Using AOP to Monitor and Administer Software for Grid Computing Environments

Mark Grechanik, Dewayne E. Perry, and Don Batory

The Product-Line Architecture Research Group
University of Texas at Austin
Austin, Texas 78712

`{gmark, batory}@cs.utexas.edu, perry@ece.utexas.edu`

Abstract. *Monitoring* is a task of collecting measurements that reflect the state of a system. *Administration* is a collection of tasks for control and manipulation of computer systems. *Monitoring and Administering computer ResourceS (MARS)* in a distributed grid computing environment (i.e. a distributed environment for coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations) is an important, expensive, and critical task. We present a novel solution based on applying crosscuts using binary rewriters and an event-based model that allows developers to create non-trivial MARS programs easily and uniformly.

Our approach converts low-level API resource calls into system-wide events that MARS programs can monitor. This is accomplished by introducing advice that contains event-generating code at join points in programs that represent computer resources. We categorize low-level resource APIs by imposing a transactional metaphor to simplify the complexity of interactions between resources and to enable reasoning about MARS programs. We report both a case study and simulation that supports the viability of our approach.

1 Introduction

Modern business enterprises have hundreds or thousands of computers running different operating systems and applications that use various resources. The task of collecting measurements that reflect the state of a system is called *monitoring*. The task of *administration* is to use the results of monitoring to effectively control and manipulate these systems. The cost of manual monitoring and administration of enterprise-level computing systems is very high and is exceedingly difficult to scale due to the extensive laborious procedures that require frequent hands-on interventions by system administrators.

Computational grids are distributed environments for coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations [1]. To view computer hardware and software as resources creates significant challenges with their fine-grained control and allocation in a grid environment. Specifically, users attempt to gain access to different resources whose owners should be able to exercise a fine-grained control of such access while ensuring that the overall security and computational integrity of the system is not compromised. This problem is exacerbated in corporate environments where a slight breach in security may lead to disastrous consequences.

Existing grid solutions are typically based on user-level programs called agents running under minimum-security privileges. These agents can only accomplish parallel data processing tasks (e.g. solving partial differential equations or computing some numerical algorithms), and fall short of enabling fine-grained access to selected resources [2][3][4]. Moreover, many agents rely on a polling mechanism that wakes them up at predefined time intervals to run some tasks and then puts these agents back to sleep [2][3][4]. Polling agents often miss events that occur in the middle of a polling interval, and waste computational resources when awakened at times when their services are not needed.

Monitoring and Administering computer ResourceS (MARS) in a distributed grid computing environment is an important, expensive, and critical task. MARS programs should be easy to develop. Unfortunately, the opposite is true. MARS is complicated by the sheer multiplicity of computer resources and technologies. For example, Microsoft Windows offers more than a hundred software development libraries to program various resources like file storage and domain name systems. Most applications created using these libraries do not have programming interfaces for their monitoring and administration. Operating systems and computer resources are not developed for easy administration and monitoring. In order to administer different computer resources, MARS programs must access memory regions and execute commands that are protected or privileged in modern operating systems. For example, a process cannot access the region of memory occupied by some other process unless it uses an interprocess communication mechanism to accept commands and data in a predefined format. Otherwise, the space of each process is protected by the operating system and cannot be intruded on. Existing MARS solutions are ineffective since they either target the source code of applications and operating system kernels or rely upon using specific vendor-dependent library APIs to write MARS programs that target specific resources. Therefore a fundamental problem of MARS is how to dynamically administer and monitor computer resources in a grid computing environment both automatically and uniformly.

We introduce a novel approach that allows developers to write MARS programs uniformly. Our approach converts low-level API resource calls into system-wide events that MARS programs can monitor by registering their listeners with special services. This is accomplished by introducing advice that contains event-generating code at join points in programs that represent computer resources. Advice is applied by instrumenting low-level API calls to produce desired notifications. By imposing a transactional metaphor on MARS systems, we simplify the event delivery mechanism reducing tens of thousands of different events to only five event categories. We report both a case study and simulation that supports the validity of our approach.

2 The MARS Model

A computer resource changes its state after a client program executes some API that modifies values of some internal variables of this resource. *This is a fundamental property upon which any administrating and monitoring solution is based.* Suppose we have an observer who “lives” inside a CPU, “watches” internal variables of computer resources, and notifies us when their values change. If this observer can also modify the values of these variables on our behalf, then we can call him/her a *MARS observer* and *manipulator.*

The behavior of the MARS observer/manipulator can be explained using *aspect-oriented programming (AOP)* concepts. The observer can be viewed as a MARS aspect that is applied to computer resources. Different APIs that are located in different libraries and programs that manipulate the same resource represent a crosscut. A MARS aspect introduces a set of standard advice to resource crosscuts. For example, handling notifications about changes in the state of monitored resources is accomplished by applying `before` advice to APIs that manipulate these resources.

We categorize APIs that change the state of computer resources. Some APIs initialize or open a resource, some APIs perform read from or write to a resource, and others close resources. By creating such categories we enable the MARS observer to notify us that some resource has just been written to by some process rather than to produce a cryptic message stating that some API has been executed with a list of its parameters.

A high-level logical view of the MARS model is shown in Figure 1. At the top level a MARS observer and manipulator detects changes in states of computer resources as well as manipulate their behavior. This observer and manipulator accomplishes work using event and AOP models that are based on binary rewriting mechanisms. Binary rewriters are a part of low-level implementation of our MARS approach and are described in the next section.

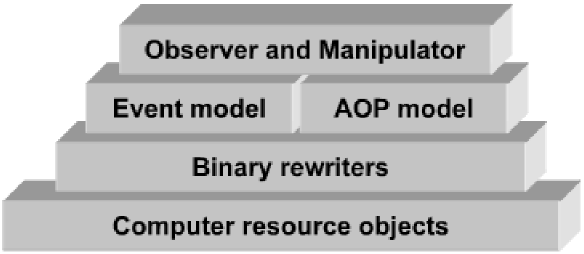

Figure 1: Logical view of the MARS model.

3 The Supporting Models

3.1 Event Model

An event model is a mechanism for delivering asynchronous data elements called *events* from *sources* to destinations called *sinks* as shown in Figure 2 [5]. A *source* is a program that generates an event and asks an event delivery mechanism either to deliver it to a sink or to put it into an event queue until some sink program requests it. The *sink* program invokes a callback function in response to the delivered events. Various architecture and object-oriented design patterns have been built around this event model (X-Windows, COM/DCOM, MS Windows). All are based on the assumption that programmers can modify the source code of sources and sinks in order to add events and their callbacks to the existing architecture.

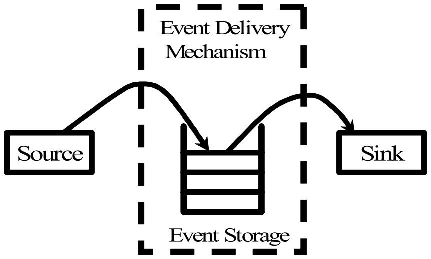

Figure 2: Event model.

We extend this event model to create a useful abstraction for MARS programs. A resource to be monitored is represented by the source and a MARS program is the sink. The event delivery mechanism is supplied by underlying MARS services. When a low-level API call is made by some application that accesses a resource then an event is generated and delivered to the MARS sink program. For example, when we need to monitor when a file is opened by some application, we can instrument a system service `fopen` to generate an event every time it is called. It is clear how to use event models when a programmer needs to generate and receive events using some event API. However, the problem is how to enable the generation of events without available program source code. We address this issue in the following sections.

3.2 Binary Rewriting Model

The majority of software resources in modern operating systems are implemented as shared libraries, dynamic-link libraries, and executable programs. Programs are linked to libraries and call their functions that in turn trap to the operating system when a system service call is made. We need to determine the joint points in the program at which we need to generate events or take some actions.

Join points are well-defined points in the execution flow of a program [6]. In our approach, join points serve as placeholders for MARS crosscuts that refine program functionality to enable monitoring and administrating tasks. Advice is inserted in the executable program code at join points using special tools described in Section 7.1.

`Before` advice is invoked when a function call is made but before the function code is executed. The typical purpose of this advice is to replace values of certain function parameters on the stack. For example, consider an application that calls the function `OpenFile` that opens the file “`myfile.txt`”. The name of the file is passed as a character string parameter to the function `OpenFile`. Suppose that every time this function is called with its file name parameter pointing to “`myfile.txt`” we want to change it to “`other.dat`” instead. This administration task is very common, and normally it requires changes in the application source code and therefore is laborious and difficult. AOP advice makes this conceptually easier to realize.

`After` advice is executed when a function call is executed but before the return instruction gives the control back to the caller. It could be used to notify a MARS program about completion of a task. Finally, `around` advice enables a call to a replacement function rather than the intended callee function.

3.3 Event Categorization

Advice communicates with MARS programs by sending events. Having each API method send a unique event to MARS programs is impractical since a computer has tens of thousands of different methods for which event objects/types would need to be defined. We solve this problem by imposing a transactional metaphor by viewing a computer as a database whose tables are resources we need to monitor. The properties of these resources are the attributes of tables in our abstraction. The APIs that manipulates resources become transactions that we execute on this resource database.

Consider Windows library APIs for fax service, *simple network management protocol (SNMP)*, and file I/O as shown in Figure 3. Each library contains various functions that manipulate some resources. The fax service library contains functions that allow users to write software that sends and receives faxes from computers connected to phone lines via modems. The SNMP library allows system administrators to configure remote devices, monitor network performance, audit network usage, and detect network faults or inappropriate access. Finally, file I/O is the most used library in Windows API since almost every program uses it to gain access to file systems.

Windows APIs contains over 13,500 calls [7]. When studied carefully, Windows APIs can be grouped into separate categories. We identify these groups as transaction types. The first group contains functions that open and initialize resources. For example, despite different names and signatures functions `FaxDevStartJob`, `SnmpStartup`, and `CreateFile` have the same semantics — they initialize and return a pointer or handler to a resource. The second group contains functions that perform operations on resources. The third group contains functions that commit or rollback transactions executed on resources. The fourth group contains functions that terminate the activity performed on resources and release handlers

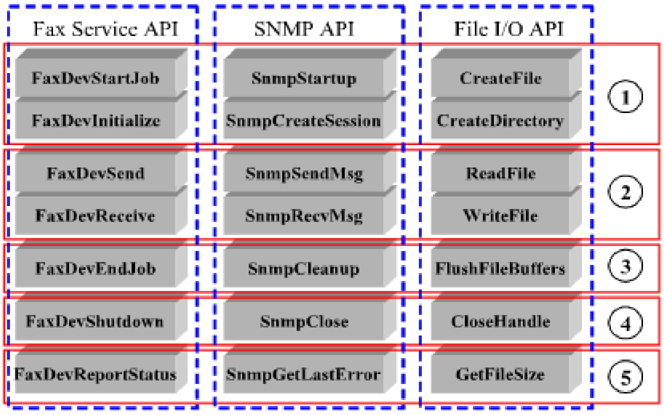

Figure 3: Categorization of fax service, simple network management protocol (SNMP), and file I/O Windows library APIs using our transactional metaphor.

that points to them. Finally, the fifth group contains functions that return status information on resources, for example, the size of a file or the error of the previously executed function. This grouping allows us to reduce the number of possible types of events from tens of thousands to only five. Each event instance contains fields that designate its category type, resource, and other resource specific information. An example of resource specific information is a message that an application attempts to write data into a smart card that is missing from its reader. The other resource specific information includes data specific to a given resource, for example, file attributes if the resource is a file.

4 A MARS Implementation

Unlike common AOP implementations based on compile-time or link-time weavings that apply aspects using compile-time weaving (e.g. AspectJ and AspectC++), MARS aspects are impractical or impossible to apply to source code of programs that represent computer resources. Source code for many commercial resources are not available to their users, or resources are required to run in reactive mode (e.g. 24x7) so that to stop a resource and recompile its code with MARS aspects applied cannot be done.

We implement MARS aspects via load-time and run-time weavings using binary rewriters that are tools used to change the structure of in-memory binary code representations. Interestingly, binary rewriters are used mainly in profilers and program optimizers. Using rewriters for instrumenting large software projects (e.g. [23]) is a relatively new field of study, and we extend its horizons in this paper.

4.1 Monitoring Resources

We break the task of monitoring computer resources into two subtasks: the instrumentation of API calls to insert event-generating code and the delivery of generated events to MARS programs. This approach solves MARS problems since the administration and monitoring tasks can be added as new features to existing functions that manipulate computer resources.

The implementation of the monitoring part of our solution consists of rewriting a binary application in three steps. First, we determine the APIs that must be monitored, their locations and signatures. Second, we build a library exporting advice which are overloaded functions with signatures that match the signatures of the APIs to be instrumented. Third, we instrument the programs by applying advice.

4.2 Administering Resources

4.2.1 The Problem

The task of administering computer resources is more complex than monitoring. When monitoring, event notifications flow from resources via the APIs that manipulate them to MARS programs. Administration tasks require changes to be made to operations and resources in order to achieve certain goals. In order to accomplish an administration task by creating and delivering administrative commands we need to enable a program that represents a resource to receive these commands, execute them, and desirably send the confirmation back to MARS programs. However, the majority of these programs are developed without special interfaces that enable their administration. They execute in the protected memory and cannot be easily tampered with. Thus, in order for resources to be administered we need to add special interfaces to programs that represent these resources that enable them to communicate with MARS programs and execute administration commands.

4.2.2 Connection and Agent Threads

The resource program should maintain a connection with a MARS program and respond to commands it receives. Since this functionality is not a part of processes that represent resources, we need to enable it. A kernel thread that is run as a child of a resource process and dedicated to establishing connections with MARS programs and receiving and processing administrative commands is called a *Connection Thread (CT)*. The other thread that is responsible for communications with instrumented event generation code is called an *Agent Thread (AT)*. These threads are not created as a result of code native to programs that represent administered resources and therefore, they must be injected into resource processes. There are several injection techniques [8][9] of which the main idea is to create a kernel thread executing some functions and attach it to a process by using binary rewriting mechanisms. This injected thread acts like an agent with respect to the process in which this thread is injected because this process is not aware of the presence of the thread. Often binary rewriters that inject CTs and ATs should have some control over the protected space of the process that is the subject of the thread injection. This control is necessary to write certain control structures in the process space that enable agent threads to act as native to the process. One way to do it is to enable a binary rewriter to act as a debugger to the resource process. In this role it can suspend the execution of the target process and write into its process space.

However, this approach requires every process to be started by a binary rewriter. Another way is to tap into operating systems services that govern the start and termination of processes. Commodity operating systems use special functions exported from a system library to instantiate any process. `CreateProcess` is an example of such a call in Windows. The algorithm of this call is rather complicated and described in detail in [10]. The important thing is that processes no matter how they are started, cannot bypass this call. By statically instrumenting `CreateProcess` we enable it to act as an injector of agent threads into the created process.

4.2.3 Solution

Figure 4 illustrates the administrative part of the MARS solution. A MARS program MP communicates via an interprocess connection C with the connection thread CT injected in process P that represents an administered resource. Both threads CT and AT execute a loop whose exit condition is triggered either by a command from MP or by terminating the process P. The CT receives commands from MP and AT receives events from native threads of P designated T_k. Recall that when we enable monitoring of resources we instrument certain APIs by embedding event-generating code. Rather than using an interprocess communication mechanism to deliver these events to MARS programs, the instrumented code sends events to the AT that executes within the same process P. This is indicated by the dashed arrow A→B. The cost of intraprocess communication among threads is cheaper than interprocess communication among threads. The event-generating function makes a blocking call to the AT and waits for instructions. The CT and subsequently the AT can be updated with these instructions on the fly. This is extremely important in an enterprise environment where software may run 24x7 and tasks may be updated hourly. The AT determines whether this call is monitoring in which case it returns the control to the calling thread immediately. Otherwise, if this is an API that requires an administrative action then the AT executes an appropriate function.

Suppose an administrator needs to propagate a task that can be described in English this way: "When program A opens file "`myfile.txt`" then it should be redirected to the file "`other.dat`" and security access privileges should be granted for the duration of the access". The administrator creates a command that specifies that if the first parameter of function `OpenFile` has value "`myfile.txt`" then it should be replaced with the value "`other.dat`". The other command instructs the AT to execute a function that grants administrative privileges at the beginning of `OpenFile` API. When a thread T_k of the process P calls the `OpenFile` function, it executes the event-generating code that sends an event describing this action to the agent thread. The AT invokes the function that replaces the parameter "`myfile.txt`" to the `OpenFile` function with "`other.dat`" value, and then executes a function grants administrative privileges to P. CT manages the table of functions that AT invokes. The thread T_k is suspended for the duration of A→E shown as a dashed arrow in Figure 4. When AT returns the control back to T_k via D→E then T_k finishes the execution of `OpenFile` with the replaced parameter.

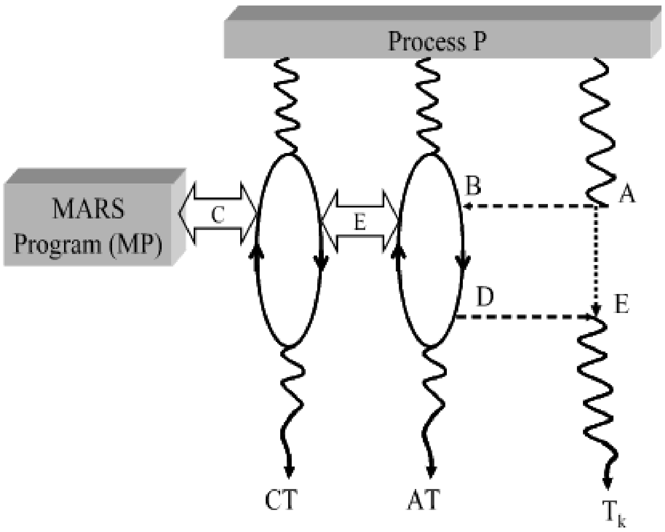

Figure 4: A schema of our MARS solution.

There are many ways to improve the performance of this MARS solution. MARS tasks can be stored in lookup tables managed by the CT. These tasks can be loaded into the table when a process is created and the CT is injected. These and other similar improvements are beyond the scope of this paper and are the subject of future work.

5 Case Study

Many grid tasks require that certain applications should execute and need to ensure that other grid programs do not "steal" resources from it. Consider a situation shown in Figure 5 when process A executes starting at time A_s^1 and finishing at time A_e^1. Process A should be given the highest priority, and the task of a grid administrator is to suspend other processes that try to run simultaneously with A. Suppose that the grid administrator is an agent that polls at times t_1 and t_2 to monitor the computer state. Between the t_1 and t_2 the process B starts at time B_s and finishes at time B_e. Thus, the pro-

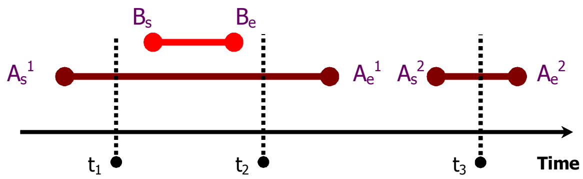

Figure 5: Process B executing concurrently with instances of the process A.

cess B is not detected by the grid agents, and it may interfere with the execution of the process A.

Suppose that process A terminated at time A_e^1 and its new instance started at time A_s^2. The polling agent detects this instance at time t_3, however, it is unable to tell whether it is a new instance. The process identifier may be reused by the *operating system (OS)* and assigned to the new instance of A. Existing grid solutions provide real-time detection of events associated with the asynchronous start and termination of programs either by involving OS kernel modifications that makes them difficult and impractical [11] or by using polling mechanisms that inherently misses important events [2][3][4]. *Our solution avoids these problems*.

We implemented a system that serves as a proof of concept. We used the Detours library [12] that is a dynamic code splicing tool developed for x86 platform by a Microsoft research group, to instrument programs and solve the problem outlined above in a real-world enterprise environment described below.

Consider a *semiconductor fabrication facility* (*fab*) that has a number of tools used in manufacturing microprocessors based on silicon wafers. In general, one or more tools are controlled by programs running on a general-purpose computer. These programs receive real-time data, analyze it, and make decisions that result in sending control signals to the tools. If for some reason a system misbehaves or some rogue program interferes with some important application that is processing real-time data, then the fab stops resulting in the loss of millions of dollars. While it is important to share computational resources in enterprise environments, it is essential to have software that monitors and controls computers in this and the other similar situations to execute a special procedure automatically to stop rogue applications from interfering with critical programs.

One of the authors (Grechanik) applied our approach to a real-time component-based semiconductor overlay analysis and control system. The Archer Analyzer is a software package geared for Archer 10 optical overlay metrology systems manufactured by the California-based KLA-Tencor Corporation [13][14]. It operates in a grid computing environment where resource sharing may lead to significant problems that require immediate solutions in dynamic and complex organizations such as semiconductor facilities.

The purpose of optical overlay measurements is to detect and fix misalignments between layers of semiconductor chips that were put on a silicon wafer using microlithography processes. Overlay or misregistration is a vector quantity defined at every point on the wafer. Ideally, the value of overlay should be zero. When nonzero overlay is detected the tool is stopped and the error is corrected as soon as possible.

We applied our MARS solution on Windows NT by instrumenting `CreateProcess` to enable it to act as an injector of agent threads into the created process. Once we detected that a new process interferes with resources used by the Archer Analyzer, we suspended the meddling process and resumed it again after the system became idle.

It was interesting to observe the reaction of company's management to our solution. As soon as they realized that we instrumented OS services, they demanded that solutions be removed from the computer. They perceived the modification of a fundamental layer (i.e. OS) upon which other software runs as a threat to the safety of the general system. Clearly, it will take some time until instrumentation of low-level services will be accepted by general software practitioners.

6 Performance Study

For the performance study we implemented a simple system based on the Detours library. When process P opens a file `myfile.txt` then the event generating function with which we instrumented Windows file I/O APIs produces event notifications that are delivered to a MARS program.

6.1 Experimental Setup

Efficient implementation of event storage is important for the overall performance of MARS. Since allocation and destruction of event structures in memory is expensive, we implement an event pool that is allocated at MARS initialization stage. The size of the event pool is fixed. When the administration thread receives an event notification from one of program's threads it locates an unused event object in the pool. Each event structure has a bit flag that is set when a structure is filled with event information and cleared when the event is delivered to the MARS program (MP). Delivering events to MPs is done via the interthread connectors and a semaphore that is set when a new event is inserted in the pool. The semaphore wakes up a delivery mechanism thread in MP that reads the event and clears the bit flag.

Our experiments consist of simulating different event generation rates and event storage pool sizes. We varied the event generation rate and measured CPU utilization (also called CPU load) by event generation and delivery mechanisms, and the average waiting time that events spend in storage plus the time they wait to be put in the storage until they are picked up by the destination process.

The main purpose of our experiments is to show that within reasonable limits of event generation rates, the CPU load is small enough, and it does not affect the overall performance of the system. We deliberately ignore user-defined load (e.g. administration tasks) that may be associated with events since it is the prerogative of an administrator to design and run such tasks. It is unlikely that a MARS user associates a time-consuming administrative task with a frequently called API. Often, it makes little sense to produce event notifications when some API is invoked frequently. For example, being notified about the change of color of every pixel carries little practical information and creates a significant load on a CPU. Our experiments provide guidance for MARS users as to what event generation rates and event pool sizes are acceptable to achieve good overall performance.

We carried out our experiments using MS Windows 2000 that ran on Intel Pentium III 850MHz CPU and 768MB of RAM. We instrumented our event simulator with performance monitoring (PerfMon) API [25] that is distributed with Windows platform software development kit. PerfMon API provides programming access to various counters that enable monitoring the use of CPU and memory by any application.

6.2 Detour Performance Characteristics

Detours library [12] is a dynamic code splicing tool developed for x86 platform by a Microsoft research group. Since we use it extensively in this paper we report its performance characteristics. Interception times are measured on our experimental platform as defined in Section 6.1. The average time to invoke different empty functions without interception is 0.043μs, and with interception using the Detours library it is 0.057μs. The overhead of the Detours library is small and within the range of 200ns. Common interception mechanisms, like breakpoint trapping, have surprisingly larger overheads (at 218μs) [12]. Thus the overhead of the Detours library is comparatively small.

6.3 Results

Our first metric of performance of MARS is the CPU utilization caused by the event generation code and our event delivery mechanism. The graph of CPU utilization is shown in Figure 6. The CPU load grows linearly with the event generation rate. We noticed that the overall performance starts degrading when the CPU utilization by the event simulator exceeds 10% that corresponds to the event generation rate above 600 events/sec. If we draw an analogy between events and requests to web servers, then the rate of 600 events/sec corresponds to over 50,000,000 requests to a web server per day. Since this rate is excessive for the sheer majority of MARS tasks, we can conclude that our system behaves reasonably well under standard loads. Of course, as soon as a load is associated with event delivery, this rate will drop. The point of the experiment is to show the efficiency of underlying event delivery framework**.**

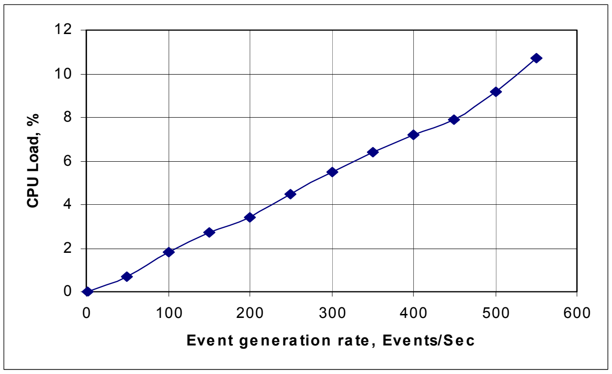

Figure 6: A graph of CPU utilization dependent on the event generation rate for event pool size equal to 1,000 events.

A graph showing the dependency of an average waiting time for a generated event to be put in the container pool from the event generation rate for different preallocated container pool sizes is shown in Figure 7. The graph shows that for sufficiently large pool size, an average waiting time is small, however, with the increase of the event generation rate the waiting time grows nonlinearly. We conclude that it is better to have a large container pool for the worst event generation rate case to avoid a significant increase of the waiting time.

The shape of graphs shown in Figure 7 can be explained using analytical results and fundamental laws of queueing theory [16]. In the best possible case the arrival rate of events is less or equal to the event processing rate of the MP, and the average waiting time for events is close to zero as it is shown in Figure 7 with the graphs being flat until the event arrival rate exceeds the event processing rate of the MP. The larger the event pool the longer the flat region of the graph. Then a backlog of unprocessed events grows continually as the event generator keeps producing events. Late events experience larger response times. As the number of events increases, more events are waiting increasingly long times. Thus, for any pessimistic bound on the MP response time it is possible

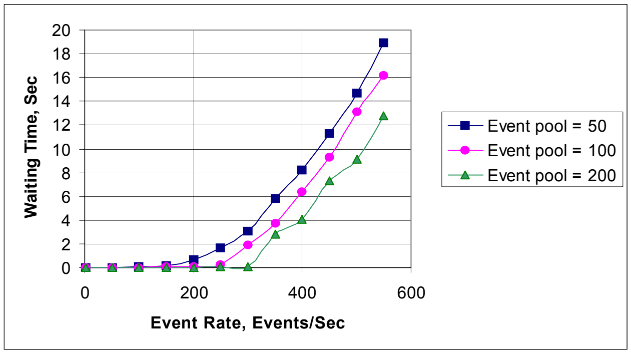

Figure 7: A graph of waiting times dependent on event generation rates for different event pool sizes.

to pick an event generation rate sufficiently large that the bound is exceeded.

The waiting time of MP (a response time of a service terminal) with respect to the increasing number of events is expressed with the following formula $R = ((N-1)D_{max})/(1+(Z/(N \cdot D)))$,

where R is the waiting time, N is the number of events, D is the total time to process all events, D_{max} is the largest time it takes to process an event, and Z is the average time required to process an event. This formula shows that the waiting time increases linearly with the number of events waiting for service. So the question that we ask is whether graphs showing the increasing response time in Figure 7 are linear after the breaking point is reached between their flat and growing parts. This breaking point symbolizes that the event generation rate sufficiently exceeded the event processing rate of the MP.

To show that the waiting time increases in a linear fashion we estimate the correlation between samples of the graph by computing the Pearson product moment correlation coefficient [17] as

$$r = \frac{n\Sigma(XY) - (\Sigma X)(\Sigma Y)}{\sqrt{n\Sigma Y^2 - (\Sigma Y)^2}\sqrt{n\Sigma X^2 - (\Sigma X)^2}}$$

Let variables X and Y stand for the event generation rates and the event waiting times respectively. The values of r for the data shown in Figure 7 are 0.93, 0.97, and 0.96 for event pool sizes of 50, 100, and 200 respectively. These values of r suggest a strong tendency the average waiting time to increase linearly as the event generation rate increases, and it serves as a good indicator that we are in the agreement with the queuing theory that governs the behavior of our system.

7 Related Work

There are two categories of related work. The first includes different tools and techniques that enable instrumentation of binary code. The other describes existing monitoring and administering solutions, their benefits and limitations.

7.1 Machine Code Splicing Solutions

There are two types of machine code splicing: static and dynamic. *Static splicing* is a technique for rewriting machine code with the subsequent storing it on some persistent storage. In contrast, dynamic splicing enables rewriting the machine code when it is loaded in the memory for executing within a process. When static splicing is applied to a program or a library then its image is overwritten and the refined code is stored on a hard drive. From this moment on this refined program code is loaded in memory to execute. Dynamic splicing requires a special program to load the program to be refined in memory with the purpose of gaining read and write access to its process image. Then the loading program applies splicing to the loaded process and allows it to run. This operation should be performed every time when a desired program is to be run.

Both approaches have been implemented and tested on a variety of platform. Etch [18] is a static and Detours [12] is a dynamic code splicing tool developed for x86 platform. Dyninst [19] provides a C++ class library for dynamic code splicing that covers a range of platforms such as IRIX (MIPS), AIX (Power), Solaris (Sparc), Windows NT (x86), and Linux (x86). EEL (Executable Editing Library) [20] is also a C++ library that hides the complexity and platform-dependent detail of editing executables. EEL provides abstractions that allow a code splicing tool to analyze and modify executable programs without being concerned with particular instruction sets, executable file formats, or consequences of deleting existing code and adding refinement feature code. EEL simplifies the construction of program measurement, protection, translation, and debugging tools. EEL also can edit fully-linked executables, not just object files, and it is portable across a wide range of systems. ATOM [21] is a single framework for static and dynamic code splicing that enables building a wide range of customized program analysis tools. It provides a powerful interface for navigating through the code of an existing application and dropping instrumentation code at join points. FX!32 developed by Digital [22] combines an emulator and translator that takes x86 code and dynamically convert it into Alpha-based instructions. Mediators [23] is a technology for instrumenting all shared library calls, monitor their behavior, integrate legacy components together, or encapsulate potentially harmful or unreliable components. They can be dynamically installed and removed during execution or installed before execution begins. Mediators are based on the Detours library.

It seems that the static code splicing approach can do everything that the dynamic approach does and more since it needs to be applied only once to splice the target code. However, the combination of static and dynamic approaches is preferable for our solution. Since static code splicing can be applied only when a program is not executing then this approach is not good for long-running processes because it requires processes to stop, apply a splicer, and restart processes. The other drawback of this approach is its legality. Many commercial software packages are sold with licenses that govern their use. A standard clause in such licenses states that no programs in these packages can be modified for any purpose. However, this clause does not apply when programs are executing in memory. Thus, this plays significant role when MARS is applied to commercially licensed software. We also consider dynamic code splicing of commodity operating system kernels. Recent paper on this topic [24] proved that dynamic code splicing of commodity operating system kernels is possible with an instruction-level precision.

7.2 Monitoring and Administering Solutions

Existing MARS solutions can be roughly divided into four groups. The first includes the software that provides remote access to the managed computers. PC Anywhere and Citrix terminal server [26][27] are examples of these approach. This solution is not scalable as it only removes the need for an administrator to be physically present at a computer. It is network intensive since it is based on screen pixel transfer between computers.

The second group includes specialized or modified OS kernels of distributed operating systems that enable administration of distributed computers with automation of some tasks. An example of this approach is the TACOMA OS [28] that implements several distributed management policies. The drawbacks of this approach are performance penalty resulting from a "heavy" kernel and impracticality of modifying existing operating systems to incorporate this strategy.

Another approach is to run an agent at the managed computer that collects information and may control some resources, but has limited capabilities to affect operating system settings and other running applications. The problem with this approach is that the agent can be killed leaving the computer unmanageable. In addition, polling agents are created using platform-dependent API, and they cannot penetrate interprocess memory to administer arbitrary applications. Monitoring and administrative agents work in polling mode, sleeping for some time and waking up to collect information and execute some administration tasks. The problem with this approach is that polling agents are often invoked when their services are not needed, and they consume computer resources to gather information about their behavior without producing any useful actions.

Finally, the trace collection approach is based on parsing text data that applications write in their log files. It also uses OS-dependent API, for example, performance monitoring API on Windows 2000 or SNMP traces in order to extract semantically relevant information that is of interest to users. This approach is extremely laborious and limited in scope, however, it is the simplest to implement considering the alternatives. BMC is one of the major MARS product companies has two solutions called GuardianAngel and SiteAngel that are based on the trace collection approach. IBM's *Tivoli Enterprise Console (TEC)* is another example of commercial monitoring and administering software that requires each controlled application to incorporate in its source code special API designed by Tivoli engineers that sends monitoring messages and accepts control requests from MARS programs [2]. HP AdminCenter [29] explains the cause of various failures in systems. While the AdminCenter uses a rule based system to show dependencies among different resources, TEC requires the monitored program source code to be modified to include diagnostic messages that have predefined format. Dolphin gathers information via SNMP or RPC. The information is stored in a proprietary internal format that can be accessed through the provided GUI.

Other commercial companies addressed this problem but with little success. For example, Microsoft's Zero Administration Kit [30] was dependent on Windows NT for clients and servers. The major part of this kit was a system policy editor with some templates. Other commercial implementations, for example, Network Computer Viewpoint Administrator by Boundless Technologies [3], which one of the authors (Grechanik) of this paper developed in 1998 is complex and requires operating system drivers while providing limited functionality to administrators.

Extensive analysis of system administration tasks such as *monitoring, diagnosing, and repairing* (*MDR*) was done in [31][32]. The proposed MDR system used information gathered and stored from enterprise distributed system with the purpose of statistical analysis. The statistics in the MDR system have to be analyzed to determine expected values and dispersions. If a problem, for example, a device failure or a CPU overload happens then administrators, users, or managers can be notified of the problem. Some problems can be automatically fixed, and for other problems the administrator can specify repairs. Administrators and users are enabled to visualize the statistics and information.

A number of systems [33][34][35][36][37][4][38][39][40] concentrate of collecting monitoring information using polling agent approach and then calculate statistical parameters. Some have very complex subsystems for monitoring computer resources using polling agent and harvesting measurement data. In most cases they differ on whether the gathering happens from a single node, or happens on remote nodes and is sent to a single node. None of these systems address the issue of writing monitoring and administration software. In fact, most do not provide any administration capabilities. Very few of them provide any form of notification more advanced then simple screen eyeballing.

8 Discussion and Future Work

An accepted paradigm in the design of MARS software is based on using conventional object-oriented programming techniques that conflicts with the underlying mechanisms of resource monitoring and administration. These mechanisms are based on viewing resources as programming objects without strict physical boundaries that exist outside the scope of MARS software and their methods are spread across different libraries. All attempts to apply conventional techniques led to ineffective MARS programs that were complex to write and hard to maintain.

MARS research is interdisciplinary. Our approach to build MARS programs is synthesized from a variety of techniques and ideas developed in operating systems, software engineering, and programming language research. It is noteworthy that binary rewriters that constitute the basis for our solution are not widely accepted in software engineering due to a common belief that it is not easy to integrate them in software development processes. Our research shows that not only binary rewriters can be easily integrated in software development and also it is difficult to solve the MARS problem if they did not exist.

We believe that our approach has potential. Not only can it be used for creation of MARS programs but also for application integration and collaborative computing. Its key advantage is that all these uses involve minimal development efforts. Architects will not disrupt their organizations by recoding existing applications in order to add new MARS functionality. It offers, for example, an attractive alternative to the way computer resources are currently administered and monitored, and it abolishes the need for any programming changes to them.

Of course, there are limitations. It is not clear how our ideas apply to real-time systems. If an application has intensive graphic front end (e.g., a game), then our approach may not be able to offer the best performance when critical resource functions are monitored and administered. Further, the libraries containing functions that constitute some resources change over time. These changes can impact a MARS application created with our technology, requiring changes to be propagated to MARS programs. For operating system modifications of system services tend to be rare, whereas for custom libraries, changes occur more often.

9 Conclusions

The administration and monitoring of computer resources especially when their source code is not available is both a difficult and fundamental problem of MARS. We have shown that a viable solution is interdisciplinary and lies in refining executable code that represent computer resources. We use the principles of binary rewriting in order to refine functions that constitute the resource interfaces. The proposed MARS aspect-oriented approach hides the complexity of the low-level code instrumentation and presents interfaces that allow programmers to write MARS programs uniformly and with minimal complexity. Our solution reduces the significant complexity associated with development of MARS software by enabling a simple and powerful event model for the monitoring task. We enable programmers to operate on resources as if they were first-class objects thereby presenting a uniform way to write MARS programs. By imposing a transactional metaphor on MARS systems we simplified the event delivery mechanism reducing tens of thousands of different events to only five event classes. We showed that our approach can solve monitoring and administration problems without incurring the complexity of existing monitoring and administering technologies. We applied our MARS implementation to a nontrivial commercial system operating in a grid computing environment where resource sharing may lead to significant problems and it demonstrated the viability of our approach and successfully tested its critical functionality.

Acknowledgments. We warmly thank James C. Browne and Michael D. Dahlin for reading this paper and providing useful comments and suggestions.

10 References

[1] I.Foster and C.Kesselman, The Grid: Blueprint for a New Computing Infrastructure, 2nd edition, Morgan-Kaufmann, 2004.

[2] Private conversations with Tivoli engineers.

[3] Boundless Technologies, "Viewpoint Administrator". http://www.internetwk.com/story/INW19990901S0011.

[4] BMC Software Corp. Guardian Angel. http://www.bmc.com/products/proddocview/0,2832,19052_19444_29401_9118,00.html

[5] D. Luckham, "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems". Addison-Wesley, 2002.

[6] G. Kiczales, J. Lamping, A. Menhdekar, C. Maeda, C. Lopez, J. Loingties, and J. Irwin, "Aspect-Oriented Programming". In M. Aksit and S. Matsuoka, editors, ECOOP, vol. 1241 of Lecture Notes in Computer Science, Springer, 1997.

[7] D. Spinellis, "A critique of the Windows application programming interface". Computer Standards and Interfaces, 20:1-8, 1998.

[8] M. Grechanik, D. Batory, and D. Perry, "Integrating and Reusing GUI-Driven Applications". International Conference on Software Reuse, Austin, Texas, Apr 2002.

[9] Matt Pietrek. "Learn System-level Win32 Coding Techniques By Writing an API Spy Program". Microsoft Systems Journal, vol. 9, no. 12, 1994, pp. 17-44.

[10]D. Solomon and M. Russinovich, Inside Microsoft Windows 2000, Microsoft Press, 2000.

[11]Sun Microsystems, N1 Grid: n Computers Operating as 1. http://wwws.sun.com/software/solutions/n1

[12]G. Hunt, "Detours: Binary Interception of Win32 Functions". Proc. 3rd USENIX Windows NT Symposium, Seattle, WA, July 1999.

[13]Semiconductor Business News, http://www.siliconstrategies.com/story/OEG20020708S0052.

[14]KLA-Tencor, "Archer Analyzer Automated, Real-Time Overlay Metrology Analysis," Technical Fact Sheet,

[15]http://www.kla-tencor.com/products/archer10/archer_analyzer_tech_factsheet.html

[16]E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik, Quantitative System Performance. Prentice Hall, 1984.

[17]G. Box, W. Hunter, and J. Hunter, "Statistics For Expirementers". John Wiley and Sons, 1978.

[18]T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad, "Instrumentation and Optimization of Win32/Intel Executables Using Etch". USENIX Windows NT Workshop, Seattle, WA, Aug 11-13, 1997.

[19]B. Buck and J. Hollingsworth, "An API for Runtime Code Patching". International Journal of High Performance Computing Applications, 2000.

[20]J. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing". SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1995.

[21]A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools". Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, 196-205, June 1994.

[22]A. Chernoff and R. Hookway, "DIGITAL FX!32 - Running 32-Bit x86 Applications on Alpha NT". USENIX Windows NT Workshop, Seattle, WA, August, 1997.

[23]R. Balzer and N. Goldman, "Mediating Connectors". Proc. 19th IEEE International Conference on Distributed Computing Systems Workshop, 73-77, Austin, TX, June 1999.

[24]A. Tamches and B. Miller, "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels". Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, New Orleans, LA, February 1999.

[25]S. Pratschner, "Instrumenting Windows NT Applications with Performance Monitor". http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnperfmo/html/perfmon.asp

[26]S. Kaplan, M. Mangus, "Citrix Metaframe for Windows Terminal Services: The Official Guide". McGraw Hill, 2000.

[27]Symantec Corp. PC Anywhere. http://www.symantec.com/pcanywhere/

[28]R. van Renesse and F. Schneider, "An introduction to the Tacoma distributed system, version 1.0". Technical Report 95-23, University of Tromso, Norway, June 1995.

[29]HP AdminCenter. http://www.networkcomputing.com/613/613f1b.html.

[30]C. Zacker, Zero Administration for Windows. O'Reilly, 1999.

[31]E. Anderson and D. Patterson, "Extensible, Scalable Monitoring for Clusters of Computers". Proceedings of 11th Systems Administration Conference, 1997.

[32]E. Anderson, "System Administration: Monitoring, Diagnosing, and Repairing". Ph.D. Qualifying Proposal, April 1997.

[33]J. Sedayao and K. Akita, "LACHESIS: A Tool for Benchmarking Internet Service Providers". Proceedings of the LISA IX Conference, 1995.

[34]C. Shipley and C. Wang, "Monitoring Activity on a Large Unix Network with perl and Syslogd". Proceedings of the LISA V Conference, 1991.

[35]R. Finkel, "Pulsar: An Extensible Tool for Monitoring Large Unix Sites". Software - Practice and Experience(SPE), Vol 27, No 10, 1163-1176, 1997.

[36]Sun Microsystems. SOLSTICE System Management. http://wwws.sun.com/software/solstice/system.mgmt.html

[37]D. Hardy and H. Morreale, "buzzerd: Automated Systems Monitoring with Notification in a Network Environment". Proceedings of the LISA VI Conference, 1992.

[38]B. Hill, "Priv: Secure and Flexible Privileged Access Dissemination". Proceedings of the LISA X Conference, 1996.

[39]C. Pierce, "The Igor System Administration Tool". Proceedings of the LISA X Conference, 1996.

[40]K. Ramm and M. Grubb, "Exu: A System for Secure Delegation of Authority on an Insecure Network". Proceedings of the LISA IX Conference, 1995.