Trust but Verify: Monitoring Remotely Executing Programs for Progress and Correctness

Shuo Yang, Ali R. Butt, Y. Charlie Hu and Samuel P. Midkiff
Electrical and Computer Engineering Purdue University
West Lafayette, IN 47907
{yang22,butta,ychu,smidkiff}@purdue.edu

ABSTRACT

The increased popularity of grid systems and cycle sharing across organizations requires scalable systems that provide facilities to locate resources, to be fair in the use of those resources, and to monitor jobs executing on remote systems. This paper describes the GridCop system which allows a computation on a remote, and potentially fraudulent, host system to be monitored for progress and execution correctness. A novel feature of our system is that it constructs cooperating *submitter* and *host* programs from the original program, and these programs allow both progress and execution correctness to be monitored with negligible overhead while providing protection against common fraudulent behaviors. Experimental results show that the overhead of this monitoring is low on both the submitting and host machines. We describe compiler algorithms that allow the required monitoring code to be automatically generated.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent programming – distributed programming; D.3.4 [**Programming Languages**]: Processor – compiler; D.2.5 [**Software Engineering**]: Testing and Debugging – monitors, tracing

General Terms

Security, Verification, Performance, Experimentation

Keywords

Grid computing, cycle-sharing, correctness verification, Progress monitoring, trustworthiness, security

1. INTRODUCTION

Computational workloads for many academic groups, small businesses and consumers are bursty. That is, they are characterized by long periods of little or no processing punctuated by periods of insufficient compute cycles. By aggregating large numbers of computers and users together, the resource demands are "smoothed out" across sub-groups even as demand remains bursty within sub-groups. Centrally managed software projects [11, 22, 27] have sprung up to access idle machine to perform computations that would be economically infeasible to solve on committed hardware. Centrally managed systems like Condor [20] and LoadLeveler [14] have been developed to allow resources to be aggregated within permanent or ad-hoc organizations.

Centralized administration of resources exists because they allow a trusted entity – the system administrators – to verify and track the trustworthiness of users given access to the resources, *and* they allow users to deal with a known, trusted entity. Certification of the user of a machine is almost always contingent on being an employer of the machine owner, or being certified by another organization which the user belongs to, and which is in turn trusted by the machine owner. This certification requires legal contracts that carefully delineate risks and responsibilities, staffs to maintain accounts, accountants to monitor funding streams and tax consequences, and generally increase the overhead and real costs of acquiring and using computing resources. This in turn restricts the domain of applications that can be run on shared resources.

The irony of this situation is that the resources to be shared are extremely perishable – cycles, bandwidth and disk space not used in the past do not create additional resources to be consumed in the future. The major value of these resources to their owner is the knowledge that they are available when needed, and the major costs of sharing unneeded cycles are the legal and administrative overheads. Eliminating these overheads would dramatically increase the quantity, and decrease the cost, of available cycles. Both the decreased cost and the ease of accessing cycles would increase the applications that could exploit them. Academics and research laboratories would have access to a vast array of machines for running simulations, benchmarking programs, and running scientific applications; small businesses would have machines available for data-mining sales, accounting and forecasting; and consumers would have machines available to perform computationally intensive, but low-economic value activities such as games and digitally processing home movies. Elimination of these overheads would allow automatic intermediation between con-

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

sumer and providers of resources, allowing shared resources to blend seamlessly with locally owned resources.

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

1. How are resources to be used discovered, and how are providers of these resources to be 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?

The first two of these items are the subject of research by the authors of this paper and others [1, 2, 3, 8, 13, 25] These systems support one or more of sandboxing applications for host safety, resource discovery and compensation.

The GridCop system described in this paper solves the third problem – how can the submitter track the progress of a job, and receive assurances that the job is executing correctly? In solving this problem, we make the following technical contributions.

- We present the first known technique to remotely monitor the incremental progress of a program that is executing in an untrusted environment.
- We present a method of instrumenting a program with *location beacons*. This technique treats a modified program control flow graph as a finite state automata (FSA), and constructs a *transducer* using this FSA. The transducer is part of the program that is executed on the host, and emits information that allows a corresponding FSA at the submitter machine to follow the progress of the job.
- Although the transducer provides excellent information about the progress of a job, at a low-overhead, it is susceptible to *replay* and *spoofing* attacks, e.g. attacks where the monitoring output from a previous run is replayed, or the output from the alleged current run is faked. We present a method of instrumenting a program with *recomputation beacons* to compute the input and output of selected program regions and to send these inputs and outputs to the submitter machine for verification.
- We present compiler techniques that can be used to automatically construct the host and submitter FSAs, and to collect input and output data for a region of the program and replay the execution of the program on the submitter machine.
- We provide experimental results measured over a wide area network showing the overhead of these techniques on both the host and submitter machines. The overhead of these monitoring techniques is less than 3.5% on the host side. The cost of monitoring a job is less than 1.6% of that of running it locally on the submitter side.

The rest of the paper is organized as follows. Section 2 presents the threat model considered in this paper. Section 3 gives an overview of our GridCop monitoring system. Section 4 presents the design of the GridCop system, and section 5 presents the experimental results showing the effectiveness of the system. Finally, Section 6 discusses the related work and Section 7 concludes the paper.

2. THE THREAT MODEL

Untrusted systems can be divided into two categories: malicious and fraudulently irresponsible (e.g. the environment targeted by the Samsara [7] system.) Malicious systems are willing to expend significant resources to damage their victims. For example, a malicious system might be willing to execute a program until the final results are to be written to disk, and then terminate it. Our system is assumed to operate in a less hostile, but perhaps fraudulent environment. In particular, we assume that hosts are motivated by self-interest, that our programs execute on unmodified JVMs, and that the source program is not altered. These are reasonable assumptions since JVMs are large complicated pieces of software distributed in binary form, and are not easy to modify without access to the source code. Our monitoring system makes it very difficult to change a program without being caught. Moreover, because we *apparently* monitor almost the entire program execution (but *actually* monitor only a small part of the execution), fully automatic tools will not be effective. Program alteration requiring human intervention is costly enough to preclude its use.

3. OVERVIEW OF THE GRIDCOP SYSTEM

Every participating node can submit jobs (i.e. be a *submitter* node) or host jobs (i.e. be a *host* node). A node can perform both roles simultaneously, i.e., its jobs can be running on remote nodes while jobs from other remote nodes are running on it. Before a program is executed, it is passed to our tool which transforms it into a program that executes on the host machine (*H-code*) and a program that executes on the submitter machine (*S-code*). The H-code is the original program augmented with *beacons* and auxiliary code that send information about the program to the submitter machine. The S-code, executing on the submitter machine (or a trusted machine accessible from the submitter machine) uses this information to track the progress, and verify the execution of, the program. In our larger system described in [3], the host and submitter nodes also have the ability to issue and receive credits for services consumed and provided. This topic is beyond the scope of this paper, and the credit system our system will deploy is described in [3].

H-code contains two types of beacons: *location beacons (L-beacons)* and *recomputation beacons (R-beacons)*.

L-beacons are placed along certain control flow graph (CFG) edges, as described in Section 4.1, and identify the location of the program that is currently executing. L-beacons provide fine-grained location information, but because their location and output are static properties of the program, they are subject to replay and spoofing attacks. For example, a tool on the host machine could extract the CFG of the submitted program and generate a program that simply sends L-beacon information periodically. To guard against these attacks, GridCop also adds R-beacons to the program, as described in Section 4.2. R-beacons send the input and result data for a code region to the submitter machine where the computation of the region can be checked. The effectiveness

of this approach requires that:

- the computation *actually* duplicated at the submitter machine be a small part of the overall computation. Otherwise the submitter machine is not efficiently offloading work to the remote system.
- the computation *potentially* duplicated at the submitter machine must be a large portion of the whole job. Otherwise, our work will allow malicious hosts to develop a program analysis tool to extract the duplicated computation from the program and only execute it.

The subset of the *potentially* duplicated regions that are *actually* recomputed is selected randomly, i.e. in a way that is not predictable by the host machine.

The S-code program corresponding to the H-code program, and generated from the original program, is executed on the submitter node. The S-code program has two functions. The first is performed by a finite state machine (FSA), derived from the original program CFG, that tracks the L-beacons sent by the H-code. As each L-beacon is received, the current state is updated. The second uses R-beacons. When an R-beacon is received, the code region selected for recomputation is identified, and the computation is performed. The R-beacon also contains the result of executing the code region, which is compared with the result of the recomputation to determine if the host faithfully executed the region, and by implication, the entire program.

4. IMPLEMENTATION OF THE GRIDCOP SYSTEM

Our compiler-assisted remote job monitoring system consists of compiler techniques to produce the S-code and H-code programs, and two monitoring modules, the L-beacon module and the R-beacon modules. Each module consists of three run-time components: (i) a beacon message creation component on the host, which is integrated with the submitted application, (ii) a beacon reporting component on the host that sends beacon messages to the submitter, and (iii) a beacon message processing component on the submitter. The monitoring system architecture is shown in Figure 1. The monitoring system can adjust the runtime configurations, such as the inter-transmission interval of the reporting components, based on the trust level between the submitter and the host. For example, when a submitter has a high level of trust in a host and would like to conduct less aggressive monitoring, it can set a longer inter-transmission interval when submitting a job.

4.1 Location Beacons

L-beacons emit information at significant program execution points. This information is sent to the submitter machine which uses it to determine what parts of the program have been executed. Placing L-beacons at every branch would provide very precise, fine-grained beacon information. Unfortunately, L-beacon processing would consume unacceptably large amounts of resources on the host and the submitter machines, since a large percentage of the instructions executed would be beacon related. Therefore GridCop places L-beacons at computationally significant points in the program. In the rest of this section, we describe how beacon code is generated for the H-code, how beacon processing code is generated for the S-code, and the actions that take

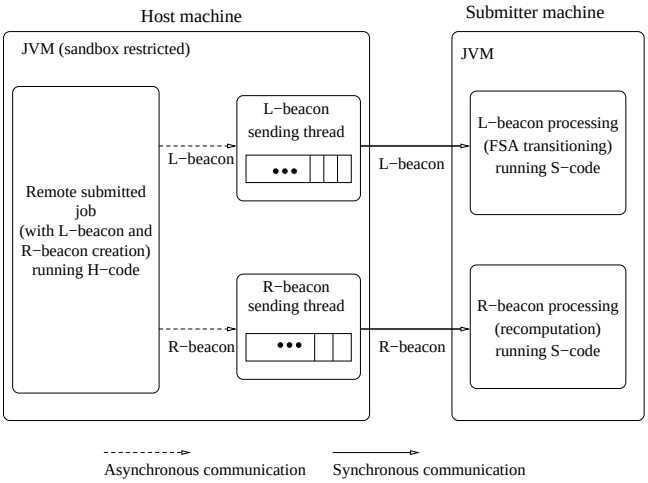


Figure 1: Components of the GridCop monitoring system

place at runtime to create, report and process beacon messages.

4.1.1 Code Generation for L-beacons

For H-code, the compiler finds computationally significant points in the program and places a beacon at each of them. It also inserts hard-coded routines into the code that aggregate and structure information provided by beacons, and that report the information to the submitter program. These routines are described in Section 4.1.2. For the S-code, an FSA is created, where each state in the FSA corresponds to a beacon in the H-code. Beacon processing consists of checking if a sequence of L-beacon messages corresponds to legal transitions on the FSA. Viewed in this light, L-beacons in the H-code can be viewed as a transducer, i.e. a finite state machine that emits information on state transitions.

Because L-beacons report the progress of a job, the time between milestones should be reasonably long. Our approach is to place L-beacons at the beginning of method calls, but only at the beginning of method calls that perform a "significant" amount of work. Two broad strategies can be used to determine these methods, profiling and compile time cost models. We have chosen to use a very crude compile-time analytical cost model, namely, methods that contain loops are considered to be computationally significant. Future work for the project may involve refining this model (although for the current benchmark set, our current model is adequate). A beacon is inserted as the first statement in each computationally significant method. The beginning of the main program also gets a beacon, as does any node that can return from the program as the result of a normal program termination. For a multi-threaded program, each `run()` function is also treated as a main program.

As described above, when beacon information is sent to the submitter machine, the submitter checks the validity of the beacon information by traversing an FSA. Each beacon inserted above corresponds to a state in the FSA, and the beacon's ID is the input for the transition to the state. The beacons inserted at the beginning and the end of the program serve as the start and the accept state, respectively, in the FSA.

$N_b \Leftarrow \phi$;
$N_{\tilde{b}} \Leftarrow \phi$;
for(each node n in G_F){
    if (n is annotated with a beacon)
        $N_b \Leftarrow N_b \cup \{n\}$;
    else
        $N_{\tilde{b}} \Leftarrow N_{\tilde{b}} \cup \{n\}$;
}
while($N_{\tilde{b}} \neq \phi$){
    $n_{\tilde{b}} \leftarrow dequeue(N_{\tilde{b}})$;
    for(each node $n_p \in predecessor(n_{\tilde{b}})$){
        for(each node $n_s \in successor(n_{\tilde{b}})$){
            $Edges(G_F) = \langle n_p, n_s \rangle \cup Edges(G_F)$;
        }
    }
    $Edges(G_F) = Edges(G_F) - \{edge\ connected\ to\ N_{\tilde{b}}\}$;
    $N_{\tilde{b}} \Leftarrow N_{\tilde{b}} - \{n_{\tilde{b}}\}$;
}
$G'_F \Leftarrow G_F$;

$State_{FSA} \Leftarrow \{nodes\ in\ G'_F\}$;
$Transition_{FSA} \Leftarrow \phi$;
for($\langle n_i, n_j \rangle \in Edges(G'_F)$){
    $Transition_{FSA} \leftarrow Transition_{FSA} \cup$
        $\{transition \langle n_i, n_j \rangle\ with\ ID\ n_j\}$;
}

Figure 2: Algorithm of Building FSA

The algorithm of Figure 2 describes how to build an FSA. We build the FSA for the submitter side in two steps. The first step transforms the control flow graph (*CFG*) G_F of the program into a graph G'_F that contains only nodes with beacons. First, we add all nodes of G_F that contain L-beacons (i.e. the computationally significant nodes) to the set N_b. All other nodes of G_F are placed in the set $N_{\tilde{b}}$ For each node $n_{\tilde{b}} \in N_{\tilde{b}}$ edges $\langle n_p, n_s \rangle$ are added to G_F, for all $\langle n_p, n_{\tilde{b}} \rangle$ and $\langle n_{\tilde{b}}, n_s \rangle$ in G_F. $n_{\tilde{b}}$ is then removed from G_F. We call the graph resulting from the above procedure G'_F.

The second step generates the FSA. The FSA transition table can be trivially constructed as follows. Let the set of states be the set of nodes in G'_F. For each edge $\langle n_i, n_j \rangle \in Edges(G'_F)$, there is a transition from n_i to n_j on the symbol which is the beacon ID of n_j. The start state of the FSA corresponds to the node at the beginning of the main program, and accept states correspond to nodes that can return from the program.

Figure 3 shows a program fragment with beacons inserted. For each computationally intensive function in Figure 3(a), the compiler inserts an L-beacon instruction at the beginning of the function to emit a unique transition ID in Figure 3(b). These IDs are treated as the transition symbols between states in the FSA. A transition ID always drives the FSA to the unique state named by the transition ID. Consider the FSA shown in Figure 3(c). For example, `bar2` in the example emits "3", which causes a transition from any predecessor of `bar2`, on the FSA, to `bar2`.

4.1.2 Runtime Support for L-beacons

At runtime, the L-beacon instructions inserted by the compiler are executed. Each L-beacon deposits a location

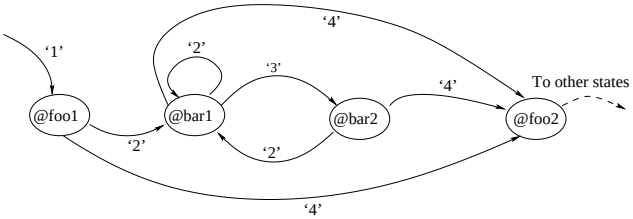
```
(in main function)
...
call foo1(...)
call foo2(...)
...


function foo1(...){
    for(...){
        call bar1(...)
        if(...) {
            call bar2(...)
            call lightbar(...) //this function is loopless
        }
    }
}
function bar1(...){ ... for( ...) ... }
function bar2(...){ ... for( ...) ... }
function foo2(...){ ... for( ...) ... }
```

(a) A piece of pseudo code from benchmark LU. All functions but lightbar() contain a loop

```
function foo1(...){ call create_beacon('1') ...}
function bar1(...){ call create_beacon('2') ...}
function bar2(...){ call create_beacon('3') ...}
function foo2(...){ call create_beacon('4') ...}
```

(b) L-beacon added at the prologues of functions with loops



(c) Part of FSA corresponding to above program: transition symbols on the edges correspond to the unique IDs emitted by the inserted beacon instructions

Figure 3: A part of FSA derived from the Java Grande LU benchmark

ID (ID_L) and the current thread ID (ID_T), which together form an L-beacon message (ID_T, ID_L) which is added to the L-beacon buffer. For multi-threaded programs, L-beacon messages from different threads are multiplexed in the L-beacon buffer. ID_Ts are used to label different threads so that the L-beacon processing component on the submitter side can use them to track the progress of different threads. A different FSA is used for each application thread, thus ID_T in a beacon message identifies an FSA and ID_L drives transitions on that FSA. The L-beacon buffer size is set to contain 3000 L-beacon messages. If the L-beacon buffer is full, an L-beacon instruction will return without depositing any information.

The reporting component on the host is in a separate thread (*L-Sender*) that transmits L-beacon messages to the submitter through a TCP connection. The reporting component uses a *paced* beacon transmission scheme. It sleeps for an interval (set by the submitter when the program is submitted to the host) and wakes up to send the L-beacon messages in the L-beacon buffer. During the sending procedure, L-Sender first copies the contents in the L-beacon buffer to a private buffer and clears the L-beacon buffer. It then sends the contents in the private buffer across the network. Thus the cross-network data transfer procedure is asynchronous to the application program.

The submitter node uses the L-beacon messages sent by the host as input to the FSAs to track the progress of the remote job. It reads each L-beacon message received from the host, locates an FSA using ID_T and drives transitions using ID_L. In this way, the L-beacon messages are de-multiplexed and processed. If the submitter receives a full L-beacon buffer, i.e., the L-buffer was full before the packet was sent, the submitter assumes that L-beacon messages may have been discarded on the host side. In this case, the submitter checks whether the first message for each thread in this packet causes a transition to a state reachable from the current state of the thread. After the first message for each thread gets processed, normal FSA transitioning continues. Because the L-beacon messages within each transmission interval are limited by the sending interval, the submitter overhead to process these beacons is also limited.

4.2 Recomputation Beacons

R-beacons check the validity of the host computation. At the start of an identified computation region (i.e. an innermost loop in the current implementation), the input of the computation along with the computation ID (ID_C) is placed in the R-beacon buffer; at the end of the identified computation, the result of the computation is added to the R-beacon buffer. An ID_C, the input, and the corresponding result form the *R-beacon message* (ID_C, input, result). The R-beacon message is transfered to the submitter for verification, with the ID_C allowing the submitter to know what computation to apply to the input. If a host has not correctly executed the program, the result from the host will differ from the result computed locally on the submitter with the same input. Thus the submitter can detect a cheating host by periodically comparing the local computation results with the remote results for the same inputs.

4.2.1 Code Generation for R-beacons

By inserting R-beacons at significant computation regions, we are potentially monitoring most of the computation. We currently use loop nests to identify significant computation regions. In particular, we place a recomputation beacon immediately before every innermost loop. Computationally significant regions could be identified with profiling information for general programs, or by using more sophisticated compile-time analytical cost models [6, 29].

After identifying a significant computation region, the compiler inserts an R-beacon into the H-Code. This is a straight-forwardly automatable procedure. At the prologue of the identified computation region, a call to `addInputToBuffer()`, which will put the ID_C and the input of this computation region in the R-beacon buffer at runtime, is inserted. At the epilogue of the identified region, a call to `addResultToBuffer()`, which will put the output of this computation region in the R-beacon buffer at runtime, is added. For the S-Code, the compiler generates a subroutine to handle all of the recomputation cases resulting from the R-beacons in the corresponding regions in the H-Code. This subroutine is called for every R-beacon message the submitter processes. Figure 4 shows the code transformation to insert a R-beacon at a desired region.

In general, the input data passed to the R-beacon must contain all upwardly exposed uses of variables in the region, that is variables that are read in the region, but the value read is defined outside of the region. For scalars, the SSA form of the program can be used to easily locate these variables. For arrays, the SSA form may indicate the entire array is upwardly exposed when only some elements are. Sending the entire array can increase the communication overhead involved in monitoring. The Array SSA form [15] allows, via compiler analysis and runtime tracking, individual upwardly exposed elements of the array to be determined, and only those elements to be sent. Moreover, by using the Array SSA form, arbitrarily small and/or complex regions of the program can be selected for monitoring.

For many dense numerical codes, subscripts are affine and are not *coupled*, i.e., each index variable only appears in one dimension of the array. For these references, a relatively simple static determination of the input for a region can be made. Since regions are delimited by innermost loops, with an index of i_m, upwardly exposed references will be in a slice(s) of arrays $a(f_0, f_1, \ldots, f_{m-1}, lb(i_m) : ub(i_m), f_{m+1}, \ldots, f_{n-1}, f_n)$, where where f_j is the value of the subscript function of dimension j of the array a,and lb and ub are the lower bound and, upper bound, respectively, of the subscript function that contains i_m evaluated over the range of i_m. Again, because the R-beacon is outside of the innermost loop, all f_j are constants within that innermost loop. In our benchmarks we use these simple slices as the inputs, and as shown in Section 5 this incurs an acceptable overhead.

4.2.2 Run Time Support for R-beacons

In Figure 4, `addInputToBuffer()` is an atomic, thread-safe operation that puts the computation input and the ID_C of an identified computation region into the R-beacon buffer. `addResultToBuffer()` is an atomic, thread-safe operation that adds the corresponding result to the R-beacon buffer. Multiple threads of an application can contain R-beacons, and we must ensure the operation of putting a R-beacon message to the R-beacon buffer is atomic. The buffer management routines ensure that the input and result data for

```
...
double dx[], dy[], da;
int dx_off, dy_off, n;
...
for(i=0; i < n; i++)
    dy[i +dy_off] += da*dx[i +dx_off];
...
```

(a) The original 'hot' spot code

```
...
double dx[], dy[], da;
int dx_off, dy_off, n;
...
Recomputationbuffer.
    addInputToBuffer(7, da, dx, dx_off, dy_off, n);
for (i=0; i < n; i++)
    dy[i + dy_off] += da*dx[i + dx_off];
Recomputationbuffer.addResultToBuffer(dy);
...
```

(b) Transformed H-Code that runs on the host node (note: ID_C=7 in this example)

```
...
//Dataobject is a Serializable object transfered from host
...
switch (Dataobject.ID_C){
...
case 7:
    double dx[], dy[], da;
    int dx_off, dy_off, n;
    da = Dataobject.input.content0;
    dx = Dataobject.input.content1;
    dx_off = Dataobject.input.content2;
    dy_off = Dataobject.input.content3;
    n = Dataobject.input.content4;
    for (i=0; i < n; i++)
        dy[i +dy_off] += da*dx[i +dx_off];
    for( i=0; i< dy.length; i++)
        if(dy[i] != Dataobject.output[i])
            report_error();
    break;
case ...
    ...
}
...
```

(c) Transformed S-Code that runs on the submitter node

Figure 4: Pseudo-code example showing a transformed 'hot spot' in LU Factorization

a single execution instance of a region are grouped together and added to the buffer as follows.

When a thread attempts to add input data to the R-beacon buffer by calling `addInputToBuffer()`, it first checks to see if the *input_outstanding* flag is false, indicating nothing is in the buffer. If the flag is false, the thread acquires a lock on *input_outstanding* and rechecks the value of *input_outstanding*. If *input_outstanding* is still false, it is set to true and the flag *thread_ID* is assigned the value of the current thread ID. The input data is then deposited into the buffer, and the lock is released. Another thread attempting to add input data will find the value of *input_outstanding* is true on either the initial check, or after acquiring a lock on *input_outstanding*. In either case, `addInputToBuffer()` returns without depositing any information into the R-beacon buffer. The unlocked check of the *input_outstanding* flag reduces, in practice, the number of locks that must be acquired, while updating *input_outstanding* and *thread_ID* under a lock eliminates data races on these flags.

When a thread attempts to add result data to the buffer by calling `addResultToBuffer()`, it again checks the *input_outstanding* flag, but this time it checks to see if *input_outstanding* is true. If it is, a lock is acquired on *input_outstanding*, and *input_outstanding* is checked again. If it is still set, the thread checks if the value of the *thread_ID* matches the current thread ID. If it does, the thread knows that it deposited the input data part of the current R-beacon message, and now deposits the result part of the message. *thread_ID* is now assigned an invalid value -1.

When the inter-transmission interval for sending R-beacons is up (two seconds in our experiments), the message in R-beacon buffer is copied to a private buffer. A lock is then acquired on *input_outstanding*, and it is set to false. At this point, a new message can be added to the buffer by another instance of a region, and the private buffer is sent to the submitter via a TCP connection.

When the submitter receives an R-beacon message, the submitter machine calls the subroutine based on the value of ID_C and performs computation with the input in the R-beacon message. It then compares its local computation result with the result in the R-beacon message. If the results are same, the verification succeeds. Otherwise, the verification fails.

To prevent replay attacks from a cheating host, we check for repeated computations using *message digests*. The submitter calculates the message digest of each input along with the ID_C in each R-beacon message using the MD5 [24] algorithm. MD5 is a fast secure one-way hash function that takes arbitrarily-sized data and outputs a 128-bit value. The message digest is then used to index into a hashtable. If a calculated message digest is not present in the hashtable, it is added into the hashtable. If it is present, appropriate action can be taken based on the likelihood of the previous redundant computations at the beacon region. Our current implementation uses a hashtable with a capacity of 6000. For a runtime configuration with inter-transmission interval of 2 seconds, i.e. highly aggressive monitoring, it can store about 2.7 hours of R-beacon inputs as message digests before reaching a load factor of 0.8. In an actual use of the system, the communication interval will likely be much longer than 2 seconds. Combined with proper replacement policies, the mechanism will be able to detect replay attacks over extremely long periods of time.

5. EXPERIMENTAL RESULTS

In this section, we present performance results showing the overhead and effectiveness of our system.

5.1 Parallel Java Grande Benchmark

The parallel Java Grande benchmark suite version 1.0 [28] is a standard benchmark suite for computationally intensive Java applications. Suite II of the parallel Java Grande benchmarks contains simple kernels which are commonly found in the most computationally intensive parts of real numerical applications. It consists of five benchmarks: `LU`, which performs LU factorization; `SOR`, which performs successive over-relaxation; `Series`, which computes Fourier coefficients of the function f(x)=(x+1)^x on the interval of [0,2]; `Sparse`, which performs a matrix multiplication of unstructured sparse matrices; `Crypt`, which performs an International Data Encryption Algorithm encryption and decryption of an array. The Java Grande benchmarks are self-initializing, i.e., there was no network activity to send program data sets. We used data size B as the input to our experiments. Programs were hand-transformed using the techniques described Section 4.

5.2 Experimental Platform

Our experiment was run on a submitter/host pair located at University of Illinois at Urbana Champaign and Purdue University. The submitter machine, located at UIUC, is an uniprocessor with an Intel 3GHz Xeon processor with 512KB cache and 1GB main memory. It runs the Sun JDK 1.5.0 and the Linux 2.4.20 kernel. The host machine located at Purdue is a Dell PowerEdge SMP server with 4 x 1.5GHz Intel Xeon processors, each with 512KB cache and sharing 4 GB main memory. It runs the Sun JDK 1.4.2 and the Linux 2.4.20 kernel. Both machines are connected to the campus network.

In our measurements, the inter-transmission intervals of the L-beacon and R-beacon reporting components were set to 2 seconds. This is a highly aggressive monitoring scenario. In an actual system, the inter-transmission interval would be in tens of seconds or minutes. Therefore, our experiment provides an upper bound of performance overhead and network traffic incurred by using our monitoring system.

5.3 Run Time Computation Overhead

To simulate long running jobs, a loop is added outside the individual kernels in the benchmarks. Each benchmark was run in 1, 2 and 4-thread modes. This is to evaluate the scalability of our system design by showing the system performance overhead with different degrees of parallelism.

5.3.1 Host side

On the host side, we first measure the time to run the original benchmarks on our host machine, which reflects the scenario of remote job execution without monitoring. These form our baseline numbers. We then run the manually transformed S-Code and H-Code versions of the same benchmarks on the submitter/host pair, which reflects the scenario of a remote job submission with monitoring. Figure 5 shows the overhead of job executions with beacons over the corresponding un-monitored baseline job execution times. In Figure 5, the overhead of our monitoring system to run benchmark SOR and LU with 4 threads is lower than that of running benchmark SOR and LU with 2 threads.

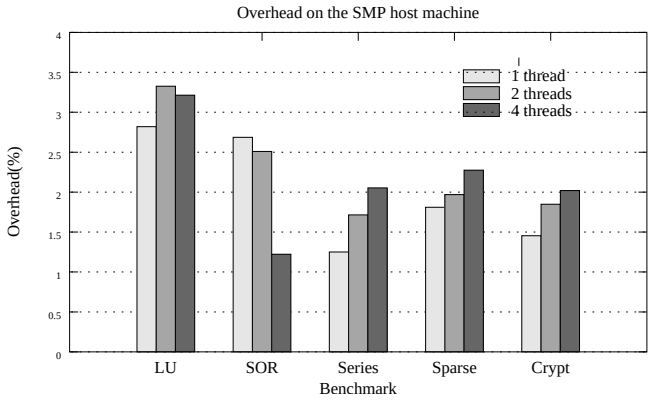


Figure 5: Overhead of executing and transferring beacons

This is because the change in the SOR and LU program speedup, going from 2 threads to 4 threads, is smaller than the increase in monitoring overhead.

Our experimental results show that the maximum performance overhead is under 3.5% and the average performance overhead of the whole benchmark suite is 2.1% in our experiments.

5.3.2 Submitter side

On the submitter side, we measured the time to execute the benchmarks on the submitter in the single-thread mode and used it as the baseline (as the submitter is not a SMP machine). We then measured the time used to process the L-beacons and R-beacons and to compute and process the message digests of R-beacon input, while monitoring remote benchmarks running in 1, 2 and 4-thread modes. These are the computation resource costs on the submitter. Figure 6 shows the ratio of CPU time used to monitor a benchmark over that of running the same benchmark in the single-thread mode locally on the submitter.

We observe that the ratio of CPU usage for monitoring over locally executing the benchmark is always under 1.6%, and the average ratio is 0.3%. This is a consequence of our paced beacon transmission technique discussed in Section 4.2.2 – the R-beacon messages that are *actually* put into the R-beacon buffer and transfered to the submitter for processing are only a very small fraction of the total potential R-beacons. The percentage of total beacon sites visited whose messages are actually sent is under 0.04% for LU and SOR, under 0.26% for Series and Crypt, and under 1.5% for Sparse. The submitter cost for monitoring benchmark Sparse is much higher than the cost for monitoring the other benchmarks because Sparse contains a single course-grained loop identified as computationally significant. Thus the percentage of the total execution that is recomputed in Sparse is much higher than that percentage for the other benchmarks. Nevertheless, the cost is still only 1.6% of the cost of executing the application locally on the submitter.

5.4 Network Bandwidth Overhead

Another important metric to evaluate our system is the network resource usage of our system. Since network resources are not unlimited, it is necessary to limit the amount of data sent from the host to the submitter node. To see how effectively we accomplished this, we measured the ac-

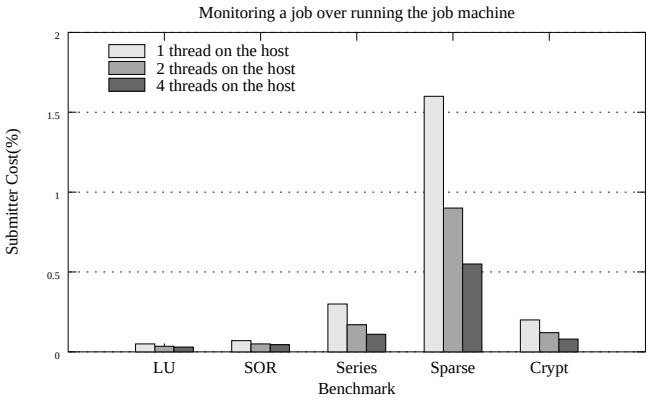


Figure 6: Submitter's computational cost of monitoring a remote job over running it locally

```
...
ByteArrayOutputStream baos =
        new ByteArrayOutputStream();
ObjectOutputStream objectos
        = new ObjectOutputStream(baos);

objectos.writeObject(serializable_object_to_send);
//serializable_object_to_send contains beacons

byte[] bytes = baos.toByteArray();
objectos.close();
baos.close();

size_sum += bytes.length;
//size_sum will provide network traffic information

...
```

Figure 7: Code to measure the network traffic caused by our approach

tual network traffic incurred by our beacon subsystem for each benchmark under different execution modes.

5.4.1 Method of Measuring Serialized Data Volume

The capability to serialize objects is one of the features of Java that simplified our implementation and increases its robustness. L-beacons and R-beacons are placed in serializable objects to make the submitter's understanding of the representation of data from the host easier. However, this mechanism transfers more data across network than just sending the raw data. Thus we need to measure the actual network traffic sent by our system. To do this, we serialize an object to be sent across network by the host into a ByteArrayOutputStream object. We can then measure the actual transfered size of a serializable object as shown in Figure 7.

5.4.2 Network Traffic Measurement Results

We measured the network traffic due to beacons for different benchmarks running with different numbers of threads. Table 1 shows that the total network traffic in bytes over the duration of each benchmark execution. Table 2 shows the highest average network traffic in unit time among our benchmarks occurs in Sparse, and is only 45.2 KByte/second. This is a consequence of our paced R-beacon transmission technique: the maximum number of R-beacon messages sent per unit time is the same as for the other benchmarks, yet the size of each of Sparse's R-beacon messages is larger than those of the other benchmarks.

	1 thread	2 threads	4 threads
LU	13.9MB / 663s	9.7MB / 399s	7MB / 311s
SOR	2.3MB / 132s	1.9MB / 91s	1.7MB / 81s
Series	600KB / 537s	340KB / 269s	180KB / 135s
Sparse	2.3MB / 138s	2.1MB / 73s	1.9MB / 43s
Crypt	52KB / 243s	29.7KB / 123s	16KB / 62s

Table 1: Network traffic during monitoring process over the execution time

	1 thread	2 threads	4 threads
LU	21.5KB /s	24.9KB /s	23.0KB /s
SOR	17.9KB /s	21.4KB /s	21.5KB /s
Series	1.1KB /s	1.3KB /s	1.3KB /s
Sparse	17.1KB /s	29.5KB /s	45.2KB /s
Crypt	0.2KB /s	0.2KB /s	0.3KB /s

Table 2: Average network bandwidth usage

5.5 Simulation of Cheating Node Detection

In this section, we evaluate the effectiveness of our approach in detecting cheating nodes, and isolating them from the system. The non-cheating nodes faithfully run the jobs sent to them, where as the cheating nodes may either refuse to run the jobs even if they are not busy, or abandon a running job without completion. For the purpose of our experiment, we simulated a pool of 1000 computing nodes. The peer-to-peer substrate Pastry [4] and the companion resource discovery as described in [3] are used to manage and discover idle nodes in the pool for job issuing.

To drive the simulation we created a job trace as follows. We selected 100 application executions uniformly randomly from the 15 different application executions shown in Figure 5. Next, we determined the mean execution time (T_μ) for the 100 selected jobs and created a random job issue sequence such that the inter-arrival time between two consecutive jobs has a uniform random distribution with a mean of T_μ. 500 of the 1000 nodes issued 100 jobs each using the created trace. We made 250 of these 500 nodes cheat. The presence of these nodes affects the time it takes for jobs issued by non-cheating nodes to complete.

Figure 8 shows the number of jobs issued by non-cheating nodes only that are issued but waiting in queue over time. Note that the total number of jobs issued by non-cheating nodes is 25,000. The topmost curve shows the scheme when no cheaters are caught. Here jobs from non-cheating nodes have to compete with jobs from cheating nodes, and hence it takes longer for them to complete. When a job is sent to a cheating node, it can either immediately refuse to run the job or abandon the job without completing it. In any case, the submitter detects that its job is not running and resubmits it to some other (probably non-cheating) node. The process is repeated till the job successfully completes.

The bottom curve in Figure 8 shows an ideal situation where all cheaters are known a priori. In this case, the non-cheating nodes send their jobs to other non-cheating nodes

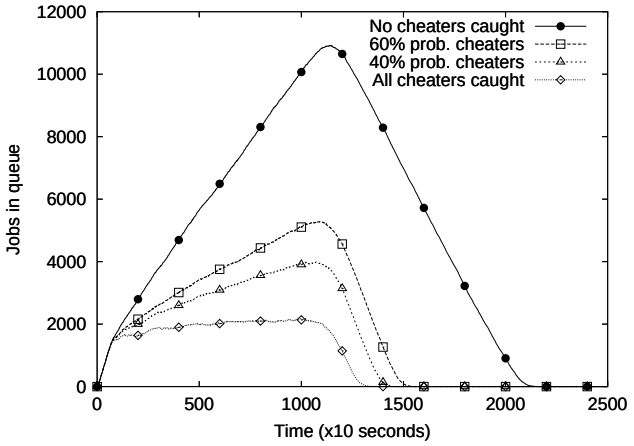


Figure 8: Number of jobs issued but not completed over time. The percentages represent the portion of cheaters that are not immediately caught.

only, and do not run any jobs submitted by a known cheating node. The number of jobs in queue submitted by non-cheating nodes increases little due to the issue sequence, and all jobs complete within 2000 seconds after the issuance of the the last job at 11730 seconds. The remaining two curves show more realistic cases where cheaters accept a job, but abandon it after processing it for a random time. We assume that within 2 seconds from a node abandoning a job, i.e., the next beacon interval, the cheating node is caught. We simulate two mixtures of cheating nodes: The percentage in the legend represents the percentage of cheating nodes that fail probabilistically over the length of the job; the remaining cheaters simply refuse to run jobs and are detected immediately. The curves show that our scheme is able to catch the cheaters, and the jobs for non-cheating nodes complete much faster than the case where cheaters were not caught at all. In this scenario both probabilistic cheater schemes were able to complete under 15560 seconds, compared to 21690 seconds for when no cheaters were caught. In contrast, the ideal case took 13730 seconds to complete all the issued jobs. The simulation shows that our scheme is effective in determining the cheating nodes, and isolating their effect from the overall system.

6. RELATED WORK

The compiler algorithm for reducing the overhead of monitoring in Section 4.2.1 is a form of program slicing, see [19, 21, 30] for a small sample of the work in this area. Most program slices are with respect to a variable v and program point p, and take a slice from p back to the beginning of the program. We differ in that our variable v is a part of an array (usually treated as atomic items in slicing, with the slice computing the entire array if any part of the array is computed) and in not taking the slice back to the beginning of the program, but rather maintaining a list of items (i.e upwardly exposed array elements) that must be acquired from outside the loop. The Array SSA [15] program representation will be necessary for doing this on less well structured programs than those in our benchmark suite. Lee and Hall [18] use a similar technique to extract slices of a program in order to do performance debugging on large applications. Their goal is different than ours, and assumes the computation host is trustworthy. Our technique of finding regular section descriptions of upwardly exposed arrays, or result arrays, borrows heavily on techniques developed for HPF compilers [12, 16], and in particular the work of Koelbel [17].

With the increasing popularity of volunteer-based cycle sharing, efficient protection against malicious machines has become an important research topic. Sarmenta discusses a spot checking mechanism to catch malicious machines (saboteurs) [26]. The central manager randomly assigns some computation, whose results are known to the central manager, to volunteer machines. By comparing the known results with the results sent by the volunteer machines, malicious volunteers can be caught efficiently. Du et al. [9] proposed a Merkle (Hash) tree based technique to detect cheating nodes when embarrassingly parallel computations are being performed. By verifying a subset of leaves in the Merkle tree, a central job manager can verify the correctness of results in the tree. Both of above techniques ensure the integrity of participant machines by checking a subset of *independent computations* completed by the participant machines. Over time, our approach monitors the correctness of all parts of an application. Moreover, our technique monitors the progress of the application, enabling partial payments or detection of errors before a long running application has finished.

Various remote debugging techniques have existed for years, see [5, 23]. Remote debugging techniques are used as a development facility to help developers to find bugs on a *trusted* remote platform during the program development phase. Our approach differs from these techniques in that our approach collects execution correctness information in an *untrusted* platform after the program has been developed and released, and in how the data is gathered.

Program monitoring is also employed in the Globus project for providing better quality of service [10]. This monitoring is either achieved indirectly by determining the resource utilization of the program, or by modifying the program to insert explicit calls to the Globus API. The motivation of our work is different in that we are using the monitoring to determine if we are getting a resource as promised.

7. CONCLUSION

We have described what we believe is the first solution to monitoring the progress and correctness of a remote job. Moreover, the overhead of performing this monitoring is shown to be insignificant– less than 3.5% on the host side. On the submitter side, the cost of monitoring a remote job is less than 1.6% of that of running the job locally. This technique, combined with our work, and the work of others, in resource discovery, sandboxed execution and automatic credit systems, opens the way for exploiting idle cycles across the Internet in a dynamic, decentralized and accountable fashion.

Acknowledgment

We thank Josep Torrellas for giving us access to his machines at UIUC to perform remote job submission and monitoring experiments. This work was supported in part by NSF CAREER award grant ACI-0238379 and NSF grants CCR-0313026 and CCR-0313033.

8. REFERENCES

[1] 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 *Proc. 19th ACM Symposium on Operating Systems Principles*, October 2003.

[2] 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: Special issue on Scalable Web Services and Architecture*, 63(10), October 2003.

[3] A. R. Butt, X. Fang, Y. C. Hu, and S. Midkiff. Java, Peer-to-Peer, and Accountability: Building Blocks for Distributed Cycle Sharing. In *Proceedings of the 3rd USENIX Virtual Machines Research and Technology Syposium (VM '04)*, May 2004.

[4] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting Network Proximity in Peer-to-Peer OverlayNetworks. Technical report, Technical Report MSR-TR-2002-82, 2002.

[5] D. Cheng and R. Hood. A Portable Debugger for Parallel and Distributed Programs. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing (SC'94)*, November 1994.

[6] M. J. Clement and M. J. Quinn. Analytical Performance Prediction on Multicomputers. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, November 1993.

[7] L. P. Cox and B. D. Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. In *Proc. 19th ACM Symposium on Operating Systems Principles*, October 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] W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable Grid Computing. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, March 2004.

[10] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proc. 8th International Workshop on Quality of Service*, June 2000.

[11] Genome@home. Genome at home. http://www.stanford.edu/group/pandegroup/genome/index.html (December 16, 2004).

[12] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo. An HPF Compiler for the IBM SP2. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*. ACM Press, December 1995.

[13] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. In *Proceedings of the 13th USENIX Security Symposium (Security '04)*, August 2004.

[14] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira. *Workload Management with LoadLeveler*. IBM International Technical Support Organization, 2001. http://www.ibm.com/redbooks (Dec. 17, 2004), publication number SG24-6038-00.

[15] K. Knobe and V. Sarkar. Array SSA Form and Its Use in Parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, January 1998.

[16] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1993.

[17] C. Koelbel and P. Mehrotra. Compiling Global Name-space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[18] Y.-J. Lee and M. Hall. A Code Isolator: Isolating Code Fragments from Large Programs. In *17th Workshop on Languages and Compilers for Parallel Computing (LCPC '04)*, September 2004.

[19] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. SUIF Explorer: an Interactive and Interprocedural Parallelizer. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, May 1999.

[20] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. 8th International Conference on Distributed Computing Systems (ICDCS 1988)*, June 1988.

[21] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving Program Slicing with Dynamic Points-to Data. In *Proceedings of the tenth ACM SIGSOFT Symposium on Foundations of software engineering*. ACM Press, November 2002.

[22] Nile. Scalable Solution for Distributed Processing of Independant Data. http://www.nile.cornell.edu/index.html (September 29, 2003).

[23] D. D. Redell. Experience with Topaz Teledebugging. In *Proceedings SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[24] R. L. Rivest. RFC 1321 –MD5 Message-Digest Algorithm, 1992.

[25] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[26] L. F. Sarmenta. Sabotage Tolerance Mechanism for Volunteer Computing Systems. In *CCGrid'01*, May 2001.

[27] SETI@home. Search for extraterrestrial intelligence at home. http://setiathome.ssl.berkeley.edu/index.html (December 16, 2004).

[28] L. A. Smith, J. M. Bull, and J. Obdrzalek. A Parallel Java Grande Benchmark Suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC2001)*, November 2001.

[29] K.-Y. Wang. Precise compile-time performance prediction for superscalar-based computers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, June 1994.

[30] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981.