Service Migration in Distributed Virtual Machines for Adaptive Grid Computing

Song Fu and Cheng-Zhong Xu
Department of Electrical and Computer Engineering
Wayne State University, Detroit, Michigan 48202
{oaksong, czxu}@wayne.edu

Abstract

Computational grids can integrate geographically distributed resources into a seamless environment. To facilitate managing these heterogeneous resources, the virtual machine technology provides a powerful layer of abstraction and allows multiple applications to multiplex the resources of a grid computer. On the other hand, the grid dynamics requires the virtual machine system to be distributed and reconfigurable. However, the existing migration approaches only move the execution entities, such as processes, threads, and mobile agents, among servers and leave the runtime services behind. They are not potent to achieve service reconfiguration in face of server overload or failures. In this paper, we propose a service migration mechanism, which moves the computational services of a virtual server, for instance a shared array runtime support system, to available servers for adaptive grid computing. In this way, parallel jobs can resume computation on a remote server without requiring service preinstallation. As an illustration of the service migration mechanism, we incorporated it into a Java-compliant distributed virtual machine, DSA, and formed a Mobile DSA (M-DSA) to accommodate adaptive parallel applications in grids. We measured the performance of M-DSA in the execution of applications from the SPLASH-2 benchmark suite on a campus grid. Experimental results show that service migration can achieve system adaptivity effectively.

1 Introduction

Virtual machine (VM) technology provides a powerful layer of abstraction for resource management in grids [5]. It enables user applications to become strongly decoupled from the system software of the underlying resources and other applications. This property facilitates the development of grid applications by allowing programmers to use well-defined service interfaces. However, the conventional VMs can not efficiently coordinate the resource sharing and scale well in wide area environments, especially across administrative domains. A remedy for these problems is to construct a distributed virtual machine, by factoring virtual machine services into logical components, replicating and distributing them throughout a network [15].

A distributed VM can be viewed as a set of virtual servers running on top of multiple physical servers. It is certainly possible to deploy its components as static computing units. However, when a server running a parallel job is reclaimed by its owner, the remaining processes of the same job have to be stopped. Moreover, the application will terminate when a server failure occurs. Research in [10] shows the CPU availability and host availability are volatile in grids. Adaptivity is fundamental to achieving application performance in such dynamic environments [2]. Therefore, the abstraction layer provided by a distributed virtual machine would not be fully exploited unless it can be instantiated and reconfigured at runtime to tackle the grid dynamics.

The existing approaches to constructing reconfigurable systems, such as process/thread migration [12], have significant limitations. They only move the execution entities among servers and leave the supporting services behind. Their effectiveness is based on two assumptions: (1) there are servers with idle resources; (2) the underlying runtime support system, as a form of grid services, has already been running on the destinations. On the other hand, parallel applications strive to distribute computational jobs evenly among servers. As a result, it is difficult for an overloaded server to steal idle CPU cycles from another one with similar processing capacity. Things become even worse when a server is going to fail. In that case, all of the computational jobs on the failing node will swarm into other existing servers. It will lead to cascaded server overloads and the entire parallel application has to be stopped, if no new servers are added to the system. As to the second assumption, it is not proper to pre-install and execute all possible services on each grid computer, across different administrative domains.

In this paper, we propose a scalable and effective approach, service migration, to achieve adaptive grid computing. The semantics of service migration is to suspend the residing execution entities, stop the service programs of a virtual server on the source node, migrate the runtime service data and states along with the execution entities’ states to a destination node, initiate a new virtual server with restored services on the destination, and resume application execution. In this way, parallel jobs can continue computation on an available server without requiring service preinstallation, when server overload and/or failures occur. Service migration provides a general approach to service reconfiguration. As a proof of concept, we incorporated service migration into our Distributed Shared Array (DSA) [1] system and designed a Mobile DSA (M-DSA). M-DSA supports a Java-compliant distributed virtual machine to accommodate parallel computation in heterogeneous grids. Service migration allows reconfiguring the virtual machine and makes it adaptive to grid dynamics.

Experimental results in a campus grid environment showed that service migration achieved system adaptivity with marginal
performance degradation, compared with that in a cluster. Our service migration method is complementary to other migration techniques moving execution entities. A hybrid migration infrastructure will make a system become more adaptive to the changing environment.

The remainder of this paper is organized as follows: Section 2 presents the related work. The overall architecture of M-DSA is given in Section 3. Section 4 describes the service migration design and implementation details. Section 5.1 focuses on the M-DSA interface to Globus service. In Section 6, we present performance results from SPLASH-2 benchmark programs. Conclusions are made in Section 7.

2 Related Work

Our service migration mechanism was inspired by the capsule migration in Stanford’s Collective project [14]. The Collective capsule encapsulates the complete state of a machine, including the entire operating system as well as applications and running processes. Virtual machine monitors are utilized to capture system state. By transferring a capsule across a network and binding it to a destination computer architecture, a user can resume her work after a commute between home and office. Capsule migration presents the possibility of moving the underlying service components between computers for computation continuity. However, as designed at the OS layer, this migration approach incurs considerable overhead.

Research interests in applying the classical VMs to grid computing are recently revived, with the Denali [19] and DVM [15]. However, these systems are not easily reconfigurable, due to the enormous execution contexts. In contrast, application-level virtual machines can be tuned to become more efficient for specific applications, and it is much easier to migrate these lightweight services in a network. The global object space in JESSICA2 [22], and the DSA [1] in Traveler [21] are such examples.

Figueiredo et al. [5] proposed an architecture to provide network-based services in computational grids, based on classical virtual machines. To efficiently manage grid resources, it allows a VM to be created on any computer that has sufficient capacity to support it. Virtual server migration is realized by moving an entire computation environment to a remote virtualized compute server. However, the migration details and associated overheads were not discussed by the authors. With the OS and middleware level support, SODA runtime system [9] constructs a distributed virtual server. Operations over shared arrays are supported by the DSA runtime system and transparent to application programs.

3 Overview of M-DSA Architecture

Service migration achieves adaptive grid computing by moving runtime support services from an overloaded or failing server to an available one. As an illustration of the service migration concept, we developed M-DSA, which extended a DSA runtime system [1] for virtually shared arrays with mobility support.

3.1 DSA Services

The DSA services provide a single system image across a set of compute nodes for distributed shared arrays. It relieves programmers from runtime scheduling of internode communications and the orchestration of the data distribution. Each DSA virtual server stores a portion of a shared array. Remote data access is supported by the DSA runtime system and transparent to application programs.

The DSA system comprises of a main API component known as the DsaAgent. Each virtual server involved in a parallel application has a DsaAgent. The DsaAgents are responsible for local and remote access to shared array elements. Parallel jobs are constructed as computational agents, distributed among the virtual servers. Operations over shared arrays include synchronous and asynchronous read and write. A SCI-like directory-based coherence protocol manages copies of an array block on different virtual servers. Besides, the DSA system provides APIs for barrier synchronization among agent threads. The original DsaAgents are stationary and cannot move between servers. As a result, a parallel application will make little progress or even be terminated, when servers become overloaded or failed.

3.2 Service Migration and M-DSA Architecture

To support service migration, we designed the M-DSA system with an architecture as depicted in Figure 1. The extended
DsaAgents constitute a set of virtual servers in a distributed virtual machine. Migration-enabled methods allow a DsaAgent to perform service migration to transfer DSA services among physical servers. We refer to the DsaAgent on a server that starts a parallel application as the coordinator. It is responsible for receiving migration requests from the LoadMonitor, and triggering a service migration at some potential migration point. The LoadMonitor makes migration decisions based on the performance information collected by sensors on each virtual server, during the application execution. We discuss the migration decision problem in Section 5.1. The LoadMonitor is an independent Java object that can be run on any server of the distributed virtual machine. To accommodate service images transferred from remote nodes, a Bootstrap daemon is pre-run on the destination node. It receives the incoming service image and initiates a new DSA virtual server with the restored services.

Different applications can multiplex a grid computer. This is realized by allowing multiple virtual servers to reside on a physical server. They provide different runtime services and accommodate the corresponding computational agents. The M-DSA runs on top of the JVMs, which facilitate the service migration in heterogeneous environments. We will present the details of service migration in the next section.

4 Service Migration for Adaptive Computation

A service migration involves collecting and packing the states of runtime services and agents, transferring them to a destination node, re-initiating the services and resuming agent execution at the statement prior to migration on the new virtual server. Compared with process/thread migration [12], service migration deals with not only moving the execution context of each process/thread, but also the reconstruction of the execution environment on a remote server. It introduces more technical challenges as a consequence. In this section, we will describe the service migration mechanism for adaptive computation.

4.1 Service Migration Algorithm

As depicted in Figure 2, the service migration mechanism proceeds as follows:

1. When the LoadMonitor detects some server in the system is overloaded or to be unavailable, based on the application performance data collected from execution monitoring sensors, it issues a “ToMigrate” message, along with the addresses of the source and destination servers, and a neighbor server to the coordinator, which will record this message in a message queue.

2. When the parallel computation reaches a synchronization point, the coordinator is requested to perform migration checking, which finds out whether there is a migration request to be served at this synchronization point.

3. If a “ToMigrate” request is found in its queue, the coordinator will send a “Migrate” message to the Bootstrap daemon of the destination server, denoted as bootstrapdst, and wait for the acknowledgement of migration completion.

4. Then bootstrapdst contacts the Bootstrap daemon on the source server, from which the DSA service pack and agent images will be transmitted to the destination. The related program codes are prefetched from the neighbor server.

5. The bootstrapdst loads the retrieved class files and builds the method area in the main memory. The DSA service and agents are reestablished by the restoration procedure. It also starts the DSA service by running the newly-initiated DsaAgent. Computational agents are triggered to resume their parallel jobs at the proper computation phase.

6. As soon as the service migration is successfully completed, the bootstrapdst sends an acknowledgement message back to the coordinator.

Next, we will present the details of service migration. Section 4.2 focuses on the monitoring of application performance, which provides crucial information for migration decision. The procedure of wrapping up and restoring runtime service is described in Section 4.3. As a complement, we briefly discuss the agent migration in Section 4.4.

4.2 Performance Monitoring for Service Migration

The overhead associated with packing, transferring and rebuilding runtime services of a virtual server is relatively high. This requires that agents on a virtual server have sufficient amount of computation and two consecutive service migration points keep distant spans in order to mitigate the impact of migration overhead.

If a distributed virtual machine support a sequential consistency memory model, the system appears like a multi-programmed uniprocessor and the virtual machine services can be migrated at any point with a guarantee of correctness of resumed execution. However, for better performance, the M-DSA system adopts a relaxed memory model to reduce both the number of messages and amount of data transfers between virtual servers. When the source server performs a service migration to a destination, the shared array partitions managed by the corresponding DsaAgent cannot be accessed during the migration. As a result, the memory consistency is complicated and it is similar to the consistency problem in “partitioned networks” [3]. In such a model, some virtually shared data between two consecutive synchronization points could be in inconsistent states. If agents access such data managed by the migrated DsaAgent, their computation may be incorrect, especially when they are used as input. To ensure correctness, service migration can only be allowed at synchronization points.
To exploit the benefits of service migration and minimize the impact of migration overhead, we monitor the dynamic performance of running applications and make migration decisions based on these accurate measurements. The Autopilot [13] toolkit is used to tune the migration timing according to the actual application progress. Each virtual server incorporates an Autopilot sensor, which extracts quantitative performance data of the corresponding application. The application program is instrumented with synchronization method calls and the DsaAgents record the execution time of each computation phase at those points. To relieve calculation burden in the LoadMonitor and avoid using much global information of the system, we specify an attached function for each sensor to normalize the execution time by the processing speed of a physical server.

When the LoadMonitor has collected the performance data from all the sensors, it compares the data with an overload threshold to make migration decisions. This threshold is a positive real number. Its value can be pre-defined or dynamically adjusted to tune to heterogenous systems. When a service migration is decided, we need to find an appropriate destination server to accommodate the migrated virtual server. This involves the resource management in M-DSA. We will present its details along with a rigorous model for migration decision in Section 5.1.

During service migration, the runtime data and states of a virtual server are transferred to a remote physical server. As a consequence, the monitoring infrastructure should be adjusted. This adjustment is small, because there is no direct connections among sensors. A sensor for the new virtual server is instantiated on the destination. Then the Autopilot toolkit registers properties of this sensor to the Autopilot manager, which resides on the LoadMonitor and acts as a name server for clients to look up sensors.

4.3 Service Wrapping Up and Restoration

In this section, we discuss the policies for wrapping up and restoring various JVM runtime data structures for service migration. The objective is to minimize the amount of data and states to be captured, while ensuring the correct resumption of services after migration.

The JVM specification [11] defines several data areas that are used during program execution, including the heap, method area, and JVM stack. They together form the execution context of a runtime service on a virtual server. To allow different applications to multiplex resources of a computer, we let the Bootstrap daemon on each node capture the service data and states, and wrap them into an application-independent form, service pack. As illustrated in Figure 3, the service pack contains components from the following areas:

The heap area for a virtual server stores the dynamically created shared arrays and associated control structures. Since they record the runtime information and contents of the shared arrays, most data structures maintained in the heap area have to be contained in the service pack. An exception is the locks for block accesses, because they can be created and initiated on the new virtual server. In M-DSA, we include the shared arrays in service pack instead of in the agent image, because they are allocated in the virtual server address space and accessed by computational agents only via well defined APIs. After the virtual server is restored on a new destination, the agents’ references to a shared array will be updated by using the array name to locate the actual array object.

The method area stores the per-class structure for each loaded class. It can be rebuilt at the destination when all the referenced classes are loaded. So, we do not include information of the method area in the service pack. To retrieve a remote class file, M-DSA applies a prefetch technique to let the new destination server retrieve the virtual server and application program codes in parallel with the transmission of a service pack, from a nearby server. We realize this technique with the aid of LoadMonitor. When the LoadMonitor triggers a service migration, it selects an existing lightly-loaded physical server that is in the same local area network as the new destination, i.e., their subnet addresses are identical. Then, a migration message containing the network addresses of (class download server, migration source, migration destination) is sent to the Bootstrap daemon of the new destination. The daemon contacts the nearby server for program codes, while waiting for the arrival of service pack from the source.

A JVM stack maintains the frames for virtual server method calls. Each virtual server provides the shared array access services to computational agents by defining a set of access methods, and these services will be re-initiated on a new physical server after migration. Therefore, the JVM stack can be rebuilt on the destination without transferring the stack frames. However, there is some control information, such as the shared arrays’ names, the number of agents, and their references, that should be kept in a service migration. These control information is well stored in the local variables of the virtual server object when a service migration occurs. Therefore, we transfer the virtual server object containing the necessary control information to the destination, and restore the object there.

If there are multiple virtual servers on a physical server, the Bootstrap daemon wraps them into separate service packs and migrates them one by one. Correspondingly, they will be restored sequentially on the destination server after migration.
4.4 Computational Agent State Capture

A computational agent contains one or a set of threads carrying out parallel jobs. In M-DSA, we transfer the states of computational agents along with the service pack to reduce migration overhead of sequential transmission of service states and agent states. The details are as follows.

Each virtual server contains a data structure recording all its residing agents and having a reference to each agent object. During service packing, as discussed in the preceding subsection, the virtual server object is serialized. As a result, all its referenced non-transient members, including the agent objects, will be included in the service pack. When the service pack is transferred to a new server, the agent thread objects will also be moved there. A challenge in agent movement is how to ensure its threads will resume execution just at the points where they are stopped.

The program counter (pc) register, in the per-thread runtime data area, contains the address of the Java instruction currently being executed by a thread. It indicates the starting point of computation after service migration. Since the potential migration points are at the computation phase boundaries, we apply a portable approach to preserve the pc register information. We represent a thread’s pc register by a set of integer numbers stored in phaseNum. After an agent thread has performed a computation phase, the phaseNum is increased by 1 to indicate the index of next phase. Then, a barrier method carries out a synchronization operation. It also makes the coordinator DsaAgent check whether there is any service migration request. To let the agent thread locate the right computation phase after migration, a if statement is inserted at the beginning of each phase to compare the value of phaseNum with the phase index of the following computation. Positive values of phaseNum indicate the agent was migrated and has executed on another server.

Unlike service packing, we extract the agent thread execution state stored in the JVM stack frames. We adopt the state capture and restoration approaches proposed in [16] to rebuild thread stack on a server after migration. At the same time, the thread pc register is set according to the value of phaseNum and the set of if statements. Thus, an agent can resume execution at the correct computation phase after a service migration.

5 Interface to Globus Service

The Globus Toolkit [6] is the de facto standard for grid computing. It provides an infrastructure and tools to manage grid resources. To construct a reconfigurable distributed virtual machine in the grid environment, we developed the interface to Globus in M-DSA for efficient and secure resource management.

5.1 Resource Management and Migration Decision

Resource management is an important component for the system startup and service migration, because we need to select appropriate physical servers to accommodate the DSA services. Figure 4 depicts the resource management architecture of M-DSA. It also presents the components that make the service migration decisions.

When the M-DSA system receives an application program, the scheduling will negotiate with the grid runtime system, e.g., Globus in our implementation, to obtain a list of available physical servers that have sufficient resources. Then it will broker the allocation and scheduling of application programs on grid resources, or start up the bootstrap daemon on the new destination server when a service migration is necessary. This step will also insert sensors to help the LoadMonitor control application execution. Next, the scheduler will generate a script file, which specifies the list of servers on which the application programs will be executed; see Figure 5 for a sample script. The Globus Toolkit will startup the application execution by running the script file.
• Migration candidate determination, concerning which server should transfer the runtime service and application programs.
• Migration timing, determining when a migration should be performed.
• Destination server selection, selecting a new physical server so that the performance gain from a migration won’t be outweighed by its overhead.

In [7], we formulated the migration decision problem as two optimization models and derived the optimal migration policy for distributed and heterogenous systems based on stochastic optimization theories. We summarize the results in the following theorem. Readers are referred to [7] for its proof.

Theorem 1 (Optimal migration timing) For any server s with capacity c in a distributed virtual machine and a given overload bound R, the optimal service migration timing l∗ in phase, is

\[ l^* = \frac{cR - w_0}{\mu}, \]

where \(c\) denotes the ratio of the current server’s capacity to that of the original server, \(w_0\) being the initial workload and \(\mu\) as the average of means of the application components’ workload changes.

When a service migration is triggered, we can calculate a lower bound of capacity of candidate servers for destination server selection. Then, the scheduler will negotiate with Globus to find an available server whose capacity equals to or is marginally above the lower bound. Our formal approaches requires the distribution information of application workload changes. We have been proposing a stochastic learning framework to obtain this distribution from the existing application performance data. In this paper, our focus is on the service migration concept and its illustrative example as M-DSA. So, we chose a simplified migration decision scheme based on workload threshold in our experiments.

5.2 Security Protections in M-DSA

The dynamic and multi-domain nature of the grid environments introduces significant security issues [18]. For instance, in a campus grid, clusters of different departments are connected via insecure campus networks. Unauthorized users may dispatch their programs to grid computers, and even try to exhaust grid resources via Denial of Service attacks. Therefore, security protection is a vital issue affecting the feasibility of a system in practice. The Grid Security Infrastructure (GSI) [6] provides authentication and authorization functionality to protect the grid resources. As a runtime support middleware on top of Globus, M-DSA utilizes GSI to ensure secure accesses to the underlying grid resources. In addition, it has its own security measures to protect the communication and service migration between virtual servers.

It is also possible that untrusted users attempt to run their private services on top of the Bootstraps. To tackle this problem, we devised service tags to distinguish registered service programs from others. Each Bootstrap daemon on a physical server maintains a list of tags, which map one-to-one to the set of registered services. The tag list is obtained from a trusted server in the system. When a service pack is transmitted to a Bootstrap daemon from a remote server, the daemon checks the tag field of the pack. If the tag matches one in the tag list, the corresponding service program is permitted to be restored and initiated on the server. Otherwise, the service pack will be discarded. To prevent malicious users on a network to extract the tag information from a valid pack, we encrypted its tag field. A digital signature is also included to ensure the pack’s integrity. The body of a service pack is not encrypted, due to performance concerns and the observation that a service pack for parallel computation usually does not contain sensitive information.

During the execution of a parallel application, DsaAgents of different servers will contact one another for array accesses. To protect their communication from being tampered by network users, we introduced a selective signing approach. That is data, travelling cross two clusters, is appended with a digital signature and the receivers check the digest. However, we assume a cluster is a trusted environment. So intracluster communication is free from digital signature to reduce delay. When a service migration occurs within a cluster, the digital signature of the service pack is also avoided. Our security mechanisms protect system resources from malicious usage, and ensure safe and efficient execution of parallel applications on M-DSA virtual machines.

6 Experiment Results

Our experiment platform was a campus grid in the Wayne State University. The cluster in the Institute of Scientific Computing was composed of eight SUN Enterprise SMP Servers. Five of them were 4-way E3500 with 1024 MB memory and the other three were 4-way E3000 with 512 MB memory. Each processor module had one UltraSPARC II with 400 MHz (E3500) or 336 MHz (E3000), and 4 MB cache. The cluster in the department of Electrical and Computer Engineering had four Dell PowerEdge2400 Servers. Each of them possessed two 733 MHz PentiumIII processor and 256 KB cache. They used the Red Hat Linux 8.0 with 2.4.18-14smp kernel. All of the grid nodes ran the Globus Toolkit v3.2 as the grid runtime system. The computers within each cluster were connected through a 100 Mbps Ethernet. The two clusters were located in different buildings and the intercluster connection had a bandwidth of 60 Mbps.

We ported the applications of LU factorization and FFT from the SPLASH-2 [20] benchmark suite to evaluate the performance of service migration on the M-DSA runtime system. All codes were written in Java, compiled in JDK 1.4.1.

6.1 Execution Time of LU and FFT on M-DSA

Figure 6(a) depicts the execution time breakdown of LU, factorizing 2048 x 2048 double matrix. Each virtual server hosts eight computational agents at outset. We changed the block size and measured the costs of computation, synchronization, shared array access and service migration in three different scenarios. In Scenario I, the LU application was executed within the SUN cluster and there was no service migration. The figure shows the overall execution time decreases as block size increases, and it increases after the block size is larger than 64 x 64. It is because
the data locality can be better exploited with a larger block, but this benefit is compromised by the false sharing when a threshold of block size is exceeded. We also notice the synchronization cost is significant, when the number of agents and computers is large. In Scenario II, we used four SUN E3500 servers and four DELL servers to execute the LU application. In the figure, the computation cost was a little greater than that in Scenario I, due to the relatively slow processing speed of the DELL servers. The intercluster communication leads to more remote access time of the shared array. As a result, the synchronization cost increased by 43.6% to 52.8%. The overall execution time is less than 17% more of that within a cluster. A service migration was occurred in Scenario III, which moved the DSA service from a DELL server to a SUN E3500 server. According to the figure, the computation cost was reduced slightly because of the increased processing speed. However, the synchronization cost was more because the average time waiting for agents to complete a computation phase was increased. The service migration overhead is less than 2.5 seconds, i.e., less than 1.5% of the overall execution time.

The execution time of FFT is shown in Figure 6(b). The execution environment was the same as that for the LU factorization. The inputs were 2048 x 2048 double arrays and each element was a complex number. FFT is a single-writer application with fine-grained access. The intensive shared array access and longer synchronization operations contributed to slow down this application. Its performance in the campus grid degraded, primarily due to the large number of intercluster shared array accesses. The service migration improved the data locality, but introduced more synchronization overheads. The overall execution time of our M-DSA system with service migration was at most 21.4% longer than that within a cluster.

### 6.2 Service Migration Overhead Breakdown

We measured the overhead breakdown of intracluster and intercluster service migrations with security protections. Figure 7 lists the experiment results in the LU factorization of 2048 x 2048 double matrix with 32 x 32 block size. The distributed virtual machine resided on four SUN E3500 servers and four DELL servers. The intracluster service migration was within the SUN cluster. The intercluster migration was from a DELL server to a SUN E3500 server, with security protection.

Given the cost of each category, we can derive the overall service migration cost, $C_{service}$, as,

$$C_{service} = t_I + t_P + \max\{t_{T_{serv}}, t_{Tcode}\} + t_{R},$$

where we take the maximum of $t_{T_{serv}}$ and $t_{Tcode}$, because service pack and codes are transferred in parallel with the prefetch technique. The $t_{T_{serv}}$ and $t_{Tcode}$ are determined by the sizes of service pack and codes respectively. According to the figure, the service packing and restoration are the most costly operations. This is partially caused by inefficiency of the Java object serialization procedure. The generated service pack is 4.2 MB of size. The security protections contributed 297 ms (25.5%) and 210 ms (24.7%) to the packing and restoration costs in the intercluster service migration. The different processing speed of source and destination computers in the intercluster migration accounts for the difference in packing and restoration time. The measured application performance information from the Autopilot sensors is piggybacked by the synchronization messages. So, the service initialization time is quite small. From the figure, we can see both types of service migration have a total cost less than 1.8% of the overall execution time for the LU application. Compared with stopping a parallel computation when overload and/or failure occur in a distributed environment, service migration is an affordable approach to system adaptivity.

### 6.3 Security Protection for Intercluster Communication

To evaluate the cost of our security approaches, selective signing, for intercluster communication, we measured the execution time for security operations and compared it with a full signing approach, that is to provide a digital signature for all intercomputer communication even within a cluster.
7 Conclusion and Future Work

In this paper, we have presented a service migration mechanism to move runtime services among networked computers. It allows reconfiguration of a distributed virtual machine for adaptive grid computing. As an illustrative example, we discuss the design and implementation details of service migration in the M-DSA runtime support system for parallel applications. Experiment results in a campus grid environment show that service migration can achieve system adaptivity effectively. Our ongoing work focuses on designing a new underlying runtime system utilizing the data management mechanisms proposed for data grids to achieve high-performance shared data accesses. More real applications will be performed to evaluate the system performance.

Acknowledgment This research was supported in part by U.S. NSF grant ACI-0203592 and NASA grant 03-OBPR-01-0049.

References


Figure 8. Security protection cost in the campus grid M-DSA.