Adaptation and Specialization for High Performance Mobile Agents

Dong Zhou and Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{zhou,schwan}@cc.gatech.edu

Abstract

Mobile agents as a new design paradigm for distributed computing potentially permit network applications to operate across dynamic and heterogeneous systems and networks. Agent computing, however, is subject to inefficiencies. Namely, due to the heterogeneous nature of the environments in which agents are executed, agent-based programs must rely on underlying agent systems to mask some of those complexities by using system-wide, uniform representations of agent code and data and by 'hiding' the volatility in agents' 'spatial' relationships.

This paper explores runtime adaptation and agent specialization for improving the performance of agent-based programs. Our general aim is to enable programmers to employ these techniques to improve program performance without sacrificing the fundamental advantages promised by mobile agent programming. The specific results in this paper demonstrate the beneficial effects of agent adaptation both for a single mobile agent and for several cooperating agents, using the adaptation techniques of agent morphing *and* agent fusion. *Experimental results are attained with two sample high performance distributed applications, derived from the scientific domain and from sensor-based codes, respectively.*

1 Introduction

Mobile agent[4, 14, 18, 42] as a new design paradigm for distributed computing potentially permit network applications to operate across heterogeneous systems and dynamic network connectivities, to reduce their bandwidth needs, and to avoid overheads caused by large communication latencies. In addition, mobile agent systems[11, 20, 24, 38] are designed to facilitate the construction of distributed programs that have the flexibility to adapt their operation in response to the heterogeneous nature of or dynamic changes in underlying distributed computing platforms.

Agent computing, however, is subject to several inefficiencies. Some of these inefficiencies are caused by the complexities of the environments in which mobile agents are deployed. Such environmental complexities include heterogeneity in architectures, communication networks (both at the hardware and protocol levels), operating systems, and agent management systems. This diversity requires agent-based programs to rely on underlying agent systems, most of which are based on interpreted languages like Java and Tcl/Tk[10, 26], to mask some of these complexities, by using system-wide, uniform representations of agent code and states to store, transport and execute agent programs. Additional inefficiencies in agent computing are caused by the dynamic nature of agent-based programs, where different components of these programs exhibit volatile 'spatial' relationships. Such 'spatial' volatility results from agents' mobility and from the runtime service/agent discovery schemes being used. The underlying agent systems 'hide' this volatility by ensuring that remote agent invocations are directed to current agent execution sites.

There has been considerable work on dealing with inefficiencies in agent computing, including the development of Just In Time (JIT) compil-

ers for agent code[23], of methods for creating efficient Java programs[37], and of performance tuning techniques and tools for distributed agent applications[15]. These efforts are particularly relevant to performance-constrained distributed applications, such as data mining, where large amount of states may have to be moved upon discovery, and interactive simulations, where application must offer real-time performance to end users [12, 21].

Our research is exploring two approaches for improving the performance of distributed, agent-based programs: (1) runtime adaptation and (2) agent specialization. The general aim of this work is to enable programmers to employ these techniques to improve program performance without sacrificing the fundamental advantages promised by mobile agent programming. This paper explores the effects of using two specialization approaches: *agent morphing* on a single mobile agent and *agent fusion* on multiple cooperating agents.

The remainder of this paper is organized as follows: Section 2 presents two applications that can benefit from the use of agent technologies while also requiring levels of performance not easily attained with current agent systems. interactive continuously Section 2 also describes the performance implications of using agent- vs. compiled object-based representations of the software components involved in interactive data viewing. Based on these evaluations, Section 3 next describes two agent specialization techniques – morphing and fusion – that address some of the runtime performance problems of applications like these. Significant performance gains are demonstrated from applying these techniques to the aforementioned applications for a variety of typical scenarios of use. Based on the improvements demonstrated in this section, Section 4 then describes next steps in our research, including the design of runtime support in which various adaptation techniques are easily applied. The paper concludes with a discussion of related research (see Section 5), conclusions, and future work (Section 6).

2 Using Agents with High Performance Applications

2.1 Mobile Agents and High Performance

Advances in the processing and communication capabilities of today's computer systems make it possible to wire heterogeneous and physically distributed systems into computational grids[8] that are able to run computation- and communication-intensive applications in real time. Consequently, end users are encouraged to interact with their applications while they are running, from simply inspecting their current operation, to 'steering' them into appropriate directions[27]. Examples of such applications include teleimmersion, interactively steered high performance computations, data mining, distributed interactive simulations, and smart sensors and instruments [12, 21, 41, 44].

Data in such applications comes from sources like sensors, disk archives, network interfaces, and other programs, is transformed while passing through the computational grid, and is finally output into sinks like actuators, storage devices, and the user interfaces employed by interactive end users. Application interfaces also permit applications to be reconfigured on-line in response to explicit user requests or to changes in user behavior. Sample reconfigurations include the creation or termination of certain application components, component replication, changes in dependencies between components, and changes in the mapping of components to computational grid elements.

Our aim is to use mobile agents to implement some of the data processing tasks of interactive high performance applications. More specifically, while it is unlikely that a high performance simulation like a fluid dynamics[39] or a finite element code will employ mobile agents for the simulation itself, it is desirable to represent as agents many of the computations and data transformations required for their interactive use. Such representations enable end users to interact with their long running simulations from diverse locations and machines (e.g., when working from home), and they permit the appropriate placement of data transformations such that data reductions are performed where most appropriate (e.g., before sending data to a weakly connected machine located in an end user's home). Our efforts are sup-

ported by several recent developments, including the creation of agent-based visualization and collaboration tools for high performance computations[13, 40].

Our second aim is to freely mix the use of agent- vs. compiled object-based representations of data transformation tasks, such that end users need not be aware of the current task representations and such that changes in task location and representation are made in response to current user behavior and needs. This paper presents our design ideas and initial implementation concerning a mixed agent/object system. This work is based on recent work elsewhere on object or agent specialization[28] and by our own work on object technologies for high performance and interactive parallel or distributed programs[6, 35, 36].

The remainder of this section describes and evaluates two applications that are representative of interactive scientific programs and sensor processing applications, respectively. The application drawn from the scientific domain, termed Interactive Access to Scientific Data (ISDA), has performance constraints due to the amounts of data being manipulated and displayed, regardless of end users' locations. The other application's performance constraints are derived from the necessity to process data in real-time or at certain rates, while sensor (source) and sink locations may change. This application, termed Parallel Scalable SAR Processing Simulator (PSSPS) is derived from the standard SAR (Synthetic Aperture Radar) benchmark originally developed at Lincoln Labs[46].

2.2 Sample Applications: Interactive Scientific Data Access (ISDA) and Parallel Scalable SAR Processor Simulator (PSSPS)

Both the ISDA and PSSPS applications are stream-based, driven by multiple inputs (stream sources) and able to service any number of end points (stream sinks). The purpose of ISDA (Figure 1) is to enable human end users to view and steer a high performance simulation (a global atmospheric model acting as a data source) via visualizations of the model's output data (ie., the stream sinks). The model simulates the transport of chemical compounds through the atmosphere. It uses assimilated windfields derived from satellite observational data for its transport calculation, and known chemical concentrations also derived from observational data as the basis of its chemistry calculations.

Of interest to this paper are the ancillary computations that 'link' the atmospheric model itself to various visualizations, where the set of these additional computations is depicted as a 'cloud' connecting the simulation to its inputs/outputs in Figure 1. Most such 'cloud elements' implement transformations that prepare model data for viewing by end users, e.g., reducing the amount of model data, transforming model data from its model-internal to a user-viewable representation, etc. Other 'cloud elements' perform additional computations like comparing model outputs with satellite observational data.

The specific cloud elements used in our work are (1) a regression model with which statistical tests may be performed on selected data, (2) a specialized data reduction code that 'clusters' scientific data[29] as per end user needs, (3) the Spectral-to-Grid transformer(s) that transforms the simulation model's internal data representation to the grid-based representation suitable for data visualization, and (4) the Isosurface calculator(s) that computes volumes of data of interest to end users and also generates the graphical primitives based on which this data may be viewed (an example of data of interest to end user is data volumes in which certain chemical constituents have equal levels of concentration, and an example of generated graphical primitives are the triangular representations of data suitable for the OpenGL rendering commands used in the 3D data visualization).

For the ISDA application, the argument for using agent-based representations for selected 'cloud' elements is apparent from the fact that the visualization engines themselves may have agent-based representations, as in the case of VizAD[40], in addition to object-based representations such as the SGI OpenInventor-based visualizations described in [30]. Specifically, when end users work in laboratories, they are likely to use high end machines capable of running the OpenInventor-based visualizations interactively in order to inspect model data in detail. When end users are simply 'looking a colleague over the shoulder' with the collaborative interfaces used in our work, then they require less detailed information and are likely to use ubiquitously runnable visualization tools like VizAD that enable such collaboration across a large diversity of machines and

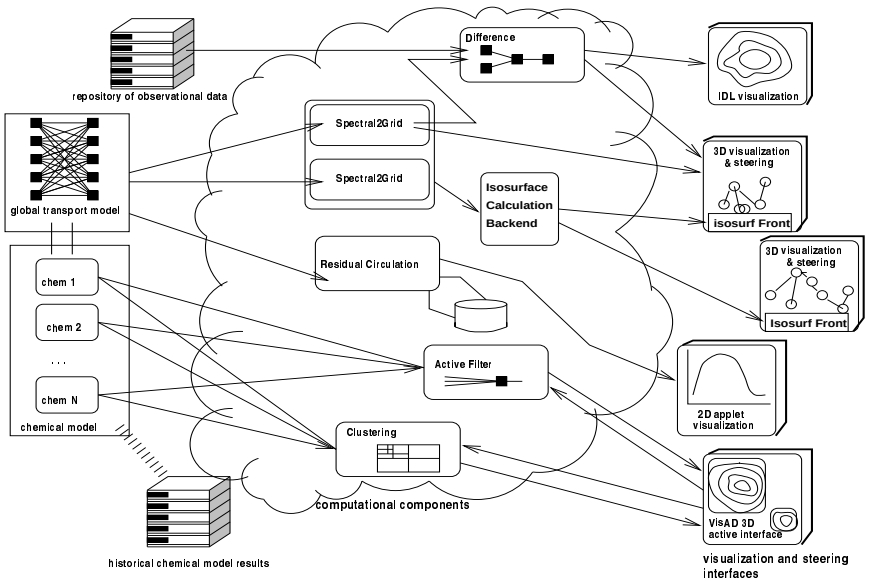

Figure 1: Computational components in ISDA.

locations.

Consequently, the associated data transformations may need to change data representations repeatedly, first from the model's internal spectral form of data to grid form, second from grid form to descriptions that may be rendered graphically, as exemplified by an Isosurface calculator. This transformer may reside as an agent on the same machine as the agent-based VizAD visualization or it may reside on a remote machine and operate as a specialized data reduction engine if the VizAD visualization is run on a weakly connected machine, such as a laptop or a computer located in a user's home. Clearly, the suitable choices of representation and the locations of agent- or object-based cloud elements depend on many factors, including current user needs and computing platform characteristics. The results presented below represent a first step toward automating choices like these, as they demonstrate the tradeoffs in performance when different element representations are used.

In PSSPS (Figure 2), data is either synthesized online or read from disk files that contain radar images. The processing of this data is performed by cloud components that include (1) selectors that filter out uninterested frames, (2) FIR filters for video to baseband I/Q conversion, (3) range compression units for pulse compression, and (4) Azimuth Compression Units for cross-range convolution filtering. Convolution results are the output strip-map images used for visualization. The implementation of PSSPS used in our work exhibits both the pipeline parallelism similar to that of the ISDA application and additional parallelism internal to pipeline stages that are able to utilize it, as is exemplified by the data parallel processing of the convolution stage for the purpose of speeding up this process.

In the PSSPS application, agent representations are useful for sensors in remote or mobile locations and/or for end users who wish to understand sensor data from mobile or remote locations. One example is a battlefield where radar data should be made accessible in some form to mobile units in the field operating at locations remote from the radar itself, only when such operational capabilities are currently required, whereas more permanent processing is installed and operated at regional or global command sites. This implies the need for flexibility in the location and execution of agent-based SAR computations associated with data sources and sinks.

2.3 Tradeoffs across Alternative Program Representations

The basic performance problems arising from the use of agent vs. compiled object representations of ISDA or PSSPS components are well understood.

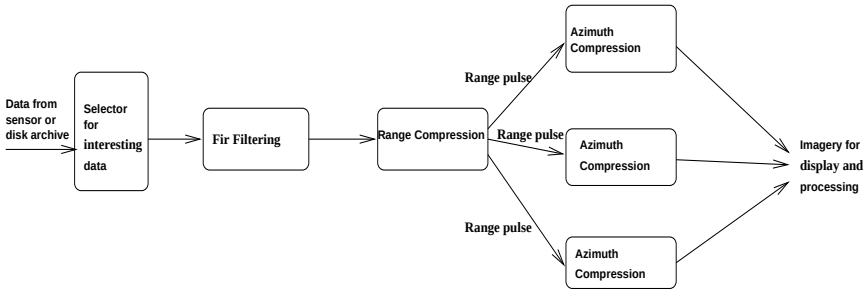

Figure 2: Structure of PSSPS.

Usage of interpreted languages, such as Java[38, 20, 24], is a major cause of these problems as is depicted by experiment results listed in table 1. These experiments use the Sun Solaris native C compiler to generate compiled code and use JDK1.2-beta3 package for the Java compiler and runtime environment (including the JIT compiler used in our later experiments). The platforms used in the experiments are the Sun Ultra-Sparc 30 uniprocessor systems with Solaris 2.5.1 and 100MB FastEthernet interconnection.

Specifically, these measurements demonstrate that for an application component like PSSPS' Azimuth computation, which has a large amount of computationally expensive floating point operations, the Java code realization runs almost 10 times slower than compiled code implemented with C. And similarly, for an application component like ISDA's Isosurface generation back-end, the Java code implementation on average takes 17 times more time than its native counterpart for a random set of grid data.

Agents	java code	compiled code	ratio
Azimuth processing	30,992	2,948	10.513
Isosurface back-end	8,348	461	18.11

Table 1: Comparing the performance of Java vs. native code(in msecs except for 'ratio').

In comparison with Table 1, the measurements presented next demonstrate the utility of JIT compilers for Java, which constitutes one way in which agent-based programs and their runtime environment may be specialized for efficient execution on target machines that have such compilers available. Specifically, Table 2 shows that with JIT, Java realizations of application components are from more than 3 times (for Isosurface back-end) to close to 5 times (for Azimuth processing) faster than those without JIT. This table also shows that static optimizations done by compilers vary in their effectiveness and that Java inter-class optimization does not much affect either of the two application components.

The performance improvements demonstrated in Table 2 might be sufficient for some applications. However, for applications like PSSPS and ISDA, their scalability and utility for large-scale data sets and for realistic execution rates would be compromised substantially by the fact that their JIT-based Java representations are 70% (in the case of Azimuth processing) to 300% (in the case of Isosurface back-end) slower than native code. However, perhaps even more important is the fact that significant additional overheads exist for distributed agent-based programs in which multiple agents must cooperate remotely, as is the case for both the ISDA and PSSPS applications. Agent-based high performance systems obviously need efficient communication mechanism to facilitate cooperation, which may involve large amounts of data, among agents. Unfortunately, our third experiment shows that Java RMI[43], which is being used for agent communication in many of the Java-based agent systems, has overheads which limit these applications' scalability in the presence of intensive agent communication.

Our experiment uses three components of PSSPS: Fir filtering, Range processing and Azimuth processing, to construct three pipelines with length of 0(Azimuth only), 1(Azimuth and Range), and 2(all three components). Our agent implementation uses Java and RMI, while the compiled object implementation uses C and OTL. OTL is the object invocation layer of the COBS CORBA-compliant object infrastructure developed at GT for high performance object-based programs[36, 7]. OTL is built

Agents		No Optimization	Normal Optimization	Inter-class Optimization
Azimuth processing	Without JIT	30,992	30,885	30,736
	With JIT	5,246	5,223	5,258
	Ratio	5.908	5.913	5.846
Isosurface back-end	Without JIT	8,348	7,452	7,469
	With JIT	1,921	1,812	1,819
	Ratio	4.364	4.113	4.106

Table 2: Effects of using JIT compilation(in msecs except for 'ratio').

on top of TCP and can perform object invocation across heterogeneous platforms.

The experimental results depicted in Table 3 demonstrate the importance of RMI performance for even the computationally intensive applications. With increased pipeline lengths, the relative performance of the compiler optimized, JIT-enabled Java representation over that of the compiled code using our efficient distributed object infrastructure gets progressively worse. We suspect that this performance problem of Java RMI is due to marshaling overhead (object creation, stream creation) and to threading and synchronization costs. There has been research on improving the performance of Java RMIs[19], but this has not yet resulted in improvements to the standard Java distribution.

pipeline length	java code	native code	ratio ratio
0	5,223	2,948	1.772
1	23,095	4,968	4.649
2	34,404	6,054	5.683

Table 3: JIT's effectiveness for pipelined applications(in msecs except for 'ratio').

The measurements depicted above demonstrate the need for additional optimizations of agent-based representations of distributed programs if they are to be used to implement high performance applications. One basic issue, we believe, is that current JIT-based optimizations lack information about the operation and behavior of these distributed applications that can be exploited to further improve their performance. Specifically, first, if the duration of an agent's operation is known, then it becomes feasible to *morph* at runtime an agent-based representation to one using native code, invisibly to end users and using techniques like cross-platform binary code generation or access to code repositories. The technique assumed in this paper relies on the presence of code repositories[2]. Second, if multiple agents residing and cooperating on one machine could be 'compiled' as if they were one unit, then this compilation could use global knowledge not accessible to either method-based JIT compilation or component-based Javac compile-time optimization, thereby able to address both intra- and inter-component (e.g., RMIs) performance issues. Such global properties are considered by the *agent fusion* technique explored in this paper, which combines multiple agents into single, more powerful and potentially, more efficient agent representations.

Morphing, fusion, and their application to the ISDA and PSSPS distributed programs are discussed in more detail next.

3 Techniques for High Performance Agent Realization

3.1 Agent Morphing

Morphing Concepts. One specialization mechanism proposed in this paper is morphing, which means changing the form of a mobile agent to adapt to the specific platform on which it is currently running. Namely, each agent may have two forms: a platform independent form – henceforth termed *neutral* form – and a platform dependent form – henceforth termed *native* form. The agent is programmed to be able to morph between these two forms, using some of the techniques exposed in Section 3 below.

This paper establishes the importance of agent morphing and describes the internal structure of morphable agents. Briefly, we assume that all agents start with their neutral forms, which implies that

they have no architectural knowledge of the hosts when they are deployed; this also facilitates the dynamic introduction of new architectures into the system. Once started, the agent will spawn a low priority thread to acquire its native implementation, and if such a native form exists, the agent can then switch from its former agent mode into native mode whenever deemed necessary. Such mode switching can occur either

- immediately after the agent has acquired its native form,
- when a morphing instruction in the program is encountered,
- or in response to end user request or to events generated by the quality of service management system[33] that controls agents/objects' efficient execution.

During mode switching, the system transforms and copies the agent-mode states into its native-mode representation. Such transformation and copying are platform-dependent, and are carried out by a set of functions within the native implementation (for our experiments, in a JNI module). This function set can either be user- or system-defined. In the latter case, the application programmer has to define an interface describing the data fields that need to be transformed and copied when morphing is performed. This interface has to be written in both native (in our case C) and agent (Java) code. System-provided state transformation functions rely on these interface definitions; they also rely on object reflection techniques to achieve transformation and copying.

An agent may also morph back from its native form to its neutral form, which happens when the agent decides to migrate. In this case, the agent first transforms and copies its native states into neutral states, then switches back into agent mode, and finally migrates, with the help from the underlying agent system[38]. In general, morphing may be triggered at any point during agent execution in response to externally generated events or by the agent itself in response to internal state changes. However, after morphing, an agent has to restart from a fixed entry point, which essentially requires an agent to record its (application) state prior to morphing.

The morphable agents designed and used with the ISDA and PSSPS applications described in this paper utilize two key abstractions: (1) invocation *adaptors* and (2) *events*.

The purpose of the adaptor is like that of the *policies* associated with objects described in [9, 16, 35]: it intercepts all incoming invocations to the morphable agent, 'translates' them to the form appropriate for the agent's current representation (neutral or native), and then directs the invocations to this representation's implementation. Each agent uses a native form adaptor and a neutral form adaptor at the same time, so that invocation clients of the agent can invoke the agent regardless of their current states. The system we are constructing assumes that each agent is morphed in its entirety, either residing in its native or its neutral state; this eliminates problems with partial state translation and state consistency when state is accessed simultaneously by native and neutral method implementations.

The purpose of events is to provide a uniform manner in which morphing is initiated, in response to the receipt of events that are internal or external to the agent. Internal events may be raised when certain state changes occur; external events may be raised by other agents or by a resource management system that has global knowledge of the agent program's behavior.

Application of Morphing to Sample Applications. Sample morphable agents have been constructed with Java, where specialized (morphed) versions of these agents are also available as native code for SUN Sparc/Solaris machines. The performance benefits of agent morphing have already been presented in tables 1 and 2, when applied to the Isosurface and Azimuth transform agents. In these particular examples, morphing overheads only come from native library loading and minimal application state translating and copying and are thus almost negligible. However, we expect such overheads to be higher when a code repository server is involved and/or when the amount of shared data is significant and morphing is applied more frequently. These agent realizations utilize the adaptors described in this section, using internally generated events. We manually program adaptors in our sample applications, but we believe that such adaptors can be readily generated by a compiler from IDL files.

The conditions under which morphing is applied are straightforward. In each example, when the amount

of data being processed by agents increase (e.g., the ISDA application's visualization wishes to view more data or the PSSPS application's image resolution is increased), then the agent implementations of these transformers do not deliver suitable performance. This fact is detected by inspection of internal data buffer fill levels[1]. To speed up data processing, the agent first acquires its native representation, then invokes an application-provided state-checkpointing method (if such a method was defined by the user), then transforms its state using the function set provided either by the system or the application programmer, and finally, initiates execution of its native form from a fixed entry point.

Discussion. Advantages of morphing include the performance improvements demonstrated in this section and also potential improvements concerning the predictability of agent execution. Predictability is particularly important for real-time and embedded applications and because Java code execution times are believed to be difficult to predict due to interpretation and garbage collection[25]. Difficulties with morphing arise from two sources. First, if native implementations cannot be acquired from a machine's local file system, then morphing overheads may become large due to the costs of access to remote repositories. Second, increased generality and complexity of native code compared to the sample agents used in our work may make morphing infeasible and/or require the provision of additional mechanisms to enable morphing. For example, with the sample applications and with the object realizations used in our research, agent safety may be guaranteed due to the sample objects' relative lack of internal complexity (e.g., no object-initiated file accesses) and due to the object system implementation's lightweight nature and heavy use of libraries. For general CORBA- or DCOM-based object implementations, guaranteeing improved performance or predictability as well as safety will require additional effort. Furthermore, internal native states like 'open file descriptors' cause problems for morphing not easily addressed for future agent systems (see [5] for a more detailed discussion of this topic).

[1] More sophisticated techniques for first detecting and diagnosing performance problems with pipeline-structured applications are described in [22, 34]

3.2 Agent Fusion

Fusion Concepts. Fusion is useful for distributed agents that communicate within and across different machines. For closely coupled cooperating agents, communication overheads can constitute significant portions of their operating costs. For example, in the sample applications described above, when an 'upstream' agent filters out much of the data, then the remaining data handed off to the 'downstream' agent may not result in significant communication-based overheads. This is the case for the 'statistics' agent operating on the atmospheric data in the ISDA application, for example. Conversely, when such filtering does not remove much data, communication overheads may be substantial, as evident for many of the PSSPS application's agents.

The intent of agent fusion is to remove communication overheads from collaborating sets of agents and to enable optimizations across agent boundaries. Briefly, these overheads include actual network transport and protocol processing times, data copying costs, and thread/process scheduling and context switching costs that arise when tasks are performed by multiple vs. single agents residing on the same machine. Possible optimizations across agent boundaries are similar to those performed by compilers across procedure boundaries, including inter-procedural analysis leading to code or data motion and procedure integration where multiple overlapping procedures are combined into a smaller number of more efficient, combined procedures[3, 31, 32]. One particularly unfortunate communication cost is that incurred by multiple cooperating agents residing on the same machine, where their shared location is due to unforeseen agent migration actions (e.g., both agents 'found' interesting data on the same machine, or both agents moved to that machine due to local resource availability). In this case, it is clear that such agents would operate much more efficiently if they were placed into the same address space and used shared user-level threads, as this would reduce agent invocation costs to the costs of a few procedure calls (via adaptors). It would also eliminate overheads associated with the implementation of asynchronous invocations, such as the use of additional threads and their scheduling and synchronization, and additional data copying due to asynchrony and kernel/user space crossings.

In summary, while agent morphing specializes individual mobile agents, fusion performs runtime opti-

mizations across multiple, cooperating agents. Fusion may be applied repeatedly, to create single efficient agents from multiple collaborating agents.

Application of Fusion to Sample Applications. Agent fusion is applied to those components in the sample applications for which co-location on the same machine and with the same internal form (native or neutral) is likely to occur. For instance, Isosurface calculations in the ISDA application may be performed outside the scope of the visualization engine (if the visualization runs on a remote or weakly connected machine) and jointly with visualization. In fact, earlier versions of ISDA always performed Isosurface calculation within the visualization agent itself, as the visualization was running on a strongly connected and high end visualization engine (i.e., an SGI Octane). The versions of ISDA now used by end users do not wish to use such an instance of the visualization agent due to their desire to operate across a wider spectrum of machines and service a larger number of end users.

Figure 3 depicts the situation in which fusion is carried out on the Isosurface front-end and back-end, where the back-end does the actual isosurface calculation and then sends the computed isosurfaces as data to front-end on which visualization agent runs. Originally, there are multiple Isosurface front-ends, and the back-end is not co-located with any of the front-ends, with the intent of minimizing overall communication cost and avoiding burden the visualization engines with isosurface calculations. When all front-ends except one complete their execution, then the back-end might migrate to the remaining front-end's location, in order to reduce the communication cost between the two parties. The system is then able to fuse the two agents to further enhance performance. Table 4 shows the potential performance improvements resulting from such a fusion action. The same table also lists the results of fusing two agents in the PSSPS application: fir filtering and range processing.

In both of the experiments shown below, fusion is done manually through simulating the actions would have been taken by the fusion compiler. Such actions include procedure in-lining, data sharing, and asynchrony elimination.

Discussion of Results. The results in Table 4 indicate the benefits of agent fusion clearly. The table's first three columns show program performance when agents interact via remote object invocation, where references to other agents are remote object references received from a registry service. The fourth column shows the performance of the same programs in which references to cooperating agents are replaced by references to local objects, but object fusion has not been performed (ie., the compiler did not compile both agents as one unit). The next column shows performance subsequent to joint agent compilation.

These experiments demonstrate that it is highly desirable to put two agents into the same virtual machine and treat them as local object to each other if they are running on the same host and closely cooperating. In these experiments, the largest gain from agent fusion is due to the replacement of remote object invocation. The gains indicated in the last column are due solely to the compiler's use of global optimizations when multiple agents' code is combined; They range from 7.88% to 30.13% and are due to inter-class optimizations like inter-procedural aliasing and procedure in-lining, and most importantly, the elimination of asynchrony.

3.3 Summary

Experiments with sample applications built on top of our current infrastructure demonstrate that both morphing and fusion have significant performance benefits. Morphing provides 70% to 300% gains in performance, while fusion provides from 7.88% to 30.13% additional improvements after co-locating two agents in the same process.

The overheads caused by the acquisition of native realizations and by state transformation in morphing are relatively low, provided that morphing is not frequently invoked and that each run of the application lasts reasonablely long. We believe that the frequency of morphing will be low in most cases, as it needs to be invoked only once, unless migration is involved. However, migration itself tends to be an expensive activity; morphing will simply add some costs to this process. Similar arguments hold for object fusion.

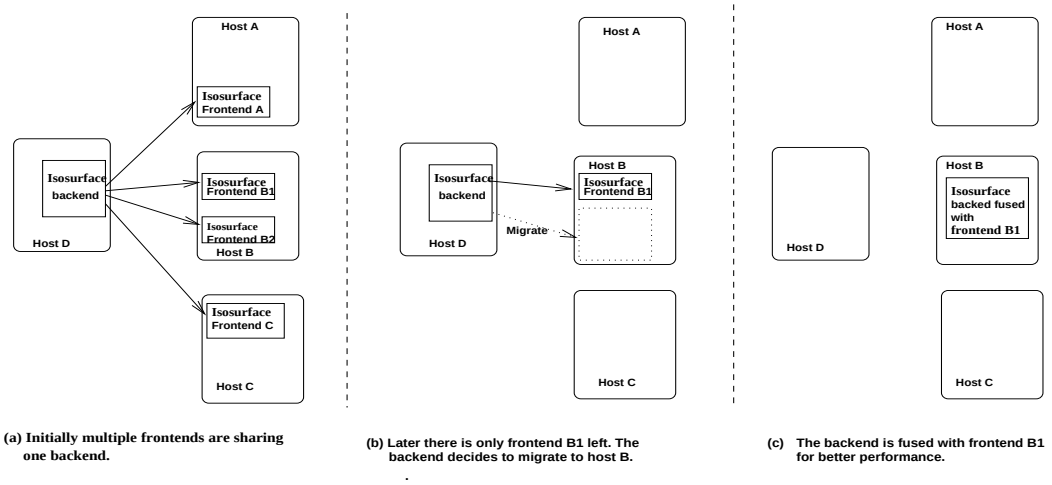

Figure 3: A scenario for fusion of Isosurface calculation front-end and back-end.

Agents	Realization	Different Hosts	Same Host Diff Proc	Same process Remote Object	Local Object	Fully Fused	Fusion Benefit
Isosurface back & front-end	Java code	8,909	22,377	12,713	2,194	2,021	7.88%
	Compiled code	4,135	4,134	575	575	511	11.13%
Fir-Range processing	Java code	17,950	28,787	21,624	2,714	2,427	10.57%
	Compiled code	4,177	4,166	1,488	1,487	1,039	30.13%

Table 4: Result of fusion applied to different agent forms.

4 Toward a System for Mobile Agent Optimization

The performance benefits derived from agent morphing and fusion presented in Sections 2 and 3 are significant. They are motivating us to construct an agent system within which agent/object–*objent*–programs are easily constructed and adapted at runtime. Such a system consists of contracts for performance requirement specifications, a notification system for contract monitoring, policies for application specified adaptation enactment, and finally, system or application defined adaptations.

This paper only addresses adaptation methods and adaptation enactment mechanisms unique to mobile agent-based applications. In our ongoing research, we are identifying other aspects unique to mobile agents and therefore, appropriately addressed by an adaptive Objent system. We expect to base this work on previous and current research in distributed adaptive systems[34, 6] and in distributed object systems[45].

Specifically, our Objent system has to address the following issues to support morphing and fusion adaptation:

1. Where and how are native agent forms created and maintained?
2. How does the system ensure consistency between the migratory and native versions of agent state?
3. When an agent's form is being or has been changed, can external agents not aware of this change continue invoking it?
4. What are useful fusion algorithms?

The basic fundamental components of the 'Objent' system we are developing have been described in a previous publication[2]. We next outline our solution approaches to the specific questions posed above.

Acquiring an Agent's Native Version. Agents are created and migrated in their platform-independent(neutral) forms. Their native forms may be created by (1) acquisition of a trusted native version from the agent's current execution site, involving agent retrieval from a local repository and/or its generation by a locally resident compiler, or (2) agent acquisition from a remote, trusted repository to which providers submit agents in forms suitable for various platforms. Initial design ideas on such a repository are described in [2].

Consistency Between Multiple Agent Forms. An agent capable of morphing never has more than at most two implementations, a platform-neutral and a native one. At any one time, only one of these implementations is active. This implies that agent morphing necessitates state copying from the previously active to the new agent form. We intend to develop methods for full state copying, for partial state copying, and for permitting developers to provide specialized state maintenance methods.

Agent Invocation. When an agent changes its form, it is not likely that the agent's new form is able to efficiently interpret the objects passed to it via invocations to its old form. Moreover, one of the purposes of agent morphing is to enable efficient and direct communications between agents in their native forms whenever possible.

The solution being developed in our current research is one that permits the (inefficient) invocation of remote agents that differ in form, coupled with the implementation of notification protocols among agents that enable agents to switch to an efficient invocation protocol whenever possible. Specifically, we employ *adaptors* that are present in all agents capable of morphing. Each adaptor has two forms: (1) the neutral form is visible to the agent's neutral implementation; (2) the native form is visible to the native code. An adaptor is much like an object 'policy' in that all invocations in the respective agent forms are directed to the appropriate adaptor. The adaptor, then, knows about the agent's current form, has methods generated from its interface definition for request translation from one form to the other, and is able to deal with issues arising from the agent's concurrent invocation in both of its forms.

The overheads of using agent adaptors are small when agents communicate in the same form, as was the case for the overheads incurred by policies evaluated in [35]. When adaptors must translate between forms, overheads depend on the complexities of invocation parameters.

Fusion Algorithms. The fusion algorithms used in our current work carry out inter-procedural optimization, and they reduce or eliminate the multithreading overheads caused by asynchronous remote agent invocation. For example, in PSSPS, an asynchronous invocation is implemented as follows: with each method in an agent's interface definition, we associate a special modifier that denotes whether the method should be invoked synchronously (SYNC_IF_FUSED) or asynchronously (ASYNC_IF_FUSED) by fellow fused agent(s). An invocation to SYNC_IF_FUSED methods by a fellow fused agent(s) is replaced by a direct local procedure call. The fusion algorithm then applies interprocedural analysis to perform aliasing and, in the case of SYNC_IF_FUSED methods, procedure inlining. Aliasing attempts to eliminate unnecessary data copying, since data formerly located in different address spaces or on different hosts may potentially be shared subsequent to agent fusion and colocation.

Fusion may be applied repeatedly, possibly later followed by agent 'splitting', if indicated. Agent 'splitting' is an agent adaptation method we are aware of, it applies *program slicing* to an agent operating on a distributed data set and distributes agent slices so that each agent slice operates on some local data which is a subset of the distributed data set.

5 Related Work

Recent active research on mobile agent systems concerns the areas of agent facility standardization, mobile agent system interoperability, and operating system support [17, 20, 24, 38]. The platforms developed by such works provide the basic agent system functionality upon which our runtime system is built. We add to this functionality the ability to adapt mobile agent and we add the event mechanisms necessary for building dynamic runtime support for monitoring and for adaptation initiation and enactment.

Research results from software specialization systems like SPIN, Exokernel, and Synthetix may be applied to our adaptable agent architecture to customize the agent system itself and/or individual agents. We will focus on customization issues more specific to mobile agent environments.

We have already benefited from research on object policies and on meta-objects[35, 16] to develop the 'adaptor' concept presented in section 3. Our work will also take advantage of current research on quality of service infrastructures like BBN's QuO[45] and Honeywell's ARA[34], but we will adapt their techniques to the mobile agent domain targeted by our work.

Our work is part of a broader project described in [1], with early results are presented in [2].

6 Conclusions and Future Work

Agent computing is subject to several inefficiencies, some of which are due to the complexities of the environments in which mobile agents are deployed. Our research is exploring runtime adaptation and agent specialization to improve the performance of agent-based programs, aiming at enabling programmers to employ these techniques and runtime adaptation in general, to improve program performance without sacrificing the fundamental advantages promised by mobile agent programming. We explore the effects of using two specialization approaches, morphing and fusion, on a single mobile agent and on several cooperating agents. Our experimental results with two sample applications, ISDA and PSSPS, show that such specialization approaches result in considerable performance improvement.

We have built a preliminary infrastructure for online morphing, which offers mechanism for interlanguage remote invocation using the model of invocation adaptors developed in our research. Infrastructures and mechanisms are applied to the ISDA and PSSPS distributed high performance applications. Also used with these applications is a realization of mobile event channels that allow reliable event delivery during end-point migration.

Our future work concerns systematic support for specialization approaches like morphing, fusion and others such as slicing. This support will comprise event mechanisms and quality of service infrastructure, both of which are important to a general agent adaptation system. We will also work on compilers for agent fusion and the adaptation of agent invocations.

Acknowledgments. Prof. Raja Das is participating in the Objent system design, with focus on the application of compilation methods to improve agent or object performance, including the use of fusion techniques. Prof. Mustaque Ahamad has been investigating the overheads of agent communications (e.g., Java RMI) and methods for creating quality-controlled Java-based objects. Greg Eisenhauer and Beth Plale are responsible for the ISDA application, Davis King provided the code for Isosurface calculation, and Fabian Bustamante is currently creating additional elements part of the ISDA application. Rajkumar Krishnamurthy provided the SAR benchmark code used in our work.

References

[1] Mustaque Ahamad, Raja Das, and Karsten Schwan. Integrating object and agent technologies for high-end collaborative applications. http://www.cc.gatech.edu/systems/facstaff/ahamad/objent.html.

[2] Mustaque Ahamad, Raja Das, Karsten Schwan, Sumeer Bhola, Fabian Bustamante, Greg Eisenhauer, Jeremy Heiner, Vijaykumar Krishnaswamy, Todd Rose, Beth Schroeder, and Dong Zhou. Agent and object technologies for high-end collaborative applications. In *OOPSLA'97 Workshop on Java-Based Paradigms for Mobile Agent Facilities*, 1997.

[3] Michael Burke and Jong-Deok Choi. Precise and efficient integration of interprocedural alias information into data-flow analysis. *ACM Letters on Programming Languages and Systems*, 1(1), Mar. 1992.

[4] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant agents for mobile computing. IBM T.J. Watson Research Center, 1995.

[5] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software – Practice and Experience*, 21(8):757–785, August 1991.

[6] Greg Eisenhauer and Karsten Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, Aug. 1998.

[7] COBS: Configurable OBjectS for High Performance Systems. College of computing, georgia institute of technology. http://www.cc.gatech.edu/systems/projects/COBS/.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[9] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[10] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification (Java Series)*. Addison-Wesley, 1996.

[11] Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995.

[12] Robert Grossman. The terabyte challenge: An open, distributed testbed for managing and mining massive data sets. http://www.lac.uic.edu/hpcc-grossman.html.

[13] Habanero. National Center for Supercomputing Applications and university of illinois at urbana-champaign. http://notme.ncsa.uiuc.edu/SDG/Software/Habanero.

[14] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile agents: Are they a good idea? Technical report, IBM T.J. Watson Research Center, 1995.

[15] Delbert Hart and Eileen Kraemer. An agent-based perspective on distributed monitoring and steering. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, Welches, Oregon, Aug. 1998.

[16] Jun ichiro Itoh, Rodger Lea, and Yasuhiko Yokote. Using meta-objects to support optimisation in the apertos operating system. In *Proceedings of USENIX Conference on Object-Oriented Technologies (COOTS)*, Jun. 1995.

[17] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agents. In *Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, Wa, USA, May 1995.

[18] Keith D. Kotay and David Kotz. Transportable agents. In *CIKM Workshop on Intelligent Information Agents*, Gaithersburg, Maryland, Dec. 1994.

[19] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementation of java remote method invocation (rmi). In *Proceedings of 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[20] Danny B. Lange and Daniel T. Chang. White paper. IBM Aglets Workbench, Sep. 1996.

[21] C. Lee, C. Kesselman, and S. Schwab. Near-real-time satellite image processing: Metacomputing in cc++. *Computer Graphics and Applications*, 16(4), 1996.

[22] Vernard Martin and Karsten Schwan. ILI: An adaptive infrastructure for dynamic interactive distributed systems. In *4th International Conference on Configurable Distributed Systems*. IEEE, 1998.

[23] Sun Microsystems. Java on solaris 2.6: A white paper. http://wwwwseast2.usec.sun.com/solaris/java/wp-java/.

[24] Dejan S. Milojicicc, William LaForge, and Deepika Chauhan. Mobile objects and agents (moa). In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[25] Akihiko Miyoshi and Takuro Kitayama. Implementation and evaluation of real-time java threads. In *Proceedings of Real-Time Systems Symposium*, Dec. 1997.

[26] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[27] Beth Plale, Greg Eisenhauer, Karsten Schwan, Jeremy Heiner, Vernard Martin, and Jeffrey Vetter. From interactive applications to distributed laboratories. *IEEE Concurrency*, 6(2), 1998.

[28] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain, Colorado, 1995.

[29] William Ribarsky, Yves Jean, Thomas Kindler, Weiming Gu, Gregory Eisenhauer, Karsten Schwan, , and Fred Alyea. An integrated approach for steering, visualization, and analysis of atmospheric simulations. In *Proceedings IEEE Visualization '95*, 1995.

[30] William Ribarsky, Yves Jean, Song Zou, Karsten Schwan, Bobby Sumner, , and Onome Okuma. A heterogeneous environment for visual steering of computer simulations. Submitted to IEEE Computer Graphics & Applications.

[31] Stephen Richarson and mahadevan Ganapathi. Code optimization across procedures. *IEEE Computer*, Feb. 1989.

[32] Stephen Richarson and mahadevan Ganapathi. Interprocedural optimization: Experimental results. *Software–Practice and Experience*, 19(2), Feb. 1989.

[33] Daniela Rosu, Karsten Schwan, and Sudhakar Yalamanchili. Fara - a framework for adaptive resource allocation in complex real-time systems. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*, Denver, USA, Jun. 1998.

[34] Daniela Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On adaptive resource allocation for complex real-time applications. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, San Francisco, USA, Dec. 1997.

[35] D. Silva and K. Schwan. Ctk: Configurable object abstractions for multiprocessors. Technical Report GIT-CC-97-03, College of Computing, Georgia Institute of Technology, 1997.

[36] D. Silva, K. Schwan, and G. Eisenhauer. Configurable distributed retrieval of scientific data. In *Second International Conference on Configurable Distributed Systems (CDS'98)*, Maryland, May 1998.

[37] Sandeep K. Singhal, Binh Q. Nguyen, Richard Redpath, Michael Fraenkel, and Jimmy Nguyen. Building high-performance applications and servers in java. In *ACM SIGPLAN Conference On Object-Oriented Programming Systems, Languages and Applications*, Atlanta, Georgia, October 1997.

[38] Markus Straber, Joachim Baumann, and Fritz Hohl. Mole-a java based mobile agent system. In *ECOOP '96 Workshop on Mobile Object Systems*, 1996.

[39] J.S. Vetter and K. Schwan. High performance computational steering of physical simulations. In *Proc. IPPS 97*, 1997.

[40] VizAD. Space science and engineering center university of wisconsin - madison. http://www.ssec.wisc.edu/ billh/visad.html.

[41] Glen H. Wheless, Cathy M. Lascara, Donald P. Brutzman, William Sherman, William L. Hibbard, and Brian E. Paul. Chesapeake bay: Interacting with a physical/biological model. *IEEE Computer Graphics and Applications*, 16(4), July/August 1996.

[42] J. White. Mobile agents. Telescript Technical Whitepaper, General Magic, Inc., oct. 1995.

[43] et. al. Wollrath. A distributed object model for the java system. *Computing Systems*, 9(4):265–290, 1996.

[44] Rich Wolski. Dynamically forecasting network performance to support dynamic scheduling using the Network Weather Service. In *Proceedings of 6th High-Performance Distributed Computing (HPDC6)*. IEEE, 1997.

[45] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, January 1997.

[46] B. Zuerndorfer and G. A. Shaw. Sar processing for rassp application. In *Proceedings of 1st Annual RASSP Conference*, Arlington, VA., Aug. 1994.