Lightweight Monitoring of the Progress of Remotely Executing Computations

Shuo Yang, Ali R. Butt, Y. Charlie Hu and Samuel P. Midkiff
{yang22,butta,ychu,smidkiff}@purdue.edu

School of Electrical and Computer Engineering
Purdue University, West Lafayette IN 47907, USA

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 presents a novel and lightweight approach to monitoring the progress and correctness of a parallel computation on a remote, and potentially fraudulent, host system. We describe a monitoring system that uses a sequence of program counter values to monitor program progress, and compiler techniques that automatically generate the monitoring code. This approach improves on earlier work by omitting the need to duplicate computation, which both simplifies and reduces the overhead of monitoring. Our approach allows dynamic and accountable cycle-sharing across the Internet. Experimental results show that the overhead of our system is negligible and our monitoring approach is scalable.

1 Introduction

Computational workloads for academic groups, small businesses and consumers are characterized by long periods of little or no processing punctuated by periods of intense computational needs. It has been observed that computational resource demands can be "smoothed out" across sub-groups by aggregating large numbers of resources and users together. Computational resources across the world naturally experience different levels of demand at any given time because of their distribution. Computational resources are perishable, thus failing to use cycles, bandwidth and disk space does not create additional resources to be used in the future. However, if resources that would otherwise go unused could be provided to other users with the promise of sufficient compensation to cover the overhead of providing the resources, along with a small profit, then these resources would yield some value to the provider.

The major value of computational resources to their owner is the knowledge that they are available when needed. The major cost of sharing unneeded cycles is the legal and administrative overheads involved in allowing others access to the resources. Allowing compensation for these administrative overheads would dramatically increase the quantity, and decrease the cost, of available cycles. Both the decreased cost and the ease of accessing cycles increase the number of applications that can exploit these resources and increase the number of users that can access 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 consumers and providers of resources, allowing shared resources to blend seamlessly with locally owned resources.

Current cycle sharing systems take two approaches to minimizing these overheads. The first approach relies on volunteers providing cycles to a trusted job provider [1–3] with no desire for real compensation. These projects have allowed large computations, which would be infeasible on committed hardware, to be performed using surplus cycles on thousands of machines world-wide, and show the value of exploiting surplus cycles. By having volunteers provide the machines and absorb the local administrative overhead of enabling the application to run, by having a single trusted application, and by avoiding compensation issues, these projects avoid the difficulties that a more general approach must tackle. Although this model performs well in its targeted application domain, it clearly cannot be generalized to support cycle sharing for applications that do not inspire similar levels of generosity.

The second approach is typified by centrally managed systems like Condor [4] and LoadLeveler [5] that have been developed to allow resources to be aggregated within permanent or ad-hoc organizations. Centralized administration of resources allows a trusted entity – the system administrators – to verify and track the trustworthiness of users given access to the resources, *and* it allows users to deal with a known, trusted entity. Although this model has allowed organizations to share unused cycles internally, it does not work well for sharing cycles across organizations.

Four technical challenges must be overcome to allow the exploitation of the massive amounts of computational resources that are going unused. Solutions to the first three of these have been developed by other projects. The first challenge is how to discover resources to be used, and how to compensate the providers of the resources and punish cheaters [6–8]. The second challenge is how to enable a *submitter* machine to generate executables compatible with the *host* platform in a heterogeneous system [9, 10]. The third challenge is how to protect the *host* machine, i.e. the machine executing the job, from hostile binaries [11, 12].

In this paper, we discuss a solution to the fourth technical challenge, the problem of allowing the *submitter* machine, i.e. the machine submitting a job, to know its job is being faithfully executed. This capability is necessary for widespread cycle sharing, both to allow submitters of jobs to have confidence that their jobs are making progress towards completion, and to allow submitters of jobs to incrementally compensate the system hosting the job, thereby bounding the risk of the host. In this paper, we assume that host nodes may act *fraudulently*, but not maliciously. That is, a host node may take actions to gain compensation for which it is not entitled, but it will not take actions that harm the submitter but which do not benefit the host. We make the following contributions to monitoring the progress of remote jobs:

- We present a novel, lightweight technique to remotely monitor the incremental progress of a program that is executing in an untrusted environment. This technique has a lower overhead than any previous technique known to the authors.

- We present experimental data showing the overhead of this system is less than 2.1% on the remote system executing the program, and is negligible on the submitter system that is monitoring the application's progress.
- We present a monitoring technique that uses characteristics of the currently executing binary to generate and encode progress information, and is impervious to replay attacks.
- We show how the relatively uniform distribution of system and library calls can be used to guide the placement of the monitoring code, allowing simple compiler algorithms to be used to generate the monitoring code. We present experimental data that validates this approach.

The rest of the paper is organized as follows. Section 2 presents our novel *binary file location beacon (BLB)* approach to remote job monitoring for native binary executable programs. Section 3 presents the implementation of our general BLB approach to target MPI programs. Section 4 presents the experimental results showing the effectiveness of the system. Finally, Section 5 discusses the related work and Section 6 concludes the paper.

2 Monitoring Progress and Correctness with Beacons

In this section, we present a light-weight technique based on *binary file location beacons (BLB)* for monitoring the progress of remotely executing programs.

We assume a generic Internet cycle sharing system where each participating node can submit jobs (i.e. be a *submitter* node) or host jobs (i.e. be a *host* node). In this paper, we focus on remote job monitoring, with other components, such as resource location and credit management, being beyond the scope of this paper.

2.1 Key Idea

Before the *submitter* node submits a computational job to the *host* node for execution, it passes the program to our tool which transforms the original program into a pair of programs, one that executes on the host machine (*H-code*) and one 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 uses this information to track the progress, and verify the execution of, the program. Figure 1 shows an example of the runtime architecture of a remote job monitoring system.

The basic idea of this paper is to use place *location beacons (L-beacons)* along the control flow graph (CFG) of a program to track the fine-grained remote job progress information. However, a L-beacon based tracking mechanism is vulnerable to a replay attack. For example, Miller et al. [13] describes how to use existing tools to replace, on-the-fly, a process with another process. Thus, if a valid L-beacon value stream of a previous execution is captured, the attacker can replace the process of the later computation with a process that emits the captured L-beacon value sequence to cheat the submitter.

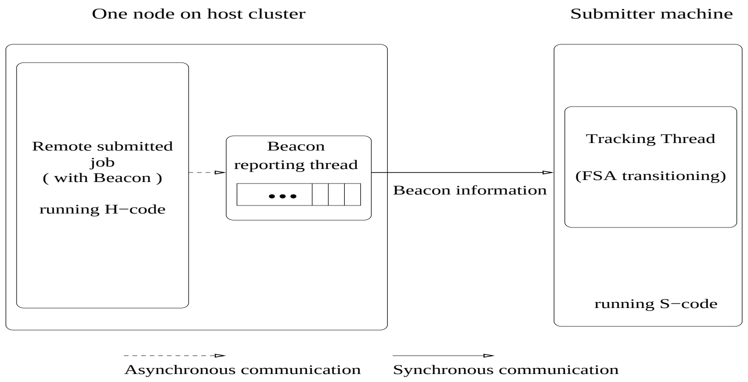


Fig. 1. The run time monitoring system for MPI programs: displaying only one tracking thread and the corresponding process it monitors

The contribution of this paper is on how to use the location beacons to monitor remote program execution. Specifically, the key idea of this paper is that by *making the value transmitted by location beacons reflect the structure of the program and by partially randomizing their placement, we can defeat replay attacks.* We call this technique *binary file location beacons*, or *BLB*-based remote monitoring for binary applications.

In our BLB-based remote job monitoring, the submitter constructs a finite state automaton (FSA) that tracks the progress of the remote job executing on the host machine. BLBs are placed along control flow graph (CFG) edges of a program and used to identify the current location of the program during its execution. At runtime, a BLB invokes a function f that stores the beacon value in a buffer, with the buffer sent to the submitter program at predetermined intervals. The value placed in the buffer is the value of the program counter at the BLB site, more precisely, it is the program counter of the instruction immediately following the call to f. By using the program counter value, the beacon value is intimately tied to the layout of the binary code generated for the program.

Our beacon insertion technique works as follows. The original program is scanned for candidate beacon insertion points, i.e., the entries to computationally significant regions. Computationally significant regions can be identified by the programmer or can be identified via an analytical cost model (see [14] for details). In this paper, we use a variant of the former approach, but instead of having the programmer explicitly identify candidate insertion points, we make use of the observation that system and library calls tend to be relatively uniformly distributed across programs, and tend to not appear within the inner loops of high performance programs. By using system and library call sites as candidate sites for placing BLBs, we do not need to use analytical models within the compiler to locate candidate sites.

2.2 Possible Attacks

Each BLB inserted will attempt to add a beacon value, which is a program counter (PC) value, into the BLB buffer to be sent to the host. An attack on our monitoring system needs to emit a stream of valid BLB values to be communicated to the submitter machine. There are two ways of doing this. The first is to capture a valid stream of beacon values from a previous correct execution of the program, and then replay this stream on future requests to execute the program. We prevent this attack by not always inserting beacons at the same locations when generating a program. At each potential BLB insertion site B, a beacon is actually inserted with probability P_B. If $P_B = 0$, no beacon is ever inserted at this site, if $P_B = 1$ a beacon is always inserted at this site. For $0 < P_B < 1$ a beacon may be inserted. By setting the values of P_B to be non-zero and less than one, each version of the program generated by our compiler will likely have a different set of beacons inserted and consequently a different set of valid beacon values. Because the values of P_B can be different at different candidate sites B the placement of beacons can be made more or less likely, depending on the hotness of a program region. In any case, attempts to replay the old beacon values will fail, with a high probability, because the replayed set of beacons will likely contain invalid beacon values. Because the binaries for programs used in high performance computing are usually orders of magnitude smaller than the data they operate on, shipping a (possibly) new binary with each execution imposes only a small overhead.

The second form of attack is for the host to analyze the binary and to extract the set of BLB call sites and the reachability information between BLB call sites necessary to construct the FSA. With this information a host can reconstruct the FSA and generate a valid sequence of BLB values. Two approaches can be used to prevent this attack. The first is to use code obfuscation to hide the control flow structure of the program, and consequently make it very difficult to determine the reachability information necessary to construct the FSA. A moderate use of jump tables to implement branches, and a moderate use of jump tables for function dispatch in code not on the critical path, should be sufficient to thwart program analysis tools. We note that simply compiling programs at high optimization levels performs a high degree of code obfuscation, and that is the technique we use now. Explicit code obfuscations techniques, such as the one described in [15] to enhance the difficulty of reverse-engineering, can be applied to our approach to further enhance the security of the system.

We note that attacks predicated on changing the binary must simultaneously preserve two structural properties of the program. First, the reachability of beacons from other beacons must be unchanged. Failing to do this will cause the host to run the risk that sequences of beacons not possible in the original program will be sent to the submitter. Second, the location of the code (explained in detail in the next section) that obtains the PC cannot be changed, since this will cause sequences of beacons sent to the submitter to contain values that are not possible in the original program.

Finally, we note that our goal is not to construct an unbreakable system, but rather to construct a system where the cost of breaking it is as high as the benefit.

3 Implementation Details

In the previous section we introduced a general BLB-based technique applicable to any binary executable program. In this section, we present a concrete implementation using BLBs in MPI message passing programs. We choose MPI programs because MPI is the most popular programming model for high performance computing. Moreover, MPI programs are able to work on more diversified platforms, including SMP and distributed memory systems, than any other programming model.

3.1 Program Counters of MPI Calls as BLB values

As described in Section 2, BLBs provide fine-grained location information about an executing program. The compiler in the monitoring system generates host code by inserting hard-coded beacon instructions at significant points in the program. Because beacon instructions that are inserted in the host code take time to execute and therefore add to the overhead of the program, the selection of locations to insert BLBs must account for the tradeoff between the granularity of monitoring and the program overhead. Thus locations chosen to insert beacon instructions should be: (i) where the overhead of executing the beacon instructions is affordable, and (ii) easily identified by a compiler as an efficient place to locate a binary location beacon.

In an MPI program, interprocess communication and synchronization are achieved by calling MPI library functions. Therefore, locations of interprocess communication and synchronization points, i.e., the MPI calls in the program, naturally satisfy the above two criteria because (i) the cost of a beacon instruction is insignificant compared to the interprocess communication or synchronization cost plus the cost of the computation performed since the last beacon, and (ii) a compiler front-end can trivially identify MPI calls.

The code in Figure 2 shows our implementation to obtain PC values to be used as BLB values. `GetPC()` is an instruction that obtains the PC value of the next instruction (the invocation of the `mpi_send` call) in a C program targeting an Intel IA32 processor running FreeBSD. Function `getPC()` returns the address that is placed on the stack frame when it is invoked, i.e. the PC of the instruction immediately after the invocation of `getPC()`. Adding the C expression `pc = getPC()` immediately before an MPI call returns the address where the MPI operation is invoked, i.e., the PC value at the MPI operation call site. We have implemented the same functionality for Fortran. The only difference between the C and Fortran implementations is how the value is returned by the respective `getPC()` functions because of the different function calling conventions.

For different machine architectures, a slightly different function needs to be provided. AMD64 family processors have the same calling convention and stack layout as the Intel IA32 architecture, and the above method to get the program counter value is valid. For architectures that allow more aggressive use of registers during code generation (e.g. the PowerPC architecture), slightly different code is generated because the return address from a function call is saved into a dedicated register instead of onto the stack. Therefore the `getPC()` function for these architectures returns the value held in the dedicated register instead of returning the value on the stack frame. For 64-bit Intel Itanium architecture, the cost of `getPC()` can be reduced by utilizing its “register stack frames” architecture, which enables `getPC()` to avoid accesses to the stack frames in main memory.

```
main(){
    ...
    mpi_send(...);
    ...
}
```

(a) *An MPI call in original code*

```
int pc;
main(){
    ...
    pc = getPC();
    mpi_send(...);
    ...
}
...
int getPC(){
    asm(”mov 4(%ebp), %eax”);
}
```

(b) *PC values returned by getPC() as BLB*

Fig. 2. Obtaining the program counter of an MPI call in a C program on IA32

3.2 FSA Constructed with Program Counters of MPI Calls

We now present the method used to construct a finite state automaton (FSA) to track legal sequences of BLB values. Each process in the host system executing the submitted program runs the same MPI executable, and their FSAs are identical.

The FSA construction algorithm is presented in [14]. This algorithm projects a complete program control flow graph onto a program control flow graph containing only nodes that are annotated with beacons. A node n_i in the new graph can reach a node n_j in the new graph if and only if n_i could reach n_j in the original graph. Using this algorithm, we construct an FSA where states in the FSA represent nodes in the new control flow graph. After compiling the code into a binary executable, we use a disassembler (`objdump()` in our case) to get the addresses of the MPI calls that are identified as beacon sites, map them onto the corresponding states in the FSA, and use the addresses as the state labels in the FSA. The address of the call to `mpi_init()` corresponds to the initial state of the FSA, and the address of the call to `mpi_finalize()` corresponds to the final state of the FSA. The transition symbol α driving a transition toward a specific state (also labeled α) is the address of the corresponding node in the binary executable.

Figure 3 shows an MPI program fragment, the corresponding H-code and the resulting FSA. In this example, we treat each MPI call as a beacon site (i.e. all $P_B = 1$). As shown in Figure 3(b), the compiler identifies each MPI call and inserts a call to `getPC()` immediately before the MPI call. The compiler also inserts a call to `deposit_beacon()`, which puts the BLB value into the beacon buffer, after each MPI call identified as a BLB. After the FSA is constructed, and after the transformed code is compiled, the BLB values are mapped onto the states and transition symbols in the FSA, as shown in in Figure 3(c).

3.3 Runtime System of Monitoring MPI Programs

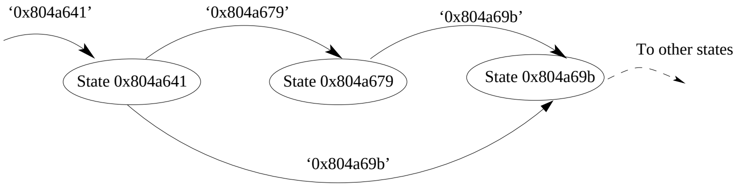
We now discuss the details of how the `deposit_beacon()` call places the BLB value generated by `getPC()` into a beacon buffer, and how the sequence of values placed into the buffer is transmitted to the submitter machine and used to monitor the progress

```
main(){
    ...
    mpi_irecv(...);
    ...
    if(...)
        mpi_send(...);
    ...
    mpi_wait();
    ...
}
```

(a) *A piece of pseudo code of MPI program*

```
main(){
    ...
    pc = getPC();
    mpi_irecv(...);// @0x804a641 in the executable
    deposit_beacon(pc);
    ...
    if(...){
        pc = getPC();
        mpi_send(...);// @0x804a679 in the executable
        deposit_beacon(pc);
    }
    ...
    pc = getPC();
    mpi_wait(); // @0x804a69b in the executable
    deposit_beacon(pc);
    ...
}
```

(b) *Generated host code*



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

Fig. 3. An example of program counter based FSA

of the program. Figure 1 shows a *tracking thread* on the submitter and the corresponding process that it monitors in the host cluster.

At the beginning of the computation on the host, the H-code performs an initialization procedure. Each MPI process allocates a beacon buffer where the BLB values generated by this process are inserted. A beacon buffer in our current implementation can hold up to 1500 beacon values. Each process also creates a separate *reporting thread*. The reporting thread on each process builds a TCP socket that connects to the monitoring S-code program running on the submitter. During the computation, the main computation thread on each process takes beacon values returned by invocations of the `getPC()` function and, via a call to `deposit_beacon()`, places the beacon value into the beacon buffer. This is shown in Figure 3(b). Periodically the reporting thread on each process sends the contents of its buffer to the submitter, and then clears the buffer to allow more beacon values to be deposited. The *pthread* mutex and condition variables are used to synchronize access to the beacon buffer by the main computation and the reporting thread.

The reporting thread on a process sends the values in the beacon buffer back to the submitter using a *paced* transmission scheme. The paced transmission scheme works as follows. The reporting thread sleeps for an interval, which is set by the submitter when the program is submitted to the host. When this interval passes, the reporting thread wakes up to send the values in the beacon buffer. If the buffer is filled before the interval expires, the reporting thread is woken up and immediately sends the buffer to the submitter node. When the reporting thread finishes sending the buffer, it sleeps for another interval. Thus, the cross-network data transfer procedure is asynchronous to the main computation of the program.

We now discuss the submitter machine actions. The submitter machine creates a dedicated thread (the *tracking thread*) for each MPI process executing on the host machine. Each tracking thread maintains an FSA, and the current state of the FSA is initialized to be the *initial* state, i.e. a state corresponding to an `mpi_init` call. Over time, the tracking thread receives buffers from its corresponding *reporting thread*, via the already established socket. Each beacon value in the buffer is processed by comparing it to states adjacent to the current state, which are found by performing a lookup in the FSA's transition table. If the beacon value does not match a valid transition from the current state, it is an illegal transition and the appropriate action is taken. Buffers continue to be received, and beacon values in the buffers continue to be processed, until the submitter receives the final state beacon value.

Finally we note that the monitoring runtime system can be configured with different setups. For example, the submitter can only build a connection to a single host process (e.g., the master node) and by receiving and tracking the single node's BLB values, the submitter can track the progress of the remote computation. Our system has sufficiently low overhead when tracking all host processes that we have not investigated this strategy further.

4 Experimental Results

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

4.1 Experimental Platform

Our experiments were run on a submitter/host pair located at the University of Illinois at Urbana-Champaign and Purdue University. The submitter machine, located at UIUC, is a uniprocessor with an Intel 3GHz Xeon processor, 512KB cache and 1GB main memory running the Linux 2.4.20 kernel. It is connected to the Internet through the campus network. The host machine is a cluster located at Purdue with 8 computational nodes, each of which has an Intel Pentium IV processor with 512KB cache, 512MB main memory, and runs FreeBSD 4.7. The nodes within the cluster are interconnected by a FastEthernet. The nodes in the cluster share a single file system, and the MPICH 1.2.5 library is installed on the cluster. Programs were hand-transformed using the approach described in Section 3.

4.2 NAS Parallel Benchmark Kernels

The NAS Parallel Benchmarks(NPB) [16] version 3.2 is a set of benchmarks developed to evaluate the performance of highly parallel computational resources. These benchmarks consist of five parallel kernels and three simulated applications. From these kernels and applications, we selected four kernels representing totally different types of computation and communication patterns to evaluate our approach.

- EP (an embarrassingly parallel kernel) represents computations without significant interprocessor communication. EP provides an estimate of the upper achievable limits for floating point performance.
- IS (a large integer sort kernel) performs a sorting operation that is important in particle method codes. IS tests both integer computation and communication performance.
- MG (a simplified multigrid kernel) performs the 3D V-cycle multigrid algorithm which solves the discrete Poisson problem with periodic boundary conditions. MG represents highly structured long distance communication and tests both short and long distance data communication.
- CG (a conjugate gradient kernel) performs the computation of the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. CG represents irregular long distance communication and unstructured matrix vector multiplication, which is typical of unstructured grid computations.

In our measurements, the inter-transmission intervals of the beacon reporting thread is set to 2 seconds, which represents a highly aggressive monitoring scenario. In an actual system, the inter-transmission interval would be tens of seconds or minutes. Also, we generate beacon information for each MPI call in the program, reflecting the case of $P_B = 1$ described in Section 2, which is the most expensive version of the H-code to monitor. Therefore, our experiment provides an upper bound on the performance overhead and network traffic incurred by using our monitoring system.

4.3 Run Time Computation Overhead

We first evaluate the scalability of our system by measuring the system performance overhead with computations running on different numbers of processors. To evaluate this, we run each of the above benchmarks with problem size-B inputs on 2, 4 and 8 processors of our cluster. We measure the time to run the original benchmarks on our cluster, which reflects the scenario of remote job execution without monitoring. These form our baseline numbers. We then run the manually transformed submitter code and host code of the same benchmarks on the submitter/host pair, which reflects the scenario of a remote job submission with monitoring. Figure 4 shows the overhead of job executions, using beacons for monitoring, over the corresponding un-monitored baseline job execution times. Our experimental results show that the maximum performance overhead is under 2.1%. We notice that the overhead does not monotonically increase with an increasing number of processors, and now explain why. Both the base line number (the computation time without monitoring) and the number of beacon calls under monitoring (the number of the MPI function calls per process at run time) decrease when the number of processes increases. There is, however, no explicit relationship between these two decreasing values. As well, our monitoring system introduces additional synchronization overhead by adding a single *reporting thread* to each process. But this

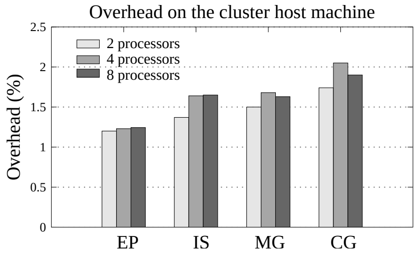


Fig. 4. Host Side Overhead with Different Number of Processors (Problem Size-B)

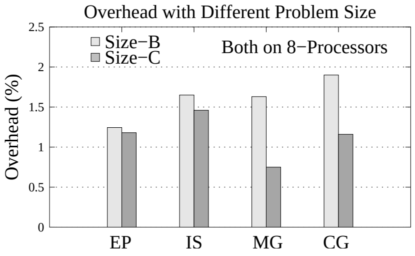


Fig. 5. Host Side Overhead with Different Problem Sizes

synchronization overhead is always one extra thread (the *reporting thread*) per process no matter how many processes the MPI code runs on.

Next we evaluate the relationship between the problem size of a monitored computation and the monitoring overhead. A problem with size-C input represents a larger problem size than the problem with size-B input. For example, for MG problem size-B uses a 256 by 256 by 256 matrix as the input data set, and problem size-C uses a 512 by 512 by 512 matrix as the input data set. Figure 5 shows that the overhead to monitor a larger computation (in this case, size-C) is always smaller than that to monitor a smaller computation (in this case, size-B). This is because the number of MPI calls in problem size-B and in problem size-C runs of each benchmark are similar. Therefore the cost of depositing BLBs into the buffer and transferring them across the network (the overhead on the host machine) for both problem sizes are similar. However, the total computation time for problem size-C is greater than that for problem size-B, thus the overhead is lower for problem size-C.

Finally, we evaluate the submitter node CPU usage to monitor a remote job. As the submitter code only performs FSA transitioning, it uses a small fraction of the CPU. We use the system `time` facility to measure the computational resources used by the verification process on the submitter. This is an imperfect evaluation because this ratio changes according to two factors: (1) the submitter's hardware, and (2) the submitter's workload while monitoring a remote job, which affects the resources available to perform the monitoring. We believe, however, that the numbers give a feel for the low overheads, and small amount of resources required to perform the monitoring. Table 1 shows the ratio of the sum of the user CPU time and the system CPU time to the wall clock time (elapsed real time) during the S-code execution.

As the results in Table 1 show, monitoring a computation with a larger problem size always takes a smaller percentage of the submitter's CPU resources than monitoring a computation with a smaller problem size. This is because the amount of beacon information processed by the submitter for problem size-B and problem size-C is same for each benchmark, while the monitoring time, i.e., the computation time on the host for problem size-C is significantly longer than that for problem size-B. These numbers were measured while one author logged into the submitter machine and launched two emacs processes, one *vi* process, and one *Mozilla* web browser process, which mimics a 'realistic' working scenario of a job submitter.

	Size-B	Size-C
EP	0.06%	0.02%
IS	0.07%	0.02%
MG	0.15%	0.03%
CG	0.17%	0.07%

Table 1. CPU Usage of Monitoring Computation with Different Problem Size Code on Submitter: 8-processors on the host in both cases

	Size-B (8 procs)	Size-C (8 procs)
EP	4.2 bytes/sec	1.0 bytes/sec
IS	101.5 bytes/sec	23.2 byte/sec
MG	21.2K bytes/sec	1.5K bytes/sec
CG	21.9K bytes/sec	7.9K bytes/sec

Table 2. Average network traffic (total traffic / execution time) in for different problem sizes

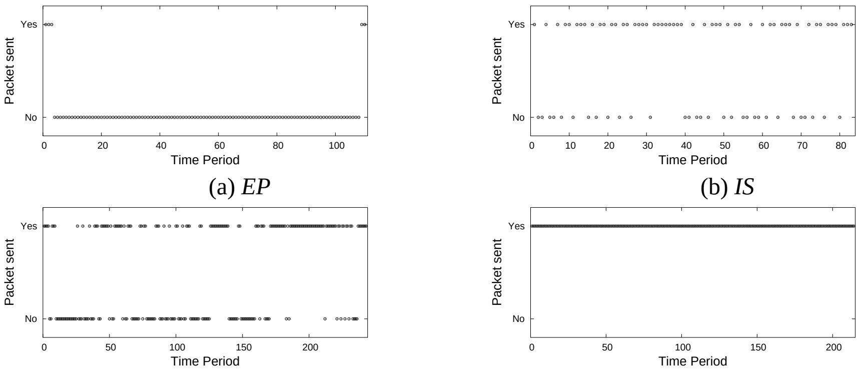
4.4 Network Bandwidth Overhead

Since network resources are finite, it is necessary to limit the amount of data sent from the host to the submitter node. We evaluate the network traffic generated by our system by measuring the number of bytes sent from the host machine to the submitter machine. In our experimental setup, each working process in the host cluster builds a TCP connection to the monitoring program (S-code, which creates tracking threads for all remote working processes) running on the submitter machine. In a real monitoring system setup, the submitter could choose to track a subset of the processes. Therefore, our experimental results reflect the upper bound of network traffic in a monitoring system.

In our experiment, we measured the average network traffic incurred by different benchmarks with problem size-B and problem size-C running on 8 processors. Table 2 shows the result of the above measurement, i.e. the total bytes of BLB values divided by the job execution time for each experiment. The results show that the larger the problem size , the smaller the average amount of network traffic the monitoring system incurs per unit time. The EP (embarrassingly parallel) kernel causes nearly zero traffic because this benchmark represents the type of computation without interprocessor communication. The BLB traffic is non-zero for EP because the benchmark uses several `mpi_reduce()` calls to get the computation result at the end of the benchmark. IS is the integer sorting benchmark and it uses a small amount of interprocessor communication to exchange the single elements at the boundaries of sub-arrays. MG and CG are typical numerical computations representing different communication patterns. These numbers show that the network traffic caused by our monitoring system is within the dial-up bandwidth.

4.5 Beacon Distribution Over Time

Our BLB based MPI program tracking approach leverages the observation that MPI calls are relatively uniformly distributed across most programs. This property enables the *incremental* progress tracking by the submitter. To verify this observation, we measure the number of beacon packets received by the submitter, i.e., the number of TCP send operations on the host machine, across the execution (monitoring) time. Figure 6 shows the number of packets received in a single *tracking thread*, which reflects the beacon temporal distribution over execution time during computations of problem size-C for a single computation process in the host cluster in our experiment. Each bin in Figure 6 represents a two seconds interval. Figure 6 shows that with the exception of



(a) *EP* (b) *IS*

(c) *MG* (d) *CG*

Fig. 6. Beacon packets sent by the host machine distributed during the execution (problem size-C) period: each bin representing a 2-second interval; 'yes' meaning there is BLB packet sent to receiver at that interval, 'no' meaning no BLB packet sent at that interval

EP, the beacon buffer packets sent by the host machine in our benchmarks are relatively uniformly distributed across the program execution time. EP is an embarrassingly parallel program and has no communication (because no data dependencies exist) during the main computation. For EP style programs, the submitter may choose to insert BLBs via an analytical cost model, as mentioned in Section 2.1 and described in [14], to get a relatively uniform BLB distribution. (We chose not to use that approach in this paper for the EP benchmark to keep our experimental conditions consistent.) We conclude that our approach of using MPI calls to place beacon calls gives good incremental progress information for most MPI programs.

5 Related Work

With the increasing popularity of grid systems and cycle sharing, efficient protection against malicious machines has become an important research topic. Sarmenta discusses a spot checking mechanism to catch malicious machines (saboteurs) [17]. The central manager randomly assigns some computations, 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. [18] 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 grant the correctness of all the 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. Our approach differs in that it monitors the integrity of all parts of an application execution. Moreover, our approach monitors the progress of the application and enables partial payments or detection of errors before a long running application finishes. We note that monitoring the progress of an execution is stricter than only checking that a remote machine has faithfully executed the program. Monitoring the progress of execution requires *incremental* confirmation of faithful execution. This is important for long running jobs so that the submitter machine does not have to wait for the job to finish to know the job's progress.

Hofmeyr et al. [19] uses sequences of system calls to detect intrusions. They built a profile of normal system call behavior for a process of interest, treating deviations from this profile as anomalies. Chen and Wagner [20] designed the MOPS system based on the formal model of a program and of a security property, which uses a finite state automaton to describe security rule of a process. Both of these techniques analyze system call sequences to achieve anomaly detection. Our approach differs from theirs in that the beacons in our monitoring system are not limited to system calls (e.g. the implementation example in this paper uses MPI function calls as beacons). Moreover, the purpose of our approach is to monitor the remote job progress instead of assuring the security of a local machine.

Our previous monitoring system [14] provides an approach to monitoring remote computations running Java bytecode. The submitter constructs an FSA to track the progress of the program, and it duplicates a portion of the computation (R-beacon) to prevent replay attacks. The BLB approach presented in this paper differs from it in that the BLB approach obviates the need for recomputation beacons (R-beacon), which are the main component of network traffic and computational burden on the submitter side incurred by the monitoring system. The BLB approach also makes the beacon location identification much easier for the compiler.

Program monitoring is also employed in the Globus project for providing better quality of service [21]. 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 receiving a resource as promised, and we do not need any special APIs in the host system, increasing the portability of our approach.

6 Conclusion

We have described a solution for monitoring the progress and correctness of a remote job. We show that the overhead of performing this monitoring is small. Although we describe our approach in the context of the MPI programming model, it is applicable to any binary. It is beneficial to both resource providers and resource consumers by limiting their risks. 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, ad-hoc fashion.

Acknowledgment

We thank Josep Torrellas for giving us access to his machines at UIUC to perform the remote job submission and monitoring experiments.

References

1. Genome@home: Genome at home. (`http://www.stanford.edu/group/pandegroup/genome/index.html(December16,2004)`)

2. SETI@home: Search for extraterrestrial intelligence at home. (http://setiathome.ssl.berkeley.edu/index.html(December16,2004))
3. David, A.P.: BOINC:A System for Public-Resource Computing and Storage. In: Proc. 5th IEEE/ACM International Workshop on Grid Computing. (2004)
4. Litzkow, M., Livny, M., Mutka, M.: Condor - A Hunter of Idle Workstations. In: Proc. 8th International Conference on Distributed Computing Systems (ICDCS 1988). (1988)
5. Kannan, S., Roberts, M., Mayes, P., Brelsford, D., Skovira, J.F.: Workload Management with LoadLeveler. IBM International Technical Support Organization (2001) http://www.ibm.com/redbooks (Dec. 17, 2004), publication number SG24-6038-00.
6. Butt, A.R., Fang, X., Hu, Y.C., Midkiff, S.: Java, Peer-to-Peer, and Accountability: Building Blocks for Distributed Cycle Sharing. In: Proc. of VM'04. (2004)
7. Castro, M., Druschel, P., Hu, Y.C., Rowstron, A.: Exploiting Network Proximity in Distributed Hash Tables. In: International Workshop on Future Directions in Distributed Computing. (2002)
8. Lo, V., Zappala, D., Zhou, D., Liu, Y., Zhao, S.: Cluster Computing on the Fly: P2P Scheduling of Idle Cycles in the Internet . In: Proc. of IPTPS'04. (2004)
9. Minchew, C.H., Tai, K.C.: Experience with Porting the Portable C Compiler. In: ACM 82: Proceedings of the ACM '82 conference, New York, NY, USA (1982)
10. PARISC-Linux: The PARISC-Linux Cross Compiler HOWTO. (http://www.baldric.uwo.ca/HOWTO/PARISC-Linux-XC-HOWTO.html(March16, 2005))
11. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proc. of SOSP'03. (2003)
12. Kamp, P.H., N.M.Watson, R.: Jails: Confining the Omnipotent Root. In: Proceedings of SANE 2000 Conference. (2000)
13. Miller, B.P., Christodorescu, M., Iverson, R., Kosar, T., Mirgorodskii, A., Popovici, F.: Playing Inside the Black Box: Using Dynamic Instrumentation to Create Security Holes. In: Proceedings of 2nd Los Alamos Computer Science Institute Symposium. (2001)
14. Yang, S., Butt, A.R., Hu, Y.C., Midkiff, S.P.: Trust but Verify: Monitoring Remotely Executing Programs for Progress and Correctness. In: Proc. of PPOPP'05. (2005)
15. Linn, C., Debray, S.: Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In: Proc. of CCS'03. (2003)
16. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., P.Frederickson, Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The NAS Parallel Benchmarks. Technical Report NAS Technical Report RNR-94-007, NASA Ames Center (1994)
17. Sarmenta, L.F.: Sabotage Tolerance Mechanism for Volunteer Computing Systems. In: CCGrid'01. (2001)
18. Du, W., Jia, J., Mangal, M., Murugesan, M.: Uncheatable Grid Computing. In: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04). (2004)
19. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security **6** (1998)
20. Chen, H., Wagner, D.: MOPS: an Infrastructure for Examining Security Properties of Software. In: Proc. of CCS' 02. (2002)
21. Foster, I., Roy, A., Sander, V.: A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In: Proc. 8th International Workshop on Quality of Service. (2000)