

Monitoring Remotely Executing Shared Memory Programs in Software DSMs *

Long Fei, Xing Fang, Y. Charlie Hu, and Samuel P. Midkiff
Purdue University, West Lafayette, IN 47907
{lfei, xfang, ychu, smidkiff}@purdue.edu

Abstract

Peer-to-Peer (P2P) cycle sharing over the Internet has become increasingly popular as a way to share idle cycles. A fundamental problem faced by P2P cycle sharing systems is how to incrementally monitor and verify, with low overhead, the execution of jobs submitted to a remote untrusted hosting machine, or cluster of machines. In this paper, we present the design and implementation of GripCop DSM, a novel incremental execution monitoring and verification scheme for software distributed shared memory (SDSM) programs running on remote clusters. Our scheme maximally leverages the shared memory abstraction provided by the SDSM system by extending the shared memory abstraction to the monitoring process by replicating one of the processes running on the host cluster to verify intermediate results at runtime. Our GripCop DSM employs two monitoring schemes: (i) a full-scale monitoring scheme that completely replicates the computation of a process running on the cluster, and (ii) a decoy monitoring scheme that deceives the host cluster into believing that full-scale monitoring is being performed without it ever actually being done, thereby incurring negligible overhead. Experiments show that the combined use of full-scale and decoy monitoring ensures faithful execution with low performance impact, even over a wide area network.

1 Introduction

Cycle sharing over the Internet has recently made large gains in popularity as a way to share idle cycles. Large numbers of Internet connected computers can join a peer-to-peer network and donate idle CPU cycles to fulfill the computing needs of peer nodes. Nodes

can receive credit, or be charged, according to their contributions to, or consumption of, resources [14, 5]. With individual hosts becoming more powerful, more idle cycles are available, making P2P cycle sharing an ideal mechanism to utilize the idle CPU resources.

Significant technical challenges must be overcome to unleash the potential of the massive computational resources that are going unused:

1. How are resources discovered, and how are providers of these resources compensated (and cheaters punished)?
2. How is the *host* machine, i.e. the machine executing the job, protected from hostile binaries?
3. How does the *submitter* machine, i.e. the machine submitting a job, know its job is being faithfully executed as the job executes?

The first two of these items are the subject of research by the authors of this paper and others [3, 4, 5, 7, 8, 10, 17]. These systems support one or more of sandboxing applications for host safety, resource discovery, and compensation.

In this paper, we focus on the third problem: how can the submitter track the progress of a job, and receive assurances that the job is executing correctly? In particular, we focus on how to remotely monitor the progress of parallel applications targetting a software distributed shared memory system (SDSM) running on clusters of PCs or workstations.

An SDSM system provides the programmer with the ease of programming a shared memory multiprocessor system and the cost efficiency of utilizing existing networked workstations. Moreover, by running an SDSM on an Internet connected remote cluster, the SDSM system can be made available for demanding computational tasks via a peer-to-peer Internet cycle sharing system. In our context, we assume clusters accessed via P2P networks are tightly coupled systems.

In this paper, we describe the design and implementation of the GripCop DSM, a novel runtime execution monitoring and verification system for a remote SDSM cluster, built on TreadMarks [1]. In the GripCop DSM,

*The research reported here was supported, in part, by National Science Foundation Grants 0325603-CCR and 0313033-CCR.

the submitting machine runs a replica of one process of the application running on the remote cluster, and performs online monitoring and verification. In the rest of this paper, we refer to this monitoring scheme as *full-scale monitoring*.

The objective of the GripCop DSM is to monitor the progress of a remotely executing job by partially duplicating the computation being done on the host machine. The novelty of the GripCop DSM is in how it leverages the shared memory abstraction provided by the underlying SDSM system by extending the abstraction to include the remote monitoring process. In doing so it greatly simplifies the tasks to be performed by the monitored process and the monitoring process in order to support runtime execution monitoring.

Full-scale monitoring, however, can incur a high overhead on the execution of the SDSM program running on the host cluster. The high overhead comes from the high cost of synchronization between the monitoring process and the host processes, and the higher round trip delay for the monitoring process to retrieve application data, due to the much higher wide area network latencies compared to local area network latencies. To reduce the overall performance impact on the program being monitored and the load on the monitoring machine, GripCop DSM only performs full-scale monitoring on a small percentage of submitted jobs. For the remaining jobs, GripCop DSM uses an extremely light-weight *decoy monitoring* scheme on the submitter side. Decoy monitoring does not replicate the computation of the target process, instead, it makes the host cluster believe it is the subject of full-scale monitoring. Whether to perform full-scale monitoring or decoy monitoring can be decided at compile time (a flag in the compiler commandline) or at runtime (by a coin-toss according to a user-specified probability). The submitted application running on the host machines notices no differences between these two monitoring modes, requiring the host cluster to always assume it is being monitored.

Our system also provides support for intermediate result verification and incremental payments. This allows the risk of both parties to be limited by verifying the intermediate results during the execution time, and making partial payments as the job progresses.

This paper makes the following contributions:

- A novel method of monitoring the progress and correctness of software DSM applications running on a remote host cluster;
- A dual-mode monitoring scheme that enables tunable tradeoffs between monitoring overhead and confidence;
- The design and implementation of an extension to the TreadMarks library that supports full-scale monitor-

ing and decoy monitoring. The extended library allows remote monitoring with little change to the existing application source code.

- Experimental results over a wide area network showing the practicality of this novel technique.

2 Shared Memory Programming in TreadMarks

TreadMarks is a state-of-the-art SDSM [13] which provides the programmer with a virtual global shared address space on distributed memory machines such as a cluster of workstations. Its design focuses on reducing the amount of communication necessary to maintain memory consistency. The major implementation features of TreadMarks are lazy release consistency [11] and the multiple-writer protocol [6]. Release consistency (RC) [9] is a relaxed memory consistency model that permits a processor to delay making its local changes to shared data visible to other processors until certain synchronization accesses occur. This delay allows RC to be implemented more efficiently than conventional release consistency [12], which requires modifications to shared memory be made visible to other processors immediately. The lazy release consistency implementation of RC in TreadMarks further reduces the number of messages and the amount of data compared to earlier, eager implementations [11] by delaying the propagation of modifications until necessary. To reduce the communication due to false sharing, TreadMarks implements a multiple-writer protocol, introduced in [6], which allows multiple processors to simultaneously modify (different parts of) their local copy of a shared page. The modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the cost of false sharing.

TreadMarks provides the following synchronization APIs:

- `Tmk_barrier()`: block the calling process until every other process arrives at the barrier;
- `Tmk_lock_acquire()`: block the calling process until it acquires the specified lock;
- `Tmk_lock_release()`: release the specified lock.

These synchronization APIs serve dual purposes. First, they are synchronization points in parallel programs. Second, in the lazy release consistency model and multiple-writer protocol, the exchange of consistency meta data (vector timestamps and write notices) is delayed until these synchronization points.

3 Threat Model

The goal of verification for our system is to ensure that cheaters cannot always cheat undetected. Three assumptions are made in our system. First, we assume the existence of a credit system (such as described in [5]) that awards credits to host machines that faithfully execute remotely submitted jobs and imposes a penalty on hosts that perform fraudulently. How to calculate credits accurately is beyond the scope of this paper. Second, we assume the owners of the host machines do not go so far as to sniff and decode network traffic. Third, and just as important, we assume participants of a cycle-sharing community are *rational*. A host machine will cheat only if the expected profit is greater than the expected cost. That is, let p be the probability of cheating and being caught, R be the profit of cheating (e.g. gaining credits by false execution), Q be the penalty if caught (e.g. being kicked out of the community), then a host machine will cheat only if:

$$(1 - p) \times R > p \times Q \tag{1}$$

Inequality (1) implies that if $p > 0$, the credit system can deter rational cheaters by having a large (Q/R) value. In other words, if the host machine is not sure that it can cheat without being caught, it will not cheat in the face of a sufficiently large penalty.

4 GripCop DSM Overview

4.1 Architecture Overview

Our monitoring and verification scheme is based on duplicating, on a trusted machine, part of the computation performed by the TreadMarks application on the remote cluster, and then comparing the intermediate results obtained on both machines. For efficiency, the monitoring machine only duplicates the computation done on one of the cluster nodes.

In a cycle-sharing community, a submitter (*submitter node*) can submit the binary of a program written with the TreadMarks library to a remote cluster (*host cluster*). When the execution starts on the host cluster, a replica of one of the processes is created on the submitter or a monitoring machine trusted by the submitter. The monitoring machine must be binary-compatible with the host cluster. We call the process being replicated the *target process*, and the replica process the *monitoring process*. Correspondingly, the machine running the target process is the *target node*, and the machine running the monitoring process is the *monitoring node*. The submitter is typically the monitoring node. The monitoring node can also be a remote

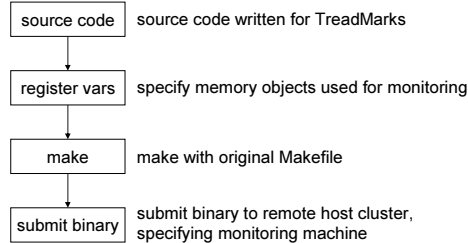


Figure 1. Steps of using GripCop DSM

trusted node or a trusted node in the host cluster (e.g. a certified machine) to achieve minimum latency.

We embed support for monitoring into the original TreadMarks library, preserving all important features, e.g., lazy release consistency and the multiple-writer protocol. In our monitoring scheme, the monitoring process synchronizes with the target process at each synchronization point (locks and barriers) to obtain up-to-date metadata, i.e., vector timestamps and write notices. However, the monitoring process obtains the actual consistency data (*diff* messages) directly from the nodes that produced the *diffs*. This design maximizes the parallelism between the monitoring process and other processes while preserving the semantics of lazy release consistency.

It requires little effort to monitor an existing TreadMarks program using GridCop DSM. Figure 1 shows the steps. The user can register a few memory objects to monitor by inserting a call to `Tmk_mon_register_object(void* start_addr, int obj_size)` for each memory object, where `start_addr` is the starting address and `obj_size` is the size of the object in bytes. These calls need to be inserted at the beginning of `main()` in the source code. The generated binary is then submitted to the host cluster for execution, and the submitter can specify a monitoring machine to use. When the program begins executing on the host cluster, a replica process is created on the specified monitoring node to remotely monitor the execution of the TreadMarks program.

4.2 Two Monitoring Modes

The monitoring processor can operate in one of two modes: a *full-scale* monitoring mode or a *decoy* monitoring mode. In the full-scale monitoring mode, the monitoring process replicates all the computation done by the target process, and synchronizes with the target process at each synchronization point (`Tmk_lock_acquire()`, `Tmk_lock_release()`, `Tmk_barrier()`). Intermediate values of the registered memory objects on the target process are compared with the corresponding objects on the monitoring process at each synchronization point. The full-scale mon-

itoring mode can significantly increase the execution time of the application for three reasons. First, when there is a high latency between the monitoring node and the host cluster over the wide area network, the round-trip latency, which incurs overhead at each synchronization point, can significantly slow down the execution. Second, additional `diff` messages between the monitoring process and the computation processes over the wide area network add to the communication overhead of the program. Finally, the additional waiting time due to different speeds at the monitoring node and cluster nodes can slow down execution if the monitoring machine is slower, since frequent synchronizations between the computation processes now need to be extended to include the monitoring process.

In the decoy monitoring mode, the monitoring process continues to send `ACK` messages to the target process as if it is responding to the synchronization messages sent by the target process. In addition, the monitoring process properly responds to other requests (e.g. connection requests, `distribute` messages) as if it is doing full-scale monitoring. To avoid the round trip latency for each synchronization incurred in the full monitoring, the monitoring process actually sends redundant dummy reply messages before each synchronization request message arrives. These reply messages are effectively buffered in the socket buffer of the target process so that its next synchronization request is “satisfied” immediately. On the monitoring side, the monitoring process can perform a reduced level of tracking of the progress of the target process execution by traversing a Finite State Automaton (FSA) in which each synchronization request message from the target process corresponds to a state. In this way, the monitoring process can check if a sequence of synchronization messages corresponds to legal transitions in the FSA [20]. Because the remote host cluster has no knowledge of the operating mode of the monitoring host, it cannot take advantage of the decoy monitoring mode without the risk of being caught cheating.

4.3 Verifying Execution Results

In the full-scale monitoring mode, the faithfulness of execution is verified by comparing the values of memory objects in the target and monitoring processes at each synchronization point. For efficiency reasons, not all shared objects are monitored. The user can specify up to K (a configurable parameter whose default is 10) objects to monitor. At each synchronization point, a 32-bit hash code is computed based on the values of the objects being monitored. The hash code is included in the first synchronization message sent from the target process to the monitoring process. If the hash code is

different from the hash code computed by the monitoring process, a fraud alert is issued.

4.4 How Fast Can We Detect Fraud?

Fraud can be detected while the host cluster cheats when the monitoring process is running in full-scale monitoring mode. Let p_{full} be the probability of using the full-scale monitoring mode. If the cheater is caught on its N th cheat, then N is a random variable with the geometric probability mass function: $P(N = n|p_{full}) = (1 - p_{full})^{n-1}p_{full}$. This random variable has expectation:

$$E\{N\} = 1/p_{full}$$

that is, on average, a cheating host cluster will be caught on its $(\frac{1}{p_{full}})$ th cheat. Suppose we need confidence δ of catching cheating on or before the host node’s n_δ cheat. Solving

$$\sum_{i=1}^n p_{full}(1 - p_{full})^{i-1} = \delta$$

we get $n_\delta = \ln(1 - \delta)/\ln(1 - p_{full})$. In particular, $n_{0.95} = \ln 0.05/\ln(1 - p_{full})$. That is, we have 0.95 probability to catch the cheater in its $n_{0.95}$ cheats. For example, if $p_{full} = 0.05$, $n_{0.95} \simeq 58$. On average, we can catch the cheater on its 20th $(1/p_{full})$ cheat.

A smarter cheater, however, will not always cheat. Suppose we have a hosting cluster that cheats only when there is sufficient profit (expected credit gain) to outweigh its fear of being caught. Without loss of generality, let us assume the credit gain (random variable C) has a normal distribution $N(\mu, \sigma)$, and the hosting cluster cheats only when $C \geq c$. The probability of cheating in this model is $p_{cheat} = Pr\{C \geq c\} = 1 - \Phi(\frac{c-\mu}{\sigma})$. Then the probability of detecting fraud in a certain execution (including cheating and non-cheating cases) is:

$$p_{detect} = p_{cheat} \times p_{full} = (1 - \Phi(\frac{c-\mu}{\sigma}))p_{full}$$

where $\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^z \exp(-\frac{x^2}{2})dx$ is the cumulative distribution function of the standard normal distribution. Similarly, we have

$$E\{\hat{N}\} = 1/p_{detect}$$

where \hat{N} means the smart cheater is caught on its \hat{N} -th service (including cheating and non-cheating services) and

$$\hat{n}_{0.95} = \ln 0.05/\ln(1 - p_{detect})$$

These formulae give the average time the cheater can proceed undetected in the community and the time to detect the cheater with 0.95 probability. For example,

0	sequence #			31
from	type	id		
real from		monitoring message type		
monitoring message sequence #				
monitored objects hash code				
message body (variable length)				

Figure 2. Monitoring message format

if the host machine cheats only when the credit gain is above average, then $p_{detect} = \frac{1}{2}p_{full}$ (here $Pr\{C \geq c\} = \frac{1}{2}$). If $p_{full} = 0.05$, we have $\hat{n}_{0.95} \simeq 118$. On average, a cheater will be caught on its 40th ($1/p_{detect}$) service.

5 Runtime System Extension

In order to minimize modifications to the user program, support for remote monitoring is implemented by extending the TreadMarks runtime system.

5.1 Extensions for Full Monitoring

The extended library for supporting full monitoring performs four new functions: (1) replicating one of the processes on the monitoring machine, (2) synchronizing the monitoring process with the target process, (3) handling `diff` requests from the monitoring machine, and (4) verifying intermediate results at runtime.

Monitoring message format: Messages between the target process and the monitoring process use an extended header (Figure 2). The first 8 bytes are the same as the original TreadMarks message header to maintain compatibility. Immediately following is the *real from* field, which is used in `diff` handling to distinguish `diff` requests from the target process and the monitoring process (and to reply to the actual requesting node). The contents of *real from* fields are the same as the contents of *from* fields in messages sent by computation nodes; *real from* is set to `Tmk_nprocs` in messages sent by the monitoring process. The *monitoring message type* field is used to specify the type of monitoring message. The type can be `MON_MSG_READY_NOTICE` (ready notice), `MON_MSG_DATA` (forwarded synchronization data), or `MON_MSG_ACK` (ACK). The monitoring message carries its own sequence number (starting from 1). The sequence number is used to eliminate duplicate messages in the case of retransmission. The last field in the header is the 32-bit hash code of monitored objects described in section 4.3.

Tmk_startup(): In the original TreadMarks, `Tmk_startup()` initiates multiple processes on the

nodes in a cluster and initializes the TreadMarks runtime system, e.g., the shared virtual memory, on all nodes. In our extended TreadMarks, `Tmk_startup()` initiates an additional process on the specified monitoring machine, which has the same `Tmk_proc_id` as the target process – the process being monitored. This allows the monitoring process to replicate the computation performed by the target process. The monitoring process performs all the initialization tasks required by TreadMarks. There are two ways to specify which monitoring mode to use in the monitoring process: according to a user specified probability i.e., a random *coin toss*, or a mode flag in the compilation commandline. The monitoring process performs synchronization with the target process at synchronization points (barriers or locks) as described below, until the target process calls `Tmk_exit()`, at which point the target process sends an exit notice to the monitoring process regardless of which monitoring mode is used, and the monitoring process exits.

Tmk_distribute(): In the original TreadMarks, `Tmk_distribute()` sends the association of a shared memory address with a variable, due to a `malloc()` in the shared memory, to other processes and waits for their ACKs. In our extended TreadMarks, the monitoring process receives `distribute` requests directly from processes on the host cluster. If the process calling `Tmk_distribute()` is the target process, it sends `distribute` requests to all other processes except the monitoring process. The monitoring process does not send out any `distribute` requests.

Tmk_barrier(): Figure 3(a) shows the synchronization between the target and monitoring processes at a `Tmk_barrier()` when the target process is the barrier manager. The target process waits until all other processes of the cluster arrive at the barrier, and then updates the monitoring process with the latest vector timestamp. The target process sends out a barrier departure notice (containing the latest vector timestamp) after it receives ACK from the monitoring process.

Figure 3(b) shows the synchronization between the target and monitoring processes at `Tmk_barrier()` when the target process is a barrier slave. The target process waits until the monitoring process arrives at the barrier. Then it sends a barrier arrival notice to the barrier manager. The target process forwards the barrier departure notice (containing the vector timestamp) to the monitoring process before it leaves the barrier.

Tmk_lock_acquire(): Figure 4(a) shows the synchronization between target and monitoring processes when the target process is not the current lock holder or the

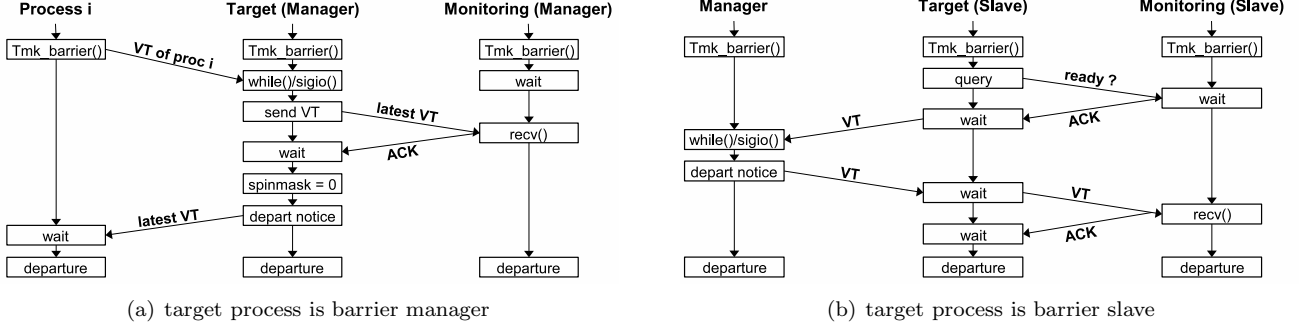


Figure 3. Synchronization at `Tmk_barrier()`

last releaser. The target process sends out a lock acquire request to the lock manager, which forwards the lock acquire request to the lock tail. When the target process receives the lock release notice, it forwards the lock release notice to the monitoring process before it actually acquires the lock.

If the target process is the current lock holder or the last releaser, it can directly acquire the lock. In order to synchronize with the monitoring process, it waits until the monitoring process arrives at `Tmk_lock_acquire()` before it proceeds. The target process blocks `lock_acquire` messages from other processes during this period.

Tmk_lock_release(): Figure 4(b) shows the synchronization between target and monitoring processes when the target process releases a lock. The target process waits for the monitoring process to reach `Tmk_lock_release()`, and then the lock is released (if there is an immediate acquirer). To avoid unnecessary interval creation when a lock release is followed immediately by a lock acquire by the same process, the original TreadMarks delays an interval creation from a lock release until the next lock acquire request arrives at the target process and is handled by `Tmk_lock_sigio_handler()`. To avoid the complexity of making `Tmk_lock_sigio_handler()` synchronize with the monitoring process, i.e., via forwarding the lock acquire request to the monitoring process and waiting for the corresponding `ACK`, in the modified TreadMarks an interval is created immediately after the lock is released, i.e., at every `Tmk_lock_release()`, and a lock acquire request is not forwarded to the monitoring process, as the monitoring process does not need to know to whom the lock is released. This eager interval creation also avoids an extra round-trip delay to the lock acquiring process.

Vector timestamp: The monitoring process must maintain a correct vector timestamp in order to perform proper page invalidation. The target process also needs to know the monitoring process' current vector

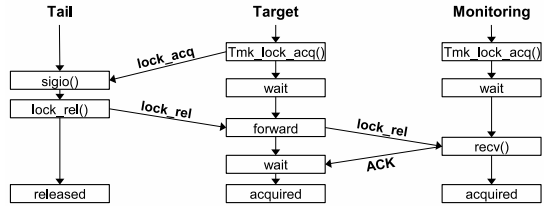
timestamp in order to generate proper write notices for the monitoring process when it is the barrier manager. In our extended process library, the target process keeps a shadow copy of the monitoring process' vector timestamp. This vector timestamp is updated properly after each synchronization point to reflect the up-to-date vector timestamp of the monitoring process. In particular, as discussed above, we use eager interval creation in `Tmk_lock_release()` to keep the consistency between the shadow vector timestamp and the monitoring process vector timestamp for lock synchronizations.

Handling diff: When the monitoring process encounters a page fault, it obtains `diffs` directly from the host cluster nodes (without going through the target process). The monitoring process will not receive `diff` requests directly from other nodes. Since the monitoring process does not receive `diff` requests from computation nodes, it generates fewer write notices than the target process. This does not affect the correctness of the computation performed by the monitoring process since pages written by other computation processes are properly invalidated at each synchronization point.

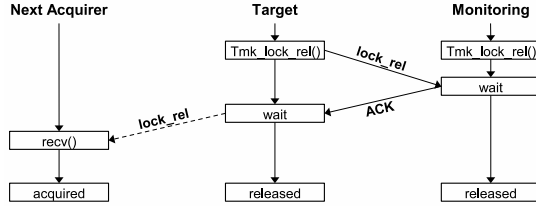
Monitoring: Our extended library supplies a new API `Tmk_mon_register_object(void* start_addr, int obj_size)` which allows the user to manually register up to `K` (a configurable parameter whose default is 10) memory objects for monitoring at runtime. `start_addr` is the starting address of the memory object; `obj_size` is the size of the object. A hash code is computed based on these `K` monitored objects at each synchronization point. The hash code is sent along with the first synchronization message sent from the target process to the monitoring process. An alert is issued if the hash code is different from the hash code computed by the monitoring process.

5.2 Extensions for Decoy Monitoring

In decoy monitoring, the monitoring process can perform a reduced level of monitoring of the progress



(a) target process is not the current lock holder or last releaser



(b) target process is the current lock holder or last releaser

Figure 4. Synchronization at `Tmk_lock_acquire()` and `Tmk_lock_release()`

of the target process by tracking the synchronization points that have been encountered by the target process. These synchronization points signify the current execution location of the target process. To track the progress of the target process execution, an FSA is automatically created by the compiler in which each state corresponds to a synchronization message in the code executed by the target process. To monitor the progress, the monitoring process checks the validity of the progress by traversing the FSA. The details of the FSA scheme can be found in [20].

To deceive the host cluster into believing that full monitoring is being performed, the monitoring process continues to react to synchronization messages received from the target process by sending ACK messages to the target process. Each ACK message has an empty message body with the *monitoring message type* field in the message header set to `MON_MSG_ACK`, and the monitoring sequence number set to a special value 0. In addition, the monitoring process properly responds to other requests (e.g. connection requests, `distribute` messages) as if it were doing full-scale monitoring.

To keep the ACK traffic volume extremely low and still produce enough ACKs to eliminate waiting at the target process, the monitoring process sends 5 ACKs (100 bytes of traffic) for each synchronization messages it receives from the target process. The extra ACKs will stay in the target process’ receiving socket buffer and is picked up immediately at its next request. Without sniffing and decoding the network traffic, the host cluster or the TreadMarks application running on the host cluster cannot tell which monitoring mode is being used. The decoy monitoring mode eliminates the computational overhead on the monitoring process; it also avoids the three factors contributing to the communication overhead mentioned in Section 4.2.

6 Experiments

6.1 Experimental Settings

Our experimental testbed consists of a host cluster located at Purdue University and a remote monitoring

node located at the University of Illinois at Urbana-Champaign (UIUC). The host cluster consists of nine nodes connected through a gigabit switch. TreadMarks applications always run on eight of the nodes. The ninth node is used as the monitoring node in one of the three experiments. Each node in the cluster has an Intel P4 3.0GHz CPU with 1G memory, and runs Red Hat Enterprise Linux WS release 3. The UIUC monitoring node has 4 Intel Xeon 3.06GHz CPUs (only one CPU is actually used in monitoring) with 2G memory, and runs Red Hat Linux release 9.

We measure the performance impact of monitoring and the corresponding traffic statistics under three scenarios: (1) FLM (Full-scale Local Monitoring) – when the host cluster and the monitoring machine are connected to the same switch, performing full-scale monitoring; (2) FRM (Full-scale Remote Monitoring) – when the monitoring machine and the host cluster are connected via the Internet, performing full-scale monitoring; and (3) DRM (Decoy Remote Monitoring) – when the monitoring machine and the host cluster are connected via the Internet, performing decoy monitoring. Scenario FLM reveals the lower bound of monitoring overhead when the network latency is minimal. It represents the real world scenario where there is a certified machine within the host cluster’s LAN. Scenarios FRM and DRM are real world scenarios when the monitoring node and the host cluster are geographically far apart. Because our extended runtime library uses eager interval creation in `Tmk_lock_release()`, we also take measurements using the original TreadMarks runtime library with eager interval creation at lock release (Tmk EIC) in order to isolate the effect of eager interval creation at lock release. Each experiment is run 5 times. The average numbers are reported.

We use 4 benchmarks: `Water-Spatial`, `MG`, `Barnes-Hut`, `tsp`. The input problem sizes and commandline parameters are shown in Table 1. All the benchmarks run with 8 computation nodes. Monitoring versions use an extra machine (local or remote) as the monitoring node. Because the algorithm used in TSP is non-deterministic, its total running time can vary based on the workload assigned to the different

Benchmark	Size/Iterations	Arg.	Synchs
Water-Spatial	262144 molecules, 32768 boxes	-X1	barriers + locks
MG	256x256x256, 20 multi-grid iterations	-X1	barriers only
Barnes_Hut.c	65536 iterations	-	barriers + locks
TSP	19 cities	-	barriers + locks

Table 1. Input problem sizes, arguments of applications, and their synchronization characteristics. “-X1” is used when needed to enlarge the shared memory by doubling the page size.

Benchmark	Tmk EIC	FLM	FRM	DRM
Water-Spatial	0.57%	-2.34%	189.32%	-1.04%
MG	0.04%	4.74%	242.26%	0.55%
Barnes-Hut	-0.40%	4.54%	300.90%	0.16%
TSP	-4.27%	-6.51%	157.83%	-6.19%
Average	-1.01%	0.11%	222.58%	-1.63%

Table 3. Execution overhead under the 4 versions relative to under the original TreadMarks

machines.

6.2 Experimental Results

Table 2 shows the statistics of benchmark applications using the original TreadMarks running on the cluster using 8 nodes. This is the baseline of our analyses below. Note that the execution time reported by TreadMarks benchmarks excludes initialization and report generation, which typically also involve lock and barrier primitives. An average lock acquire time of 0 indicates there are no contending locks in execution path. Without loss of generality, we set process 1 to be the target process in the experiments with monitoring.

6.2.1 Runtime Overhead

Table 3 shows the runtime overhead using eager interval creation at lock release and with the three monitoring scenarios. Negative overhead means the version linked with the modified runtime library runs faster than the version linked with the original TreadMarks.

Eager interval creation in `Tmk_lock_release()` alone, with no monitoring involved, has little effect on the running time of the benchmarks. The exception is TSP, whose time to completion could vary with the order and time that the better paths are found.

In FLM, we use a ninth machine in the cluster of the monitoring machine. On average, this yields an increased execution time of 1.12% on top of the EIC results. This can be viewed as a lower bound on full-scale monitoring when the connection between host cluster and monitoring machine has minimum latency. We consider TSP an exception due to its nondeterministic nature.

In FRM, we use the UIUC machine as the monitoring node. This scenario represents a real world full-scale monitoring case where the monitoring machine and the host cluster are geographically separated. This scenario yields an average monitoring overhead of 222.58%. The significantly higher monitoring overhead is a consequence of the retransmission of synchronization and diff messages (discussed in Section 6.2.2).

In DRM, we use the same setting as with the second scenario except that the monitoring process performs decoy monitoring. This scenario represents a real world decoy monitoring case. The change in execution time in this scenario is insignificant compared to EIC, except for TSP, whose execution time fluctuates by nature.

The other aspect of monitoring overhead is the CPU overhead on the monitoring node. In the full-scale monitoring, since the monitoring process mirrors the computation performed on the target process, the CPU overhead on the monitoring node is application-dependent. In addition, our experiments show that computing the hash has negligible overhead compared to the average lock acquire time and barrier time. Hashing of 100KB of data, which far exceeds the need of any realistic application, incurs an overhead of only 0.12ms. In decoy monitoring, we observe no noticeable CPU overhead (almost 0.0% CPU usage) on the monitoring node.

6.2.2 Runtime Statistics

To analyze the overall execution overhead shown in Table 3, we measured the runtime statistics of the 4 benchmark applications under 5 different settings: original TreadMarks, Tmk EIC, FLM, FRM, and DRM. For ease of comparison, we normalized the data (except retransmission statistics where the baseline is 0 for some applications) under Tmk EIC, FLM, FRM, and DRM with respect to the data under the original TreadMarks, i.e., each number reported below is the *ratio* of the datum to the corresponding datum obtained using the original TreadMarks.

Lock acquire time and barrier manager time:

We first measured the average lock acquire time and barrier manager time (from when the barrier manager enters `Tmk_barrier()` to when it leaves) of the computation processes. Table 4 shows the normalized average lock acquire time and barrier manager time. The average lock acquire time of `MG` and `Barnes_Hut` are omitted because there are no contended locks. We observed a significant increase in lock acquire time and barrier manager time when performing the full-scale remote monitoring (FRM). The difference between the speeds of the monitoring and target processes leads to *retransmissions* of synchronization messages and `diff`

Benchmark	Running time (s)	# lock acquires	Avg. lock acq time (s)	# barrier arrivals	Avg. barrier time (s)	# msgs	Traffic volume (MB)	# diffs
Water-Spatial	37.45	96	0.0059	14	0.0781	117583	354.88	42843
MG	71.08	10	0	804	0.0423	339080	1426.46	121222
Barnes-Hut	16.05	12	0	19	0.9835	899816	324.35	100899
TSP	6.14	698	0.0551	3	0.0031	18273	5.41	3813

Table 2. Runtime statistics using the original TreadMarks on 8 nodes

Benchmark	Average lock acquire time				Average barrier manager time			
	Tmk EIC	FLM	FRM	DRM	Tmk EIC	FLM	FRM	DRM
Water-Spatial	1.02	0.97	265.14	1.11	1.34	0.74	32.60	0.76
MG	–	–	–	–	1.00	1.08	5.99	1.01
Barnes-Hut	–	–	–	–	0.99	1.07	5.79	0.99
TSP	0.95	0.93	2.77	0.93	1.25	1.08	4.44	2.71
Average	0.98	0.95	133.96	1.02	1.14	0.99	12.20	1.37

Table 4. Normalized average lock acquire time and barrier manager time

Benchmark	Synchronization messages					diff requests				
	Orig. Tmk	Tmk EIC	FLM	FRM	DRM	Orig. Tmk	Tmk EIC	FLM	FRM	DRM
Water-Spatial	1	1	1	379	12	0	0	0	148	0
MG	0	0	0	669	3	0	0	0	0	0
Barnes-Hut	48	47	52	322	48	0	0	0	367	0
TSP	0	0	0	0	8	0	0	0	0	0

Table 5. Retransmission of synchronization messages and diff requests

request among the computation nodes in some applications (Table 5), due to request-reply timeouts.

Table 4 also shows that FLM and DRM have only a slight impact on lock acquire time and barrier manager time compared to the original TreadMarks and Tmk EIC. The increased barrier manager time in FLM for `Barnes_Hut` and in DRM for `TSP` is also due to an increased number of *retransmissions* of synchronization messages as shown in Table 5.

Traffic: When measuring traffic statistics, we count only *received* messages. This is consistent with how the original TreadMarks counts its runtime statistics. In TreadMarks, each node maintains two sockets for each of the other nodes. One of the sockets (*request socket*) is used to send requests to the other node and then wait for its reply. The other socket (*reply socket*) is used to *listen* for requests from the other node and then send the corresponding replies. The interrupt service function (`sigio_handler()`) polls (using `select`) only the reply sockets for incoming requests. In DRM, the periodic dummy ACK message is sent to the *request socket* of the target process, therefore it does not trigger a `recv()` operation (and thus is not counted as a received message) unless the target process is expecting an ACK (when it is counted as received message). Table 6 presents the normalized number of messages received by the computation nodes and the total size of received messages. On average, performing full-scale monitoring (FLM or FRM) increases the number of messages by 6% and increases the traffic volume by

Benchmark	# of msgs received			Size of received msgs		
	FLM	FRM	DRM	FLM	FRM	DRM
Water-Spatial	0.06	0.06	0.0008	0.12	0.12	0.0002
MG	0.04	0.04	0.0049	0.10	0.10	0.0002
Barnes-Hut	0.04	0.04	0.0001	0.08	0.08	0.0001
TSP	0.06	0.06	0.0096	0.13	0.13	0.0060
Average	0.05	0.05	0.0038	0.11	0.11	0.0016

Table 7. Normalized number of messages and their total size received by the monitoring node

1%. In contrast, performing decoy monitoring (DRM) increases the number of messages and total traffic volume slightly (less than 1% on average).

Table 7 presents the number of messages received by the monitoring node (FLM and FRM) and the total size of the received messages, normalized to the corresponding numbers of the original Treadmarks. We can see that the monitoring node receives about 5% of the messages and about 11% of the traffic (compared to the total in the original Tmk). These numbers are reduced to 0.38% and 0.16% in DRM.

7 Related Work

The untrusted nature of the P2P environment results in ensuring fairness in charging for the service and verifying the execution results to be essential problems in P2P cycle sharing. On one hand, the client may cheat by not paying the computing server for its work; on the other hand, the server can cheat the client by charging for non-existent or false computation. While the former can be discouraged by maintaining a rep-

Benchmark	# of messages received				Total size of received messages			
	Tmk EIC	FLM	FRM	DRM	Tmk EIC	FLM	FRM	DRM
Water-Spatial	1.00	1.06	1.07	1.00	1.00	1.00	1.00	1.00
MG	1.00	1.06	1.06	1.01	1.00	1.00	1.00	1.00
Barnes-Hut	1.00	1.05	1.05	1.00	1.00	1.01	1.01	1.01
TSP	1.00	1.07	1.07	1.01	1.00	1.02	1.02	1.01
Average	1.00	1.06	1.06	1.00	1.00	1.01	1.01	1.00

Table 6. Normalized number of messages and their total size received by the computation nodes

utation system in the P2P network [2], the latter is much harder to detect and defeat. Existing schemes are based on code encryption (in special cases) [18], trusted hardware [21], a combination of hardware and encryption [15], dummy objects [16], or a quiz-scheme [14]. Recent work has addressed the problem of monitoring the progress of submitted jobs and verifying the correctness of remote execution. Ali et al. [5] propose a progress monitoring scheme for Java programs based on finite state machines. Yang et al. [19] propose a monitoring scheme for Java using a location-beacon-based finite state machine and partial replay of the computation to support monitoring for progress and correctness. Our monitoring differs from all of these schemes in that we exploit the design of TreadMarks to replicate the computation performed at one of the host nodes to ensure the faithful execution of submitted program.

8 Conclusions

Verifying the faithful execution of a program on a remote node is an important problem in peer-to-peer cycle sharing. In this paper, we present the design and implementation of GripCop DSM, a novel incremental monitoring and verification scheme for TreadMarks programs submitted to a remote cycle-sharing host cluster. GripCop DSM replicates the computation on one of the host cluster nodes on a trusted monitoring machine, and employs either a full-scale monitoring mode or a decoy monitoring mode. The full-scale monitoring mode performs replicated computation and hence rigid monitoring of execution progress and correctness. The decoy monitoring mode does not perform replicated computation, but deceives the host cluster into thinking it is performing the replicated computation. Hence it is light-weight when the monitoring machine connects to host cluster via the Internet. A combination of these two monitoring modes provides a remote execution monitoring scheme that guarantees overall efficiency and is capable of detecting fraudulent nodes quickly.

References

- [1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.
- [2] N. Andrade, F. Brasileiro, and W. Cime. Discouraging free riding in a peer-to-peer cpu-sharing grid. In *HPDC '04*, 2004.
- [3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP '03*, October 2003.
- [4] A. R. Butt, S. Adabala, N. H. Kapadia, R. J. Figueiredo, and J. A. B. Fortes. Grid-computing Portals and Security Issues. *Journal of Parallel and Distributed Computing*, 63(10), October 2003.
- [5] A. R. Butt, X. Fang, Y. C. Hu, and S. Midkiff. Java, Peer-to-Peer, and Accountability: Building Blocks for Distributed Cycle Sharing. In *USENIX VM '04*, May 2004.
- [6] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *SOSP '91*, Oct. 1991.
- [7] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM Computer Communications Review*, 33(3), 2003.
- [8] A. P. David. BOINC:A System for Public-Resource Computing and Storage. In *Proc. 5th IEEE/ACM International Workshop on Grid Computing*, November 2004.
- [9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90*, May 1990.
- [10] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. In *USENIX Security '04*, August 2004.
- [11] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA '92*, May 1992.
- [12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.
- [13] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.
- [14] V. Lo, D. Zhou, D. Zappala, Y. Liu, and S. Zhao. Cluster computing on the fly: P2p scheduling of idle cycles in the internet. In *3rd IPTPS*, 2004.
- [15] S. Loureiro, L. Bussard, and Y. Roudier. Extending tamper-proof hardware security to untrusted execution environments. In *Proceedings of the Fifth Smart Card Research and Advanced Application Conference*, 2002.
- [16] C. Meadows. Detecting attacks on mobile agents. In *Foundations for Secure Mobile Code Workshop*, pages 64–65, 1997.
- [17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, November 2001.
- [18] T. Sander and C. F. Tschudin. Towards mobile cryptography. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1998.
- [19] S. Yang, A. R. Butt, Y. C. Hu, and S. Midkiff. Trust and Verify: Monitoring remotely executing programs for progress and correctness. In *PPoPP '05*, 2005.
- [20] S. Yang, A. R. Butt, Y. C. Hu, and S. P. Midkiff. Lightweight monitoring of the progress of remotely executing computations. In *LCPC '05*, 2005.
- [21] B. Yee. *Using secure coprocessors*. PhD thesis, Carnegie Mellon University, May 1994.