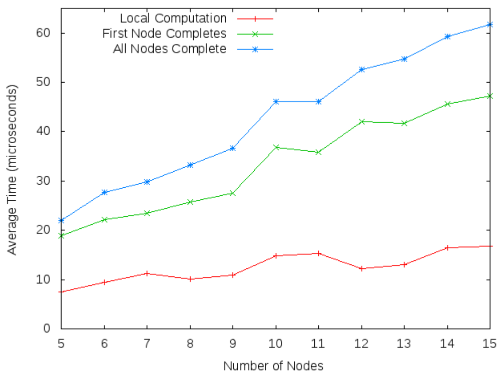


Figure 13: Time for nodes to install new routing tables after a link change

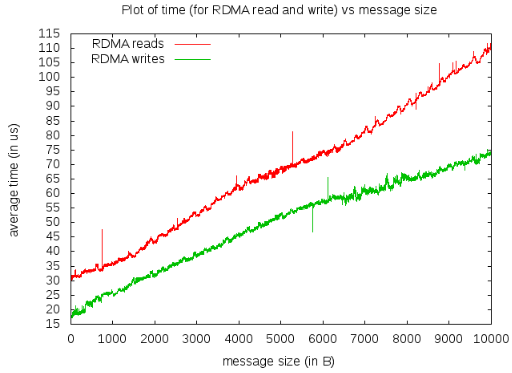


Figure 15: RDMA latencies on Marmot

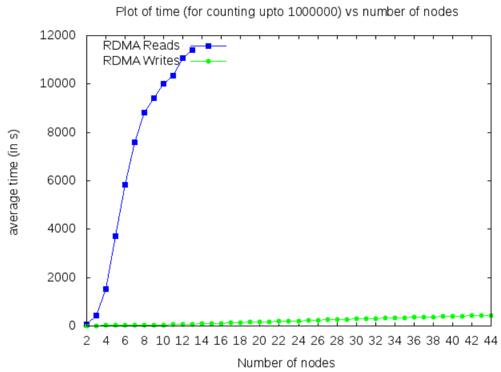


Figure 16: Distributed counting experiment on Marmot

s/writes to node B less frequently than the other nodes and possibly in a separate thread.

We explored this issue in a real setting on the Texas Stampede cluster which has faster infiniband than Susitna, 56 Gbps, but runs a scheduler that allots the nodes, so we have no control over whether they are on the same rack. Nodes on different racks on Stampede are connected via 3 different switches, the respective top of rack switches and the aggregator switch, which means that the RDMA operations latency between them is twice as high as compared to the latency for nodes on the same rack. Additionally, a rack has 2 IB switches connected to 20 nodes each and each leaf switch is oversubscribed to the core switch, so that the nodes on the same rack, but connected to different switches get only 80% of the bandwidth. We ran the distributed counting experiment on 8 and 16 nodes on a total of 3 racks. For comparison, we ran the same experiment on 2 nodes on the same rack and switch. The Table 14 shows the data contrasting it with that for Susitna. As we can see, there is a significant improvement on Stampede for 2 nodes because of the higher bandwidth, but the numbers for the 8 and 16 nodes are comparable because of higher latency between nodes on different racks or same rack but different switches. We also observe that the throughput is still reasonably high, even for this adverse case.

Number of nodes	Stampede	Susitna
2	0.91	3.23
8	8.66	7.09
16	13.16	13.42

Figure 14: Sequence of operations for the two nodes

5.8.2 *Old RDMA hardware*

We ran some of our experiments on Marmot, part of Emulab which is only 1 Gbps Infiniband. There are two major changes to note. Even though, the RDMA operations' latency varies linearly, we find that the latency of writes is much better than the reads. Figure 15 compares them.

However, because of this, SST-reads scales very badly. The distributed counting experiment's result for Marmot are shown in Figure 16. However, SST-writes still scales linearly.

5.9 Advantages of SST-writes

The design of SST-writes ensures that we have fewer RDMA operations in most use cases. The SST-writes allows a node to update a single state variable at remote nodes, leading to their faster detection. This is observed in the predicate complexity and scalability experiments. However, for a lot of other experiments, SST-reads has comparable performance with marginally better latencies.

6. RELATED WORK

The shared table model is a familiar paradigm: use cases include routing tables, load balancing, autonomic or other forms of self-adaptive behaviors, "affinity" based task placement, intrusion detection, etc. We see all of these as areas that might benefit from SST because the exceptionally high performance and low latencies achieved by the solution are orders of magnitude better than what was achievable a decade or more ago, and hence the option of much more aggressive dynamism arises.

Work on distributed expression evaluation predates modern computing systems, including the META system [14], which evaluated expressions over sensors, modelling each sensor as a publish-subscribe group in which the sensor pushed updates to the interested receivers. This style of computation is common with pub/sub systems, such as the Isis Toolkit [2], TIB/RendezVous [], Gryphon [12], and Siena [3]. In addition, there is a large body of research on event-structured database systems, including systems for running transactions both on streaming data and on stored data.

Important work on systems that focus on large scale systems include Astrolabe [13], BigTable [4] and Google's Spanner [5]. The Astrolabe system, designed for scalable data mining in very large data centers used a hierarchy of tables, and the BigTable platform used within Google brings table structured data to the file system. But none of these focus on low-latency scenarios.

Recently, there is a lot of work on systems using RDMA for scalable DHTs, such as Farm [6], Pilaf [10] and HERD [9]. These systems focus on the server-client model of computation, completely different than what we target with SST.

Thus, we believe that SST is unique in this space: focusing on a pure sharing model, with the simplest possible lock-free sharing approach, and an emphasis on raw speed and low latency in smaller datacenter deployments: rack-scale computing, control programs for switches and routers, etc. Here, there is little relevant prior work. Indeed, the closest fit would probably be a technology like the MPI library [7], which supports barriers and shared memory backed by Infiniband. However, MPI is oriented towards HPC, and adopts a gang-scheduling model in which all the programs are replicas, with one designated as the leader and the others as workers, and is not able to adapt to add or drop new members. None of these limitations apply to SST.

7. CONCLUSIONS AND FUTURE WORK

Our paper introduces a simple shared state table implemented over RDMA, and demonstrates that the SST abstraction is a fast and convenient way to share system state on RDMA networks. We implement and compare two versions of SST, one based on one-sided RDMA reads, and the other on one-sided writes, and find that the writes version scales better and involves fewer RDMA operations. Evaluation shows that the solution works well both with full-bisection bandwidth in a single rack, and with routing.

In the future, we would implement a replication system which uses SST for state sharing. There are various optimizations possible for doing predicate evaluation within SST which we hope to pursue, including common subexpression detection for faster evaluation of predicates, priority-based evaluation of predicates and modifying the pattern of RDMA operations to update the most used state variables faster.

8. REFERENCES

[1] Ken Birman. Vsync Cloud Computing Library, 2015.

[2] Kenneth P. Birman. Replication and Fault-tolerance in the ISIS System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 79–86, New York, NY, USA, 1985. ACM.

[3] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, August 2001.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[5] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[6] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[7] Message Passing Forum. Mpi: A message-passing interface standard. Technical report, Knoxville,

TN, USA, 1994.

[8] Intel. Intel 64 and ia-32 architectures software developer's manual. 2015.

[9] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[10] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, 2013. USENIX.

[11] J. Moy. OSPF Version 2. Technical Report RFC2328, RFC Editor, April 1998.

[12] Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An information flow based approach to message brokering. In *IN PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING*, 1998.

[13] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, May 2003.

[14] Mark Wood and Keith Marzullo. The design and implementation of meta. In *Reliable Distributed Computing with the Isis Toolkit*, pages 309–327. IEEE Computer Society Press, 1994.