Wax: A Wide Area Computation System

Peter D. Stout

December 1994

CMU-CS-94-230

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This research was sponsored in part by the Advanced Research Projects Agency under contract number DABT63-93-C-0054 and in part by the Intel Corporation under an Intel Graduate Fellowship.

Keywords: Distributed Systems, Parallel Processing, Networks

Abstract

This dissertation explores the use of machines connected to wide-area networks to provide the parallelism needed to work on large coarse-grain parallel applications. Large applications, such as circuit simulations, integer factoring, graphics applications, and NP-complete approximations, have potential coarse-grain parallelism that exceeds the parallel processing facilities at any single site. At the same time, the machines connected to wide-area networks represent a large and growing source of under-utilized computing power. However, in order to transform the idle machines on a wide-area network into a large scale, wide-area distributed computing system, a number of problems need to be addressed: latency, failure, security, process management, scale, and system administration. This dissertation describes a prototype wide-area distributed computation system, called *Wax*, that has been built and used to explore solutions to these problems.

Wax implements a programming model that takes into consideration the problems that exist in distributed computing environments, particularly wide-area ones. The key differences from traditional parallel programming models are a reduced global consistency guarantee, the explicit incorporation of failure into the model, and reduced access to the local execution machine. The first two differences allow *Wax* to expose the effects of some these wide-area problems (failure, latency, scale) to the application programmer. The most appropriate solutions to these problems often require application-specific knowledge. The third difference allows *Wax* to incorporate mechanisms to protect the resources used to process applications from abuse by those applications.

Instead of providing a global consistency guarantee, which not all applications require, *Wax* handles some of the other wide-area problems (processor failure, process management) that existing distributed computation systems leave to their users. By deemphasizing consistency, *Wax* is also able to provide applications with greater potential parallelism.

Since the *Wax* programming model differs from those supported by existing distributed computation systems, the dissertation also identifies a class of applications that can be solved using the model. In addition, the dissertation reports the performance of *Wax* implementations of several applications from the general class.

Acknowledgements

I would like to thank the many people who have helped to make this dissertation possible. I will start out by thanking my advisors, in reverse chronological order: Brian Bershad, Eric Cooper, and Alfred Spector. All of whom somehow managed to leave CMU before I did.

The rest of my committee (Doug Tygar, Adam Beguelin, and Mark Manasse) contributed a variety of useful advice while I was writing my dissertation. In addition, Jeannette Wing provided helpful feedback on the definition of the *Wax* programming model. In addition to my committee, Beth Bottos and Hugh Stout have provided editorial and proofreading advice for various drafts of this dissertation.

Wayne Sawdon, Mary Thompson, and Matt Zekauskas made their primary workstations accessible to *Wax* and provided feedback on how the system affected their day-to-day use of those machines. I am also indebted to John Mount for providing me with a concrete demonstration of the usefulness of my work, *and* some very pretty pictures.

Susan Hinrichs, Hugo Patterson, Tom Warfel, John Hampshire, Conal Elliott, and Mark Derthick helped over the years to make our office a pleasant place to spend the long hours that are part of graduate school. I would also like to thank Garth Gibson for providing the timely advice that landed us in our current location.

I would also like to acknowledge the support of the members of the Camelot project when I first started at CMU. They helped smooth my transition from chemist to computer scientist. One member of the project deserves special recognition: Lily Mummert. Over the years she has provided a great deal of advice, feedback, and motivational support.

Outside of Wean Hall I would like to thank the folks on the various CS softball teams who have helped me while away many a summer weekend. While at CMU I have belonged to a couple of cooking groups that have helped to keep me well fed and given me the opportunity to experience many new dishes. Among the people who have participated in these dinner groups are Bill Webster, Kim Ginther-Webster, Lee Ann Patterson, Joanne Karohl, and Randy Dean. Several people have not only helped feed me, but have been willing to share a home with me: Brian Zill, Christian Lebiere, Vince Cate, and Allan Heydon.

Finally, a special thanks to my family and my fiancee. All four of my parents (Virginia Stout, Bill Sieverling, Hugh Stout, Kit Ellis) have been a source of support, advice, and inspiration for many years. My brother, Eric Sieverling, provided me with a demonstration of courage and self-awareness that I value far more than I think he realizes. And finally, without the moral support of Beth Bottos, my fiancee, I am not sure that this work would ever have been finished.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large applications, such as circuit simulations, integer factoring, graphics applications, and NP-complete approximations, have potential coarse-grain parallelism that exceeds the parallel processing facilities at any single site. The largest multi-processing supercomputers have on the order of 1000 nodes, but such machines are not common. My thesis is that the machines connected to a wide-area network can provide the parallelism needed to work on large coarse-grain parallel applications. In order to prove my thesis, I have built a prototype wide-area distributed computing system called *Wax*. My work provides a number of contributions in the area of distributed and parallel computing:

- I have identified a high capacity computing resource and the problems that must be addressed in order to turn the raw resource into a useful multiprocessor.
- I have designed a programming model that recognizes many of the problems which exist in a distributed computing environment and is, therefore, feasible to implement and use in such an environment. Since the model differs from those supported by existing distributed computation systems, I have also identified a class of applications that can be solved using the model.
- I have built and evaluated a prototype wide-area distributed computing system, called *Wax*.

The remainder of this introduction provides an overview of the research I performed to demonstrate my thesis and a road map through the remaining chapters of my dissertation.

The impetus for this thesis was the realization that wide-area networks represent a large and growing source of under-utilized computing power. More and more workstations and personal computers are being connected to local-area networks, which are themselves being linked to wide-area networks, such as the Internet. As of January 1994, the Internet connected 2.2 million machines and had a growth rate of about 5% a month [Lot94]. In addition, surveys [DO89, Nic87, TLC85] of local-area networks have shown that many of the machines connected to such networks are idle at any given time.

In order to transform the idle machines on a wide-area network into a large scale, wide-area distributed computing system, a number of problems need to be addressed:

- *Latency.* The communication links that make up a wide-area network tend to have low throughput and high latency. The communication speeds of wide-area networks range from .1 megabits per second (Mb/s) to 5 Mb/s, as compared with 10 Mb/s to 1000 Mb/s for local-area networks and traditional multiprocessors. The physical separation of machines on wide-area networks provides a lower bound on the communication latency is approximately 1ms per 200 miles. In general, latency is higher than this lower bound due to delays introduced by intermediate network nodes.

- *Failure.* Machine and network failures happen regularly. The mean-time-to-failure (MTTF) for a host on the Internet is approximately 18 days [LCP91]. The MTTF of a distributed computation decreases linearly as the number of machines that are used increases. Furthermore, in a system that only uses otherwise idle resources, computations will also be interrupted when a previously idle machine is used locally. Any interruptions or failures must be handled without requiring that the entire computation be restarted.
- *Process Management.* This problem can be divided into two pieces: resource identification and resource allocation. Resource identification covers the mechanisms that allow application programmers to identify which machines can be used by a computation, and resources providers to control which machines can be used by the distributed computing system. The mechanism used by resource providers will typically have both a static and a dynamic part. The (relatively) static part defines the set of machines that can potentially be used, while the dynamic part determines which of those machines is currently available. The definition of "available" is a major policy issue for a distributed computing system. The resource allocation issues include how to assign machines amongst multiple computations and what to do when a failure occurs. The latter issue, at least in the case of network failures, can be viewed a computation competing with itself for resources.
- *Security.* The distributed computation system must protect the resources that people and organizations are making accessible for use in a wide-area computation. For small systems running on private networks with all machines belonging to one group, this issue may not be very important, but as the system grows larger (used by more people or multiple organizations) they become critical. Mechanisms are needed for preventing accidental or malicious attacks on the resources that are local to the remote machines. There should also be a means of tracking where a job has executed, so that problems can be identified if necessary. Protecting the applications that are being processed by the computation system is harder (impossible), because the remote machine that is executing the computation has complete control over that application.
- *Scale.* Many of the proceeding problems become harder as the number of machines involved grows. A large-scale distributed computation system should be able to manage thousands of processors; individual computations might use hundreds of processors. To be able to grow to such size, the mechanisms used to solve the problems in the distributed computation system must scale appropriately.
- *System Administration.* The structure of a distributed computation system must not conflict with the autonomy of the various organizations whose machines are connected to the system. Within one organization, there may exist multiple levels of control, for example: company/university, a department, a group, and the primary user of a machine. A distributed computing system needs to allow resource providers to quickly and independently recover the use of their resources from the distributed computing system.

With more and more machines being networked together, these problems can arise even in inter-networks, such as the one at CMU, that geographically one thinks of as being local-area networks. Therefore, in this thesis, I will use “wide-area distributed computation system” to refer to any distributed computation system that must handle two or more of these problems.

Chapter 2 examines how these problems have been handled in existing distributed computation systems. While these systems are a source of ideas, the direct applicability of their solutions is unclear, because the systems are, in general, intended for use by medium-sized groups of people sharing a common pool of resources on a local-area network. The larger, more public, and less secure environment of a wide-area, distributed computing system requires solutions that are more tolerant of failures, more security conscious, and more scalable. This thesis provides a stepping stone across the gap between the existing distributed computation systems and a large scale, wide-area, distributed computation system that would allow programmers to exploit the computing resources of the Internet as they now do the data resources of the Internet [Cat92, Kah91, Sch89].

One of the stepping stones provided by my thesis is a programming model that reflects the problems that exist in distributed computing environments, particularly wide-area ones. The key differences in the model, as compared to traditional parallel programming models, are a reduced consistency guarantee, the explicit incorporation of failure into the model, and reduced access to the local execution machine. The first two differences allow an implementation to expose the effects of some problems (failure, latency, scale) to the application programmer. Another effect of the second difference is that an implementation of the model must handle some of the problems (processor failure, process management) that the existing distributed computation systems leave to their users. The third difference allows an implementation to incorporate mechanisms to protect the resources used by the implementation. Chapter 3 describes the model in more detail.

In Chapter 4, I describe the implementation of the prototype system, *Wax*, that I built to demonstrate the utility of my programming model. *Wax* also provides a means of exploring the security, process management, and resource management issues raised by large scale distributed computing. In particular, *Wax* incorporates a number of mechanisms that allow resource providers to control when and by whom their resources are used and to protect those resources while they are being used by *Wax*. *Wax* has been in regular operation at CMU and the University of Washington for the past nine months.

What is *Wax*? To the application programmer, *Wax* is a library of routines that, when linked into an application, (potentially) allow the programmer to execute pieces of the application in parallel and to share data between those pieces. To the user of such an application, *Wax* looks much like a traditional batch system. The user submits a job to *Wax* for processing, the system schedules the job for execution, and then returns any output. Behind these interfaces are a collection of system processes that make *Wax* function (see Figure 4.1).

In Chapter 5, I identify the class of applications that can be solved with this model. The applications in the class generally have a large amount of coarse-grain parallelism and do not depend much on global consistency. The class includes applications such as independent distributed simulation [JCRB89], graphics and film animation, many probabilistic algorithms, and integer factoring [LM89], that require no communication between the elements of the computation. It also includes applications, such as the quadratic assignment problem [PC89], that share intermediate results in order to reduce the amount of computation. Chapter 5 also sketches out possible *Wax* implementations for several applications, and provides a framework that application programmers can use to evaluate the feasibility of implementing a particular application using the *Wax* programming model.

In Chapter 6, I evaluate the contributions of my thesis by examining *Wax*'s performance for several applications and its impact on resource providers. I also explore the lessons that can be learned from having built *Wax*, and suggest some future work along the path towards the goal of building a large scale, wide-area, distributed computation system.

I conclude the thesis by summarizing my results and contributions in Chapter 7.

Chapter 2

Related Work

The earliest distributed computation systems ran on the Arpanet in the late 70's. One of the first was *Creeper*, which moved between Tenex machines as it printed a file. The "Worm" programs [SH82], built by Shoch and Hupp at Xerox PARC in the early 80's, created and used the first general-purpose distributed computation system. The Cambridge Processor Bank [NH82] was another early system to support remote execution. It dynamically assigned processors selected from a pool of machines to users when they logged in.

A number of distributed computation systems have been developed since the early 80's. These systems can be divided into general-purpose systems, which support the execution of arbitrary application code, and special-purpose systems, where the remote execution mechanism is an integral part of a specific application. The next section provides a brief overview of a number of systems, which cover the range of possible implementations. Four criteria will then be used to examine how *Wax* fits into the existing body of research: programming model, failure model, process management, and system security.

2.1 Current Systems

Distributed computation systems have existed for many years. They have been implemented both as user-level tools, and as part of operating systems. Most of the current distributed computation systems are user-level implementations. Two exceptions are discussed in this section: *Sprite* [DO89] and the *V* system [The86, TLC85]. The user-level implementations tend to be more portable, since many operating systems support similar application programming interfaces. On the other hand, direct access to the operating system internals can make some features easier to implement, such as process migration.

All of the user-level implementations have system processes that run on any machine that is to be accessible to the computation system. These system processes typically monitor the status of the machine and handle task creation and termination. The user-level implementations also limit users to machines for which they have login accounts.

The rest of this section provides brief sketches of *Wax* and a number of the other existing distributed computing systems. These sketches will serve as a basis for comparing some of the important features, such as programming model, failure model, security, *etc.*, of the various systems later in the chapter.

2.1.1 Butler

The *Butler* system [Nic90] at Carnegie Mellon University allows users to access other machines in the same cell of the Andrew File System (AFS[1]) as their home machine. A special command is used to start a process on a remote machine with a specified machine type and software environment. The system starts the task if an appropriate machine is available in the pool of idle workstations. *Butler* does not provide mechanisms for monitoring tasks or automatically restarting tasks. In particular, if the owner of a machine returns, the process started by *Butler* is terminated by the local *Butler* daemon. AFS is used for data sharing, and provides a common execution and authentication environment.

2.1.2 Condor

Condor [LLM88], developed at the University of Wisconsin, allows users to build a multiprocessor out of the machines connected to a local area network. The machine from which the user starts a computation is the master processor. The system dynamically allocates other idle machines to serve as slave processors. The only supported inter-process communication mechanism is the file system on the master machine. *Condor* uses a special library to redirect file operations back to the master processor. This dependence on the master machine means that it must always be available. *Condor* can checkpoint and move tasks between remote machines when a slave processor is reclaimed by its owner.

2.1.3 Marionette

The *Marionette* [SA89] system from the University of California at Berkley uses a master-slave model of computation and requires the user to define the set of available machines. It also provides facilities for using a collection of heterogeneous machines. Marionette provides shared data that can be updated by the master and read by the slaves. A special program is used to create and distribute the slave executables to the available slaves and then start the master process. The slave operations are automatically assigned to available processors by the run-time system. While *Marionette* does not support explicit process migration, it will restart an operation on another slave if a slave crashes.

2.1.4 PVM

PVM [GBD$^+$93] provides users with a "parallel virtual machine" made up of a heterogeneous collection of machines connected on a network. Users can program this virtual machine much like a conventional multiprocessor, as *PVM* supports both message-based communication and either distributed shared-memory (versions 1 and 2) or a sychronous blackboard (version 3).

A *PVM* application consists of one or more user defined *components*. When instantiated as a *PVM* process, a component will perform one piece of a computation, for example, solve a matrix. A component defines a mapping between a name and a set of executable images that can be used to create *PVM* processes, one for each machine type. When

[1] AFS is a registered trademark of the Transarc Corporation.

instantiating a component, the system selects an available machine from the pool defined by the executable images associated with that component.

2.1.5 Spawn

Spawn [WHH+92] is a distributed computing system developed at Xerox PARC. *Spawn* is designed to run on idle workstations connected in a local area network. It serves as both a source of computing cycles and as a test bed for exploring resource allocation policies. Computing resources are allocated using an auction mechanism.

A *Spawn* computation is structured as a tree of tasks, where each *Spawn* task is actually a pair of processes: an *application manager* and an *application worker*. The *application worker* performs the desired computation and transmits (implicitly or explicitly) the results to its corresponding *application manager*. This manager process then merges the results with those produced by any sub-tasks that it has spawned and passes the merged results to its parent manager. With an appropriate application manager, the application worker can be a program that does not directly interact with *Spawn*.

2.1.6 Sprite

The *Sprite* [DO89] network operating system, which was developed at the University of California at Berkeley, is designed to run on a local-area network. Every machine in the network runs an operating system kernel that provides shared access to a common distributed file system. The system supports two forms of task migration. The first allows users to explicitly start or move tasks to a remote machine. The second is used by the system to return a migrated task to its originating host when the workstation on which it is executing becomes busy. The system does not support the direct migration of tasks between remote machines.

2.1.7 Utopia

Utopia [ZWZD92], developed at the University of Toronto, is designed to balance the load on a heterogeneous collection of networked computers. The system does not directly provide any inter-process communication mechanisms, but depends on the availability of a distributed file system with a uniform name space on all machines.

Utopia is primarily intended to function in a local-area environment, but it has a clustering mechanism that can be used to extend the system to larger numbers of machines or more widely distributed groups of machines. The system load balances only at process startup. It does not provide any automated recovery mechanisms.

Utopia and its commercial derivative, *LSF*[2], provide a wide range of user interfaces. A user can access the system implicitly using a modified version of the Berkeley C-shell or explicitly from either shell scripts or C programs or by using a distributed batch system.

[2]LSF is a registered trademark of the Platform Computing Corporation.

2.1.8 V

V [The86, TLC85] is a network operating system developed at Stanford. The *V-kernel*, which implements processes, address spaces, and network transparent inter-process communication, runs on every machine in the network. Most traditional operating system services are implemented by servers outside of the kernel. These servers, and all other tasks, can move among machines to balance the work load.

2.1.9 Zilla

Zilla [Cra90] is a distributed computation system for networks of Next computers. To perform a computation, a user creates a "network" of machines, assigns a program to each one, and starts them. A master process is used to control the slave programs on the remote machines. NFS is typically used to collect the results. There are limited facilities for attaching non-Next computers to the *Zilla* network.

2.1.10 *Wax*

Wax allows programmers to use the computing power available in the idle workstations connected to local and wide-area networks as though it were a giant multiprocessor. *Wax* supports a programming model that provides a task heap-based execution model and a shared-data store that provides only limited consistency guarantees. The data store, called a blackboard, is an associative memory with information stored as (key, value)-tuples. With this programming model *Wax* is able to avoid performing global synchronization operations that can be costly in environments that have relatively slow, high latency, communication links and may involve large numbers of nodes. The system provides automated task management and uses process checkpointing to recover from failures and support process migration.

The *Wax* blackboard is loosely modeled after the tuple-spaces used in the *Linda* system [CG89]. The key difference between the two mechanisms is the consistency that they guarantee. *Linda* maintains a globally consistent tuple space, which allows tuples to be used for inter-task synchronization. *Wax*, on the other hand, only guarantees that individual data store read and write operations are atomic. Different *Wax* tasks may see different sets of keys in a blackboard and/or different values for a particular key.

In order to provide the organizations making their machines available to *Wax* with autonomous control of those machines, *Wax* is structured as a collection of cooperating domains. Each domain consists of machines belonging to one organization.

2.1.11 Special Purpose Systems

While there are no general purpose wide-area computation systems, there are several special purpose ones. These include network games, such as NetTrek, and the system built by Lenstra and Manasse [LM89] for factoring large numbers. NetTrek uses a central server to coordinate communication among a small number (up to 16) of clients. Each client maintains its own view of the system based on the the updates reported by the server.

The players must adapt to the delays and inconsistencies introduced by slow or congested network links.

Lenstra and Manasse's system uses available machines to attack one problem at a time. A master machine assigns parts of the factorization to participating hosts. Electronic mail is used for inter-host communication. Additional instances of the system must be run to work on more than one factoring problem.

Another important special purpose system was the Internet Worm [Spa88], which raised concerns among potential resource providers about the potential risks of supporting distributed computation. One concern is that an application could monopolize the resources on a machine or network. Another is that a distributed computation system could be used to examine or modify data that is not otherwise accessible across the network.

2.2 Programming Models

While the previous section described each system in isolation, this section, and the rest of the chapter, examines their commonalities and differences in a number of areas. The purpose of this examination is to determine both which features should be incorporated in large scale computation system, and which features may be difficult to support. The rest of this section focuses on the programming model supported by the various systems.

Existing distributed computation systems have typically attempted to make a collection of workstations connected to a local area network appear to be a single machine. They, therefore, support programming models that are similar to those used on uniprocessors and conventional multiprocessors.

A programming model defines a process or execution model, one or more inter-process communication (IPC) mechanisms, and a data consistency model. The process model determines how an application can be decomposed into pieces for scheduling and possibly parallel execution. The data consistency model defines guarantees about the ordering of process and IPC operations performed by an individual process and by processes executing in parallel.

2.2.1 Process Models

The existing distributed computation systems support a variety of process models. Each model is suited to different types of applications.

Butler, *Condor*, *Marionette*, and *Zilla* implement a master-slave model. In the master-slave model, the initial, master, process in a computation controls the execution of remaining (slave) processes. Slaves will generally communicate directly with only the master. This model is appropriate for computations where the various components are independent of each other.

Spawn, *Sprite*, *Utopia*, and *V* support a task tree model. The task-tree model is a generalization of the master-slave model that allows multiple levels in the process hierarchy. This is a natural process model for applications that use recursive decomposition to solve a problem, for example, branch-and-bounds searches.

PVM uses a peer-to-peer model. The peer-to-peer model treats all processes as equals, which makes it more flexible. In particular, any process can wait for the termination of any other, and communication links may exist between any pair of processes. A master-slave or task-tree process model can be implemented on top of a peer-to-peer system. The opposite, however, may not be true, because the termination of a non-leaf process in a master-slave or task-tree system may terminate, or disconnect, all processes below the dead process. A peer-to-peer model is best suited for computations where multiple processes are performing similar tasks on a common data store.

Wax uses a task-heap model. In the task-heap model a computation is divided into pieces that can be, but do not have to be, executed in parallel. There is no guarantee made about which, if any, of the tasks in the task heap will be executed in parallel. Applications using *futures* for speculative execution or probabilistic algorithms are well suited to this model.

2.2.2 Inter-Process Communication

The existing distributed computation systems also support a wide variety of inter-process communication (IPC) mechanisms. While *Butler*, *Utopia*, and *Zilla* do not directly support any IPC mechanisms, they all require the environment in which they are used to provide a distributed file system. A user of any of these systems could, in principle, implement any other IPC mechanism they needed. This dependence on a distributed file system is a possible impediment to using these systems across organizational boundaries, because cooperating organizations would need to support the same distributed file system. In addition, if authenticated access to the file system is desired, then some means of inter-organizational authentication would also be necessary.

IPC in *Condor* is also done through the file system, but *Condor* does not require a distributed file system. Applications are linked with a library containing special versions of the C input/output routines, which redirect file operations back to the master machine.

Marionette goes beyond file-based IPC and also provides a restricted form of distributed shared-memory, where only the master process may modify the shared-memory. *Marionette* provides both function-shipping and data-shipping mechanisms for updating the shared memory. All update operations are ordered relative to the invocation of "worker" operations. A worker operation performs an idempotent piece of the whole computation. In particular, a worker operation is defined to be a function of the arguments passed to the operation and the state of the shared data structures at the time the operation is invoked. Each slave process will typically execute many worker operations in the course of a computation. The state of a slave process only needs to be updated at the start of an operation. *Marionette* delays propagating updates of shared memory in order to eliminate intermediate updates.

Message passing is the primary form of IPC in *Spawn*. Tasks may communicate through a distributed file system if one is available.

PVM provides its users with the broadest range of communication options. While message passing is the recommended form of IPC, the system also provides distributed shared memory and permits tasks to access files.

2.2.3 Data Consistency

All of the existing distributed computation systems provide consistency guarantees that are similar to those provided on uniprocessor and traditional multiprocessors. While strong data consistency guarantees make it easier for programmers to reason about the behavior of their programs, they come at a cost: additional processing, communication overhead, and extra delays to maintain event ordering. These costs can limit the parallelism available to applications. Furthermore, the performance cost of maintaining data consistency is often exacerbated by the failure of the machines and networks being used by a computation.

Wax provides more limited consistency guarantees: different processes can observe different views of the shared data store, and individual processes can be executed more than once. Limited blanket guarantees and mechanisms that allow programmers to enforce consistency as necessary make it possible for *Wax* to maximize the parallelism available to applications.

2.3 Failure Models

Distributed computation systems generally ignore the issue of failure. Given that they were designed for use in local-area networks, where failures are infrequent, this decision is neither surprising nor unreasonable. All of the systems, except *PVM*, are vulnerable to the failure of a single process: the master process or the root of the task tree.

Only *Condor* and *Marionette* have mechanisms for recovering from the failure of slave processes. The ordering of memory updates and task invocations used by *Marionette* to implement its distributed shared memory is also used to recover from the failure of a slave. If a slave crashes, the necessary application state is restored in another slave by replaying the recorded memory updates.

Condor periodically writes checkpoint records so that it can quickly release resources when a machine is reclaimed. These checkpoint records are stored on the machine running the master process. If a machine crashes, *Condor* can recover the interrupted process by shipping the most recent checkpoint record to another machine. The *Condor* programming model leaves no residual dependencies on previously used slave machines.

The remaining systems only provide mechanisms to monitor the status of executing processes. They leave process recovery entirely to the application programmer. Any application-specific failure recovery mechanism for the systems using master-slave or task-tree process models would have to be incorporated into the master or root process.

A *PVM* application can be made failure tolerant by adding a task management component. An instantiation of this component monitors the status of the other processes working on the computation and initiates appropriate recovery code when a failure is detected. Multiple instances of the task management component are needed to avoid having a single point of failure. In PVM versions 1 and 2, this mechanismc could only handle the failure of a *PVM* user process. In those versions of *PVM*, the death of a *PVM* system process would cause the remaining system processes to exit.

2.4 Process Management

A basic requirement of a distributed computation system is the ability to locate an available machine and to start a task on that machine. All of the existing distributed computation systems provide this; however, the definition of "available" varies from system to system. In *Butler*, *Condor*, *Sprite*, *Wax*, and *Zilla* a machine is available if no one is using the machine's console and the system load is below a certain threshold. *Marionette*, *PVM*, *Utopia*, and *V* always consider machines to be available. While *Utopia* and *V* will choose the least loaded machine, *Marionette* and *PVM* use round-robin assignment. In *Spawn*, an auction mechanism is used to allocate idle machines.

Condor, *Sprite*, *V*, and *Wax* also support process migration. *Condor* and *Wax* migrate tasks to other available machines, when the machine currently being used by a task becomes busy or crashes. *Sprite* returns a task to the machine where it was created, when the current machine becomes busy. *V* uses process migration to balance the load amongst the pool of available processors.

2.5 System Security

The security requirements of distributed computation systems can be divided into two broad categories: protecting resources and protecting the system's users. In principal, the resources connected to a distributed computation system should be as secure as they are in the absence of that system. To achieve this goal, resource providers must be able to control who can use their resources. In addition, a task executing in the computation system must have no more ability to attack the machine on which it is executing than if it were executing elsewhere in the network.

2.5.1 Protecting Resources

Protecting the resources used by a computation system has usually been a secondary consideration in the design of such systems. The existing systems have tended to either ignore the problem or rely on the security mechanisms of the underlying operating system.

Condor, *Marionette*, *PVM*, and *Zilla* users must have a login account on every machine that they use. The executing processes then have whatever privileges the user has on that particular machine.

Butler relies on the security provided by AFS. Given that *Butler* was designed to run in clusters of public workstations where all personal files are stored in AFS and users have accounts on all machines, this is probably a reasonable decision.

The mechanism used by *Condor* to relay file operations back to the host running the master task leaves that machine open to attack from any machine that is currently executing a slave process. The proxy process on the master machine that performs file operations does not distinguish Condor slave processes from other process executing on slave processors.

A problem with relying on the operating system for security is that an operating system consists of a large amount of code and has many interfaces. Ascertaining that applications cannot exploit security weaknesses in that code or those interfaces is difficult. As a result, the absence of system-specific security measures leads potential resource providers to worry that connecting machines and networks to a distributed computation system makes the resources more susceptible to attack.

Wax incorporates a number of mechanisms to protect the resources it makes available. In particular, Access Control Lists (ACLs) are used to protect all machines used by the system. In order to be able to enforce the access controls, a mutual authentication protocol is performed when communication links are set up. For most *Wax* communication links, the authentication protocol is implemented as part of a communication package that is similar to the secure remote procedure call (RPC) mechanism developed at Xerox PARC by Birrell [Bir85]. *Wax* also logs where all computations are processed, so as to provide an audit trail should problems arise.

2.5.2 Protecting the User of a Computation System

Protecting the user of a distributed computation system is harder than protecting resources. It is currently beyond the scope of a distributed computation system to provide an execution environment that is as secure as a single machine connected to a network. Just as a user is ultimately forced to trust the machine to which they are directly connected, they are also forced to trust all of the machines that are used to process their computation. While a task is executing, the host on which it is running has complete control over all data that is accessible to that task.

At present, the only way to protect tasks from attack is to allow the users of a distributed computation system to control which machines are used to process a computation. With this ability, users can choose the right balance between security and computing power. To facilitate tracing problems that are discovered after a computation has been processed, it should also be possible to trace where the various pieces of the computation were executed.

The development of secure co-processor systems, such as the Dyad system from CMU [TY91, Yee94], offers the prospect of a future network environment in which a distributed computation system will be able to provide better protection for both resource users and resource providers. A secure co-processor system uses a physically secure component (a special chip or card), containing a processor and some non-volatile memory, to bootstrap a secure execution environment on the primary processor of a standard PC or workstation. In an environment where co-processors are in use, a distributed computation system could be configured to use only those machines that are known to running secure operating systems or users could be allowed to specify that a computation should only use such machines. In addition, a secret key stored in the co-processor could be used to uniquely identify each of the machines used by the distributed computation system.

2.6 How Wax Compares

Figure 2.1 lists the major features provided by each of the distributed computation systems discussed in this chapter. Two major features distinguish *Wax* from all of the other distributed computation systems: integrated security mechanisms and better tolerance for failures. The ability of *Wax* to tolerate failures is in part due to the limited consistency guarantees of its shared data store. While none of the existing systems provides the same

	Butler	Condor	Marionette	PVM	Spawn	Sprite	Utopia	V	Wax
Remote Execution	x	x	x	x	x	x	x	x	x
Process Migration		x				x		x	x
File Access	x	x	x	x	x	x	x	x	1
Message-based IPC				x	x	x		x	
Shared Data Store			x	x					x
Machine Identification by Name		x	x	x	x	x	x		x
Machine Identification by Feature	x			x					x
Recovery from Process Failure		2	2	3					x
Lacks Single Point of Failure									x
Integrated Security									x

Figure 2.1: This table lists the features provided by the distributed computation systems discussed in this chapter. An x indicates the system supports the feature, while a blank means the feature is missing. A 1 means that the system emulates file access on top of its native shared data store. A 2 means that the system automatically recovers from the failure of slave processes, but not the master process. A 3 means that the system provides mechanisms which allow the application to be notified of failures.

set of services as *Wax*, each of *Wax*'s other features has been implemented in one or more of the other systems.

Chapter 3

Programming Model and Interface

This chapter describes the programming model seen by users of *Wax*. The chapter starts with an overview of the features of the programming model. It then discusses some of the features that the model lacks. Following this overview is a description of failure-free behavior of the operations provided by the model. The last part of the description is a discussion of how applications can synchronize the execution of component tasks, given the lack of a global consistency guarantee. The remainder of the chapter focuses on the existence of failures is incorporated into the model.

As was described in the previous chapter, distributed computation systems that support traditional multiprocessor programming models do not deal well with failure, and will have difficulty scaling to large numbers of processors. The *Wax* programming model was specifically designed to allow the implementation of a computation system capable of tolerating the latency and failures that occur in distributed systems. There is a clear feedback loop between the model and the implementation. As expected, the model drives the implementation, but some features of the model are present strictly because they admit a reasonable implementation. For example, the *Wax* data model does not automatically guarantee that data written by one machine will be instantly, or even ever, seen by another machine. Instead, the system implementation distributes writes through to other machines in the network as resources permit. Where synchronous distribution of data is required, *Wax* provides interfaces that enable it, but with a cost that is clearly exposed to the application programmer. There are also features in the model that are included to handle real-world problems that an implementation would face. This applies, in particular, to the inclusion of access controls.

3.1 Basic Concepts

The *Wax* programming model supports a number of concepts that can be applied to parallel distributed applications:

- processors and memory to execute tasks
- automated process scheduling and management
- a namespace for shared data
- atomic read and write operations

Object	Operations
Job, Task	spawn, list, kill, exit, status
Blackboard, Tuple	read, write, delete, directory, fetch
ACL	initialize

Table 3.1: Summary of operations that can be performed on each of the objects defined by the *Wax* programming model.

- mechanisms for controlling the distribution of shared data

The model does not, however, provide several features that are common in other distributed programming models:

- a guarantee that processes will have consistent views of the shared data store
- at-most-once or exactly-once execution of processes
- a guarantee that processes will execute in parallel
- unrestricted access to the processors used by the application

The *Wax* programming model includes five basic objects: job, task, blackboard, tuple, and ACL. A *job* represents a single distributed parallel computation. As shown in Figure 3.1, each job consists of a set of *tasks*, each of which is a basic piece of the computation, and a set of *blackboards*, which are used to share data among the tasks in the job. A job is the *Wax* unit of resource allocation, somewhat analogous to a UNIX[1] process. A task on the other hand represents a locus of execution, and is equivalent to a thread in many other parallel programming models. A blackboard is an associative memory with information stored as *tuples*, each of which consists of a unique key and a set of values. The set of values associated with a key can be arbitrarily large, as can each of the values in that set. An ACL allows the owner of an object to restrict access to the object. For each job, there is a *source* that creates the job and initializes its state using resources, such as files and user input, that are outside the scope of the *Wax* programming model. Similarly, each job has a *sink* that consumes the results produced by the job.

The basic model of a *Wax* computation is a loosely-coupled collection of tasks that are cooperating to solve a problem. Tasks and blackboard data diffuse out from where they are created. As tasks execute, their results diffuse to other tasks. Eventually, the diffusion process will return the results of a computation to its creator.

Table 3.1 summarizes the operations that can be performed on blackboards, tasks, and ACLs. The operations will be described in greater detail later in the chapter.

Figure 3.2 shows a general pseudo-code framework for a *Wax* application. The function *Source* is the piece of the application that is executed by the source of the job. It creates the job and sets up its initial state. The function *Task*, which is executed by each task in the job, performs the actual work of the job. If a task is initially unable to read the input data, it uses the fetch operation to actively force the system to move the data. Using the

[1]UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

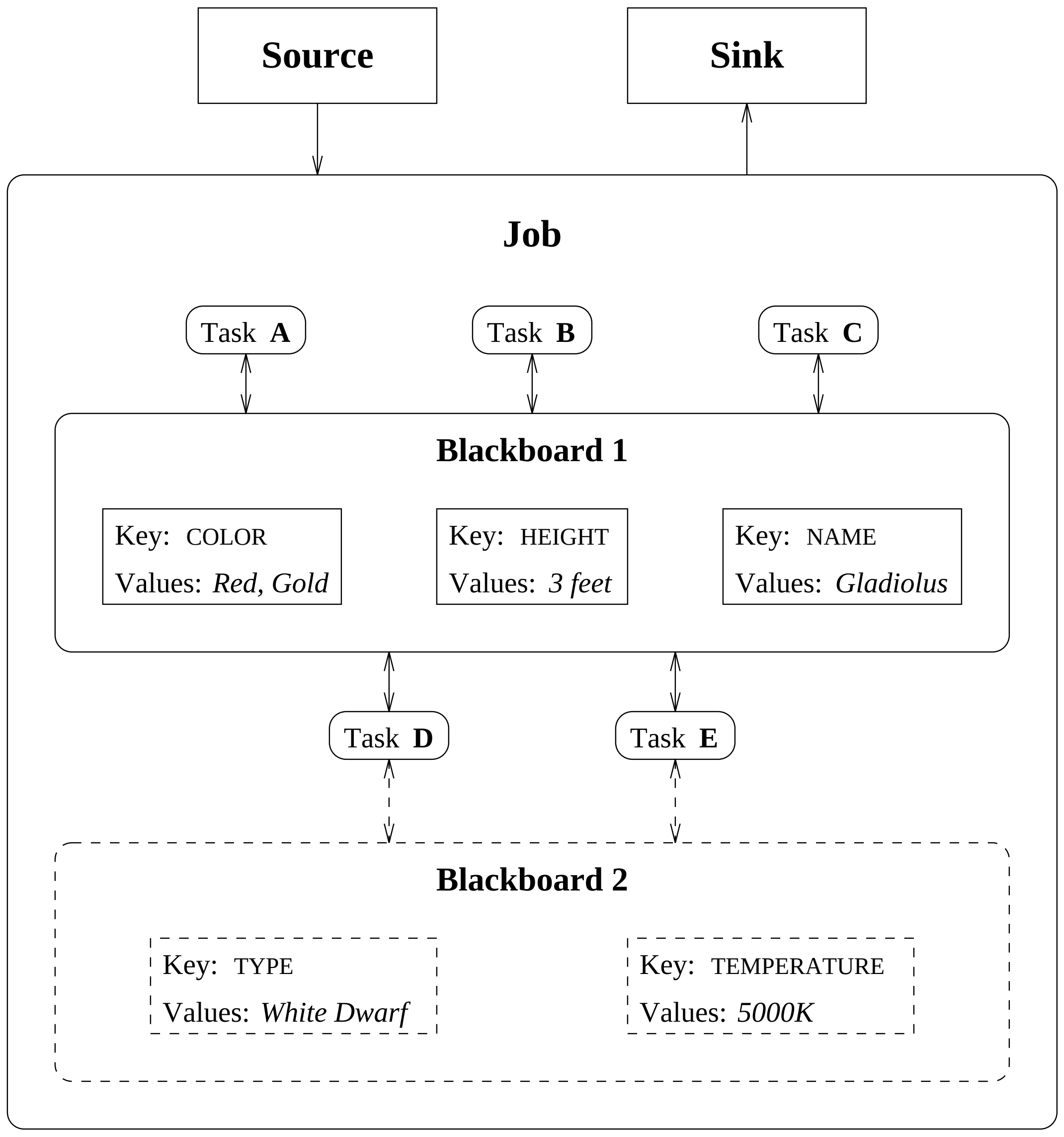

Figure 3.1: Logical structure of a *Wax* job. A *Wax* job corresponds to a single, loosely-coupled, parallel distributed program made up of one or more tasks, where each task is a unit of work that can be independently scheduled on a computer. Within a job, tasks interact through shared data stores, called blackboards. Each job has access to at least one blackboard. Information on a blackboard is stored as tuples, each of which has a unique key and one or more values.

```
Source()
{
  Create job and blackboard;

  Write application input to blackboard;

  Partition problem;
  For each piece of problem {
    Spawn a Wax task;
  }
}

Task()
{
  Process arguments to identify assigned portion of the computation;

  Read input;
  If input not found {
    Fetch input;
    Read input;
  }

  While not done {
    Compute;
    If assigned portion is too large {
      Divide portion and spawn additional tasks for each piece;
    }
    Write output to blackboard;
  }

  Call exit;
}

Sink()
{
  While computation not done {
    Wait for new output to appear;
    Read new output;
    Combine new output with existing output;

    Decide whether computation is done;
  }

  Delete job and blackboard;
}
```

Figure 3.2: A pseudo-code example of a *Wax* application.

fetch operation is a better choice than waiting passively for them, because while the task is waiting for the data it is tying up a processor without performing productive work. After reading the input, the task computes the assigned problem and writes output to the blackboard. If a task decides that its problem is too big, it can spawn additional tasks to spread the workload. The *Sink* function collects the output produced by the tasks within the job. The example sink function waits passively for the system to make the task output available, because the data may not be written immediately. Each fetch operation, regardless of whether it moves any user data, consumes communication resources, which are limited in any real world implementation of the model. If the termination criteria for the application depend on the status of the tasks within the job, the sink operation can execute list and status operations to retrieve that information. Additional examples are described in Chapter 5.

3.1.1 Execution Model

A job defines the state that is visible/accessible to the tasks within that job. The blackboards associated with a job are the only shared data stores available to the tasks within the job. A task may only interact with other tasks of the same job through a blackboard.

Limiting the resources accessible to a task allows an implementation of the model to restrict the interface that a task has to the machine executing it. In particular, the machines used to execute *Wax* tasks are only visible to those tasks as sources of processor cycles and memory (physical and virtual). This restricted interface allows, for example, an implementation of the model to provide tasks with no access to the file systems accessible from the machines used to process them. Such a restriction helps convince resource providers that the system cannot be used to snoop or modify data that should otherwise be inaccessible to the creator of the job.

In the *Wax* programming model, tasks are automatically assigned to processors for execution. This relieves programmers of having to identify a set of processors to use and of having to monitor the status of each of those processors. For the same reasons that tasks are not guaranteed consistent views of a blackboard, the programming model does not define the order in which tasks will be executed. In particular, tasks may be executed in a different order from the one in which they are created. In addition, they may, or may not, be executed in parallel.

The execution order is undefined, because ordering task execution implies synchronization with all its associated costs and sensitivity to failures. In particular, some of the synchronization required to implement any task ordering would occur before scheduling a task for execution can be done. If a communication failure occurred during such synchronization, some or all of the processors being used by *Wax* might sit idle, even though there were tasks waiting to be executed. Without a defined execution order, tasks can continue to be processed as long as there are compatible processors available.

A side effect of not defining the execution order of tasks is that a job that has inter-task dependencies may deadlock. For example, if tasks **A** and **B** each depend on intermediate output from the other, and *Wax* has only one processor available, the job may never terminate. However, if the number of dependencies is small relative to the number of available processors, deadlock is unlikely to occur. If necessary, an application can use the list and directory operations to monitor the global progress of the computation and to detect deadlocks.

3.1.2 Data Model

Blackboards provide a distributed program a convenient mechanism for naming shared data and communicating between tasks. Because real distributed programs consist of asynchronously executing processes connected by fallible, finite-speed communication links, the *Wax* programming model does not guarantee that all tasks within a job will have consistent views of the tuples on a blackboard. Allowing a blackboard implementation to be inconsistent makes the implementation scalable and fault-tolerant. It is scalable because the blackboard can be allowed to become “more” inconsistent as the system grows. It is fault-tolerant because a failure (network or processor) merely results in another inconsistency in the contents of the blackboard; some processors may observe updates, while others may not. Application writers are responsible for handling any inconsistencies observed by their jobs.

The programming model allows a job to have multiple blackboards, so that in a distributed implementation of the model, a large data set can be stored in a blackboard that is shared by several jobs. This would allow a job to benefit from any data distribution and caching that was done during the execution of previous jobs. In addition, if such a shared blackboard is used only as a source of input, the blackboard could be write-protected. This would prevent the machines processing the *Wax* tasks that need to read the blackboard from modifying the data it contains.

The only operations for which the *Wax* programming model provides any atomicity guarantees are reading or writing blackboard tuples and spawning a new task. Read operations are guaranteed to return only values that have been previously recorded by a write operation. The write and spawn operations are atomic in that other tasks will never be able to examine the contents of the blackboard or job and observe a state in which such an operation is partially performed.

3.2 Controlling Execution

The *Wax* programming model supports creating, monitoring, and terminating tasks. The source and sink for a job must specify the job being examined or modified as an explicit argument, because they are outside of the *Wax* programming model and may therefore be able to access multiple jobs. For *Wax* tasks, on the other hand, the job is implicitly specified, since a task belongs to exactly one job and is only able to interact with tasks and tuples that are part of that job.

The *spawn* operation creates a new task in the specified job. The arguments to the operation are the program that implements the functionality of the task, the arguments that should be passed to the task when it is started, and the ACL for the new task. The operation records the new task in the meta-data for the job and returns a unique identifier for the new task. The operation will fail and signal an error if the invoker of the operation does not have permission to create new tasks.

The *list* operation returns a list of the tasks in the specified job. The operation will fail and signal an error if the invoker of the operation does not have permission to examine the job.

The *kill* operation takes a task as an argument and records that the specified task has terminated. If the task is currently active on a processor, then the system may halt its execution. The operation will fail and signal an error if the invoker of the operation does not have permission to kill tasks.

The *exit* operation halts execution of the task that invokes the operation and records its termination in the meta-data for the job.

The *status* operation returns the current status of the specified task. If the returned status indicates that the task has called exit, then this operation guarantees that a fetch operation has been successfully executed for all tuples written by the specified task and that all tasks spawned by the specified task can be observed by invoking a list operation. The operation will fail and signal an error if the invoker of the operation does not have permission to examine the job.

3.3 Manipulation of Data

The operations provided by the *Wax* programming model for manipulating blackboards can be divided into two groups. The first group (read, write, delete, directory) consists of the operations that examine or modify the contents of a blackboard. The *write* and *delete* operations modify the contents of the blackboard by adding or removing tuples, respectively. The *directory* and *read* operations return information about the contents of the blackboard namespace or of a tuple, respectively. The second group, which consists of the fetch operation, provides the application programmer with a tool to combat the inconsistencies that can arise in a real world implementation of the model. All of the operations require a blackboard to be passed as an argument. The additional arguments and effects of each operation are described in the remainder of the section.

3.3.1 Tuple Operations

The *write* operation is passed a key and value to be added to the specified blackboard. If a tuple with the specified key does not exist in the blackboard, then a new tuple is created with the specified key and the set of values initialized to contain the specified value, otherwise the specified value is added to the set of values in the tuple. The operation will fail and signal an error if the invoker of the operation does not have write access to the blackboard.

The *read* operation takes a key as an argument and returns one of the values from the tuple in blackboard with a matching key. The operation is non-destructive. If the matching tuple has multiple values, then the choice of which value to return is left to the implementor of the model. The operation will fail and signal an error if either the invoker of the operation does not have read access to the blackboard, or the specified key is not found in the blackboard.

3.3.2 Blackboard Namespace Operations

The *delete* operation takes a key as an argument. The operation removes the tuple with the specified key and all of its associated values from the blackboard. The delete operation is intended to provide jobs with a mechanism for handling the real world problem that there is only finite space available to store tuples. The operation will fail and signal an error if either the invoker of the operation does not have delete access to the blackboard, or a matching tuple is not found in the blackboard.

The *directory* operation allows its caller to determine what tuples exist in the blackboard, and to wait for new tuples or values to be written. The operation takes a regular expression and returns the keys from all accessible tuples whose keys match the given regular expression. This operation does not cause data to be moved. The operation will fail and signal an error if the invoker of the operation does not have read access to the blackboard.

3.3.3 Data Distribution Operations

The *fetch* operation takes a regular expression as an argument. This operation allows the invoker to make sure that it can access the tuples whose keys match the given regular expression that were produced (written, but not deleted) by tasks that the invoker can observe with the list operation (see Section 3.2). These semantics allow a job to define a partial ordering of blackboard operations, without requiring that an implementation of the model provide consistent global snapshots of a blackboard. A fetch operation will fail and signal an error if the invoker of the operation does not have read access to the blackboard.

The *Wax* programming model does not provide a blackboard flush operation, which would push data away from the invoker, rather than pull it as the fetch operation does, because correctly implementing such an operation might require global synchronization prior to executing a task. Consider, for example, an implementation that caches the contents of a blackboard in multiple locations. The problem arises when such an implementation decides to use an additional cache. Before starting a task using that cache, the implementation would have to identify all flush operations that were performed by other tasks that could have known of the existence of the new task, and ensure that the cache contains all of the data that was distributed by those operations. An implementation might, however, provide programmers with a flush operation that could be used to indicate to the implementation which tuples the job considers most important to distribute.

An implementation is free to distribute data between caches independent of any explicitly invoked flush or fetch operations. Such asynchronous transfers would hopefully allow the communication latency to be hidden by overlapping it with task internal computation.

3.4 Controlling Access

Every *Wax* job, blackboard, task, and machine used to process tasks belongs to a user (*owner*). The owner of an object may attach an ACL to the object in order to restrict who may access the object. An ACL identifies the set of principals (users and machines) that can access the associated object, and the type of access each principal is allowed. The set of supported access types includes at least read, write, and administer. The last access right would grant the specified principal the right to manage the object.

The *Wax* programming model defines only one ACL operation, *initialize*, for the ACLs associated with jobs, blackboards, and tasks. When one of those objects is created, the initialize operation is automatically invoked to set the ACL on the object. The ACL on such objects cannot be changed, because that would require a globally consistent update. While the existence of ACLs to protect the machines used to process *Wax* tasks is part of the *Wax* programming model, the mechanisms for initializing and changing them are not.

3.5 Synchronization

The *Wax* programming model does not provide any global consistency guarantees, because maintaining global consistency requires synchronization that would make an implementation too sensitive to processor and network failure. In fact, the model does not provide any mechanism for ensuring that an observed state of a blackboard is globally consistent. A programmer cannot determine, on the basis of blackboard operations alone, whether the value read for a particular key is the only value associated with that key or one of many values. Similarly, a programmer cannot be sure that the set of keys returned by a directory operation represents a consistent snapshot.

The model does, however, provide an application programmer with two guarantees about the ordering of operations that will be observed by individual tasks. If a task, **Alice**, has observed the existence of a tuple with the key TO-DO-LIST (via a directory operation for example), then **Alice** will be able to read that tuple, as long as no task invokes a delete operation with the argument TO-DO-LIST. **Alice** will be able to observe the effects of all blackboard and task operations performed by another task, **Bob**, before being able to observe that the fact that **Bob** has invoked the exit operation.

3.5.1 Barrier Synchronization

If finer-grained operation ordering is required, application programmers can use the fetch and directory operations to synchronize the execution of tasks. For example, consider the following possible implementation of barrier synchronization. If an application can enumerate the set of synchronizing tasks, a barrier can be implemented as a collection of keys in the namespace of a blackboard. The set of synchronizing tasks can either be embedded in the program used to instantiate the tasks or recorded in blackboard tuple that is retrieved by each of the synchronizing tasks. When a task arrives at a barrier for the Nth time, it writes a tuple of the form *barrier-id.N.task-id*. A task can only pass a barrier when it observes that such tuples have been written by all tasks with which it is synchronizing. A task can wait by alternately executing directory operations and some form of internal delay operation. Fetch operations can be added to the loop, if desired, to potentially reduce the wait.

3.5.2 A Mutual Exclusion Primitive

The global consistency guarantees of the *Wax* programming model are weak enough that a mutual exclusion primitive cannot be constructed within the model. Two obvious candidate techniques for constructing a mutual exclusion primitive (a master task, and Lamport's algorithm [Lam87] using atomic reads and writes) require more consistency than the model provides.

The master task mechanism uses one task to arbitrate access to the critical sections of the application. This technique achieves mutual execution by relying on there being a single instance of the master task. Unfortunately, the *Wax* programming model does not guarantee that tasks are executed at-most once.

Lamport's technique uses a set of variables to implement a sequence of barriers that a process must pass through without being interrupted by another process in order to acquire the lock. In principle, the values of the necessary variables could be stored either in the values of blackboard tuples or encoded in the names of a set of tuples. Using tuple values will not work, because the *Wax* read operation, while atomic, does not guarantee that two tasks will necessarily read the same value for a tuple that has multiple values. Using the names of tuples to encode the values does not work, because the model does not provide a mechanism for taking a globally consistent snapshot of the namespace of a blackboard.

At a fundamental level, the concept of mutual exclusion runs against the grain of the *Wax* programming model. Mutual exclusion requires the synchronization and global consistency that the *Wax* model is attempting to avoid. As a result, a distributed computation system that implemented the *Wax* programming model would have to step outside of that model to support applications that require mutual exclusion.

3.6 Dealing with Failures

The processors and networks that make up a distributed computation system can fail. If an implementation of the *Wax* programming model caches the state of a job and its blackboards in multiple locations, these failures can create inconsistencies in the contents of the various caches. Since the model does not guarantee the consistency of the distributed caches in the absence of failure, jobs must already incorporate any mechanisms that they need to handle such inconsistencies. Processor failures can also result in the loss of the state internal to an executing task.

Another possible source of "failures" in a distributed computation system is the reclamation of resources, machines and cache space, by their providers. Such failures, however, can be modeled as an appropriately chosen set of processor and network failures.

The ideal-world behavior of the blackboard and task operations is modified slightly, so as to expose the transitory communication failures that will occur in a distributed implementation. In particular, an implementation will signal an error if a temporary communication error prevents it from performing an operation successfully. Requiring an implementation to signal an error is in keeping with the model's basic design idea that failures should be exposed to an application, so that it can chose the most appropriate recovery mechanism.

Under the *Wax* programming model, a job will be able to make forward progress as long as the sink for the job does not fail permanently and there exists at least one task within the job that is able to make forward progress. A job may actually continue making forward progress after the permanent failure of its sink, but the progress will be undetectable outside of the job.

The inability of a job to make forward progress, because one or more tasks are incompatible with all of the processors that can access those tasks is beyond the scope of the *Wax* failure model. The model cannot force an implementation to create resources that do not exist. As an example, consider a job that contains a task, **PartN**, with an ACL that restricts **PartN** to running on a machine named **Cray**. If there does not exist a machine with that name, then **PartN** will never execute and the job will stop making forward progress when all of the other tasks in the job terminate.

A task will be able to make forward progress so long as not all of the work done while executing on a processor is lost when that processor fails. The work done by a task consists of the sequence of operations that the task executes that change either the job containing the task (writing/deleting tuples, spawning tasks) or the state of the processor (memory and registers) executing the task. Assuming that the probability that a machine will crash or that a task will be evicted by the owner of the machine is constant, the probability that a failure will occur at some point during the execution of a task increases with the execution time of the task.

In order to support tasks with arbitrarily long execution times, an implementation of the model must provide some mechanism that enables such tasks to continue to make forward progress. One possible mechanism is to periodically checkpoint executing tasks. This would allow the implementation to resume interrupted tasks from checkpoints of intermediate states. A task will then be able to make forward progress if, each time it is restarted after an interruption, it is restored to a state later in the work sequence than the states restored by all previous restarts. Since the checkpoint process itself is susceptible to failure, task checkpointing only makes it probable that a task will be able to make forward progress, where the probability is a function of the time between checkpoints and the expected time between failures. In general, for a given failure rate, shortening the time between checkpoints will increase the probability that a task will be able to make forward progress.

3.7 Conclusion

The *Wax* programming model is designed to allow an implementation of the model to make as much as possible of the parallelism provided by the underlying computing resources available to executing jobs. This is accomplished by exposing the inconsistencies that latency and failure create in any asynchronously executing, distributed system, and providing application writers with the mechanisms needed to enforce any application-specific consistency requirements. Each job only has to pay overhead for the synchronization overhead that it requires.

The *Wax* programming model provides the users of coarse-grain parallel applications with several benefits, including potential access to a new computation resource and automated process management. By simplifying the implementation of a distributed computation system, the *Wax* model facilitates the creation of a system that would allow the machines connected to wide-area networks to be used as a distributed multiprocessor. Such a system provides the users of coarse-grain parallel applications with a powerful new computing resource. In addition, the semantics of the *Wax* programming model require that an implementation of the model handle the assignment of processes to processors and automatically recover from the failure of those processors. This automated task management can ease the user's job of managing the execution of large computations. An implementation could expose this management mechanism in order to allow the user to monitor, and perhaps modify, the computation's behavior.

Eliminating the consistency guarantees of traditional parallel programming models simplifies the implementation of a distributed computation system that supports the *Wax* programming model. It also makes the resulting system easier to scale and more tolerant of the failures that will occur in a system spread across a wide-area network. Chapter 4 describes the implementation of a prototype wide-area distributed computation system that supports the *Wax* programming model.

Chapter 4

The Implementation of *Wax*

In order to show that the programming model described in Chapter 3 is feasible to implement in a distributed environment, I built *Wax*, a prototype distributed computation system. *Wax* further shows that the limited processor interface defined by the model allows an implementation to prevent *Wax* tasks from abusing the resources local to the processors used to execute those tasks. The rest of this chapter describes the implementation of *Wax*.

The chapter begins with an overview of the structure of *Wax*, which provides a framework for the presentation of the implementation details. The system is described from the bottom up: first the tools used to implement various pieces of *Wax*, then the implementation of those pieces, and finally the programming and shell-level interfaces provided by *Wax*. At each stage, the security mechanisms built into *Wax* are discussed along with the objects that they protect. At the end of the chapter, the implementation of *Wax* is evaluated.

At the bottom level, *Wax* consists of a library of interface routines, which are linked into application programs, and a collection of system processes that make *Wax* function. Figure 4.1 shows the overall structure of *Wax* and and the interactions between the various pieces of the system.

External processes, which are processes that can access both *Wax* and resources external to *Wax*, such as disks and monitors, that are beyond the scope of the *Wax* programming model, serve as bridges between those external resources and *Wax* computations. They create jobs, blackboards, and tasks, and provide the input for and consume the output of the computations performed by those tasks.

Wax uses three types of system processes to manage the processing of tasks and the distribution of data. The Task Managers distribute tasks to processors for execution. A Host Manager serves as *Wax*'s local agent on each processor used to execute tasks, and handles the details of executing those tasks. The Distribution Managers move data between blackboard caches.

Processors are grouped into *domains*, where each domain consists of machines belonging to a single organization. Each domain maintains its cache of blackboard data. The Task and Distribution Managers can be replicated to improve fault tolerance and support larger domains.

Each domain must have one or more trusted, physically secure, operator-attended machine on which to run the trusted components of *Wax*: the Task and Distribution Managers. The rest of the machines connected to *Wax*, the ones used to process tasks,

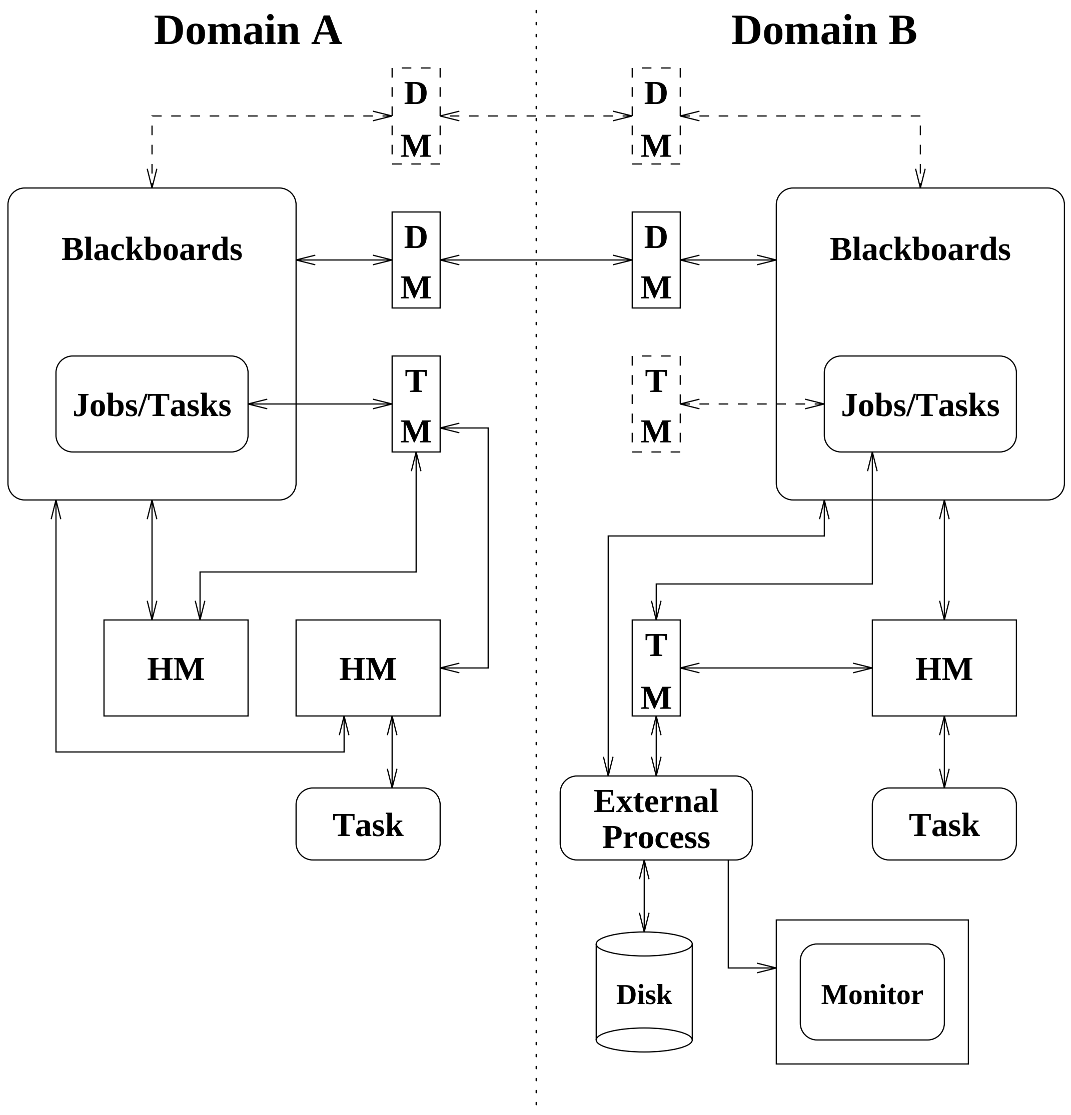

Figure 4.1: The structure of *Wax*. A domain consists of a group of machines belonging to a single organization. The processing of *Wax* tasks is controlled by the Task Manager (TM). Every machine that is used by *Wax* to process tasks runs a Host Manager (HM). Host Managers directly perform any blackboard operations requested by executing tasks, and relay task operations to a Task Manager. The Distribution Managers are responsible for moving data between domains in response to flush, fetch, distribute, or collect operations. The dashed boxes represent possible replication of the Task and Distribution Managers.

are expected to be workstations under the control of one or more primary users, and thus neither physically secure nor trustable.

By partitioning domains along organizational boundaries, *Wax* allows organizations to retain autonomous control of the resources they are providing to *Wax*. Default access and resource usage policies are expected to be defined on a per-domain basis. For example, the current implementation allows domains to assign jobs created in other domains a lower priority than the default priority of locally created ones. In a production version of the system, organizations might want the ability to allocate resources on an even finer-grained basis, such as per user or group of users.

In addition to the benefits they provide to the resource providers, domains also make *Wax* easier to scale. Domains, and their users, can potentially make use of machines in other domains without having to know how many machines there are, or what their names are. Each *Wax* domain only needs to known about the machines that are available internally, and any other domains with which it cooperates.

Domains also allow resource providers to hide information. Most of the internal implementation and structure of a domain is invisible outside of that domain. The only part of a domain that must be externally visible in order to interact with other domains is the interface to the mechanism used to exchange blackboard and job data between domains. (see Section 4.6.) In general, a domain will also want to provide other domains with general information about the domain, such as, types of processors available and amount of cache space available.

The libraries and server processes that constitute *Wax*, and the RPC interfaces that interconnect them, are implemented on the *Mach* 3.0 operating system [ABB+86]. *Wax* exploits several features of the *Mach* operating system environment. Specifically, it relies on the following operating system services:

- An RPC interface and messaging system that can be used to pass control and data between cooperating processes on the same and different machines.
- Multiple threads of control within the same address space. This is used in implementing the *Wax* system services, and can also be used in applications.
- A system-call redirection mechanism that can insulate a process and its host from one another.
- A virtual memory system that allows one process to read and write the address space of another. This is used to implement a task checkpoint and restart mechanism that can reduce the amount of computation lost when a task is interrupted.

While, in general, using *Mach* 3.0 limits the size of the available host base, there were a large number of such machines available at CMU when the implementation of *Wax* began. Furthermore, I felt that it was important to work within the context of an operating system offering more modern facilities that are likely to become commonplace over time. Section 4.8 examines how the system could be ported to other, more widely available, operating systems.

Having sketched the outline of the implementation of *Wax*, I will now start filling in the details. The first step is to examine how the various processes that make up a *Wax* computation communicate with each other. The focus will then shift to how the various processes can identify with whom they are communicating, which is necessary in order to be able to control access to resources. After authentication and authorization are covered, the presentation will move on to the implementation of the objects defined by the *Wax* programming model: blackboards, jobs, and tasks. This followed by a discussion of how *Wax* tasks are executed, and the techniques used to create a secure execution environment on each processor used by *Wax*. From there the presentation moves on to the *Wax* interfaces that are directly visible to the application programmers and users.

4.1 Inter-process Communication

Wax uses the *Mach* RPC facilities to communicate between processes running in the same domain. The Mach Interface Generator (MIG) [DJT89] is used to produce the code that marshals and unmarshals the data exchanged via RPC. These interface stubs then use the message passing facilities provided by the *Mach* kernel to move the data between *Mach* tasks. While the *Mach* kernel only supports passing messages between tasks on a single machine, the *Mach* Network Message Server (NetMsgServer) [Gro89] transparently extends the message-passing mechanism between machines.

Wax provides and uses a "connection" module that manages the creation and destruction of communication links, such as the ones between a *Wax* user and a *Wax* system process or a pair of *Wax* system processes, that may cross machine boundaries. The connection module provides functionality similar to Birrell's secure RPC mechanism [Bir85]. An important implementation difference between the two systems is that the security features in the *Wax* connection module are implemented on top of, rather than as an integral part of, the RPC mechanism. While this difference allows *Wax* to use the standard *Mach* RPC tools, it does complicate the implementation of the security mechanisms, as is described in more detail in Section 4.2.2.

Inter-process communication (IPC) in *Mach* uses kernel-protected capabilities, called *ports*, to represent the end-points of communication channels. Within a task, each port has a unique integer name. However, different tasks may, and usually do, have different names for the same port. For example, port **P** might be named 17 in task **A** and named **26** in task **B**.

A task may transmit a message to any port for which it holds a *send* right. Similarly, a task may receive a message from a port only if it holds a *receive* right. The *Mach* 3.0 IPC mechanism also supports a *send-once* right, which allows the holder to send only one message to the port. For every port, there is exactly one task holding a receive right and zero or more tasks with send or send-once rights. The holder of a send or receive right may create additional send rights that can be sent to other tasks in a message.

For task **A** to communicate with task **B**, **A** must have a send right to a port **P** for which **B** holds a receive right. To facilitate the creation of such links, the NetMsgServer provides a name service, which tasks can use to register send rights that other tasks can look up. For example, **B** could register **P** with the NetMsgServer under the name "`my-port`", which **A** could then retrieve. However, **A** needs to have a send right for the port that the NetMsgServer uses to accept name service requests. This bootstrapping problem is solved by the *Mach* kernel automatically providing every task with a send right for the port used by the NetMsgServer.

4.2 Controlling Access to Resources

As required by the programming model, *Wax* provides ACLs to protect the resources accessible through the system. Enforcing access controls requires the ability to name, authenticate, and authorize every entity attempting to access a protected object. Section 4.2.1 describes the naming scheme used by *Wax*. Section 4.2.2 presents the mechanisms used to authenticate the identity of an entity. Section 4.2.3 covers the features of the ACLs themselves.

4.2.1 Naming

Every principal[1] (user, machine, and *Wax* system process) that may access *Wax* is assigned a unique identifier. Each domain is allowed to independently assign names. The canonical form of a principal's name consists of a domain-specific identifier, a separator ('`@`'), and the name of the principal's *home* domain. For example, my user name in the CMU *Wax* domain is `pds@cs.cmu.edu`. The home domain of a user is the domain that provides the user with access to *Wax*. A user allowed to natively access *Wax* in multiple domains may have multiple canonical identifiers. The home domain of a machine is the domain that has physical possession of the machine. Within the home domain of a principal, the domain-specific identifier may be used without the separator and domain name.

In the current implementation, principal names that begin with the string "`hm:`" are interpreted as the name of a machine connected to *Wax*, all other names are considered to be user names. The canonical form of a machine name includes the name of the machine as registered with the Domain Name Service (DNS) [Moc87a, Moc87b, Moc89, Man92] of the appropriate domain. Synonyms consisting of the prefix and any portion of a machine's name that the DNS of the appropriate domain will map to the complete name are also allowed. For example, `hm:pds@cs.cmu.edu` and `hm:pds.mach.cs.cmu.edu@cs.cmu.edu` both refer to the same machine. The prefix is needed to be able to distinguish between users and machines in some cases. For example, without the prefix, the meaning of `pds@cs.cmu.edu` would be ambiguous.

Wax also provides support for groups of principals. In the current implementation, this is limited to two groups, `all-users` and `all-hm`, whose membership is implicitly defined. The first group refers to all users within the domain, while the second refers to all machines within the domain.

4.2.2 Authentication

Controlling access to resources requires that the *Wax* agent checking an access request be able to determine the identity of the requester. This implies a need for some form of authenticated communication. The best technique for performing the authentication depends in part on the type of communication being performed: local to one machine, between two machines in a single domain, and between two machines in different domains. In general, the operating system on a machine can be trusted to protect local

[1]This terminology is adopted from the *Kerberos* system, which is used to implement authentication within a domain.

communication.[2]

For communication between two machines, the authentication problem is harder. In general, the machines connected to *Wax* are neither physically secure nor trusted, in a distributed sense. For example, the primary user of a machine could modify the machine's operating system so that that user could claim any desired identity. Furthermore, the links that make up local- and wide-area networks are rarely secure. In particular, a user with access to one machine may be able to eavesdrop on all communication on any network connected to that machine. In addition, the user may also be able to modify messages exchanged among other machines and/or to masquerade their machine as a different host.

A distributed authentication mechanism is required. For communication between machines within a domain, *Wax* uses a mechanism based on version four of the *Kerberos* system [SNS88] from MIT. The *Kerberos* Authentication Server uses secret-key cryptography, based on the Data Encryption Standard (DES) [Nat77], to provide trusted third-party authentication. For communication between domains, *Wax* uses a mechanism based on public-key encryption. I chose not to use a *Kerberos*-based mechanism for such communication, because just as the users of one machine may not be willing to trust the operating system on another machine, *Wax* domains may not be willing to trust *Kerberos* servers running in another domain.

The Intra-domain Solution

The *Kerberos* system supports the definition, and authentication, of *principals*, and specific *instances* of those principals. A *Kerberos* principal typically identifies a particular user or system service, while instances are used to define additional attributes about the principal. For example, a *Kerberos* principal might have a number of instances, each of which was only to be used on a particular host. If a request from such an instance was received from a different host, it might imply the existence of a security problem.

Every *Wax* principal has a corresponding *Kerberos* principal or instance. The mapping between *Wax* user principals and corresponding *Kerberos* principal is the identity function. The mapping for *Wax* machine principals involves transforming the canonical form of the principal into an instance of a single *Kerberos* principal by replacing the "`hm:`"-prefix with "`wax.`". For example, `hm:pds.mach.cs.cmu.edu` becomes `wax.pds.mach.cs.cmu.edu`, which *Kerberos* interprets as being the `pds.mach.cs.cmu.edu` instance of the *Kerberos* principal *Wax*.

A *Kerberos* principal, or an instance thereof, verifies its identity to the *Kerberos* Authentication Server by demonstrating the knowledge of a password/key that is known only to the principal, or the appropriate instance of the principal, and the Authentication Server. In general, a user does this by entering the secret password at the beginning of a login session, perhaps as part of the login process, or when starting an authenticated process. Server processes typically read their *Kerberos* key from a (protected) file stored in the local file system of the machine on which they run. The security of this key depends, of course, on the security of the local file system.[3] This dependency can be avoided if the

[2]If this is false, then there exists no other mechanism that could protect such communication.

[3]The possible security problems created by this dependency may well be the motivation for *Kerberos*

server process is always started by a system operator, who would also enter the server's password. Alternatively, if there is a secure co-processor attached to the machine, the key can be stored in the non-volatile memory of the co-processor.

After the Authentication Server has verified the identity of the principal, it issues the principal a *ticket-granting-ticket*, which is valid for a fixed period of time.[4] While a ticket-granting-ticket is valid, a principal can use it to acquire a *service-ticket* for any principal/instance with which it wants to communicate. The service-ticket can then be used to perform mutual authentication when establishing a communication link, and to establish the secret-key to be used to protect/authenticate the data exchanged over that link (see [SNS88] for more details).

The connection module, mentioned in Section 4.1, automatically performs *Kerberos*-based mutual authentication each time a communication link is established. It also deals with renewing the authentication when the service ticket expires, assuming that the principal has acquired a new ticket-granting-ticket if necessary. Authenticated communication allows the Task or Distribution Manager to properly control access to resources used by and the data stored in *Wax*. It also prevents attackers from masquerading as one of these trusted components.

If a *Wax* user has a valid ticket-granting-ticket, then the *Wax* utilities, at both the shell and library-level, will automatically acquire the service-tickets needed to authenticate the user to the *Wax* system processes. In an environment such as the one at CMU, where users already depend on *Kerberos* for a variety of other services, this means that authenticated access to *Wax* is provided seamlessly, without any additional user effort.

Data Integrity

In conjunction with authenticating the principals at each end of a communication channel, *Kerberos* provides each of the communicating processes with a DES key, the *session* key, that can be used to protect data exchanged on the channel. The connection modules at each of the channel use the key to generate cryptographic checksums of each message that is sent. These checksums allow the receiving party to determine whether the data has been modified while in transit. The checksums do not, however, prevent third-parties from reading the message data.

The checksum is generated by performing a DES encryption, in cipher-block-chaining mode [Nat80], of the user data in the message. The final block of the ciphertext used as the checksum value. Because the *Mach* kernel may change some of the system data, in particular, the names of port rights, the checksum is only performed on the user data in the message. Also, the connection module is built on top of the MIG generated stubs, so it does not have access to the additional data that MIG adds to messages. From the perspective of MIG, the checksum is just another argument in the RPC.

The DES checksum value, the number of arguments to the RPC, an error code, the current time, and a flag indicating whether the message is a request or a reply message are then encrypted, using the appropriate session key, and the resulting ciphertext included in the message that is actually sent across the network. The error code is included in

supporting multiple (potentially location-specific) instances of a principal.

[4]The Authentication Servers at CMU will issue tickets with lifetimes ranging from five minutes to 30 days.

this data in order to work around an optimization used by MIG. In reply messages, MIG only includes the values for user arguments, including the checksum itself, if the server indicates that the operation was completed successfully. The work around has the RPC operations report their true return code as part of the checksum data, and always return a success status to MIG. In request messages, the error code is always zero. The timestamp and the direction flag allow the receiver of a message to protect against reply attacks. While the connection module does not currently check for duplicates, such checks could easily be added if deemed necessary.

The Inter-domain Solution

The only time in *Wax* that processes in different domains need to communicate is when Distribution Managers are exchanging data between a pair of domains. These communication links are authenticated using public-key cryptography and the protocol proposed by Needham and Schroeder [NS78]. The advantage of this mechanism, over one based on secret-key cryptography, is that no trusted third-party is necessary, as long as the domains have a secure means of exchanging public-keys, such as a phone call between domain administrators.

Public-key cryptography uses different keys to encrypt and decrypt data. Each principal publishes one of these keys as its public-key, and retains the other as its private-key. A principal, **Alice**, can create a message that only a specific principal, **Bob**, can read by encrypting the message with **Bob**'s public key. Similarly, **Alice** can use encryption with her private key to "sign" messages. Any principal with access to **Alice**'s public key will be able to read such messages, and will know that no other principal could have written the message.

4.2.3 Authorization

This section describes the features of *Wax* ACLs, in particular, the external format and the supported access rights. Other details about ACLs, such as the exact meaning of each access right, how an ACL is stored and enforced, *etc.* depends on the type of object to which the ACL is attached, and are therefore described later in the chapter, along with the implementation of each of type of object. *Wax* ACLs are modeled after the ones used in the Andrew File System (AFS) [SHN$^+$85].

Every job, blackboard, task, machine, and domain has an associated ACL. Each ACL has a set of positive entries and a, possibly empty, set of negative entries. An ACL entry consists of a principal identifier and a set of access rights. A *Wax* principal may access an object if the requested type of access is associated with the principal in the positive list, but not in the negative list. The types of access granted to a principal is the union of the rights in applicable positive entries minus (set difference) the union of the rights in the applicable negative entries. The current implementation of *Wax* supports three types of access rights: read, write, and administer.

ACL entries may also contain identifiers that refer to groups of principals. Such entries give all members of the specified group the same positive, or negative, access rights. Additional rights can then be granted, or denied, to specific users by mentioning them

```
+ Positive Entries
g all-users rw
g all-hm rw
u wax.TaskManager rwa
u wax rwa
- Negative Entries
u pds w
.
```

Figure 4.2: An example *Wax* access control list (ACL).

explicitly in the positive, or negative, set of entries.

The external form of a *Wax* ACL is a sequence of newline separated records. Five record types are supported: blank, comment, section header, ACL entry, and terminator. Blank records, which consist of only whitespace, and comment records, which consist of optional whitespace and the comment character ('%') followed by arbitrary text, are ignored. The section headings indicate whether entries encountered before the next section heading, or terminator, are positive entries or negative entries. A section heading line contains optional whitespace, and a single control character ('+' for positive entries, '-' for negative entries). Any text between the control character and the end of the of the section heading record is ignored. A section that contains no ACL entries is considered to be empty. Each ACL entry consists of three required fields and an optional comment. The first field in an entry is either a 'g' or a 'u', and indicates whether the identifier in the second field refers to a group or a user, respectively.[5] The third field is a set of access rights: 'r' for read access, 'w' for write access, and 'a' for administer access. The terminator record, which marks the end of an ACL, consists of a period ('.'), optionally surrounded by whitespace. A valid ACL has at least one non-empty positive section and a terminator record. The order of the sections in an ACL is not significant.

Figure 4.2 shows an example *Wax* ACL. The effect of the ACL is that all users and machines in the local domain, except `pds`, `wax.TaskManager`, and `wax`, have read and write access to the associated object. The users `wax.TaskManager` and `wax` also have administer access, while `pds` only has read access, because that user's write access is removed by the negative entry. The read and write accesses listed in the entries for `wax` and `wax.TaskManager` are not strictly necessary, because the `all-users` group entry all ready gives them those rights.

The nesting of various objects defined by the *Wax* programming model, for example, tasks inside of jobs, creates an ACL hierarchy, which is shown in Figure 4.3. The ACL on a domain defines a default ACL for all machines in that domain and for all jobs created in that domain. Within the constraints of this default ACL, users can further restrict access to their machines and/or jobs by reducing the scope of the positive rights, increasing the scope of the negative rights, or both. Hierarchical ACLs allow domain administrators to define a base-level of security, while leaving *Wax* users the flexibility to define more restrictive controls.

[5]This is intended to allow the support for groups to be generalized from the two pre-defined groups provided by the current implementation to arbitrary groups, which may result in users and groups with the same name.

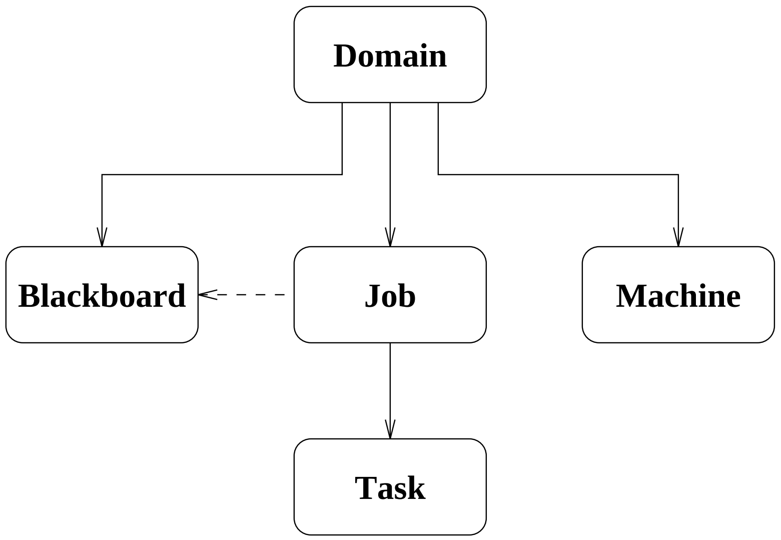

Figure 4.3: The ACL hierarchy supported by *Wax*. Starting from the top, the solid arrows indicate the sequence of ACLs that must be checked in order to access each type of object. The dashed arrow between a job and a blackboard is there because while a blackboard can be accessed independently from the job(s) that use it, access to a job is necessary in order to examine the list of blackboards it uses, which is generally a pre-cursor to accessing a blackboard.

The ACLs on domains and machines are initially defined when the resource is made accessible to *Wax*. The administrator of a domain or machine can change the ACL on the object at any time.

The ACLs on jobs, tasks, and blackboards are defined when the object is created, and they may not be changed thereafter. The ACLs cannot be changed, because the ACL information is itself stored as a tuple in a blackboard, and there is no mechanism for globally updating the value of a tuple.

The ACL that is actually attached to a *Wax* object is a function of the ACL specified by the creator of the object (the specified ACL) and the ACL attached to parent of the object in the ACL hierarchy (the parent ACL). In other words, *Wax* pre-evaluates the restrictions defined by the parent in order to simplify the access checks that need to be performed when accessing an object. If the specified ACL has no positive or negative entries, then the parent ACL is attached to the new object. If the specified ACL has only negative entries, then an ACL consisting of the positive entries from the parent ACL and the union of the negative entries from the parent and specified ACLs is attached to the new object. Otherwise, an ACL consisting of the intersection of the positive entries and the union of the negative entries of the parent and specified ACLs is attached to the new object. The slightly unusual behavior in the case of a specified ACL containing only negative rights is to simplify the use of redundant computation to verify the results produced by a task, see Section 4.4.4. Furthermore, combining an ACL with no positive entries with any other ACL yields an ACL that allows no one access, which is not particularly useful. [6]

4.3 Blackboards

The *Wax* blackboards are implemented using a library of subroutines and RPCs, AFS, and the *Wax* Distribution Manager. AFS is used to store and transfer blackboard data within a domain. The Distribution Managers move blackboard data between domains. And, the library of subroutines and RPCs provides a wrapper to glue everything together and hide the details of the implementation from the user.

Each blackboard is stored in a separate AFS directory, with tuples stored as separate files. Each domain records only one of the values associated with a key. This is sufficient to satisfy the *Wax* data model, and saves space compared to storing the entire set.

AFS uses a local disk cache, managed by the AFS Cache Manager, to reduce network communication requirements. The consistency of the cached files is maintained using a callback-based mechanism. When a file is updated, the changes are not sent to the file server until the file is closed. At file close time, the copies cached by all other, accessible, Cache Managers are invalidated. *Wax* relies on these mechanisms to propagate blackboard updates within a domain.

By default, AFS does not wait for a modified file to be stored on the file server before returning from the UNIX close system call. If a communication error occurs while the file is being stored, then the file can be corrupted by truncation. The first mechanism used by *Wax* to detect such failures was to store the length of a tuple value, as a newline-terminated ASCII string, at the beginning of the file containing the value. The second mechanism, which was discovered and implemented later, involves forcing AFS to wait for the file to be stored before returning. This allows the failure to be reported to the invoker of the operation, rather than be observed as data corruption later. While the second mechanism makes the first mechanism unnecessary, the former has not been removed from the system.

Wax assigns each blackboard a unique two-part identifier. The first part identifies the domain in which the blackboard was created. The second part distinguishes the blackboard from all others created by the same domain. A blackboard identifier is represented internally as two 32-bit values and externally as an ASCII encoding of those values. The value identifying a domain can be chosen arbitrarily, so long as it is distinct from the values used by other domains. Because I chose the IP address of the machine `cs.cmu.edu` as the domain identifier for the like named *Wax* domain, the ASCII representation of the domain portion of a blackboard identifier is formated using the standard Internet "." notation. For example, if the internal identifier for a blackboard has a domain identifier of 02030410 (hex), and a per-domain identifier of 1, then the ASCII representations of the identifier will be "`<1@2.3.4.16>`".

Each blackboard contains two distinct namespaces: one external, the other internal. The external namespace contains user data, that is the tuples that are read and written by *Wax* tasks. The internal namespace is used by *Wax* itself to record system data. The name of a tuple file starts with a prefix that identifies the namespace to which it belongs: "`int_`" for internal tuples and "`ext_`" for external ones. The rest of the name of a tuple file is the

[6] At least, in environment where you cannot subsequently modify the ACL.

```
/afs/cs.cmu.edu/project/wax/blackboards/<1@128.2.222.173>
IntegerFactoring
pds@cs.cmu.edu
+
g all-hm@cs.cmu.edu rw
u wax.TaskManager@cs.cmu.edu rwa
u pds@cs.cmu.edu rwa
u wax@cs.cmu.edu rwa
-
 .
```

Figure 4.4: An example blackboard descriptor. The first line of the descriptor is the path of the directory used to store the blackboard. The second line lists the name of the blackboard. The third line contains the principal who created, and therefore owns, the blackboard. The rest of the value is the ACL for the blackboard.

key portion of the tuple. For example, if a task wrote a tuple with the key COLOR, the tuple would be stored in the file "`ext_color`".

The *Wax* blackboard access mechanism is implemented in a recursive fashion. Each domain has a master blackboard, which is stored in a well known location.[7] The other blackboards that exist in a domain are described by tuples recorded in the master blackboard. Figure 4.4 shows the information stored in a blackboard descriptor tuple. This is, in fact, the descriptor for the blackboard used by the first *Wax* job.

4.3.1 Implementation of the Blackboard Operations

This section describes the implementation of the *Wax* blackboard operations. The first part explains the internal structure of the libraries that application writers use to access blackboards. This is followed by a description of the functionality common to all of the blackboard operations, which includes mapping blackboard identifiers to AFS directories and validating the keys. The remainder of the section then focuses on the implementation of the individual operations.

As mentioned previously, the user interface to the blackboard is implemented as a library of subroutines and RPCs. This library supports two different interfaces: one for use by external processes and another for use by *Wax* tasks. The routines in the first interface directly invoke the internal library of routines that implements most of the blackboard functionality. The routines in the second interface are wrappers around RPCs to the Host Manager executing the task. The Host Manager then invokes the appropriate internal library routine on behalf of the task. *Wax* system processes all directly invoke the internal interface to access the blackboard, in part, because it provides functionally, such as creating and deleting blackboards, that is not directly exported to *Wax* applications.

The names of the routines in the two user interfaces are identical except for the interface prefix: "`Wax_`" for the first interface and "`WaxExt_`" for the second. The only difference between the two interfaces is that the routines in the "`WaxExt_`" interface require a blackboard identifier as an argument. A Host Manager can infer the necessary

[7]In the *Wax* domain at CMU, this is: /afs/cs.cmu.edu/project/wax/blackboards/master.

blackboard identifier for operations performed through the "Wax_" interface because the current implementation supports only one blackboard per job.

When a process attempts to access a blackboard, the internal library subroutines look for a mapping from the blackboard identifier provided by the caller to the path of the AFS directory used to store the blackboard. An internal cache is checked first. If no mapping is recorded in the cache, the library attempts to read the blackboard descriptor tuple from the master blackboard. When such a tuple is found, the mapping it contains is recorded in the internal cache for use by subsequent operations. If a mapping is not found, then all of the blackboard operations will return an error code that indicates which of the following errors occured: no mapping exists for the identifier or a communication error occured while checking for a mapping. The well-known mapping for the master blackboard is inserted into the internal cache by the initialization code for the library.

Once the blackboard identifier has been mapped into an AFS directory path, the key passed to the operation is validated. *Wax* currently supports two types of keys: strings and regular expressions. A string key is valid if is a NUL-terminated sequence of ASCII characters, except for "/", and the sequence is less than 224 characters long. The length restriction and the restriction on the character set is dictated by the use of keys in file names. A regular-expression key is a NUL-terminated character string, and is considered valid if it can be parsed by the regular expression package used by *Wax*.[8] While the fetch, flush, and directory operations accept both types of keys, the read, write, and delete operations, which operate on individual tuples, accept only string keys.

Each of the exported library routines uses a series of UNIX file operations on files stored in the blackboard's AFS directory to implement the operations defined by the *Wax* programming model. All of the operations will fail and signal an error if a communication failure occurs during the execution of the operation. The remainder of this section provides the implementation details for each of the exported routines.

The Bb_Write routine implements the *write* operation. It creates a new tuple file containing the value specified as an argument. The value, which the caller specifies by address and a length, is treated as an uninterpreted byte sequence by *Wax*. The operation is made to appear atomic by writing the new value to a temporary file, rather than directly to the tuple file. If the temporary file is successfully stored on the file server, then it is renamed to correct name for the tuple file. In order to ensure that the names assigned to the temporary files are unique, they are given names based on the host on which the operation is performed and the unique identifier of the thread executing the operation.[9] The operation will fail and signal an error if the invoker of the operation does not have write access to the blackboard, or there is not enough space to store the new value.

The Bb_Read routine implements the *read* operation by reading the tuple file that corresponds to the specified key and returning the recorded value. The AFS caching/callback mechanism is used to ensure the atomicity of read operations. If the caller of the routine does not have read access to the blackboard or necessary tuple file does not exist, the operation will signal an error.

[8]This package is a thread-safe re-implementation of the standard UNIX regular expression utilities.

[9]This naming scheme is flawed, because thread identifiers are only guaranteed to be unique within a task. The scheme could be fixed by adding a process identifier to the name.

The `Bb_DeleteEntry` routine implements the *delete* operation by removing the tuple file that corresponds to the specified key from the AFS directory of the blackboard. The routine will fail and signal an error if either the invoker does not have write access to the blackboard, or the appropriate tuple file does not exist.

The `Bb_GetList` routine implements a generalization of the *directory* operation. The routine allows the caller to specify lower and upper bound times, so that the caller can use the routine to wait for new tuples to appear. The routine scans the directory containing the blackboard looking for the tuples files that match the specified key and have been modified since the lower bound time. As long as no matching tuples are found, then the scan is repeated every three minutes until the upper bound time is reached. If a string key is specified, then the size of the tuple is also returned. The routine will fail and signal an error if either the invoker of the routine does not have read access to the blackboard or the upper bound time is reached without finding a matching tuple.

The `Bb_Fetch` routine implements the *fetch* operation. After checking the validity of its arguments and the ability of its caller to access the specified blackboard, this routines use an RPC to the forward its arguments to a Distribution Manager for actual processing.

The `Bb_Flush` routine allows users to provide the system with a hint that they would like a set of tuples distributed to other tasks. If the routine returns successfully, then the matching tuples have been distributed to all tasks that were executing when the operation was invoked. The routine is implemented as an RPC to a Distribution Manager.

The `Bb_Create` routine creates a new blackboard. The operation is passed the name of the blackboard, its owner, and the ACL to associate with the new blackboard. The operation first creates an AFS directory to hold the new blackboard. The specified *Wax* ACL is then used to set the AFS ACL on the newly created directory. The mapping between *Wax* and AFS ACLs is described in Section 4.3.2. Finally, the path to that directory and the arguments passed to the operation are recorded in the descriptor tuple for the new blackboard. This operation will fail if the ACL on the local domain does not give the invoker of the operation write access.

The `Bb_Delete` routine removes a blackboard and its contents from the local cache. This involves deleting the blackboard's associated AFS directory, and its contents, and removing the descriptor tuple for the blackboard from the master blackboard. This operation will fail and signal an error if the caller does not have administer access on the blackboard.

The `Bb_` interfaces also provides routines that allow a tuple to be read or written in pieces, much like a file. The library ensures that the values read and written in this manner are consistent with the atomicity guarantees required by the programming model. In particular, the technique of writing to a temporary file and renaming is used to make the write operations appear atomic to other tasks or processes accessing the blackboard. The ability to read and write tuples in pieces is used by *Wax* when accessing large tuples, such as the programs used to instantiate tasks (see Section 4.4) and the checkpoint records used to save the intermediate state of executing tasks (see Section 4.5.3).

4.3.2 Blackboard Security

Wax depends on AFS to control access to blackboards, since external processes and Host

Managers access blackboards without passing through one of the trusted components of *Wax*. These processes are allowed direct access in order to avoid creating a potential performance bottleneck. When an AFS directory is created to cache the contents of a blackboard in a domain, its ACL is set from the *Wax* ACL associated with the blackboard. The AFS and *Wax* support similar, but not identical ACLs, which makes mapping from *Wax* to AFS straightforward.

AFS ACLs support finer-grained types of access than *Wax*. There is, however, a simple one-way mapping between the set of access types (read, write, and administer) used by *Wax* and those provided by AFS. In particular, the following set of AFS access types are used to implement each of the *Wax* access types:

- *Wax* read -> AFS read, lookup
- *Wax* write -> AFS write, insert, lookup, delete
- *Wax* administer -> AFS administer

The mapping between *Wax* principals and AFS principals is similarly straightforward, however, part of it is many-to-one, which eliminates some of the fine-grain control that *Wax* ACLs are intended to provide. Both *Wax* and AFS use *Kerberos* to implement their authentication mechanisms, therefore, every *Wax* user principal is expected to have an identically named AFS identity. The *Wax* group `all-users` is mapped to the AFS group `system:authuser`.

Unfortunately, AFS does not support *Kerberos* concept of instances, which creates a problem for the Host Manager instances. To work around this deficiency, all Host Manager instances, and the *Wax* group `all-hm`, share a common AFS identity: `wax.hostmanager`.[10,11] In order to minimize the security impact of sharing a common identity, the Host Managers do not have access to the password for their AFS identity. Instead, they request the necessary AFS authentication token, which is actually a *Kerberos* service ticket, from a Task Manager, which has the password and is therefore able to acquire such tokens. If the security of a host is compromised and this is detected, then the vulnerability of the blackboard cache is limited by the lifetime of an AFS token. Furthermore, this mechanism allows a host to be shut off from *Wax* without having to distribute a new password for the Host Manager identity to all of the remaining machines.

The use of AFS to store the blackboards weakens the data integrity guarantees that *Wax* can provide. While AFS authenticates communication links, it does not protect user data against modification. All data stored in or communicated via AFS are, therefore, subject to modification by anyone with access to the local network. This lack of security forces users to trust all machines in a domain, and limits the utility of being able to restrict tasks to particular machines. Within a domain, this problem could be solved by modifying the blackboard module to use a different, *secure*, data transport and storage mechanism.

In order to ensure the integrity of the system data that *Wax* stores in AFS, such data could include a cryptographic checksum that is signed by the agent that wrote the data. This would serve to verify both the integrity of the data and the author of the data. The data that would need to be protected are the descriptor tuples for jobs, tasks, and blackboards.

[10]Unlike *Kerberos*, AFS treats the "`.`" as part of the principal name, not as a separator between principal and instance.

[11]While it would be possible to allocate a separate AFS identity for each Host Manager, the administrators of the CMU CS computing facility balked at doing this for a prototype system.

While the appropriate places for performing such checksumming have been identified in the current implementation, it has not actually been implemented.

4.4 Jobs and Tasks

Every *Wax* job is part of the *task heap*, from which *Wax* selects tasks for processing. This task heap is implemented as a collection of *Wax* blackboards. Each domain has one or more Task Managers that maintain that domain's cache of the *Wax* task heap. The Task Managers control the processing of tasks within a domain. Any task that is not recorded as terminated in the portion of job visible in a particular domain is eligible for assignment to a processor in that domain. The number of Task Managers used by a domain's administrators depends on the number of machines in the domain and the degree of failure tolerance desired.

Using the blackboard mechanism to store jobs provides *Wax* with the same failure tolerance that the mechanism provides to *Wax* applications. It also means that the Task Managers must handle any inconsistencies that may occur due to communication and/or processor failures. In some sense, the Task Managers are tasks in a meta-*Wax* job whose computation involves processing *Wax* tasks.

Each *Wax* job and task is assigned a unique identifier of the same format as blackboard identifiers. In fact, the same identifier allocation mechanism used to generate blackboard identifiers is used to create both the job and task identifiers. The domain field of a task identifier specifies the domain where that particular task was created, and may be different from the domain field of the identifier for the job containing the task. While task identifiers happen to be globally unique in the current version of *Wax*, they are only guaranteed to be unique within a job since tasks only exist within the context of a job.

Even though the *Wax* programming model allows jobs to use multiple blackboards, *Wax* currently supports only one blackboard per job. In fact, a single blackboard is used to store both user and system data. A one-to-one correspondence between jobs and blackboards makes the prototype system slightly simpler to implement, and provides equivalent computational power. The drawback is that jobs that use identical sets of input tuples cannot share those tuples: to save blackboard space or to make use of previously distributed values.

4.4.1 External Data Structures

Job data can be divided into three groups: information about the job itself, information about the programs used to instantiate the tasks within the job, and information describing each of those tasks.

The first group of data consists of the name of the job, who created it, the ACL attached to it, and an execution priority. All of this information, except the execution priority, is passed along to the `Bb_Create` operation used to create the blackboard that will contain the job, and is therefore stored in the blackboard's descriptor tuple in the master blackboard. The execution priority of a job is recorded in a tuple stored in the job's blackboard.

The second group of data contains the programs that are available to instantiate the tasks in the job. Programs are stored separate from the task descriptor tuples, because multiple tasks will often share a single program. Storing a copy of a program in each

```
99556
mpqs_piece
1
99556
+
g all-users@cs.cmu.edu rw
g all-hm@cs.cmu.edu rw
u wax.TaskManager@cs.cmu.edu rwa
u pds@cs.cmu.edu rwa
-
 .
```

Figure 4.5: An example task descriptor taken from the integer factoring application. The first line of the tuple is the name of the task. The second line identifies the program that should used to instantiate the task. The third line specifies how many 'command line' arguments the task has. Those arguments follow immediately after the count, one to a line. After the arguments is the ACL for the task.

task descriptor could waste a lot of blackboard space, and network bandwidth. Each program has a user-defined name and one or more processor-type-specific instances. The name of a program is defined when an external process loads it into the blackboard. Instances of a program are stored in tuples with keys that include the program name and the machine architecture for which the instance is appropriate. For example, the instance of the program called "`mpqs_piece`" that would run on a PC running *Mach* 3.0 would have the key `Program.mpqs_piece.i386_mach`.

The third group of data consists of the tuples that describe each of the tasks in the job. All of the information that is specified when spawning a task is recorded in a single tuple. This includes the name of the task, the arguments that should be passed to it when it is scheduled, the name of the program to use to instantiate the task, and the ACL for the task. Figure 4.5 shows an example.

Every *Wax* task is in one of the following states:

Queued - The task is waiting to be executed.

Checkpointed - The task is waiting to be executed, and it will be resumed from an intermediate state.

Active - The task is believed to be executing.

Exited - The task has terminated.

Killed - The task was terminated by a user or another task.

Died - The task is considered to have terminated, because the process used to instantiate it died before invoking the *Wax* exit operation.

The first three states are used to manage the execution of *Wax* tasks unterminated tasks. For a particular priority level, a Task Manager will attempt to schedule **Checkpointed** tasks before **Queued** tasks, so as to be able to recover the resources used to store the checkpoint record. The three terminated states are intended to provide users will additional information about the execution of their computations. In addition, when a Host Manager reports that a task has entered the **Died** state, the Task Manager recording that event will not attempt to terminate any other executing instances of that task. The idea behind this

```
<4015@128.2.222.173>.Checkpointed.200.0.0.0.0
<4015@128.2.222.173>.Queued.0.0.0.0.0
<4015@128.2.222.173>.Active.0.128.2.242.194
<4015@128.2.222.173>.Active.0.128.2.205.7
```

Figure 4.6: A hypothetical set of tuples describing the state of task `<4015@128.2.222.173>`. The first tuple records the existence of a checkpoint record that can be used on processors of type 200 (hex), which are PCs running *Mach* 3.0. The second tuple contains the information needed to instantiate the task. The last two tuples indicates that the task is currently assigned to hosts `128.2.242.194` and `128.2.205.7`. A status query on this task would indicate that the task is active.

distinction is that an instance of a task might die due to local resource exhaustion on one machine, but another instance may successful complete execution.

The current state of a task is recorded using the keys of one or more tuples. Each key includes the unique identifier of the task to which the state information applies, the name of a task state, and up to two more pieces of state-specific information. The key for a task in the checkpointed state will indicate the type of processor that wrote the checkpoint record. The key for an active task record includes the host on which the task is believed to be executing. Keys for tuples recording the various terminated states will include the host where the operation that produced the termination record was invoked and the type of termination recorded by that operation. Since a task may be executing on multiple machines, and/or have checkpoint records for multiple processor types, there may be multiple tuples recording such states.

The current state of a task is determined by examining the keys of all of the tuples that are prefixed with the unique identifier of the task. For example, based on the tuples listed in Figure 4.6, task `<4015@128.2.222.173>` would be considered to be in the active state, because instances of the task are believed to be executing on hosts `128.2.242.194` and `128.2.205.7`. The task also has a checkpoint record that can be used by processors of type 200 (hex), which are PCs running *Mach* 3.0.

The task state information is recorded in the keys, rather than in a tuple value, because a value-based mechanism could lose state information. If the current state was recorded by writing it to a single tuple, then that tuple would have multiple values, and the *Wax* programming model does not define which value is returned if a tuple has multiple values. While the current blackboard implementation deterministically returns the most recently written value, relying on that behavior would make the Task Manager sensitive to changes in the internal implementation of the blackboard mechanism. If a value-based mechanism, instead, deleted any existing version of the state tuple before writing a new value, then the state of the task would be lost should the Task Manager crash after deleting the old value, but before writing the new value.

4.4.2 Job and Task Operations

The user interface to the *Wax* task heap is implemented as a library of wrappers around RPCs. As with the blackboard module, the library provides separate interfaces for external processes and for *Wax* tasks. The routines in the first interface are wrappers around RPCs directly to a Task Manager. The routines in the second interface are wrappers around RPCs to the Host Manager executing the *Wax* task, which then forwards the RPC to a Task Manager.

The rest of this section describes the implementation of the various job and task operations. The descriptions of the operations are grouped according to which interfaces they are part of. The first group describes the operations that exist in both interfaces. The second group covers those routines that exist in the interface used by external processes. These include operations for manipulating *Wax* jobs. The third group describes one operation that only *Wax* tasks can invoke.

All of the job and task operations that manipulate an existing job will fail and signal an error if the the servicing Task Manager is unable to map the job identifier into an accessible blackboard. Similarly, all task operations that manipulate an existing task will signal an error if the servicing Task Manager specified task identifier does not correspond to a task in the specified job.

Routines Common to Both Interfaces

The `GetTaskList` routine implements the *list* operation. The routine returns a list containing the current state, status, and location information for tasks in the specified job. The list will include all tasks known to the servicing Task Manager for which the invoker has read access. Successive operations that are serviced by different Task Manager may return different sets of tasks. If the caller of the routine does not have read access to the job, the operation will signal an error.

The `TaskGetInfo` routine implements the *status* operation. It returns the name of the specified task and its current state, status, and location. If the caller of routine does not have read access to the specified job and task, the operation will signal an error.

The `TaskKill` routine implements the *kill* operation. If the specified task has not already exited, the Task Manager will record the task as having been killed in the task heap. The servicing Task Manager will attempt to notify any Host Managers that the Task Manager believes to be processing the task. Since the *kill* operation is only intended to be used to optimize resource utilization, no error is signalled if the Task Manager is unable to contact such a Host Manager.[12] If the caller of the routine does not have write access to the specified job and task, the operation will signal an error.

The `TaskSpawn` routine implements the *spawn* operation. It creates a new task in the specified job. The caller of the routine passes the name of the program to be used to instantiate the task, the arguments to passed to the task when it is first started, and the ACL for the task as arguments to the routine. The servicing Task Manager records that information in the tuple that indicates that the task is in the **Queued** state (see 4.4.1). If the caller of the routine does not have write access to the specified job or the program needed to instantiate the task does not exist, the operation will signal an error.

[12]Even if all *known* active instances of a task are interrupted, active instances may still exist due to previous network partitions or the limited consistency maintained between Task Managers.

Additional Routines in the External Interfaces

The `WaxExt_JobCreate` routine creates a new *Wax* job. The operation is passed the name of the job, the execution priority for the job, and the ACL to associate with the new job. The operation uses this information to create a new blackboard, which will be used to store the user and meta-data for the job. Wax requires that the jobs belonging to a particular *Wax* principal must have unique names. If a job with the specified name already exists, the operation will signal an error. The operation will also fail if the Task Manager processing the request is unable to parse the ACL for the new job.

The `WaxExt_GetJobByName` routine returns the unique identifier for a job given the user assigned name of the job. The operation will fail if the caller of the routine does not have a job with the specified name.

The `WaxExt_GetJobList` routine returns the unique identifier of each job for which the invoker of the operation has read access.

The `WaxExt_JobAddProgram` routine adds a program image to a job. The caller specifies a job, the (network accessible) path of the program to add, the job internal name for the program, and type of processor on which the program can run. If the Task Manager can access the external copy of the program, then a copy of the program is stored in a tuple on the blackboard for the specified job. If the program has previously been added to the job, then the old copy of the program is replaced. The operation will fail if the the caller of the routine does not have administrative access to the specified job.

The `WaxExt_JobDelete` routine deletes the specified job. If there are still executing instances of tasks in the job, the servicing Task Manager will attempt to terminate the execution of those instances. The blackboard used to hold the job will then be deleted. If the caller of the routine does not have administrative access to the job, the operation will signal an error.

The `WaxExt_JobGetInfo` routine returns the owner, name, and execution priority of a job. If the caller of the operation does not have read access to the job, the operation will signal an error.

Additional Routines in the *Wax* Task Interface

The `Wane` routine implements the *exit* operation. The operation informs the *Wax* system that a task has finished executing and provides an exit status code that can be retrieved using the `TaskGetInfo` operation. In addition, the operation terminates execution of the *Mach* task that is used to instantiate the *Wax* task. This operation never returns to its caller.

4.4.3 Task Manager

When a Host Manager requests a task, the Task Manager selects the highest priority task whose resource requirements are satisfied by the requesting machine. After recording the machine that will receive the task, the Task Manager sends a descriptor for the task to the Host Manager. The task descriptor consists of the job to which the task belongs, its blackboard, the name of the task, the program to be executed, and any arguments specified by the spawning task. The Task Manager monitors the status of all machines executing tasks. If it determines that a machine is down or malfunctioning, it marks the task as available for assignment to another machine.

All operations that read or write task heap information are performed by a Task Manager. Each time a principal makes a request, the Task Manager checks the appropriate job and task ACLs before executing the requested operation. Task Managers check machine ACLs when assigning tasks to machines. After finding a task that the requesting machine can access, the Task Manager checks that the principal who owns the task may use the requesting machine. If the principal may not use the machine, the Task Manager will look for another task to assign.

A Task Manager functions independently of any other Task Managers that may be active. Each maintains an internal cache of the jobs and tasks that exist in the local domain. Initial scheduling decisions are made using the data in this internal cache.

The only communication between Task Managers is through *Wax* blackboards. In particular, a Task Manager does not know whether there are any other Task Managers active in the same domain. In order to keep multiple Task Managers loosely synchronized, each instance will periodically attempt to scan all of the jobs known in the local domain. The inability of a Task Manager to complete a scan due to network or processor failures does not effect its ability to correctly schedule tasks, since the *Wax* programming model provides at-least-once semantics for the execution of tasks.

4.4.4 Enforcing Job and Task ACLs

The ACL for a job is defined by the user creating it. Tasks inherit the ACL of the job to which they belong, subject to additional restrictions specified at task creation. ACLs for tasks and jobs can only be specified when they are created, because there is no mechanism in *Wax* for synchronously updating all copies of a blackboard entry, such as, a job or task descriptor.

The ACLs on jobs and tasks can also be used to protect *Wax* computations from malicious resource providers, by allowing users to control the spread of their computations. When a user creates a job, *Wax* is provided with a list of the machines the user trusts to perform the computation. If a user wants to use a domain or machine that is not fully trusted, redundant computation can be used to check the results. For example, if a task was processed by a suspect machine/domain, another task could be spawned to perform the same piece of the computation, with the machine or domain used by the first task added to the negative ACL of the new task to avoid having the same machine perform both tasks. The user or "master" task controlling the redundant computation would use the list operation to determine where the first task had executed.[13] In this usage, the ACLs might be bettered viewed as distribution control lists.

[13]The reliability of this technique, given that *Wax* may migrate executing tasks among several machines, is unclear.

4.5 Host Managers

Every machine that executes tasks for *Wax* runs a Host Manager. The Host Manager monitors activity on its machine, starts tasks for the system, and connects tasks executing on its machine to the rest of *Wax*. When the Host Manager determines that its machine is idle, it contacts the local Task Manager, requesting a task to execute. If there is a task available that can execute on the Host Manager's machine, the Task Manager sends it to the Host Manager. The Host Manager then starts the task and waits for it to finish. As described in Sections 4.3.1 and 4.4.2, a task sends all of its blackboard and task operations to its local Host Manager via RPCs. The Host Manager performs the blackboard operations directly and relays the task heap operations to a Task Manager.

4.5.1 Monitoring Machine State

In addition to serving as *Wax*'s local agent on a machine, the Host Manager works on behalf of the primary users of the machine to keep *Wax* tasks from interfering with local work. To achieve this goal, the Host Manager monitors process and user activity. Process activity is monitored by using the *Mach* system call *host_info* to track the average number of *Mach* threads that are executing. A one minute load average is used, because that is longest period provided by *Mach*. By default, the Host Manager will check the load every 15 seconds. This period was chosen as a compromise between checking only occasionally so as to not consume too many CPU cycles when the machine is busy with local work, and checking frequently so that *Wax* tasks will be shutdown quickly when local activity resumes after the machine has been idle. The Host Manager monitors user activity by checking the access and modification times of the console device attached to the machine.

A machine is considered to be busy with local work if the machine's console is being used or the average process load exceeds a certain threshold, by default 1.4 runnable threads on a single processor workstation. Conversely, a machine is considered to be idle if the machine's console has not been used recently and the process load is below a certain threshold. The default thresholds for a machine to be considered idle are that the console has not been used for ten minutes and the load average is below 0.25 runnable threads. The various parameters that affect the determination of whether a machine is considered busy or idle can be adjusted with command-line arguments to a Host Manager.

Using the access times on a machines console is not necessarily a reliable means of detecting when a user is actively using the machine. Windowing systems, particularly older releases of X Windows [SG86], do not necessarily update the access time of the console device when a user types on the keyboard or clicks a mouse button. As a compile-time option, Host Managers can be configured to also query a locally running X server for keyboard activity. This option is disabled by default because linking the Host Manager with the necessary X libraries significantly increases the size of the resulting binary, and most machines at CMU are now running version of X that do update the access time on the console device. Furthermore, the X libraries are not reentrant, with the result that X activity can only be monitored until the X server is shutdown for the first time after a Host Manager is started. While this problem may be fixed in Release 6 of X11, and could be avoided by restarting the Host Manager, whenever the connection to the X server is closed, the incremental benefit of being able to identify more types of user activity may not be worth the cost in monitoring overhead.

4.5.2 Starting *Wax* Tasks

When a Host Manager receives a task from a Task Manager, the Host Manager first reads the tuple that contains the task descriptor for that task. If the Host Manager is unable to read the necessary tuple, it will report the failure to the Task Manager and request a different task. Once the task descriptor has been successfully read and parsed, the Host Manager will attempt to read a checkpoint record for the task. If such a tuple exists, the Host Manager will restore the saved state into a new *Mach* task, otherwise the Host Manager will (re-)start the task from the beginning. The Host Manager always attempts to read a checkpoint record, because such records may be created without the Task Manager being informed. In particular, a Host Manager may crash after writing a checkpoint record, but before it is able to notify a Task Manager that it checkpointed the task.

In the *Wax* programming model, the machines that execute *Wax* tasks only need to provide processor cycles and (virtual) memory. This limited model of a machine allows *Wax* to isolate executing *Wax* from other tasks and resources on the same machine. A Host Manager uses two features of *Mach* 3.0 to isolate executing *Wax* tasks. First, the *Mach* 3.0 system-call redirection mechanism [Jon92] is disabled. Second, the Host Manager prevents the *Mach* tasks used to process *Wax* tasks from receiving a copy of the send right needed to access the name service provided by the NetMsgServer.

In *Mach* 3.0, most of the typical operating system services are provided by servers running outside of the *Mach* kernel. A system-call redirection mechanism in the kernel is used to redirect hardware system calls made by a task to one or more emulation routines in that are located in the task's virtual memory. The emulation routines either perform the operation or forward the request to the appropriate server. In particular, the standard UNIX system calls are provided on most machines running *Mach* by the UX server [GDFR90]. The UX server automatically installs the necessary emulation code in the address space of every task created with the UNIX *fork* operation. When the redirection mechanism is disabled, the kernel returns an error whenever a task makes an unsupported system-call. *Wax* tasks are run without special privileges so the *Mach* 3.0 system calls cannot be used to examine objects outside of a task.

The send right needed to access the NetMsgServer is propagated from task to task by the *Mach* operation, *task_create*, used to create new tasks. The *mach_ports_register* operation allows a task to change, or remove, the send right that any tasks subsequently created by the first task could use to contact the NetMsgServer. Host Managers use *mach_ports_register*, when they first start running, to remove the send right, thereby isolating *Wax* tasks from all tasks that do not explicitly provide them with send rights.

A Host Manager instantiates a *Wax* task using a *Mach* task, one or more *Mach* threads, and either a checkpoint record or the program specified in the task descriptor for the *Wax* task. A Host Manager creates a *Mach* task and initializes its virtual memory to contain the text and data stored in the task's program or checkpoint record (see Section 4.5.3). When the *Wax* task is started for the first time, that is from the program rather than a checkpoint record, the Host Manager will also allocate and initialize a stack segment for the first thread. The stack segment is set up to contain the arguments listed in the *Wax* task's task descriptor. The arguments are arranged such that the C runtime initialization code in the task will be able to use them as the standard `argc` and `argv` arguments to the routine in the program called `main`.

Having prevented the newly created *Mach* task from accessing the NetMsgServer, the Host Manager must explicitly give the task the send rights it will need in order to communicate with the Host Manager. The Host Manager does this with the *Mach* system call *mach_port_insert_right*, which allows one task to insert port rights into another. The Host Manager gives the port a name that is known to the *Wax* runtime library.

Once a Host Manager has set up the *Mach* task and the necessary communication links, it uses *task_resume* to set the task running, and then waits for the task to invoke *Wax* operations.

Implementation Bug

Having a Host Manager re-start a task when a network failure prevents it from reading the checkpoint record for the task breaks the forward progress guarantee of the *Wax* programming model. If a checkpoint record exists for a task, but is inaccessible when the task is assigned to a machine, the Host Manager manager on that machine will restart the task, and may then overwrite the checkpoint record with one for a less advanced state. This bug can be fixed by having the Task Manager tell the Host Manager when a checkpoint record must exist. If the Host Manager was unable to find the checkpoint record for such a task, it would then report the error to a Task Manager and request a different task.

4.5.3 Process Checkpointing

Host Managers checkpoint executing tasks at hourly intervals and when a machine becomes busy with local work. The checkpoint record for a task is recorded incrementally. The first step in the process is to force the *Mach* tasks and threads into a state where a consistent snapshot can be taken that does not leave any residual dependencies on the current host. The second step updates the resource usage information that is kept for every *Wax* task. Finally, the internal state of *Mach* task is scanned and then recorded in a tuple in the internal namespace of the job's blackboard. The key for the tuple storing the checkpoint record consists of the unique identifier for the task, the string "`.Checkpoint.`", and a string identifying the processor-type of the host performing the checkpoint operation. For example, a checkpoint record for the task "`<2@128.2.222.173>`" made by a Host Manager running on a DECstation 5000 would stored with the key "`<2@128.2.2.22.173>.Checkpoint.pmax_mach`".

Implementing a checkpoint mechanism for the processes used to run *Wax* tasks is easier than building such a mechanism for standard UNIX processes. The restricted machine model provided by the *Wax* programming model reduces the possible dependencies that *Wax* task can have on the machine executing it. As a result, checkpointing a *Wax* task involves recording the state of the *Mach* task and threads that are being used to process the *Wax* task, and the state of the communication links between the task and its local Host Manager.

The *Mach* system call *task_suspend* is used to stop the execution of all the threads that exist in the *Wax* task. This prevents the threads from executing any further user-level code, but threads may still be executing, or blocked, inside of the *Mach* kernel. *Mach* provides the *thread_abort* function that forces threads to return to user-level code. When the aborted thread is allowed to execute again, it will either repeat the *Mach* operation or return an error code indicating that it was interrupted. Since *thread_abort* has no effect on threads that are not executing *Mach* system calls, the Host Manager calls *thread_abort* on every thread that exists in the task being checkpointed. After this has been done, all of the threads are in states where they can be migrated to another machine.

Once the *Mach* task has been forced into this completely suspended state, information about the resources used by the *Mach* task are retrieved from the *Mach* kernel. These new statistics are merged with the information from any previous periods of execution, and recorded in the task checkpoint record.

Determining the state of *Mach* communication links is the most complicated step in the checkpoint process. *Mach* provides a mechanism, *mach_port_names*, for determining what ports exist in a *Mach* task and extracting the task-specific state of those ports. Unfortunately, *Mach* does not provide as straightforward a mechanism for identifying to what other tasks, and which ports in those tasks, the first set of ports connects. The restricted machine model supported by *Wax*, however, helps the situation by restricting the range of possible communication endpoints. Without access to the *Mach* NetMsgServer (see Section 4.1), all ports in the task being checkpointed must be for communication links between the task and the Host Manager, between the Task Manager and the *Mach* kernel, or within the task itself. This restricted domain for the ports in the task, enables the Host Manager to identify the other endpoints for each of the ports under most circumstances.

The port identification process consists of extracting each unidentified send right from the target task into the Host Manager. The name of extracted port right is then checked against the set of end points that are known to exist: in the kernel for task and thread ports, or for any of the communication links that exist between the Host Manager and the task. If the port matches any of the known communication links, then the pairing is recorded. If it does not, and the task has only send rights for the port, the Host Manager aborts making the checkpoint, because the complete communication state of the task could not be successfully checkpointed.[14]

A similar matching process is performed for the receive rights found in the target task. Since the Host Manager has no means of verifying that it has found all of the matching send rights for a receive right, the Host Manager simply assumes that it has, which is a reasonable assumption for well-behaved tasks.

There are unfortunately a few types of port rights that are impossible to checkpoint, because *Mach* provides no means of identifying the other end of the communication link. While most of these rights are rarely used, the *send-once* right is used by the *Mach* RPC mechanism for most reply ports. Based on the thousands of checkpoint records that have been written by the prototype system, this does not seem to be a problem that is encountered often. Computationally intensive tasks, which are the type of task best suited to *Wax*, are unlikely to spend much time sending messages. So, most checkpoints will

[14]This has never been observed in practice, and would probably be a sign that the security mechanisms in *Wax*, and/or the *Mach* kernel, had been breached.

probably be made when such tasks are computing, rather than communicating.

Once all the ports in target task have been identified, the information relating to each port, including any queued messages, is extracted from the kernel and written to the checkpoint record. After the ports have been recorded, the register state of each thread in the tasks is written to the checkpoint record. And, finally the current state of the virtual memory of the task is recorded.

If all these operations are completed without errors, the record is saved in the blackboard, and in any case the *Mach* task is terminated. The task must be terminated, because the checkpoint process corrupts the state of the communication links used by the task. In particular, the Host Manager removes all receive rights from the task, so that it record any messages that may be waiting to be received. *Wax* originally checkpointed tasks only when the task was going to be terminated because the local machine had become busy, which meant there was no need to leave the state unchanged, and assuming that the task will be terminated simplifies the effort of extracting the port information. As a result, the incremental checkpointing that *Wax* now provides requires checkpointing the task and creating a new *Mach* task from the new checkpoint record. This two step mechanism has idled machines unnecessarily when temporary network failures have prevented the storing or retrieving of the checkpoint record.

A Host Manager will fail to write a checkpoint record in several situations. These situations include the blackboard used by the task being unreachable, the Host Manager being unable to completely determine the communication state of the executing task, and the checkpoint record being too large to store in the blackboard. The first situation is the only one that has actually been observed with the prototype system. If the Host Manager is unable to checkpoint a task, any work since the last successful checkpoint that is not yet recorded in the blackboard of the task is lost.

Restoring a task from a checkpoint record is a simple matter, only slightly more complex than setting up a task from a program. The Host Manager reverses the process of checkpointing the task. First, the statistics are read and saved for later use. Then, the port information is read from the checkpoint record and used to created the necessary ports in the new *Mach* task. In particular, the links between the task and the Host Manager and the kernel are reestablished. Then, the thread information is extracted, and new *Mach* threads are created and initialized. Finally, the contents of the *Mach* task's virtual memory are initialized form the checkpoint record.

4.6 Distributed Blackboard and Job Data

Distribution Managers are the only externally visible part of a domain, and allow an organization to be part of *Wax* without having to reveal its internal structure. Distribution Managers transfer tasks among domains and provide remote domains with copies of blackboard tuples that can be cached. When moving data for a job between domains, the Distribution Managers always update the contents of the job's blackboard and propagate new task creation records before transferring any task termination records. By ordering data transfers in this way, a *Wax* user in the domain where a job was created can always determine whether the job has terminated without having to check with other domains.

Distribution Managers also enforce restrictions on the spread of information between domains. ACLs attached to every task and blackboard specify the domains that may access the object. These ACLs are specified by the task creator. Every domain has a resource usage policy that defines which internal users and which external domains may use the domain's resources and to what extent. When Distribution Managers exchange tasks and blackboard entries, the sender checks the ACL on the entry. If the entry may be sent to the remote domain, the receiver checks whether the creator of the entry may use the domain's resources. The sharing of information between domains does not have to be symmetric; the fact that one domain may process tasks from another does not require that the second domain also process tasks from the first.

A Distribution Manager always pulls information from other Distribution Managers, information is never pushed onto a domain. Pulling data allows a domain to fetch data at its convenience, rather than at the demand of another domain. While data is always pulled into a domain, a domain can request that another domain initiate a fetch. The ability to make such requests is needed to support the `Bb_Flush` operation.[15]

The ACL on a domain **cs.cmu.edu** defines the set of domains that may access data belonging to **cs.cmu.edu**. The Distribution Managers in **cs.cmu.edu** also use the domain ACL to determine the domains from which to accept data. For example, if the ACL on **cs.cmu.edu** lists domains **cs.washington.edu** and **wesleyan.edu**, then the Distribution Managers in **cs.cmu.edu** will only send data to Distribution Managers in those domains and will only accept data belonging to domains **cs.cmu.edu**, **cs.washington.edu**, and **wesleyan.edu**. While exchanging data, Distribution Managers also check the ACLs on individual blackboards for additional restrictions.

4.7 Additional *Wax* Interfaces

Wax provides users with several additional interfaces to the system beyond those needed to support the *Wax* programming model. In particular, *Wax* provides an alternate interface to the blackboard, as a replacement for the UNIX system calls that manipulate files in order to simplify the effort required to port an application to *Wax*. The other interfaces are a set of shell-level tools that allow users to examine and interact with *Wax* jobs.

4.7.1 Alternate Blackboard Interface

The *Waxio* interface, modeled after the UNIX *stdio* routines, maps file operations into operations on blackboard tuples. With this interface, an applications can be ported to *Wax* without writing any new code, if the programmer can identify all of the files accessed by the program. First, a program that can be used to instantiate a *Wax* task is created by linking the existing application with the *Wax* libraries. Second, a companion shell-level tool `wax_run`, described in Section 4.7.2, is used to submit the new binary to *Wax* for processing.

[15]In the absence of the second domain needed to test, and use, Distribution Managers, I have not completed the implementation of the *Wax* Distribution Manager.

```
/afs/cs.cmu.edu/user/pds/application
/afs/cs.cmu.edu/user/pds/application/input WaxIo.1 0
/afs/cs.cmu.edu/user/pds/application/output WaxIo.2 0
```

Figure 4.7: This is the contents of the *Waxio* file descriptor tuple for a hypothetical job. The first line specifies the current working directory for any tasks in this job. The remaining lines identify all the files that tasks may attempt to access using the *Waxio* package. The first field on each line is the name the file that the application expects to be accessing. The second field is the name of the blackboard tuple that contains, or will contain, the contents of the corresponding file. The third field is used to mark files that the *Waxio* package should automatically associate (open) with particular file descriptors. Zero indicates that the file will only be explicitly opened.

The *Waxio* package provides a simple emulation of the UNIX file operations that is sufficient to support applications that use files to store input and output. Because *Wax* blackboards do not provide the same consistency semantics as are provided by the UNIX file system, *Waxio* is not capable of supporting simultaneous access to a file by different tasks.

The package uses a separate blackboard tuple to store each file that the application will access. When a task opens a file for reading or updating, the tuple is read into the task's virtual memory. All read and write operations effect the contents of the tuple cached in the task's virtual memory. When a task closes a file that it had opened for writing, the package will write a new tuple to memory.

The file names used by a *Waxio* application cannot be directly used as the names of the tuples that will store the file data, because they may be too long or contain characters ('/') that are not allowed in tuple's names. To solve this problem, the *Waxio* package uses an extra tuple to record a mapping between file names and tuple names that is determined when a job is first created, usually by the `wax_run` program. When a task first attempts to access a file, the package reads the tuple that contains the mapping and caches it.

Figure 4.7 contains an example of the extra tuple used by the *Waxio* package to map between file names and tuples. The first line of the tuple specifies the initial current working directory for all tasks in the job using the descriptor. This allows the *Waxio* package to map any relative pathnames used by the application into the abolute pathnames recorded in the descriptor tuple. The remainder of the tuple defines the mapping between two files (input and output) and their corresponding blackboard tuples.

4.7.2 Shell-level Tools

The tools described in this section allow users to interact with the *Wax* system. Each of the user-level programs is presented separately.

wax_admin

The wax_admin program allows domain administrators to control and monitor the status of Task Managers. In particular, administrators can use this program to force a Task

Manager to re-read the file that contains the ACL for the local domain. The program also can be used to get a Task Manager to dump status information about memory usage and open communication links to a log file.

wax_extract

The wax_extract program removes tuples created by an application using the *Waxio* package from the blackboard of a job and stores them in the files the application code believed it was accessing. Command line switches allow the invoker to extract all of the tuples, or only those tuples that correspond to particular files.

The program can also be used to list the status of some or all of the tuples used by the *Waxio* package. The difference between using wax_extract and wax_ls to list this information is that wax_extract will print the information in terms of the application specified file names, while wax_ls would list the name of the tuples used to store the files.

wax_ls

The wax_ls program enables users to monitor the status of jobs that have been submitted to *Wax*. The program also allows users to list the blackboard contents of their jobs, but there is not currently a shell-level *Wax* tool for accessing blackboard tuples.

wax_rm

The wax_rm program removes a job and its associated blackboard and tasks from *Wax*. Any active tasks are killed, and the blackboard storage used by the job is freed.

wax_run

The wax_run program allows users to submit jobs that will use the *Waxio* package. The program accepts a list of input and output files that the a job will access and a list of tasks to spawn. The program creates a job, copies in all the input files to the blackboard of the job, writes the mapping tuple needed by the *Waxio* package, and spawns the specified tasks.

4.8 Evaluation of the Implementation

This section examines the implementation of *Wax*. The examination focus, in particular, on the implementation's use of various features of the *Mach* operating system. While *Mach* has provided a useful testbed for building and evaluating *Wax*, its limited availability restricts the use of *Wax* at other sites. Several design choices are evaluated and alternatives to Mach-specific solutions are proposed.

4.8.1 The Use of the Mach System Call Mechanism

The *Mach* system call redirection mechanism is one of the key techniques that *Wax* uses to protect resources connected to the system. Unfortunately, the technique is not directly portable to other operating systems. However, the "`/proc`" file system provided by System V compliant operating systems, which include OSF/1 and Solaris, is a possible replacement. Among other features, the file system allows one process to trap the execution of all system calls performed by another process. The Host Manager would use this feature to check that *Wax* tasks were only accessing the files used to store blackboard tuples and no others.

4.8.2 The Use of Mach IPC

The use of *Mach* IPC between processes on different machines was a mistake. The NetMsgServer was designed to extend the IPC mechanism provided by *Mach* 2.5 kernels. Since very few people use *Mach* IPC between machines, the code has not been heavily tested, or well maintained. In addition, it does not support any of the new features provided by *Mach* 3.0 IPC. In particular, a task waiting for a message to arrive on a port will not be notified that a message can never arrive, because no tasks have send, or send-once, rights for the port. Given the ways that *Wax* uses the RPC mechanism, this problem typically occurs as the result of a machine crashing. If this happens when a task is waiting for the reply message of an RPC, the task will wait forever. The only mechanism that *Mach* provides for avoiding this problems is the use of a timeout on the RPC, which places an arbitrary upper bound on the time a server has to perform an operation.

To avoid having to rely on a time out mechanism to recover from such failures, the connection module explicitly manages the ports used to receive reply messages for any connections that may go between two machines. The connection module associates each reply port **R** with the port **P** (send right) that is used to send the request message. If the task holding the receive right for **P** dies, the *Mach* kernel, perhaps with the aid of the NetMsgServer, will eventually notify the connection module that **P** is no longer valid. When the connection module receives such a notification, it deallocates **R**, which will cause any pending *Mach* message operation, in particular, *mach_msg* with the receive option, using **R** to return an error.

4.8.3 The Use of Mach/C Threads

I have made heavy use of *Mach* threads, in general, and the C-Threads package [CD88], in particular, in implementing the *Wax* system processes. As a result, those processes are only readily portable to other operating systems that provide pre-emptively scheduled threads, such as Solaris and OSF/1. Such operating systems presently, or will soon, support the emerging POSIX threads (Pthreads) standard [P10], which provides replacements for the C-Threads routines that *Wax* currently uses. Most of the work needed to port *Wax* from C-Threads to Pthreads is isolated in two files.

4.8.4 AFS as the Blackboard Store

AFS is not the right storage mechanism for *Wax* blackboards. While it provided a convenient mechanism to use in the prototype implementation, the AFS failure model is too pessimistic for *Wax*. In order to approximate the consistency guarantee of the UNIX file system, AFS will not allow processes to access cached files when the Cache Manager is unable to contact a file server to verify the validity of the cached data.

Another problem with AFS is that it locks up a machine when the Host Manager records a checkpoint record. This is particular troublesome when a Host Manager is attempting to checkpoint a task when a user reclaims the machine. AFS operations that are in progress seem to interfere with the UX server's ability to perform other, non-AFS, operations. From simple experiments using FTP, the interference seems to be greater for AFS related operations than for other network operations.

A possible alternative to using AFS would be the Coda file system [SKK+90, KS92]. Coda, which is derived from an early version of AFS, uses optimistic replication to improve data availability. The limited consistency guarantees of the *Wax* data model would make it easy to automate failure recovery. The drawback to Coda is that it is not as widely distributed as AFS.

4.8.5 The *Waxio* Package

The *Waxio* package was added to *Wax* largely as an after thought. In hindsight, this package is an important part of *Wax*, because it can greatly simplify the work that an application programmer must perform to port an application to *Wax*. For computations, where many runs of a serial application are needed, porting can be as easy as relinking the object file.

Chapter 5

Applications

Chapter 3 described a programming model that was designed to be feasible to implement in a distributed computing environment. In order for that model, or an implementation of the model, such as the one described in Chapter 4, to be interesting and/or useful, there must exist a class of applications that can be solved using the model. In this chapter, I identify and describe such a class of applications. In addition to describing the general characteristics of applications that can be solved using the model, this chapter outlines the implementations for several example applications. The remainder of the chapter presents guidelines to help an application programmer determine whether or not a particular application can be implemented using *Wax*, and explores some of the programming issues that need to be considered in implementing an application using *Wax*.

The *Wax* programming model can, in principle, be used to solve all applications that have only a limited reliance on global data consistency. This class of applications includes applications where the individual tasks are independent, that is pairs of task do not exchange results. Examples of such applications include independent distributed simulation [JCRB89], graphics and film animation [LC92], and many probabilistic algorithms, such as Manasse's integer factoring application [LM89]. The class also includes applications, such as the quadratic assignment problem [PC89], that share intermediate results solely in order to reduce the amount of computation. In addition, applications that have multiple phases with data shared between tasks in different phases can also be implemented with the aide of the barrier synchronization mechanism described in Section 3.5.1.

The applications that will perform well on a distributed implementation of the model have the following additional characteristics:

- coarse-grain parallelism
- individual pieces that are computationally intensive
- moderately-sized data sets

The first two properties are needed to take advantage of the parallel processing capabilities of an implementation in a distributed computing environment and to minimize the impact of the overhead of distributing pieces of a computation to processors. However, coarse-grain parallel applications in which large amounts of data are shared among the processing elements may not be able to take advantage of the parallelism provided by such an implementation, because of the need to wait for large amounts of data to be moved across (relatively) slow networks.

Wax **Model Provides**	**The Programmer Provides**
Access to computing resources, such as processors, physical and virtual memory, and blackboard storage	Hints about the resources needed to process a job/task
Automated task scheduling, including checkpointing active tasks and restarting failed tasks	Mechanisms to recover from inconsistencies created by multiple execution of tasks
A shared-data store (blackboard)	Mechanisms to maintain application required consistency of the data store

Table 5.1: The division of features between those provided by *Wax* and those provided by the application programmer.

For an implementation that uses the workstations of a distributed computing environment, the cost of communication will prevent some applications from being implemented on top of *Wax*.

5.1 Deciding Whether to Use *Wax* for a Particular Application

This section provides application programmers with a framework for evaluating the feasibility of implementing a specific application using, in general, the *Wax* programming model and, in particular, a distributed implementation of that model.

As part of the evaluating an application, an application programmer should consider whether there is more than one algorithm that can be used to solve the problem. Different algorithms may result in different data consistency requirements and/or affect the size of the blackboard working set of individual tasks. While a distributed computation system implementing the *Wax* model provides the programmer with a shared data store, the algorithm that works best on a (tightly-coupled) shared-memory multiprocessor may not be the right one to use with the distributed system, since the relative costs of computation and communication in a distributed computation system differ from those of a traditional multiprocessor.

In order to use the evaluation framework, a programmer must first understand what the *Wax* programming model provides, and what it does not provide. Table 5.1 lists the features that the model supplies to an application programmer and those that the programmer must supply. The programmer can then use the flow charts in Figures 5.1 and 5.2 to evaluate the applicability of *Wax* to specific applications. The first flow chart outlines the evaluation process in general, while the second flow chart focuses on the questions used to characterize the global consistency requirements of the application. The rest of the section explores in greater detail each of the decision points listed in the two flow charts.

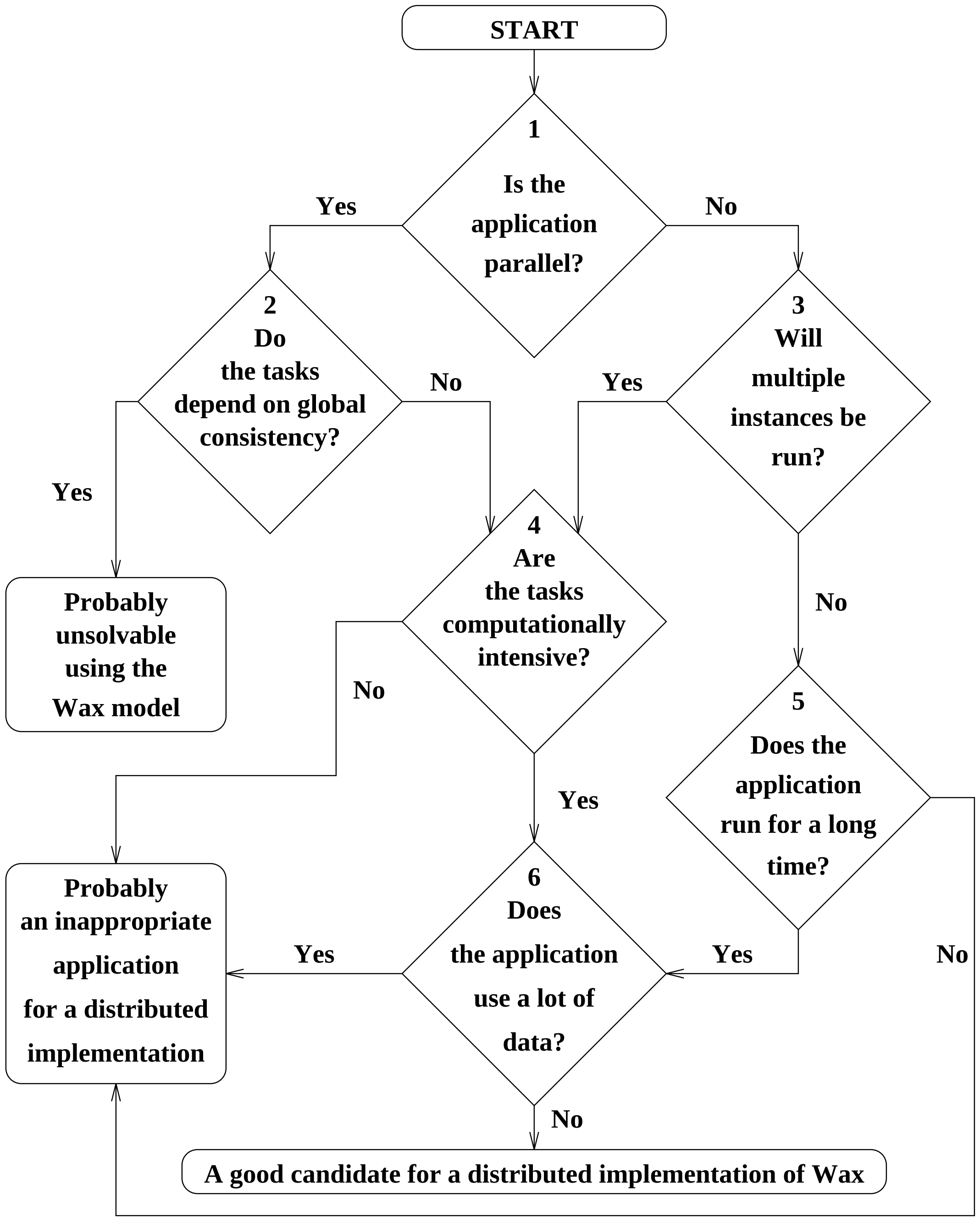
START
1
Is the application parallel?
Yes
No
2
Do the tasks depend on global consistency?
Yes
No
3
Will multiple instances be run?
Yes
No
Probably unsolvable using the Wax model
4
Are the tasks computationally intensive?
No
Yes
5
Does the application run for a long time?
Yes
No
6
Does the application use a lot of data?
Yes
No
Probably an inappropriate application for a distributed implementation
A good candidate for a distributed implementation of Wax

Figure 5.1: This flow chart outlines the process that an application programmer should use in evaluating whether or not a particular application can be solved using the *Wax* programming model, and whether the application is compatible with a distributed implementation of the model. Question 2, about global consistency, is expanded in more detail in the subsidiary flow chart shown in Figure 5.2.

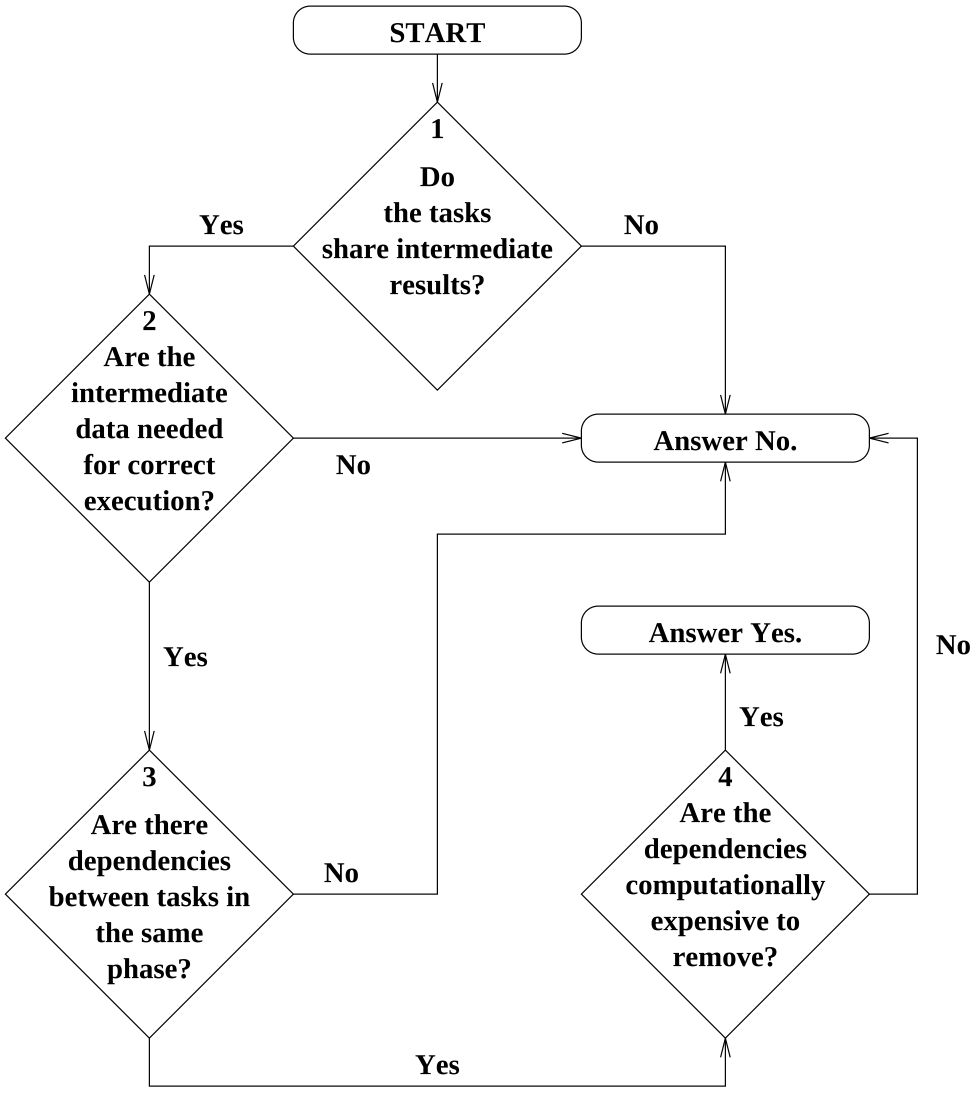
START
1
Do the tasks share intermediate results?
Yes
No
2
Are the intermediate data needed for correct execution?
No
Answer No.
Yes
Answer Yes.
No
Yes
3
Are there dependencies between tasks in the same phase?
No
4
Are the dependencies computationally expensive to remove?
Yes

Figure 5.2: This flow chart outlines the process that an application programmer should use in determining if an application requires global consistency.

5.1.1 Application Parallelism

The first decision point in the evaluation process is whether the application contains any internal parallelism, that is: can it be broken up into two or more pieces? Such parallelism is both good and bad. On the positive side, a parallel application will be able to use the potential parallelism provided by a distributed computation system. On the negative side, that parallelism raises the possibility that the various component tasks may require global consistency to execute correctly. Global consistency requirements can, at the least, create additional programming work for the application writer, and, at the worst, make the application impossible to solve within the *Wax* programming model.

If the application has no internal parallelism, the next step in the evaluation is to determine if there are other reasons for using the *Wax* model. Several possible reasons are explored in Section 5.1.2. If the application does have internal parallelism, the next decision point involves evaluating the global consistency requirements of the application, which is described in Section 5.1.3.

5.1.2 Serial Applications

All serial applications can, in principal, be implemented using the *Wax* programming model, since they have only local consistency requirements. The task performing the application might still need to execute fetch operations to bring the input data to where it is executing, but that is a data availability, rather than data consistency issue. For serial applications, using a distributed computation system is an uncertain means of a reducing execution time, particularly in an environment where the machines available to the computation system are identical to those directly available to the people using such applications. As a result, the decision to use such a distributed computation system for serial applications depends on what other benefits the system provides.

A serial application can benefit from a distributed computation system if it has external parallelism, that is, it will be run multiple times. In addition to the obvious benefit of parallel execution, the automated process management of a distributed computation system can relieve the application's user of the need to schedule and monitor the individual runs of the application.

Even if the programmer expects an application to be run only a few times, the fault tolerance provide by a system implementing the *Wax* model may make using such a system valuable if each run of the application will execute for a "long time." The exact definition of what is a "long time" depends on the MTTF of the machines that the programmer would otherwise use to process the application, and the programmer's tolerance for risk. For an observed MTTF and a given expected execution time, the programmer can determine the probability that the machine executing the application will crash before the application finishes. A long time is then that length of time at which the risk of failure exceeds the programmer's risk tolerance.

Applications that will only be run a few times and do not execute for a long time can be solved using the *Wax* programming model and a distributed computation system, but they are unlikely to benefit from the parallelism and fault tolerance that such as system provides. If the extra processor cycles available through the computation system are not important, then the application programmer is probably advised to use the local resources of a single personal workstation.

The evaluation of serial applications that the programmer decides to implement using the *Wax* model continues in Section 5.1.4.

5.1.3 Evaluating Global Consistency Requirements

The *Wax* programming model was designed to allow distributed implementations of the model to provide applications with as much of the potential parallelism available in the underlying resources as possible. The model achieves this goal by allowing such an implementation to reveal the inconsistencies and failures that occur in a distributed computing environment. The price of this parallelism, however, is that application programmers must explicitly maintain the consistency requirements of their applications.

If an application is dependent on global consistency, then the blackboard fetches needed to maintain the global consistency guarantees may (significantly) reduce the amount of parallelism that can be realized. For all such applications, the effective parallelism will be reduced by the delays caused by waiting for the consistency enforcing operations (fetches) to successfully complete.

Applications may depend on global consistency in several ways: through constraints on shared data, by relying on the order in which tasks are executed, and/or by assuming that individual tasks are executed exactly once. Some of the consistency requirements of an application's shared data are obvious, for example, a counter that is used to order events clearly needs to be updated atomically; while requirements, such as a dependence of one piece of the application on the output of another piece of the application, might not be so obvious.

Question 1 in Figure 5.2 explores whether the various tasks in the application interact. If the tasks are independent, that is they share neither data nor depend on the execution order of other tasks, then the application has no global data consistency requirements. The evaluation of such application continues in Section 5.1.4.

When an application uses interacting tasks, the programmer must, as indicated in Question 2 of the consistency flow chart, characterize the type of each interaction. If the interaction is used solely as an optimization hint, then it does not create a global consistency requirement. If the interaction is needed for the correct execution of the algorithm, then there is a global consistency requirement. If the all of the interactions are optimizations, then the evaluation of the application continues in Section 5.1.4.

Correctness dependencies are more of a problem for potential *Wax* applications. Questions 3 and 4 in the consistency flow chart (Figure 5.2) are basically prompts to the programmer to consider whether there are ways to work around such dependencies. Question 3 looks at whether the dependencies are related to different phases in the application, for example, is the output of tasks in one phase used as input by tasks in the next phase. If so, then the barrier synchronization primitive, described in Section 3.5.1, can be used to maintain the necessary dependencies.

Question 4 explores the possibility of using other techniques to avoid the dependency. The programmer can look for brute force means, such as by duplicating computation, to turn a correctness dependency into an optimization. As an example, consider an application, such as a particle simulation, where the output of one phase is used as input for the next, and each phase of the computation could be divided amongst a number of *Wax* tasks. When a task finished with its assigned portion, it could start working on the piece assigned to another task that is not currently known to be executing. Alternatively, the programmer can explore restructuring the application in order to remove the dependency. For example, if many runs of a particle simulator are required, a single task could be used for each run and parallelism achieved by executing multiple runs simultaneously.

If the correctness dependencies can be managed, or avoided, then the application can be implemented using the *Wax* programming model. The feasibility of computing the application on a distributed implementation of the model is evaluated starting in Section 5.1.4. If the dependencies cannot be managed, then the application, or the particular algorithm under consideration for performing the application, cannot be implemented using the *Wax* programming model.

5.1.4 Practical Considerations in Using a Distributed Computation System

When the programmer has determined that the application can be implemented using the *Wax* programming model, the focus shifts to whether the application is feasible to process using a distributed computation system. The answer to this question depends, in part, on the characteristics of the distributed computing environment that is used by the computation system. In order to provide a general guideline, suggestions will be made based on experiences with the prototype system described in Chapter 4 and the network environment at Carnegie Mellon University and across the Internet.

As in any parallel programming environment, the time required to create a task, assign it to a processor for execution, and recover its results is overhead that needs to be balanced against the benefit of overlapped (parallel) execution. The programmer must keep in mind that this overhead will be much higher in a distributed computation system than on a traditional multiprocessor, since each of these activities requires network communication. For example, in the prototype system, assigning a task to a processor and starting it running typically takes 3-5 seconds. As a result of the higher overhead, the application programmer may need to adjust the decomposition of the application into tasks so that each task does more work.

As with traditional multiprocessors, the amount of data used by a *Wax* application will affect its performance. Because the communication bandwidth available to tasks executing in a distributed computation system is less than that available on most traditional multiprocessors, the size of the data set will have a greater performance impact on the *Wax* implementation than on a traditional multiprocessor implementation. In particular, the blackboard mechanism in the prototype system can move data across a local-area network at 100-200 kilobytes/second (KB/s). Golding's measurements of end-to-end performances between various hosts on the Internet indicate that the effective communication rates across wide-area networks are in the 5-50 KB/s range [Gol92].

Another consideration for applications with large data sets is how much of the data is needed by individual tasks. A large per-task blackboard working-set may reduce the number of processors available to a task, because some *Wax* processors may not have enough accessible cache space to hold the working-set. When the accessible cache space is exceeded, a task will be unable to fetch or write additional blackboard data. Once this occurs, all blackboard operations that would need to fetch or write data will fail and return an error code.

To avoid these problems, the application programmer must develop estimates of the blackboard working-set requirements for each task and job. The programmer can then compare those estimates with the distributed computation system's estimates of the available resources. A computation system could also use the programmer's estimates in making decisions about where to assign a particular task for execution.

5.1.5 Termination of *Wax* Computations

Detecting and handling the termination of a *Wax* job is left to the application programmer and user. *Wax* will continue to process the tasks in a *Wax* job as long as there exist tasks that have not terminated and processors that are compatible with any of those tasks. In some applications, a task will be able to determine that the job has achieved its goal and interrupt the execution of any other active tasks. In others, the user of an application will have make that determination based on observation of the blackboard of the job in question. In both cases, the extraction of the results of a job from its blackboard, and the subsequent deallocation of any *Wax* resources used by the job, must be performed by some process external to *Wax*.

5.2 Issues in Programming using *Wax*

After deciding to implement an application using *Wax*, the programmer must consider how to map the pieces of the application and their associated data structures onto *Wax* tasks and blackboard tuples. Choosing a mapping is often an iterative process with each iteration consisting of three phases. The first phase requires identifying the parallelism that exists in the application and assigning pieces to *Wax* tasks. The next phase breaks the data structures used by the application into the pieces that will be stored in separate tuples. The last phase closes the loop by examining how the selected decomposition of the data structures affects the implementation of the individual tasks. If the effects on application performance or complexity of the implementation are considered too great, then the process can be repeated using a different partitioning of the application's parallelism.

5.2.1 Mapping Application Parallelism to *Wax* Tasks

The partitioning of an application into tasks focuses on the costs of and benefits provided by each additional task. The costs include the previously described generic overhead of creating, scheduling, and reaping the task, and perhaps application-specific overhead due to additional mechanisms needed to handle failures or possible data inconsistencies. The benefits of using additional tasks include overlapping execution and access to more physical and virtual memory.

An additional issue to consider in splitting an application into tasks is deciding whether this division should be done once when a job is first started, or dynamically adjusted during the course of its execution. If the maximum potential parallelism can be determined when a job is created and the variance in resource requirements (CPU, memory) of the various pieces is not expected to be a problem, then static partitioning is probably the right choice, since it is generally easier to implement than dynamic partitioning. Otherwise dynamic partitioning will probably provide better performance. The class of applications that are well suited to static partitioning includes integer factoring, 3-D perspective rendering, and applications, such as simulations, that generate their parallelism by processing multiple runs simultaneously. Applications that use a branch-and-bound algorithm, in which the pruning of the search space can result in significant variation in the CPU and memory usage of individual tasks, should use dynamic partitioning.

5.2.2 Decomposition of Data Structures

The process of decomposing an application's data structures can be divided into three steps. The first step is to figure out what data need to be moved into and out of the blackboard used by a job. The second is to map each piece of the application data onto one or more blackboard tuples. The last step is to decide how to recover from the potential unavailability of the data stored in a tuple.

The *Wax* blackboard contains the only shared data that is accessible to *Wax* tasks. Therefore, the process creating a job must copy any input from wherever it is stored outside of *Wax* onto the blackboard used by the job. An analogous extraction step must be performed to remove any results that the submitter of the job wants to save.

In deciding how to group pieces of data into tuples, the application writer should consider whether there are pieces that are needed simultaneously in order for any one of them to be useful. If such groups of data exist, then the application may want to write the entire group to the blackboard as a single tuple. If the pieces are recorded as multiple tuples, they may not all diffuse at the same rate with the result that none of the individual pieces would be usable.

Similarly, groups of information that need to be internally consistent should also be written as single tuples. For example, a task that produces an $N \times M$ matrix, where N and M are variable, would probably want to record N, M, and the contents of the matrix as a single tuple. Since *Wax* guarantees that individual blackboard writes are atomic, such a tuple would always be internally consistent. Whereas, if two or three tuples are used to hold the matrix and its dimensions, and two instances of the task produced matrices with different dimensions, then another task that attempted to read the matrix might read the values for N and M written by one instance and the contents of the $N' \times M'$ matrix written by the other instance. Furthermore, while the reader task would be able to detect the inconsistency if $N \times M \neq N' \times M'$, because the contents of the matrix would be the wrong size, the inconsistency is not (immediately) detectable if $N \times M = N' \times M'$.

On the other hand, the programmer may not want to combine un-related data, or pieces of data that are used by different tasks. The larger a tuple is the longer it is likely to take to distribute, since the minimum transfer time for a tuple is the size of the tuple divided by the effective network bandwidth. If a task needs only a small portion of the data stored in a tuple, then the unused data simply increases the communication cost (and delay) of moving the needed data.

```
Seller_Task()
{
  key_t *       list;
  unsigned      itemPrice = 199;
  unsigned      numItem;
  int           count;

  Wax_BlackboardWrite("Price", itemPrice);      /* Write price to blackboard */

  while (Wax_BlackboardRead("Order", &numItem) != SUCCESS) {
    Delay();                                    /* Wait for an order to arrive */
  }

  /* Process order */
}

Buyer_Task()
{
  key_t *       list;
  unsigned      itemPrice;
  unsigned      numItem;
  int           count;

  while (Wax_BlackboardRead("Price", &numItem) != SUCCESS) {
    Delay();                            /* Wait for a seller to post a price */
  }

  numItem = 1000 / itemPrice;           /* Determine how many items to order */

  Wax_BlackboardWrite("Order", &numItem);       /* Submit order */
}
```

Figure 5.3: Example of how a dependence on the execution order of tasks can result in deadlock. If only one processor is available, the `Wax_Read` **operation executed by whichever of the Buyer_Task or Seller_Task is selected for processing will never indicate successful completion, because the desired tuple will never appear.**

5.2.3 Deadlock-Free Parallelism

Another issue that application programmers must consider is the degree of deadlock-free parallelism that exists in an application. This is the number of tasks that can simultaneously be in the task heap, such that the job will execute correctly, *and* continue to make forward progress, regardless of the order and degree of concurrency with which those tasks are executed. For example, the hypothetical application shown in Figure 5.3 will deadlock if only one processor is available. If the **Seller_Task** is selected for execution, it will spin waiting for the ORDER tuple to appear, which will never happen. If instead the **Buyer_Task** is selected for execution, it will spin waiting for the PRICE tuple to appear.

Strictly speaking, the degree of deadlock-free parallelism is the maximum amount of parallelism that can be used by the *Wax* implementation of an application. However, this may often be an overly conservative bound, as there will almost always be multiple processors available to a job. The list and status operations defined by the *Wax* programming model (see Section 3.2) provide programmers with the means of estimating the number of currently available processors (by counting the active tasks). An implementation might also provide an interface for explicitly determining the number of available processors. The programmer can then adjust the behavior of the application based on the current size of the processor pool.

Any application that relies on tasks executing in parallel for correct execution is susceptible to deadlock when failures reduce the number of available processors. For example, the barrier synchronization primitive described in Section 3.5.1 provides a convenient method of synchronizing the phases of a computation, but it will deadlock if not all of the tasks rendezvousing at the barrier are executing at the same time. The list and status operations can be used to detect such a deadlock, since the barrier primitive requires each participant to know the identity of all the other participants. When a deadlock is detected, there are at least two possible recovery techniques. The application can recover by having one or more of the executing tasks perform the work that would have been executed in parallel by the other tasks if there had been more processors available, and write the necessary synchronization tuple. Alternatively, one, or more, of the waiting tasks could make processors available by invoking the exit operation. Before calling exit, a task would spawn a new task that, when assigned to a processor, would perform the rest of computation assigned to the first task.

This issue could be rendered moot by modifying *Wax* to support multi-programming of idle hosts. The multi-programming could be accomplished either cooperatively or preemptively. In a cooperative implementation, a task that was waiting to synchronize with other tasks would signal a Task Manager, via its local Host Manager, that it should be checkpointed and another task allowed to execute. In a system supporting preemptive multi-programming, Task Managers would periodically notify each Host Manager to checkpoint the currently active task, and start executing another task.

5.3 Example Applications

This section describes how several applications have been or might be implemented using *Wax*. The first example, which is a hypothetical implementation of an application that uses a branch-and-bound algorithm, is described in moderate detail, including pseudocode for the pieces that are most important to understanding the *Wax* programming model. The remaining applications, which are presented in less detail, include: Manasse's integer factoring application, an application that generates images of the Mandelbrot set, an application that produces an oracle for counting problems, a three dimensional perspective rendering application, and RAIDSIM, a simulator for a redundant array of inexpensive disks (RAID).

5.3.1 Branch-and-Bound Algorithm

This example examines a hypothetical application that uses a branch-and-bound algorithm to find all optimal solutions to a problem. The branching part of the algorithm produces a tree of solutions by selecting an existing partial solution and generating new solutions that are one step closer to complete solutions to the problem. The bounding part uses a cost metric to prune partial solutions that cannot lead to complete solutions that are at least as good as those solutions already found. Branch-and-bound algorithms can be used in applications such as the quadratic assignment problem [KB57]. The quadratic assignment problem involves identifying (approximately) optimal pairings of one set of objects to a second set of objects, for example, assigning buildings to sites. In such an instance of the quadratic assignment problem, the cost metric is a function of the flow of material between buildings and the distance between sites.

Applications that use branch-and-bound algorithms are well suited to a distributed implementation of the *Wax* programming model. The searching of each portion of the solution space is essentially independent. The only data that tasks might share are improvements in the bound value. The exchange of such values can optimize the search of the solution space, but is not necessary for the correct execution of the algorithm. In addition, the data requirements, at least for the quadratic assignment problem, are minimal: a few kilobytes.

Within each task that searches the problem space for optimal solutions, there are two threads: one to monitor the blackboard and one to search the portion of the problem space assigned to the task. The first thread, Scanner, shown in Figure 5.4, uses the *Wax* directory operation to check for tuples that contain new bound values that have found by other tasks. When it detects new tuples, the thread reads them and uses their values to update the bound value that is used internally by the other thread. The second thread, Searcher, shown in Figure 5.5, does the actual searching for solutions. Figure 5.6 shows the routines used to communicate between the two threads. Finally, Figures 5.7 and 5.8 show the source and sink for the job, respectively.

A Problem with Multi-valued Tuples

There is a subtle, but important, point to made about when the tuples recording solutions are written to the blackboard by the searcher thread. If a task is not idempotent with respect to the set of tuples and values that it writes to the blackboard, then some of those values may become lost. A value can be lost if it is only written in tuples that have multiple values, in which case, subsequent *Wax* read operations may always return one of the other values associated with those tuples.

In an application designed to find *all* solutions with the same best cost metric, the solutions cannot be written until the task has searched all of its portion of the problem space. Because each instance of a task will use the same sequence of keys to record the best solutions it finds, writing the solutions to the blackboard as soon as they are found (replace `RecordNode()` in Figure 5.5 with `Wax_Write()`) may cause some solutions to be lost if a task is executed multiple times. Since the execution of these tasks is affected by the *accessible* results of other tasks, their execution is not deterministic. As a result, two instances of the task may record a different number of solutions that are optimal when they are found, but not at the end of the search. If this happens, then one or more of the keys used to record solutions will be associated with values that include both optimal and non-optimal solutions, which could result in an optimal solution being lost. The potential existence of multiple values is not an issue for the keys used to record the bound value, because the values observed for those keys only optimize the performance of the algorithm. They do not affect its correctness.

```
/*
 * BOUND_PATTERN is a regular expression that matches the keys used to record bound
 * values.  (See MaybeUpdateBound() for an explanation of the naming scheme.)
 */
#define BOUND_PATTERN   "Bound\\.[0-9]+\\.[0-9]+"

/*
 * Scanner Thread - Monitors blackboard for output of other tasks.
 */

Scanner()
{
  key_t *       list;
  key_t *       oldList;
  value_t       value;
  int           i, count, oldCount;

  oldList = NULL; oldCount = 0;
  while (TRUE) {
    /*
     * Read the list of blackboard tuples whose keys match the pattern specified by
     * the first argument to appear.
     */
    Wax_BlackboardGetList(BOUND_PATTERN, &list, &count);

    for (i = 0; i < count; i++) {
        /*
         * Check whether the tuple has been seen (and therefore read) in a previous scan.
         */
      if (KeyNotInList(list[i], oldList, oldCount)) {
        Wax_BlackboardRead(list[i], &value);    /* Read a value associated with a key */
        MaybeUpdateBound(value, FALSE); /* Possibly update the bound value */
      }
    }
    oldList = list; oldCount = count;   /* update for next iteration */

    Sleep();                            /* wait for a while before doing another scan */
  }
}
```

Figure 5.4: Example of how to use *Wax*. The thread executing this routine scans the blackboard for bound values recorded by other tasks.

```
/*
 * Searcher Thread - Executes a Branch-and-Bound search of the assigned portion
 * of the problem space.
 */

Searcher()
{
  key_t         key;
  node_t        curNode;
  value_t       value;

  curNode = SelectNode();               /* Select a node from the search space. */
  while (curNode != NULL) {             /* While we have not exhausted the space */
    value = EvaluateNode(curNode);      /* Compute the cost metric. */

    if (value <= CurrentBound()) {      /* Compare cost of current node to the bound */
      MaybeUpdateBound(value, TRUE);    /* Possibly update the bound value */
      if (Complete(curNode) {           /* Is this a complete solution to the problem? */
        RecordNode(curNode);            /* Yes, save the result. */
      } else {
        /* Starting from the current partial solution, generate a set of nodes
         * that are one step closer to a complete solution. */
        ExpandNode(curNode);
      }
    }
    curNode = SelectNode();
  }

  /*
   * Now record all the the best solutions in the blackboard.  GenerateKeyForSolution()
   * produces a key of the form Solution.<taskId>.<count>, where the taskId field is
   * some identifier that is unique amongst all tasks, and the count field is the
   * number of solution tuples previously written by this task.
   */
  curNode = GetRecordedNode();          /* Get a saved result. */
  while (curNode != NULL) {
    key = GenerateKeyForSolution();
    Wax_BlackboardWrite(key, curNode);  /* Record it on the blackboard. */
  }
}
```

Figure 5.5: Example of how to use *Wax*. The thread executing this routine performs the search of the assigned portion of the problem space.

```
static value_t  bound;

value_t CurrentBound(void)
{
  value_t      value;

  LockBound();                         /* Prevent other threads from accessing bound. */

  value = bound;

  UnlockBound();                       /* Allow access to other threads */

  return value;
}

MaybeUpdateBound(value_t value, boolean_t local)
{
  key_t        key;

  LockBound();                         /* Prevent other threads from accessing bound. */

  if (value < bound) {
    bound = value;
    if (local) {
      /*
       * If this value was found by this task, then the new bound should be recorded
       * in the blackboard so the tasks can use it.  The function GenerateKeyForBound()
       * produces a key of the form Bound.<taskId>.<count>, where the taskId field is
       * some identifier that is unique amongst all tasks, and the count field is the
       * number of bound tuples previously written by this task.
       */
      key = GenerateKeyForBound();
      Wax_BlackboardWrite(key, bound);  /* Make the new bound available to other tasks. */
    }
  }
  UnlockBound();                       /* Allow access to other threads */
}
```

Figure 5.6: Example of how to use *Wax*. These routines handle the communication between the Scanner thread (Fig. 5.4) and the Searcher thread (Fig. 5.5).

```
/*
 * Start a job and use the program whose name is given by programName and
 * executable is pointed to by path.
 */
StartJob(const char *path, const char *programName, wax_acl_t aclBuf)
{
  wax_job_t             job;
  wax_blackboard_t      blackboard;
  value_t               bound;

  WaxExt_JobCreate("JobName", /* priority */ 1, aclBuf, &job);

  /* Add program instances for PCs and Pmaxen */
  WaxExt_JobAddProgram(job, programName, path, PC_ARCHITECTURE);
  WaxExt_JobAddProgram(job, programName, path, PMAX_ARCHITECTURE);

  WaxExt_JobGetBlackboard(job, &blackboard);

  bound = ComputeInitialBound();
  WaxExt_BlackboardWrite(blackboard, "InitialBound", bound);

  CopyInputToBlackboard();              /* Copy any initial state into the blackboard */

  /* Create the individual tasks that will perform the computation. */
  Partition computation;
  For each piece of the computation {
    WaxExt_TaskSpawn(job, programName, taskArguments);
  }
}
```

Figure 5.7: Start a branch-and-bound job in *Wax*.

```
boolean_t JobDone(wax_job_t job)        /* Has the job terminated? */
{
  wax_task_t *  taskList;
  int           taskCount;
  boolean_t     done;

  /* Get a list of all tasks in this job. */
  WaxExt_JobGetTaskList(job, &taskList, &taskCount);
  done = TRUE;
  for (i = 0; i < taskCount; i++) {
    done &= Wax_TaskDone(taskList[i]);  /* Has this task terminated? */
  }

  return done;
}

MonitorJob(wax_job_t job, wax_blackboard_t blackboard)
{
  key_t *       keyList;
  key_t *       oldKeyList;
  int           count, oldCount;
  value_t       value;
  node_t        node;

  /*
   * Collect the results.
   */
  oldList = NULL; oldCount = 0;
  while (! JobDone(job)) {
    WaxExt_BlackboardGetList(blackboard, ".*", &keyList, &count);
    for (i = 0; i < count; i++) {
      if (KeyNotInList(keyList[i], oldKeyList, oldCount)) {     /* Is the tuple new? */
        if (Matches(BOUND_PATTERN, keyList[i])) { /* Is this tuple a the bound value? */
          WaxExt_BlackboardRead(blackboard, keyList[i], &value);
          if (value < bound) {
            bound = value;
          }
        } else {
          WaxExt_BlackboardRead(blackboard, keyList[i], &node);
          /* Combine node with existing results */;
        }
      }
    }
    oldList = list; oldCount = count;   /* update for next iteration */

    Sleep();                            /* wait for a while before doing another scan */
  }
  Print final results;
}
```

Figure 5.8: Monitor a branch-and-bound job running in *Wax*.

The possibility of losing results exists even if the application only needs to produce *a* single solution, though the reasoning is slightly different. One might conclude that since the last solution written by an instance will always be locally optimal, and it doesn't matter which of the optimal solutions is returned by the *Wax* read operation, that the results can be written immediately. This is true if all task instances execute to completion. However, if there are multiple instances of a task executed, *Wax* may interrupt an instance in the middle of execution with the result that the last solution tuple written by the interrupted instance may not be one of the optimal solutions. If the interrupted instance also happened to produce more (non-optimal) solutions than any of the other instances, because it was not able to prune the search space as aggressively, then all of the solution keys will be associated with at least one non-optimal value, which could result in all of the optimal solutions being lost.

An Alternate Implementation

The implementation described above partitions the search space into a fixed number of pieces and uses one task per piece. An alternative solution would dynamically partition the search space. Such an implementation would be more complex to write, but it could perform better, because it has the potential to better balance the execution times of the various tasks. In a branch-and-bound application, not all portions of the tree are equally difficult to search, because the amount of pruning achieved varies from task to task. In an implementation that used dynamic partitioning, tasks that detected that their memory usage or running time had exceeded a pre-determined threshold could spawn additional tasks to handle pieces of the search space.

5.3.2 Integer Factoring

Another example *Wax* application is the integer factoring problems that Lenstra and Manasse [LM89] are investigating. Their solution uses a two stage multiple polynomial variation of the quadratic sieve algorithm. In the first step, which is readily parallelized, so-called *relations* are collected. In the second step, Gaussian elimination is used to combine these relations and produce the factorization.

This application is well-suited to a distributed computation system implementing the *Wax* model. The tasks executed during the parallel step of the algorithm share data only as an optimization and each of them requires days of CPU-time to execute to completion. In addition, the data requirements of the application are relatively modest: tens of kilobytes of output per task per CPU-day.

To factor a number using this algorithm and *Wax*, the user creates a job and blackboard, stores the integer to be factored and various other input parameters in the blackboard, and then spawns the tasks that compute the relations. When each task starts running, it checks the blackboard for a tuple containing the set of prime numbers that are need to use the quadratic sieve algorithm. If the tuple is not found, either because it has not yet been written to the blackboard or it is not recorded in an accessible blackboard cache, the task will generate the set of primes and attempt to record them on the blackboard.

As a task generates relations, it records them on the blackboard, one per tuple. A per task unique identifier and relation counter are used to generate the tuple key. The factoring algorithm is designed to eliminate duplicate relations, so different instances of a task writing different values to one tuple key wastes computing power, but does not affect the correctness of the factorization.

The Gaussian elimination step of the algorithm can be implemented using *Wax* or a separate external (to *Wax*) computation. If it is done within *Wax*, then every time N new relations are produced, another task will combine the new and old relations and eliminate duplicates. This processing can be done either by one of the tasks generating relations or by a separate task. When enough relations have been found, the tasks that are generating relations are killed and a task is spawned to perform the Gaussian elimination. The factorization would then be recorded on the blackboard for later retrieval by the user.

The alternate (external) solution works in much the same manner. The difference is that the relations rather than the factorization are extracted from the blackboard. In the existing *Wax* implementation of Manasse's program, the relations are extracted from the blackboard and mailed to the factoring coordinator, where they are combined with relations produced by others using the original non-*Wax* implementation of the algorithm. In either implementation, the user is able to monitor the progress of the computation by examining the contents of the blackboard.

5.3.3 Mandelbrot Set Generator

The third example application is one that generates images of the Mandelbrot set [Man83] for display with X Windows [SG86]. The application has essentially an arbitrary amount of parallelism since each pixel computation is independent. The computation for each pixel in the image involves iterating the equation: $Z = Z^2 + C$, where Z is initially zero and C is the location of the pixel in the complex plane. The equation is repeatedly evaluated a until Z exceeds a certain threshold or a specified number of iterations have been performed. The value assigned to a pixel is the number of iterations performed. Given that for moderate numbers of iterations ($\sim$256) the individual pixel computations do not take long to execute, each task is assigned a group of pixels to generate.

5.3.4 Generating an Oracle for Counting Problems

Diaconis and Efron [DE85] developed a statistical test procedure to compliment the Fisher significance test [MP83]. Unfortunately, the procedure requires the ability to solve a counting sub-problem that is $\sharp P$-hard. Mount [Mou94] has, however, developed a method for precomputing a counting "oracle" that can solve the counting sub-problem. Generating an oracle requires the determination of the coefficients for a large number of multivariate polynomials.

The application is a good *Wax* application for several reasons. First, the determination of each polynomial is independent. Second, for even relatively small oracles, such as the one for 4 by 4 tables, the number of polynomials that need to be determined is in the thousands. Third, the determination of the coefficients for a polynomial is computationally intensive. Determining one of the polynomials for the 4 by 4 oracle typically requires 3-4 CPU-hours on current workstations. Furthermore, both the number of polynomials and the cost of determining the coefficients for one of those polynomials grow rapidly with increasing problem size.

5.3.5 Three Dimensional Perspective Rendering

Another application that can be solved using the *Wax* programming model is three dimensional perspective rendering, which is used to produce two dimensional images from an input database that describes a three dimensional surface. The rendering can be done with either the *ray identification* algorithm [LC92] or a *ray casting* algorithm [HHM86].

The *ray identification* algorithm developed by Li and Curkendall identifies where, if anywhere, each location in the input database will appear in the output image. The evaluation of each input point depends only on that point. The computed value of each pixel in the output image, is then a function of all of the input points that intersect that pixel.

The *ray casting* algorithm works in the other direction: from output image to input database. For each pixel in the output image, the ray casting algorithm "casts" a ray into the input database to identify the value of the output pixel. Parallelism is achieved by partitioning the output image, since the computation of each output pixel is independent.

For an implementation that uses a distributed computation system, the ray identification algorithm is more appropriate, because it requires a smaller per-task blackboard working set. With the ray identification algorithm, each task uses only a piece of the input database. With the ray casting algorithm, however, each task may need to access the entire input database.

The ray identification algorithm has three phases: setup, computation, and collection. The first phase, which would be performed by the source for the job, is to copy the input database on to the blackboard, partitioned so that the data needed by each task is stored in a separate tuple, and spawn the desired number of tasks. In the computation phase, each task identifies which of the points in its piece of the input database may appear in the output image. After identifying all such points, each task would write a description of the partial image on the blackboard. The collection phase involves merging the partial images generated by the tasks in the computation phase into the final image. This can be done either using a *Wax* task or a process external to *Wax*. Additional parallelism can be achieved by using one job to render multiple images.

5.3.6 RAIDSIM

The RAIDSIM program, which was developed at the University of California at Berkeley [Lee90], simulates the performance of a RAID system. For each simulation run, the user specifies the configuration of the array and a test workload. The program then uses co-routines to simulate processes performing the specified workload. A simulation runs until it determines the average response time of an operation within a specified confidence interval. The program then prints information about the performance of the simulated RAID configuration.

RAIDSIM is well suited to a distributed computation system, because evaluating one configuration will generally require the execution of a number of runs, each with a different workload. Each run is independent so there are no data consistency requirements. In addition, the data used or produced by a run is only a few kilobytes in size, so the communication costs of a distributed computation system should not be an issue.

5.4 Conclusion

In this chapter, I have identified a class of applications that can be implemented using the *Wax* programming model, and presented an evaluation process that application programmers can use to determine whether or not a particular application can or should be implemented using the model. I have also described how several applications can be implemented using the *Wax* model. In the next chapter, I will examine the performance of the *Wax* implementations some of these applications.

Chapter 6

Evaluation

This chapter presents an evaluation of the prototype system described in the Chapter 4. The first part of the evaluation examines the performance of *Wax* at the micro-benchmark level. The next part looks at the performance of *Wax* implementations of several of the example applications described in Chapter 5. The last part examines the ability of process checkpointing to insulate *Wax* applications from interruptions due to machine failure or machines becoming busy with local work.

6.1 Performance of *Wax* Primitives

Table 6.1 shows the cost of executing various *Wax* primitives in the absence of failure. The execution times were determined by executing each operation 500 times on a PC with an Intel 486DX2/66 processor. All of the operations were measured using the routines in the "`WaxExt_`" interface. *Wax* tasks do not have access to a precise clock, so the direct measurement of the routines in the "`Wax_`" interface difficult. The cost of those operations should be essentially identical to the corresponding external operation, since the only significant difference in the implementation of routines in the two interfaces is the added RPC from the *Wax* task to the Host Manager for routines in the "`Wax_`" interface. The cost of a local *Mach* RPC is on the order of a millisecond.

The cost of a *Wax* read operation when the tuple is already in the local AFS cache is essentially the same as reading a similarly sized local file (~.01 seconds), which is within the precision of the available clock. I expect that the cost of the read operation when the tuple is not in the local cache will be approximately the same as a write operation.[1]

6.2 Performance of *Wax* Applications

In order to demonstrate the utility of *Wax*, and the *Wax* programming model, I have ported several applications to run on *Wax*. These applications include a quadratic assignment problem (QAP) solver, the integer factoring application described in Section 5.3.2,

[1]I have not directly measured the execution time of uncached reads, because setting up a test to reliably measure such operations is difficult due to the need to flush the AFS cache between each read operation.

Operation	Mean Time	Std. Dev.
Spawn	.14	.05
Write (128 bytes)	.075	.013
Write (1k bytes)	.077	.013
Write (4k bytes)	.091	.015
Write (8k bytes)	.108	.018
Write (16k bytes)	.136	.015
Write (32k bytes)	.21	.05
Write (64k bytes)	.35	.05
Write (128k bytes)	.72	.12

Table 6.1: Execution time of various *Wax* primitives in the "`WaxExt_`" interface on a PC with an Intel 486DX2/66 processor. Times are in seconds. TBD = to be determined.

the Mandelbrot set generator described in Section 5.3.3, and the program described in Section 5.3.4 that generates an oracle for counting problems.

The timings presented for the QAP solver and the Mandelbrot set generator were measured using an early version of *Wax* that was running on a collection of DECstations 5000s, Omron Luna88ks,[2] and Intel 386 and 486-based machines at CMU. The integer factoring and counting problem jobs were processed by a *Wax* domain consisting of DECstations and Intel x86-based machines at CMU and the University of Washington.

Although the measurements encompass one domain and utilize a pool of at most 25 processors, they illustrate both the practicality of the programming model, and the value of using a collection of hosts to solve a problem. *Wax* has not been tested on a larger collection of machines, because many of the machines that once ran Mach 3.0 have been de-commissioned or are now running other operating systems.

6.2.1 Generating an Oracle for Counting Problems

As part of his Ph.D. research, John Mount used *Wax* to generate the oracle needed to settle the 4 by 4 contingency table counting problem. As mentioned in Section 5.3.4, such oracles are of interest, because they can be used to implement the statistical test procedure developed by Diaconis and Efron [DE85]. Producing the 4 by 4 oracle requires the determination of 3694 degree 9 polynomials in 7 variables (yielding 11440 coefficients and subproblems each). With current workstations, the determination of each polynomial takes several hours. The oracle generator application is the newest *Wax* application, and the first ported and run by someone other than myself.

Mount initially implemented his algorithm, described in [Mou94], as a standard UNIX program, which he was running on a few HP 700-series workstations. He ported the application to *Wax* by compiling the original application code and linking the resulting object files with the *Waxio* package described in Section 4.7.1. The individual jobs were then submitted and managed using the various *Wax* shell-level tools described in Section 4.7.2.

[2]The version of *Wax* for the Omron is no longer up to date, because *Mach* 3.0 is no longer supported on that platform.

Wax was used to determine approximately 2500 of the polynomials. The *Wax* portion of the computation was performed by a pool of 24 machines over the course of 47 days. In that time, the computation consumed 399 CPU-days. Even though some of the machines in the pool were the primary workstation for users, and others were also being used as cycle-servers for other projects, *Wax* was able to provide the application with the equivalent of 8.5 dedicated processors. The resulting oracle occupies 180MB of storage and solves previously intractable problems in seconds.[3]

The performance of *Wax* on this application has led the application writer to consider generating other oracles (3 by 5 and 4 by 5), which he had previously considered impractical to compute. In computing the oracles for larger tables, the application may use multiple *Wax* tasks to compute each polynomial, because the number of sub-problems, and hence the expected memory usage, grows rapidly with increasing table size.

In addition, the ease of porting applications to *Wax* and the computational power provided by the system has resulted in Mount using the system for other computational projects. In particular, he has used *Wax* to generate large, high resolution images (2048 by 1365 pixels, 24-bits per pixel) using the algorithms generated by the Internet genetic art project.

6.2.2 Quadratic Assignment Problem

The quadratic assignment problem determines the optimal assignment of a collection of buildings to sites, given the distance and amount of communication between each pair of sites. The *Wax* implementation, which uses a branch-and-bounds algorithm to find all optimal solutions, is based on the work of Pardalos and Crouse [PC89]. The *Wax* version of the application is structured along the lines of the generic branch-and-bounds algorithm described in Section 5.3.1.

I have used the QAP solver to explore the performance of *Wax*. Table 6.2 shows the elapsed time, as a function of the number of processors used, to find the optimal solutions for the Nugent15 [NVR68] input. The baseline (first line of Table 6.2) for our comparisons is the running time of Pardalos and Crouse's original FORTRAN program on a single DECstation 5000.

Pardalos and Crouse also report the performance of their system on a dedicated IBM 3090-400E multiprocessor. Using all four processors, they observed speed ups of up to 3.6. However, they were unable to run the four processor version on the Nugent15 input to completion due to memory constraints.

During the last run listed in Table 6.2, a random Host Manager was killed every two minutes, left dead for one minute and then restarted. Despite the failures, *Wax* executed the computation much faster than the original version. The failure of a *Wax* task results in only a portion of the computation being lost. With the original FORTRAN version, in contrast, the entire computation would be lost. For computations such as the quadratic assignment problem, where the computation time grows exponentially with the size of the input, the improved failure tolerance provided by *Wax* can be of great benefit.

[3]This oracle is available as a WWW service via `http://mixing.sp.cs.cmu.edu:8001/form4_4.html`.

# of Processors	Elapsed Time in minutes	Speedup
1	140	–
5	43	3.3
13	26	5.4
25	13	11
25 with Failures	17	8.2

Table 6.2: Elapsed Time for Quadratic Assignment Problem Solver to produce the optimal solutions for the Nugent15 input. The last line in the table shows the performance when every two minutes a Host Manager was killed, left dead for one minute, and then restarted.

# of Processors	Elapsed Time in minutes	Speedup
1	15	–
10	17	.9
14	13	1.2
20	11	1.4

Table 6.3: Elapsed Time for Generating a 512x512 pixel image of the Mandelbrot set.

6.2.3 Mandelbrot Set Generator

The Mandelbrot set generator statically partitions the requested image into a number of pieces and uses a *Wax* task to generate each piece. As the pieces are produced the master portion of the application displays the resulting image.

Table 6.3 shows the elapsed time, as a function of the number of processors used, to generate a 512 pixel by 512 pixel by 8 bit image of the Mandelbrot set. The one processor case is measured using a version of the program where the entire computation is executed in a single process. In the multiple processor cases, the image is divided into 64 squares that are 32 pixels on side. A separate *Wax* task is used to generate each square.

The results indicate that the static partitioning of the image does not yield good performance. I suspect that a substantial portion of the problem is due to the wide variation in the amount of work required to generate a block of pixels. When only a few processors are available, there is a good chance that the computationally expensive squares will not be evenly distributed over all the processors. Whichever processor receives more of the computationally expensive squares, then becomes the limiting factor in overall performance. In addition, the cost of storing the shared data in AFS, versus shared-memory in the single machine case, further reduces the benefit of using multiple processors.

There are several possible ways to improve the *Wax* performance of this application. A static partitioning scheme, for N tasks, that assigned a task every Nth pixel from the image should produce tasks with more balanced execution times. Alternatively, dynamic partitioning of the image would also help balance the work load and probably improve performance. Finally, a modified implementation that produced higher resolution images, larger images, or multiple images in parallel would also make better use of *Wax*.

6.2.4 Integer Factoring

The *Wax* integer factoring program is based on the multi-polynomial quadratic sieve program distributed by Lenstra and Manasse as part of their "factoring by E-mail" project [LM89]. The *Wax* version was created by replacing the input and output routines (a few hundred lines of C code) in the original version with calls to access a blackboard. The bulk of the original code, which deals with the actual computation of relations, is unchanged. The *Wax* version records the relations on the blackboard, rather than in the local file system. I then periodically extract the tuples from the blackboard and mail the relations to the coordinator of factoring project.

The performance of the *Wax* version is similar to the original version, which is to be expected given that the bulk of the program is unmodified. Over the course of the nine months that *Wax* has been running, the application has used over 460 CPU-days, and individual tasks have executed for over a CPU-month.

6.3 Process Checkpointing

The process checkpointing mechanism provided by *Wax* has allowed the applications computed by the system to ignore process failures and interruptions due to users reclaiming their machines. To demonstrate that process checkpointing is useful, I have used the logs kept by the Host Managers to determine the execution time of each of the tasks processed by *Wax* and the length of the time periods a machine was computing on a particular task. Most of the tasks in this sample were part of the integer factoring and oracle generator applications.

Figure 6.1 shows the total execution time of each of the tasks spawned by the oracle generator application. The mean task execution time is 227 minutes, and the median time is 200 minutes.

Figure 6.2 shows the distribution of the length of time a task was able to execute on a machine before terminating or being interrupted.[4] The time span shown in the graph covers 93% of the execution periods observed in the history of *Wax*. The mean length of a task execution period is 88 minutes, and the median length is 38 minutes. The length of these execution periods suggests that the tasks whose execution times are shown in Figure 6.1 were typically checkpointed several times before terminating.

Figures 6.3 and 6.4 show the distribution of the length of task execution periods broken down by the type of the *Wax* host: personal workstation or computation server. A machine is considered to be a personal workstation if it is located in a user's office, where it is primarily used for interactive work. A machine is considered to be a computation server if it is not located in a user's office. The results show that computation servers tend to be idle for longer periods of time than personal workstations. The time span shown in the figures covers 97% of the execution periods on the personal workstations, and only 90% of the execution periods on the computation servers.

The integer factoring application provides even stronger support for the importance of process checkpointing. Individual tasks in that application have execution times on

[4]The length of execution periods that are terminated by machine failures cannot be readily determined from the Host Manager logs.

the order of weeks or months.[5] Such execution times are an order of magnitude greater than the longest available periods observed during the history of *Wax*. They are also substantially longer than the reported MTTF of hosts on the Internet [LCP91].

6.4 Conclusion

In this chapter, I have examined the performance of *Wax* and several applications that have been worked on using *Wax*. The data show that *Wax* is capable of providing its users with significant quantities of processing power. The data also demonstrate the importance of providing automated task management and process checkpointing in a system that uses idle workstations as a computational resource. These mechanisms allow *Wax* users to submit large number of tasks and/or long running tasks without having to deal with the effects of process failure. Finally, user feedback suggests that the *Waxio* package is an important part of *Wax*, because it provides an easy means of porting applications to *Wax*.

[5]One task that has been computing off-and-on over the last year has consumed over 49 CPU-days.

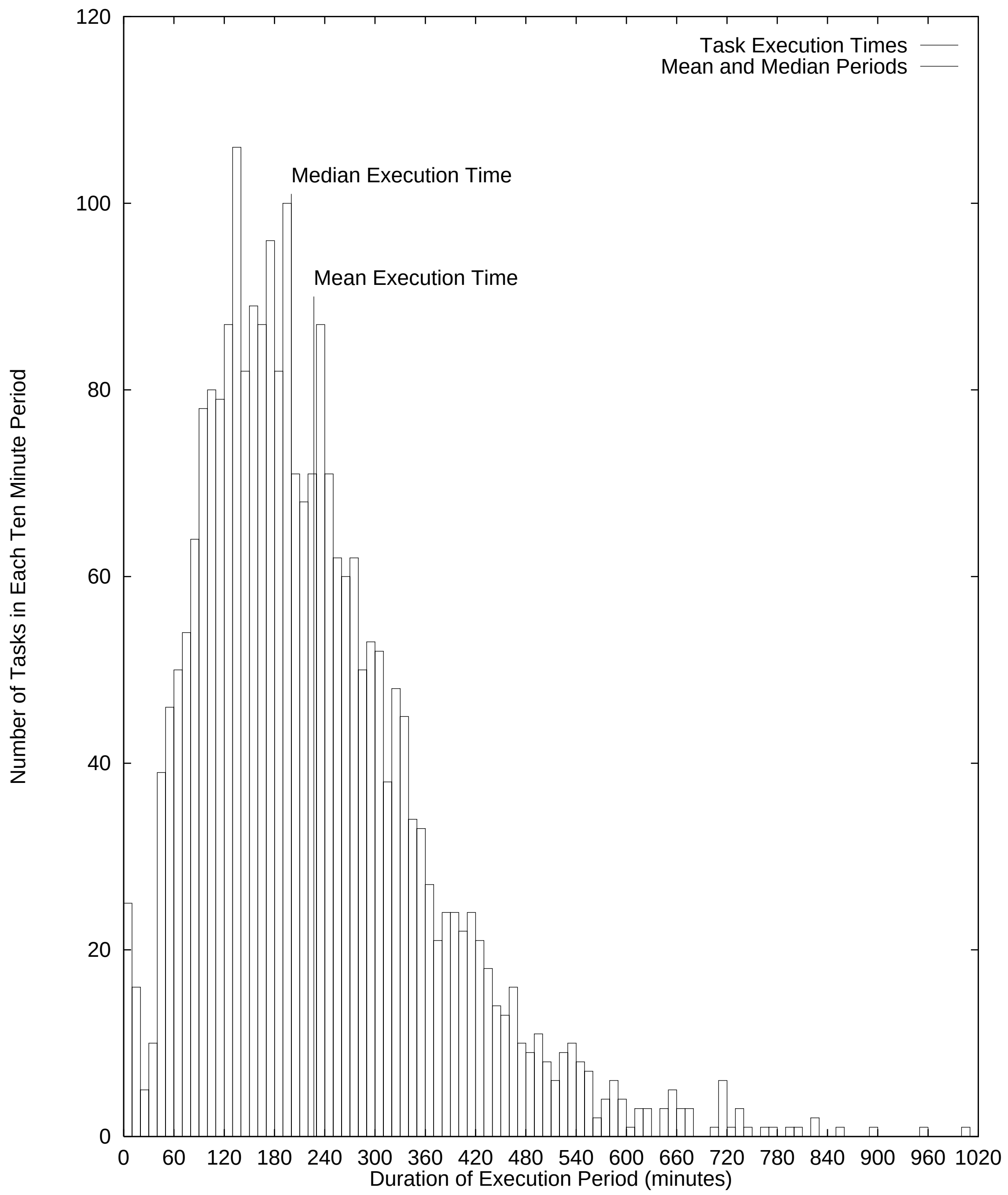

Figure 6.1: The distribution of task execution times for the tasks used to compute the oracle for the 4 by 4 contingency table counting problem. Each box shows how many tasks had execution times within a specific ten minute time period.

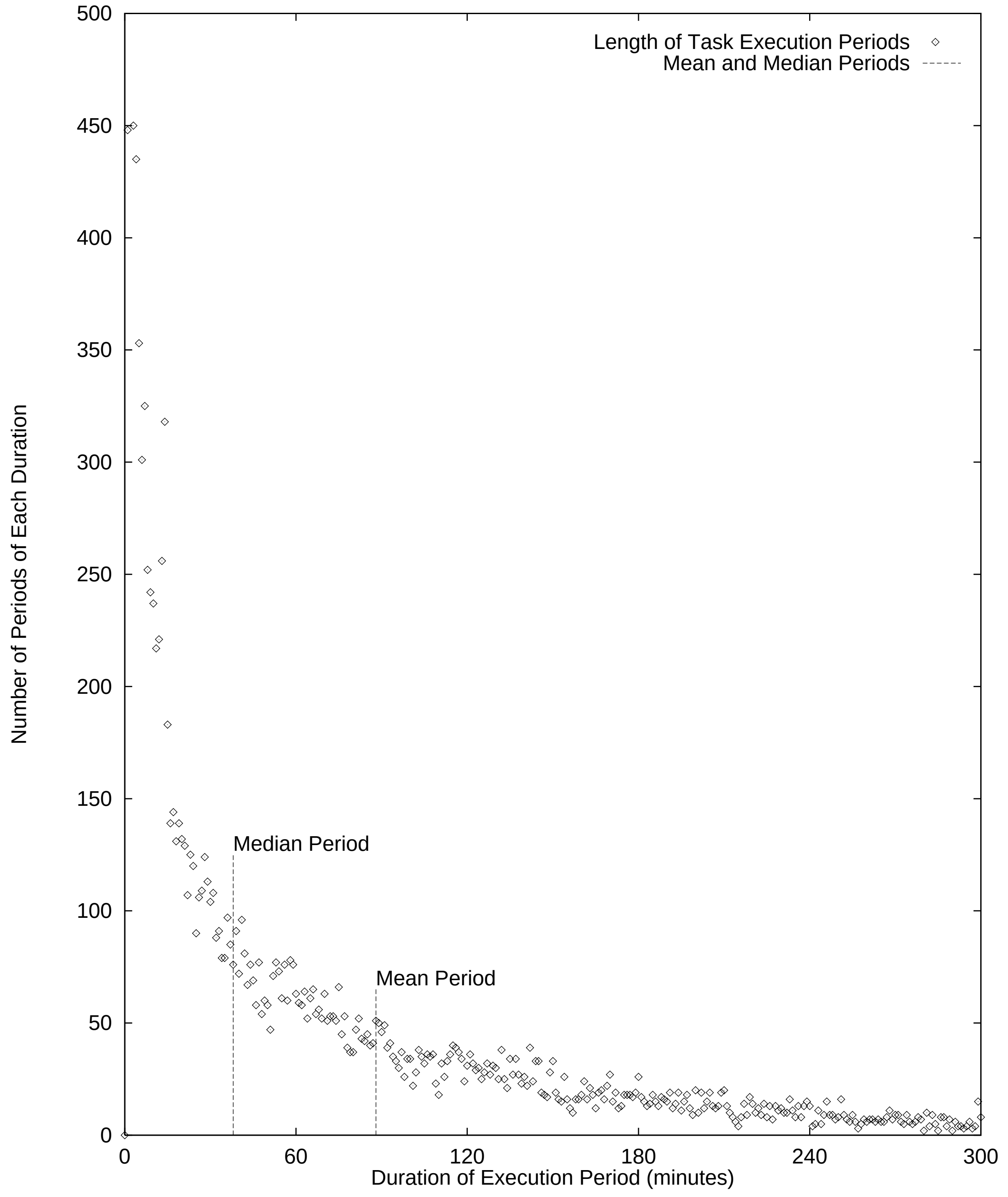

Figure 6.2: This graph shows the distribution of the length of time a task was able to execute before terminating or being interrupted. This data covers all machines that have been used to process *Wax* tasks. The graph covers about 15,000 execution periods. The point for the two minute execution periods, which is off the top of the chart, is at 791.

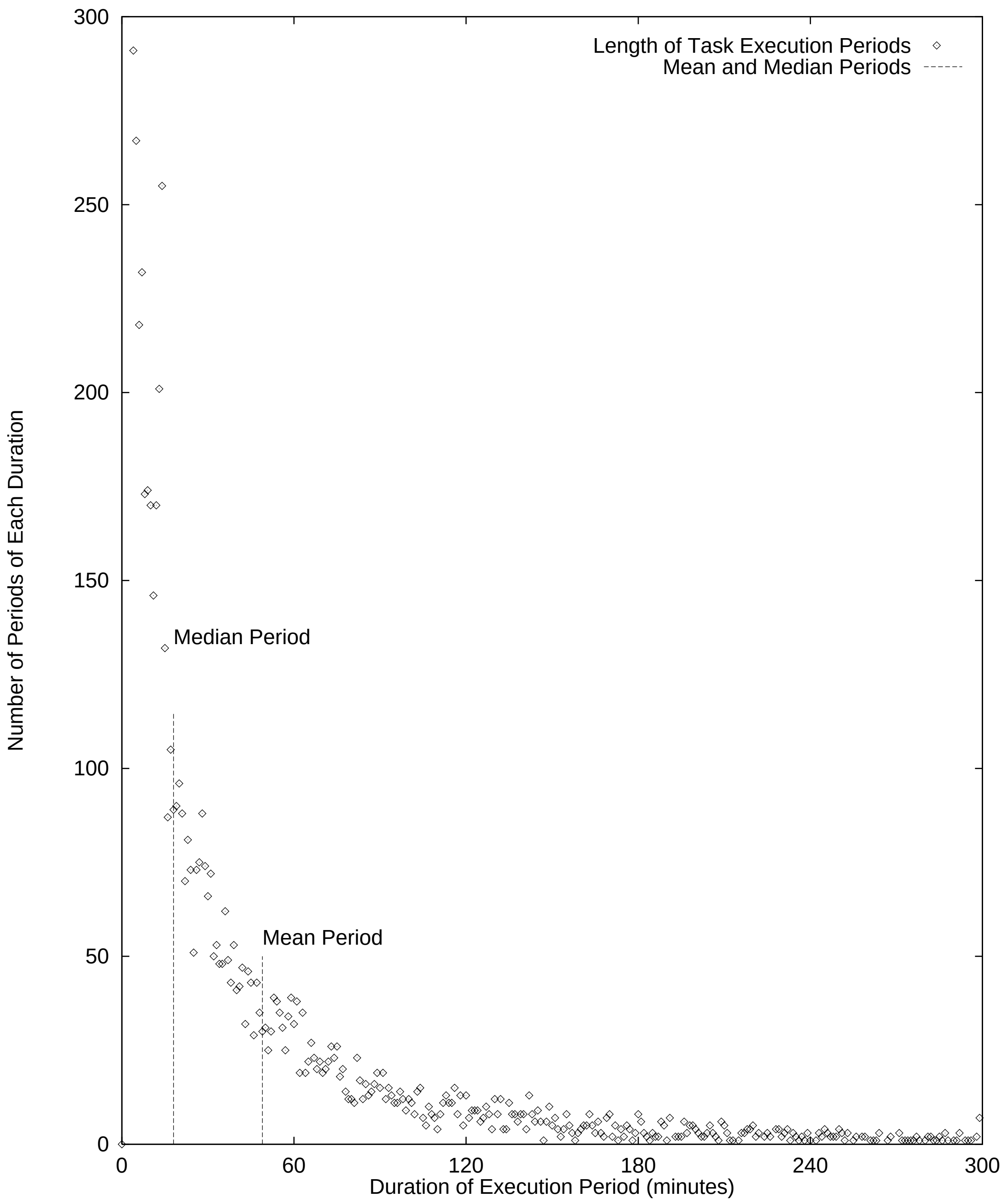

Figure 6.3: This graph shows the distribution of the length of time a task was able to execute before terminating or being interrupted on the seven machines used by *Wax* that are the primary workstation of a user. The graph includes about 7,700 execution periods. The points for execution periods of one, two, and three minutes, which are off the top of the chart, are at: 360, 450, and 320, respectively.

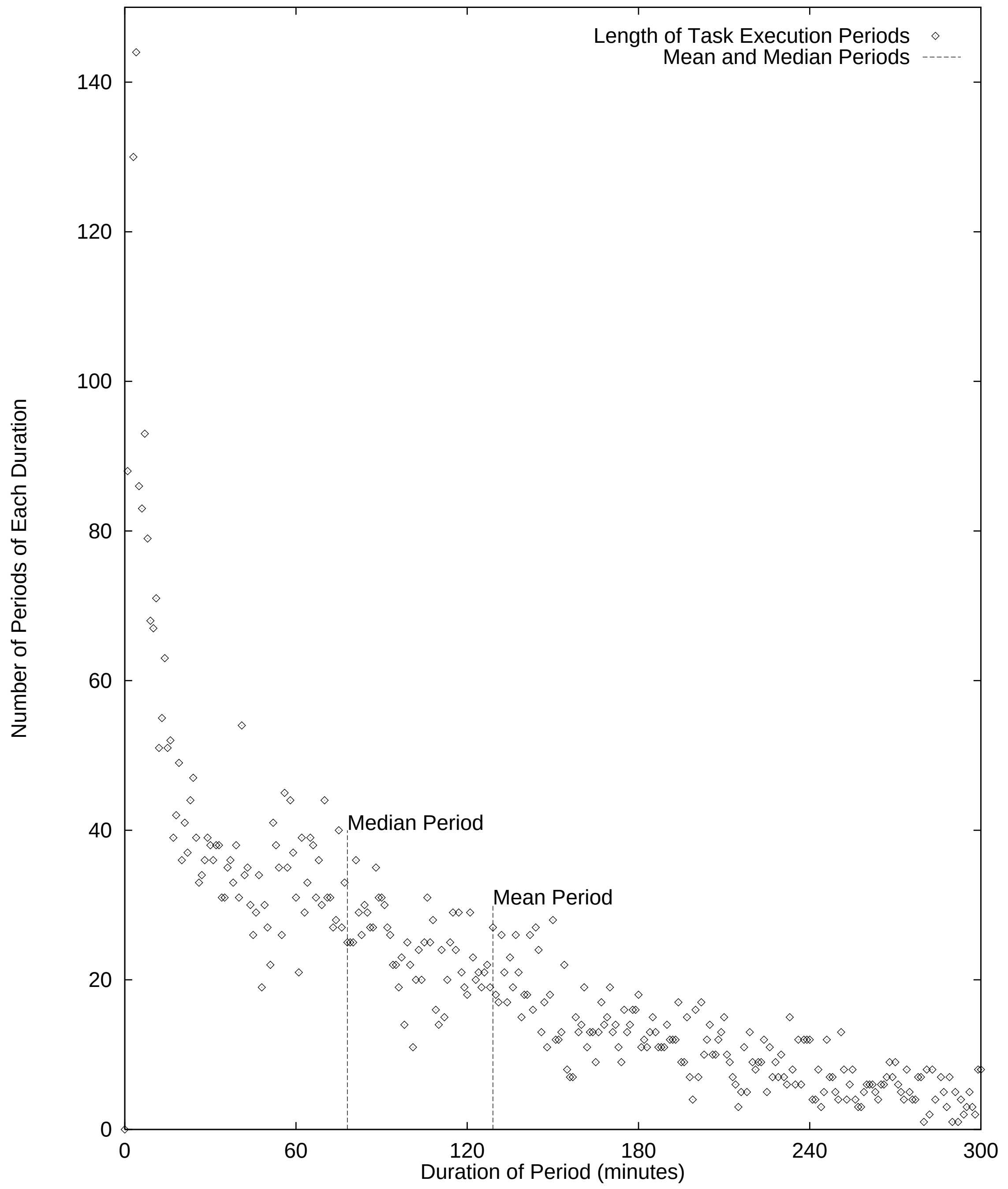

Figure 6.4: This graph shows the distribution of the length of time a task was able to execute before terminating or being interrupted on the 18 machines used by *Wax* that are part of a pool of computation servers. The graph includes about 7,300 execution periods. The point for the two minute execution periods, which is off the top of the chart, is at 341.

Chapter 7

Conclusion

In this dissertation, I have described my research into the use of the machines connected to a wide-area network as a distributed computation system. In the first part of the dissertation, I identified the machines connected to a wide-area network as a powerful, and under-utilized, computational resource, and enumerated six issues that needed to be addressed in order to use that resource as a wide-area multiprocessor. I then presented a set of solutions to those issues in the form of a programming model and a prototype wide-area distributed computation system, *Wax*. Finally, in order to demonstrate the utility of the resulting system, I described a class of applications that can be solved using *Wax*, and examined the performance and functional benefits of using *Wax* to solve some applications from that class.

The rest of this chapter is divided into three parts. I start by recapitulating the contributions provided by my research. I then suggest some possible areas for future work. I conclude the thesis with a brief summary of my research.

7.1 Contributions

The research described in this dissertation makes a number of contributions in the area of parallel and distributed computing:

- I have identified a high capacity computing resource and the problems that must be addressed in order to use that resource as a distributed computation system.
- I have designed a programming model that takes into account many of the problems that exist in a distributed computing environment and is, therefore, feasible to implement in such an environment. I have also identified a class of applications that can be solved using the model.
- I have built and evaluated a prototype wide-area distributed computing system, called *Wax*.

The overall result is a demonstration that it is feasible to use the idle workstations connected to a wide-area network as a multiprocessor. This conclusion is supported by the design of the *Wax* programming model and the implementation and performance of the prototype system.

I have designed a programming model that explicitly exposes the inconsistencies that can arise from the latency and failures that exist in a wide-area distributed computing environment. The model neither guarantees that tasks will observe consistent views of the blackboards they use to exchange data, nor that tasks will execute in any particular order. By avoiding consistency guarantees, and the global synchronization that they imply, the *Wax* programming model allows the various components of a distributed implementation the flexibility to work around the failures and delays that occur in a distributed computing environment. Avoiding such guarantees also makes an implementation of the model easier to scale, since the global data can be allowed to become "more inconsistent" as the system grows.

In order to avoid making the implementation of *Wax* susceptible to the problems that exist in a distributed computing environment, the *Wax* Task Managers function much like the tasks in a *Wax* job.[1] In particular, the *Wax* blackboard mechanism is used to store the data needed to manage the processing of computations submitted to the system, and Task Managers only interact with each through the blackboard. They are therefore only indirectly aware of the existence of other Task Managers and/or other *Wax* domains. This allows additional Task Managers and/or domains to be added as needed to handle the addition of machines to the system. Furthermore, the failure of one Task Manager does not affect the ability of other Task Managers to continue processing tasks.

Additional support for the conclusion is provided by the data in Chapter 6, which demonstrate that useful work can be done based on the *Wax* programming model and that applications can continue to make progress in the face of failures. In particular, the integer factoring and oracle generator tasks (described in Sections 5.3.2 and 5.3.4, respectively) processed by *Wax* have produced useful results for other researchers. While the measurements were made using only a single domain consisting of 25 machines, the system continued operation in spite of the same sorts of machine and network failures that will occur in a system spanning a wider area and multiple domains.

7.2 Future Work

While *Wax* and this dissertation do demonstrate the feasibility of wide-area distributed computation systems, there are still a number interesting issues to explore in this area. One important research area is the choice of appropriate mechanisms for resource identification and allocation. Another area focuses on mechanisms for billing users of a distributed computation system, and appropriately rewarding the organizations whose resources are being used by the system. In addition, there is the porting work described in Section 4.8 that is necessary to remove *Wax*'s dependencies on the *Mach* operating system.

Wax monitors processor load and user activity to identify the availability of a machine. A machine is considered idle when the load is below a particular threshold and the time since the last user activity exceeds another threshold. This means that a *Wax* task will execute only when a Host Manager expects the task to have the whole machine itself for a while. While this is a reasonable first approximation, it can both waste resources and lead to thrashing. Additional work should be done to explore the possibility of using dynamic thresholds and/or resource "sharing". By resource sharing, I mean allowing the

[1]The first version of *Wax* was implemented in a transactional style, and, as a result, was susceptible to delays when failures occurred.

distributed computation system to use some of the resources on a machine even when a user or other computations are active.

Wax largely ignores the resource allocation issue, since one of the basic premises in the design of the system is that computation is cheap. The most obvious impact of this premise is the belief that redundant computation can, and should, be used to recover from the latency and failure problems that exist in a wide-area environment. Since the machines being used by *Wax* are otherwise idle, this assumption was a reasonable one. However, a problem arises when a distributed computation systems become popular enough that the computations submitted to it exceed the available idle capacity. The distributed computation system must then incorporate mechanisms for allocating the limited resources amongst the waiting computations.

A real-world problem that *Wax* has not addressed is billing for resources. Many resource owners will desire a tangible benefit before they will be willing to allow a distributed computation system to use their resources. In order to reward resource providers, a distributed computation system will need to be able to reliably monitor resource usage and bill the consumer. In an environment where failures are expected to occur, implementing such a mechanism is a non-trivial task. Unfortunately, the lack of a solution to this problem is probably the principal barrier to the realization of production quality, wide-area distributed computation system.

7.3 Conclusion

In summary, my work has demonstrated the feasibility of building wide-area distributed computation systems. With the appropriate choice of a programming model, a distributed computation system can be implemented that is capable of tolerating the failure and latency problems that exist in a distributed computing environment. The implementation of *Wax* also demonstrates that a useful distributed computation system can incorporate security at a basic level. However, the transition from prototype to production system will also require the development of solutions to the resource allocation and billing issues described above.

Bibliography

[ABB+86] Mike Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, Atlanta, GA, July 1986.

[Bir85] Andrew D. Birrell. Secure Communication Using Remote Procedure Calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.

[Cat92] Vincent Cate. Alex — A Global Filesystem. In *Proceedings of the Usenix File Systems Workshop*, pages 1–11, Ann Arbor, MI, May 1992.

[CD88] Eric C. Cooper and Richard P. Draves. *C Threads*. Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.

[CG89] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[Cra90] Richard E. Crandall. Tales of Godzilla: Adventures in Distributed Computation. *NeXT on Campus*, 1(1):15,21, Summer 1990.

[DE85] P. Diaconis and B. Efron. Testing for Independence in a Two-Way Table: New Interpretations of the Chi-Square Statistic. *Annals of Statistics*, 13(3):845–874, 1985.

[DJT89] Richard P. Draves, Michael B. Jones, and Mary R. Thompson. MIG - The Mach Interface Generator. Available by anonymous FTP from mach.cs.cmu.edu as mig.doc,ps in the directory /usr/mach/public/doc/unpublished., November 1989.

[DO89] Fred Douglis and John K. Ousterhout. *Transparent Process Migration for Personal Workstations*. Technical Report UCB/CSD 89/540, Computer Science Division, University of California at Berkeley, November 1989.

[GBD+93] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM 3 User's Guide and Reference Manual*. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993. Also available by anonymous FTP from netlib2.cs.utk.edu.

[GDFR90] David Golub, Randall Dean, Alesandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Gol92] Richard A. Golding. *End-to-end Performance Prediction for the Internet*. Technical Report UCSC-CRL-92-96, University of California at Santa Cruz, June 1992.

[Gro89] Mach Networking Group. Network Server Design. Available by anonymous FTP from mach.cs.cmu.edu as netmsgserver.doc,ps in the directory /usr/mach/public/doc/unpublished., September 1989.

[HHM86] K.J. Hussey, J.R. Hall, and R.A. Mortensen. Image Processing Methods in Two and Three Dimensions Used to Animate Remotely-sensed Data. In *Proceedings of IGARSS '86 Symposium*, pages 771–776, Zurich, 1986.

[JCRB89] D. W. Jones, C.-C. Chou, D. Renk, and S. C. Bruell. Experience With Concurrent Simulation. In *Proceedings 1989 Winter Simulation Conference*, E. A. MacNair, K. J. Musselman, and P. Heidelberger, editors, pages 756–764, San Diego, CA, December 1989. SCS.

[Jon92] Michael B. Jones. *Transparently Interposing User Code at the System Interface*. Ph.D. thesis, Computer Science Department, Carnegie Mellon University, September 1992.

[Kah91] Brewster Kahle. *An Information System for Corporate Users: Wide Area Information Servers*. Technical Report TMC-199, Thinking Machines Corporation, April 1991. Also available from WAIS server wais-docs.src.

[KB57] T.C. Koopmans and M.J. Beckman. Assignment Problems and the Location of Economic Activities. *Econometrica*, 25:53–76, 1957.

[KS92] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[Lam87] Leslie Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[LC92] P. Peggy Li and David W. Curkendall. Parallel Three Dimensional Perspective Rendering. In *Parallel Computing: From Theory to Sound Practice. Proceedings of EWPC '92, the European Workshops on Parallel Computing*, W. Joosen and E. Milgrom, editors, pages 320–331, Barcelona, Spain, March 1992. IOS Press.

[LCP91] D. D. E. Long, J. L. Carroll, and C. J. Park. *Estimating the Reliability of Hosts Using the Internet*. Technical Report UCSC-CRL-91-06, University of California at Santa Cruz, 1991.

[Lee90] Edward Kihyen Lee. *Software and Peformance Issues in the Implmentation of a RAID Prototype*. Technical Report UCB/CSD-90-573, Computer Science Division, University of California at Berkeley, May 1990.

[LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor — A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[LM89] Arjen K. Lenstra and Mark S. Manasse. Factoring by Electronic Mail. In *Advances in Cryptology: EUROCRYPT89*, J. J. Quisqualer and J. Vandewalle, editors, volume 434 of *Lecture Notes in Computer Science*, pages 355–371. Springer-Verlag, 1989.

[Lot94] Mark Lottor. *Internet Domain Survey*. Technical report, Network Information Systems Center, SRI International, January 1994. See also RFC 1296 and Technical Report UCSC-CRL-92-34, University of California at Santa Cruz, June, 1992.

[Man83] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, New York, New York, 1983.

[Man92] B. Manning. *DNS NSAP RRs*. Technical Report RFC 1348, Network Working Group, July 1992.

[Moc87a] P. Mockapetris. *Domain Names - Concepts and Facilities*. Technical Report RFC 1034, Network Working Group, November 1987.

[Moc87b] P. Mockapetris. *Domain Names - Implementation and Specification*. Technical Report RFC 1035, Network Working Group, November 1987.

[Moc89] P. Mockapetris. *DNS Encoding of Network Names and Other Types*. Technical Report RFC 1101, Network Working Group, April 1989.

[Mou94] John Mount. Uniform Generation of Contingency Tables. A draft of this paper is available on-line in the file: /afs/cs.cmu.edu/project/contingency/doc/chamber.ps.Z., July 1994.

[MP83] C. R. Mehta and N. R. Patel. A Network Algorithm for Permforming Fisher's Exact Test in $r \times c$ Contingency Tables. *Journal of the American Statistical Association*, 78:427–434, 1983.

[Nat77] National Bureau of Standards. *Data Encryption Standard*. Federal Information Processing Standards Publication 46, U.S. Department of Commerce, January 1977.

[Nat80] National Bureau of Standards. *DES Modes of Operation*. Federal Information Processing Standards Publication 81, U.S. Department of Commerce, December 1980.

[NH82] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System*. International Computer Science Series. Addison-Wesley Publishers Limited, London, England, 1982.

[Nic87] David A. Nichols. Using Idle Workstations in a Shared Computing Environment. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 5–12, Austin, Texas, November 1987. ACM.

[Nic90] David A. Nichols. *Multiprocessing in a Network of Workstations*. Ph.D. thesis, Carnegie Mellon University, February 1990. Also available as Technical Report CMU-CS-90-107, Carnegie Mellon University, February 1990.

[NS78] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978. Also available as Technical Report, CSL-78-4, Xerox Research Center, Palo Alto, CA.

[NVR68] C. E. Nugent, T. E. Vollman, and J. Ruml. An Experimental Comparison of Techniques for the Assignment of Facilities to Locations. *Operations Research*, 16:150–173, 1968.

[P10] *Threads Extension for Portable Operating Systems*.

[PC89] Panos M. Pardalos and James V. Crouse. A Parallel Algorithm for the Quadratic Assignment Problem. In *Proceedings Supercomputing '89*, pages 351–360, Reno, Nevada, November 1989. ACM.

[SA89] Mark Sullivan and David Anderson. Marionette: A System for Parallel Distributed Programming Using a Master/Slave Model. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 181–188, Newport Beach, California, June 1989. Also available as Technical Report UCB/CSD 88/460.

[Sch89] M. F. Schwartz. The Networked Resource Discovery Project. In *Proceedings of the IFIP XI World Congress*, pages 827–832, San Francisco, California, August 1989.

[SG86] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2), 1986.

[SH82] John F. Shoch and Jon A. Hupp. The "Worm" Programs — Early Experience with a Distributed Computation. *Communications of the ACM*, 25(3):172–180, March 1982.

[SHN$^+$85] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 35–50. ACM, December 1985. Also available as Technical Report CMU-ITC-039, Carnegie Mellon University, April 1985.

[SKK$^+$90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Availabe File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[SNS88] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Usenix Conference Proceedings*, pages 191–200, Dallas, Winter 1988.

[Spa88] Eugene H. Spafford. *The Internet Worm Program: An Analysis*. Technical Report CSD-TR-823, Department of Computer Sciences, Purdue University, December 1988.

[The86] Marvin Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems*. Ph.D. thesis, Department of Computer Science, Stanford University, 1986. Also available as Technical Report STAN-CS-86-1128.

[TLC85] Marvin Theimer, Keith A. Lantz, and David R. Cheriton. *Preemptable Remote Execution Facilities for the V-System*. Technical Report STAN-CS-85-1087, Department of Computer Science, Stanford University, 1985.

[TY91] J. D. Tygar and Bennet S. Yee. *Dyad: A System for Using Physically Secure Coprocessors*. Technical Report CMU-CS-91-140R, Carnegie Mellon University, May 1991.

[WHH$^+$92] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[Yee94] Bennet S. Yee. *Using Secure Coprocessors*. Ph.D. thesis, Carnegie Mellon University, 1994.

[ZWZD92] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle. *Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*. Technical Report CSRI-257, Computer Science Research Institute, University of Toronto, April 1992.

Index