*18 January 1993*

# Performance analysis of Amadeus on top of OSF/1 MK

*Youcef Laribi*

OSF Grenoble RI

**Abstract:**

This report describes a performance analysis of Amadeus, a Comandos implementation, that has been ported on top of OSF/1 MK. The goal of the analysis is to understand the different components of Amadeus, evaluate their adequacy on top of a server/microkernel architecture represented by OSF/1 MK and identify the bottlenecks due either to Amadeus itself or to the platform (OSF/1 MK) supporting it. For this purpose, benchmarking coupled with profiling techniques were used as basic methods to evaluate the performance. The most frequent operations in Amadeus applications of the Amadeus interface were measured under different circumstances by ad−hoc benchmarks. This study allowed to not only evaluate Amadeus itself but also to understand what is relevant to measure in a Comandos−like platform and what are the critical components, and execution paths in such architecture that should be performant.

## 1 Introduction

In order to prove the adequacy of a server/microkernel architecture to supporting object−oriented packages, Amadeus, a distributed object−oriented platform, was ported on OSF/1 MK, a system structured in a such way . However, if the micro−kernel based systems have many known advantages, the performance of Applications running on those environments is not well understood. Moreover, applications, like Amadeus, using either microkernel services or host server features make their performance evaluation difficult.

The goal of the performance study of Amadeus on top of OSF/1 MK is three−fold. First, it allows to understand well the dependencies between the different parts of the system. It also permits to evaluate the use of OSF/1 and Mach3.0 features as a support platform to Amadeus, or what are the features that are more adequate to use than others. Last bottlenecks inside Amadeus software, can be identified and then removed or improved.

Benchmarking coupled with profiling methods were used to evaluate Amadeus platform. Benchmarking is useful to focus on specific components of the system to study their

performance while profiling is used to dissect the overall results obtained by benchmarks into individual internal system modules results.

The Comandos project is part of ESPRIT, the EC project. Comandos aims at providing a computational and development environment for distributed applications basically used in a LAN environment. The most important innovation of Comandos, is its approach to integrate the operating system, programming languages and development tools in a high comfortable object−oriented environment that hide the distribution behavior of the system, and provides the programmer with a computational model, in which parallelism and shared data are the keys.

This report is structured as follows: First an overview on Amadeus model and architecture is given to familiarize the reader with the terminology used later. Then C**, the Amadeus language is presented and its interface to Amadeus system is described. Different measurement methods are presented in section 4 and the choices adopted are justified. We then present the different evaluations done on the Amadeus interface primitives exported to applications and present the relevant results. The study of the impact of Mach3.0 features in Amadeus is the subject of section 7. Last, conclusions of this study are given in section 8.

# 2 Amadeus Overview

The Amadeus project provides an integrated platform for the development of applications which run in a distributed and persistent environment. An application which runs in this environment is structured as a collection of co−operating objects which may be dispersed throughout a distributed UNIX system [1].

The Amadeus environment ensures an object−oriented support for writing distributed applications by providing the programmers with high−level object−oriented languages. The programmer has not to deal with the location of his/her objects. He/She can specify which type of objects are persistent data and which are not by means of language constructors. The Amadeus environment also provides the user with features to take advantage of the parallelism offered by a distributed system. The user can invoke many operations on objects in parallel. Many applications can share objects, leading to an easy development of cooperative applications (CAD, Document processing, Information systems, Databases, ... ).

The Amadeus environment consists of two components: The generic runtime, and the Amadeus kernel. The generic runtime is layered above the Amadeus kernel and provides a language−independent method of accessing remote objects.

On each node of the Amadeus cell runs an instance of the Amadeus server. Application processes running on that node are called Amadeus clients. Amadeus applications are written in high−level object−oriented languages. Currently, only the C** language, an extension of C++ language that supports distribution and persistence is provided. The compilers of these languages should provide a language specific runtime which fits the

languages representations to Amadeus generic representations. To access Amadeus services, the applications are linked to the Amadeus library which contains all the generic runtime and the bulk of the Amadeus kernel.

## 2.1 Computational model

As each system gives the programmer a virtual machine view, the Amadeus environment provides the programmer with the following abstractions :

- Jobs.
- Activities.
- Objects.

### *2.1.1 Jobs*

Jobs can be assimilated to (heavyweight) distributed processes. It's a distributed address space on which objects used by the activities of that job are mapped in. On top of Unix, it's implemented as a set of Amadeus clients. When an activity of a job invokes an operation on an object and that object is on another node, the job diffuses its address space to encompass that object by creating an Amadeus client on that remote node. Recall that an Amadeus client is implemented as a Unix process (See Fig. xx).

Users can create new jobs by running amadeus applications from the Unix shell (as for Unix applications) or by using amadeus interface. The created job will have the same text image as its creator. When created, a job consists of a single Amadeus client (or context). Within this context, a single activity is created. If the job is created from the shell, then the first activity (see 2.1.2) carries out the mainline of the Amadeus application. If it's created by another job, then it carries out a specified operation on a specified object, as indicated in the invocation frame passed to it by its creator [2]. The Amadeus clients created as a result of a job diffusion, still exist after accomplishing the invocation that caused their creation. This is, to avoid recreate an Amadeus client on a node, each time the job has to do processing on that node.

### *2.1.2 Activities*

Activities perform successive invocations on objects; they are the processing entities in the Amadeus model. Activities can be assimilated to lightweight, distributed processes. An activity is local to one job but can span many contexts of that job. On top of Unix, they are implemented by a simple lightweight process package. An activity is represented by a single lightweight process (thread) when it is created. The local invocations, where the invoking and the invoked object are in the same context are handled immediately like function calls. If the activity performs a remote or cross−context invocation, the local thread is stopped , and a new thread is created in the target context where the object is mapped to act on behalf of the activity in that context.

### *2.1.3 Objects*

Objects represent the structure unit of applications. In object−oriented environments, computations are handled as successive invocations on objects. New objects can be created during the computation and already existing objects (persistent) can be retrieved from the backing store, modified and then stored again. It's much like, the mapped files feature provided on some Unixes. The communication between processing entities (activities) is also handled by object sharing. Indeed, the Amadeus and in general the Comandos computational model, use the sharing memory scheme for communication. However in the current versions of Amadeus, synchronizing access to shared objects is still the responsability of the programmer. Once an object is mapped in some context, all invocations on that object take place in that context. Hence, contexts in Amadeus operate as object servers. There are two types of objects in Amadeus :

- Real objects.
- Proxies.

A real object is the common sens of an object, while a proxy is a representative of a real object on some context. This concept can be compared in RPC scheme, to function stubs. A proxy object has exactly the same operations as the object it represents except that the operations have not the same implementation. The proxy implementation is automatically generated by the C** compiler to interact with the Amadeus system to perform the invocation if the real object is mapped in another context, or if the object is not yet mapped anywhere. When the real object is mapped in a context, it replaces its proxy object in that context transparently to the application. The first time an activity references an object not mapped yet in a local context, a proxy for that object is created using the object *stub*(1). An object stub is a large object reference, holding information used to find the object and to create a proxy of that object. Stubs are objects of a generic runtime class, and are created when objects are. When the state of an object refers to other objects, these references are actually pointed to stubs of these objects which are stored just after the state of the object.

## 2.2 Amadeus Architecture

As we said earlier, The Amadeus environment is made up of the amadeus server and the Amadeus library linked to Amadeus applications. Internally, the Amadeus environement consists of the following components :

- The storage system provides a long−term storage for objects.
- The Generic runtime is the interface of Amadeus kernel to the implemented languages, providing them with a generic interface to implement distributed and persistent objects.
- Remote Location Service which locates where clusters are stored or mapped.

---

(1) To not onfuse with RPC stubs

- Activity manager which provides facilities to create jobs and activities and implement operations on these. This component also provides a remote and cross−context invocation service.
- Communication system which implements the inter−kernel message protocol (IKM).

### *2.2.1 Storage system*

In Amadeus, objects are stored into containers, which constitute the manageable units of storage and can be assimilated in classical filesystems to volumes. Each node may manage one or more containers. On top of Unix, containers are implemented as directories.

Within a container, objects are grouped into clusters. Usually, clusters contain related objects, and a cluster is used as a mapping unit when an object fault occurs in the virtual object memory. The clusters are implemented as files of the directory representing their container. To allow that this persistent store is accessible to all nodes participating in the Amadeus environment, a distributed filesystem is used. On Ultrix implementation, NFS is used. On each node, all the containers should appear with the same path name. Amadeus may contain up to $2^8$ containers. Each container may contain $2^{24}$ objects.

### *2.2.2 Generic runtime*

The Generic runtime (GRT) is the interface of the Amadeus environment to applications and languages. To allow the programmers to benefit from the Amadeus computational model, high−level object−oriented languages should be extended to add persistence and distribution to their object model using for that, the Amadeus interface. Amadeus implementors have prefered to not put all the interface between Amadeus and languages into the compiler to avoid to repeat this operation for each new implemented compiler. Rather, the interface between the language and Amadeus, is divided into a language specific runtime, generated automatically by the compiler and a generic part which performs actions in a language−independent fashion and hence, is reused for all languages. When a language specific information is needed, the generic runtime make upcalls to code supplied by the language. The main functions, a generic runtime accomplish are :

- Detecting object/cluster faults.
- Mapping and unmapping clusters.
- Marshalling parameters for remote invocations.
- Dispatching incoming invocations.
- Managing object creation and promotion.
- Garbage collection.

The generic runtime part of Amadeus is written in C++ which facilitates its use by C** applications.

Notice, The GRT handles also the Amadeus name service. The naming scheme is quite simple. It relies on Unix naming for files. The name service associates pathnames with object

identifiers. Two operations are exported by the name service: The checkin operation to associate an object identifier with a name (string) , and a lookup operation which allows to retrieve the identifier of an object given its name. Each association <name, identifier> is represented by a file in Amadeus, so that hierarchical name space of Unix can be reused for Amadeus naming.

### *2.2.3 Remote location service*

The remote location service (RLS) is responsible for locating objects given its generic identifier. For that, the RLS implements a shared table on each node called the Kernel Cluster Table (KCT) which is shared by all Amadeus clients and the local Amadeus server within a node. This table contains an entry for each mapped cluster in the local node. Another table is used for clusters mapped at remote nodes for each remote container. This table is called the MCT (Mapped Cluster Table), and is also shared by all the contexts of the local node. This table contains for each cluster contained in the corresponding container, the node where it is mapped. These tables and others are implemented using the shared memory feature of the Ultrix system.

### *2.2.4 Activity Manager*

The activity manager manages jobs and activities including implementing remote and cross−context invocations. It also implements a lightweight user−level thread package and supports all operations on activities and jobs together with providing low−level synchronization primitives.

### *2.2.5 The Communication system*

The Communication system implements the primitives needed to establish communication between contexts either on the same or different nodes. Hence, it constitutes the basis of cross−context and remote invocations. UDP internet domain sockets are used for remote communications and Unix domain sockets for local communications.The communication layer overcomes the limitationsof the UDP protocol such as lost and duplication of messages and Unix domain socket for limited−size messages.

## 3 Some C** concepts

Amadeus supports C++, the only language provided with it currently, with some extensions; This extended language is called C**. C** remains upwards compatible to C++; Any C++ program, can be compiled and run in an Amadeus environment. The extensions of C++ are fundamentally for persistence and distribution features. C** lets the the programmer free to choose which data is persistent. The same rule for distribution is applied. Lets describe how C** supports both persistence and distribution.

## 3.1 Persistence support

Persistence in C** is supported via the concept of persistent classes. The instances of such classes may be persistent or not, depending if there are other persistent objects refering to them. The persistence is also integrated in the inheritence tree, so that classes derivative from other persistent classes, are also persistent. The instanciation of persistent classes is more expensive than for ordinary C++ classes, because it incurs overheads in code and space. The roots of persistence are named, and registered in the name space.

Amadeus supports the persistence as an optional attribute of objects, because it assumes that programmers know better what they want, and do not want to pay for something they don't use. This is why, using non−persistent objects has the same cost in C** as in C++.

## 3.2 Distribution support

Likewise, distribution is supported at the language level. The programmer specify if she wants to use an object remotely. The distribution is completely dissociated from persistence. An object may be persistent but not distributed and vice−versa. However, in Amadeus, distribution is associated with individual members of the class, rather than to the whole class as was for persistence. The class members preceded by the term *global* can be used remotely. This means that an invocation where the invoking and the invoked object are in two different contexts is possible if the invocation occurs on a *global* member. A member which is not global, cannot be used remotely. A class which contains at least one *global* member is called a *global* class.

The use of remote objects is possible, because of the proxy concept. The compiler generates proxy classes for global classes. These proxy classes contain the global members of the original class plus some extra operations. The invocations to global members are actually transformed by the compiler to invocations on the corresponding proxy member of the proxy class. The implementation of proxy members, tries to find out if the object is local. If it is not, then a cross−context or remote invocation is performed if the object is mapped elsewhere. If it was in the backing store, it is mapped into the context. This is why calling global members, incurs some extra cost, even in local. If an object refer to other objects and these objects were instanciated from global classes, then the mapping of the refering object, doesn't automatically mean the mapping of the refered objects. This is why, these objects are represented only by proxies until the first global operations are invoked on them. The proxy operations forward the operation to a real object or replace the proxy by the real object if it is possible to map it locally. Non global members have not such proxy members generated and so cannot be used remotely

# 4 Methodologies of measuring

Performance may mean timing deadlines, constraints in the use of resources, or a required degree of reliability. In other cases performance is a qualitative measure. Through out all the report, we use the first definition of performance, that is, timing constraints.

Commonly performance studies aim at understanding existing systems in order to tune them or to lead to a new design based on the results observed on the existing system.

There are many methodologies used to evaluate the performance of a running system, we can quote :

- Profiling the system if the host offers primitives and compilers that can produce a profiled version of the target system and an analyser of the profiling data.
- Benchmarking : consists of writing typical applications that use the target system, and measure the performance of these applications. It's commonly used to compare between similar systems.
- Instrumenting or monitoring the target system by inserting code that collects data from the running system. It's used commonly by developer teams to collect information about the parts of the system that need enhancements.

In our approach to evaluate Amadeus, we used a combination between profiling and benchmarking. Instrumentation needs a complete and complex software that is based on collecting data based on event occurrence and tools for viewing the collected data. It's commonly used in the development phase. Benchmarking, however, allowed us to have the point of view of the system user, and to measure what is the impact of the system performance on user application performance. We combined benchmarking with profiling in a complementary fashion. When we want to understand some unexpected results, we use the Unix tool gprof to obtain the trace of the system internal called operations, and the time spent in each one. Profiling can be used only on coarse−grained time operations, since it's based on periodic sampling of the program PC. Times spent in each procedure are computed using a statistical approach, which means that the results are meaningful, only if the system is running for long period (many seconds). Some operations does not exceed one millisecond, so we cannot use profiling in such cases.

Our benchmarks were written to measure the basic and frequent operations in Amadeus. Before that, we had to determine what are these operations, and what are the parameters that influence the operations. These parameters may be direct parameters (such as object size in creation) or indirect parameters (such as the used filesystem). We have also taken care of side−effects of certain operations on others. Hence we have separated different operations to be executed in different address spaces (Amadeus Jobs).

# 5 Measurement precautions

On top of OSF/1 MK 4.1, Amadeus system is running on BM600. These machines have a 386 CPU, 16 MB RAM, 256 KB cache memory, 25 Mhz frequency, IDE disk controller, WD0 ethernet card. The system clock resolution is of 10 ms (100 tick per second). Thus, the results obtained by using this clock have a margin of error of 10 ms.

Since the elapsed time of some operations is less than the clock resolution, we can't measure directly the operation by taking the time before and after the operation, even if this is the best method to have trusted results. To obtain more precise results, statistic techniques are used to approximate the target operation timing. For example, local synchronous invocations take only some microseconds, say t is the real value, and our clock is far from having a microsecond resolution. Hence, the best method to approximate the real value is to repeat the invocation sufficient number of times, say X is this number and T is the total elapsed time measured to perform X invocations. X should be chosen so that T >>10 ms. The approximate value of the time needed to make an invocation is computed as :

$$T / X$$

The absolute error of this value is of (10 / X) ms. To have a margin of error of the order of a microsecond, the operation should be repeated at least 10.000 times.

If this method is accurate to measure some operation timings, it has however a drawback when measuring side−effects operations. If we take for example, the object creation operation, measuring this operation by repeating it thousands of times may lead to inaccurate results, since the creation of an object may have a side effect on the creation of the subsequent object. The cost of creating the first and the last object is not the same. So computing the timing of this operation as the arithmetic mean of the total time is not correct. Another problem due to the statistical approach is that invoking an operation thousands of times may lead to exhausting the resources of the system. For example, the OT (Object Table) where objects are registered since mapped has a limited size. Hence mapping thousands of objects may exceed the resource limit and oblige to recompile the whole Amadeus software for higher resource limits. Sometimes these resources are not of Amadeus but the OSF/1 resource limits such as exceeding the allowed paging space. Moreover, even if the resource limits are not exceeded, the extensive use of some system resources may influence the results obtained. Invoking thousands of objects, lead to touching many pages of the process sparse address space, and hence, induces an important paging activity, which may interfere with the results obtained.

Apart from the precautions that should be taken when using a statistical method, measuring requires the collector(2) to ensure that during the measurement, the status of the system is normal. Measuring programs while other cpu−greedy applications are running is not a good idea.

---

(2) Person or program who is measuring

Evaluating the performance of an operation, may require preparing the environment needed to the experiment. For example, before starting the measurement of a time required by looking up an object, the objects required by the experiment should be created by a separate application. This is why preparing some experiments and obtaining the results may take many hours.

Each designed experiment is repeated many times so that to avoid collecting abnormal values due to an intensive paging activity or a instantaneous congestion of the network. Some experiments may have different values, for each new execution. For example, reading a file, depends if the file is already cached in the filesystem cache or not and since this information cannot be perceived by users, it is sometimes difficult to decide which value to take. This is why we resort to take the mean time of all the values obtained.

Taking these precautions, the results obtained for different Amadeus exported operations are exposed together with the benchmarks used to collect them in the following section.

# 6 Measuring Interface primitives

What the programmer perceives from Amadeus is its interface. In Amadeus, the interface is provided in two ways: (1) by language constructors, and (2) by run–time interface implemented as a library. The library contains the whole generic runtime and the bulk of Amadeus kernel. Each Amadeus application is linked to this library. The performance of an application relies in part, on the performance of the underlying primitives exported by the Amadeus virtual machine and which the application may use explicitely or implicitely.

We want to measure how the different components of the Amadeus virtual machine are efficient and performant. We also want to identify bottlenecks and to analyze what is the impact of the use of Mach3.0 features in the version running on OSF/1 MK.

In this section we try to analyse the most used services by Amadeus applications. These services are :

1. Object's creation and mapping.
2. Name service usage.
3. different schemes of object invocation.

## 6.1 Object creation

We are only interested in persistent objects, because volatile objects are handled as in C++. Amadeus handles transparently the call to Amadeus runtime primitives to provide object creation service. The C** compiler generates for the *new* operation,(3) a call to Amadeus runtime for objects of persistent classes. The generic runtime performs all related operations to the creation, then it returns a pointer to the newly created object as in

(3) The C++ operation for creating objects.

conventional C++ programming. Of course, the cost of this operation is much higher than in C++, because it's not just a matter of allocating space for the newly created object. To evaluate the cost of this operation, we should take into account, two parameters:

- The size of the created object.
- The number of objects already mapped into the current context.

Since it's difficult to estimate the time needed to create one object, we have resort to measure the time needed to create many objects of a given size and to observe the behavior of the system as the address space is filled up with objects of that size. The time needed to create an object is computed as the mean time.

In our benchmarks, we aimed at evaluating three aspects :

- The influence of object size on the cost of its creation.
- The influence of the data organisation on the creation cost;
- The cost of the creation op. depending on how full , in terms of objects, the context is.

The first point is straightforward. However, let's have an example for the second point:

Suppose that we have an array of 1000 objects. Is it more efficient, to have this array stored in one object, or having it split into many smaller objects. In other words, what is the nature of variation when creating small comparing to big objects.

The third criterion is used to observe the uniformity of creation of objects while the address space (context) is filled up. In other words, is the cost of creating the first object the same as creating the 1000th object of the same type ?

To observe the behavior of the creation operation regarding these three criterions. We have designed one dedicated benchmark, that takes into account the three criterions.

**Description**

The benchmark defines many object classes. Each class, represents an array of bytes. The only difference between these classes is the size of that array. Given a class, the benchmark instantiates that class a number of times (N). The value of N itself varies, and for each fixed value N, a mean time of creating an object of the total time needed to create N objects is computed. To avoid interference between different measurements, each experiment [(4)]takes place in a sperate context. So, the innermost loop of the benchmark varies the N value while the outermost loop varies the object class. In the innermost loop a separate context is created to handle the experiment.

Fig.1 shows the performance of this operation for different sizes. The X axis is representing the number of created objects from a given size. The elapsed time needed to create that number of objects is represented on Y axis. We can compare the different curves corresponding to different sizes. Fig.2 highlights the criterion 2 by showing the cost of creating a given size of data by varying the object's size containing that data.

---

(4) The measurements done for a fixed N and a fixed class are considered as a sperate experiment.

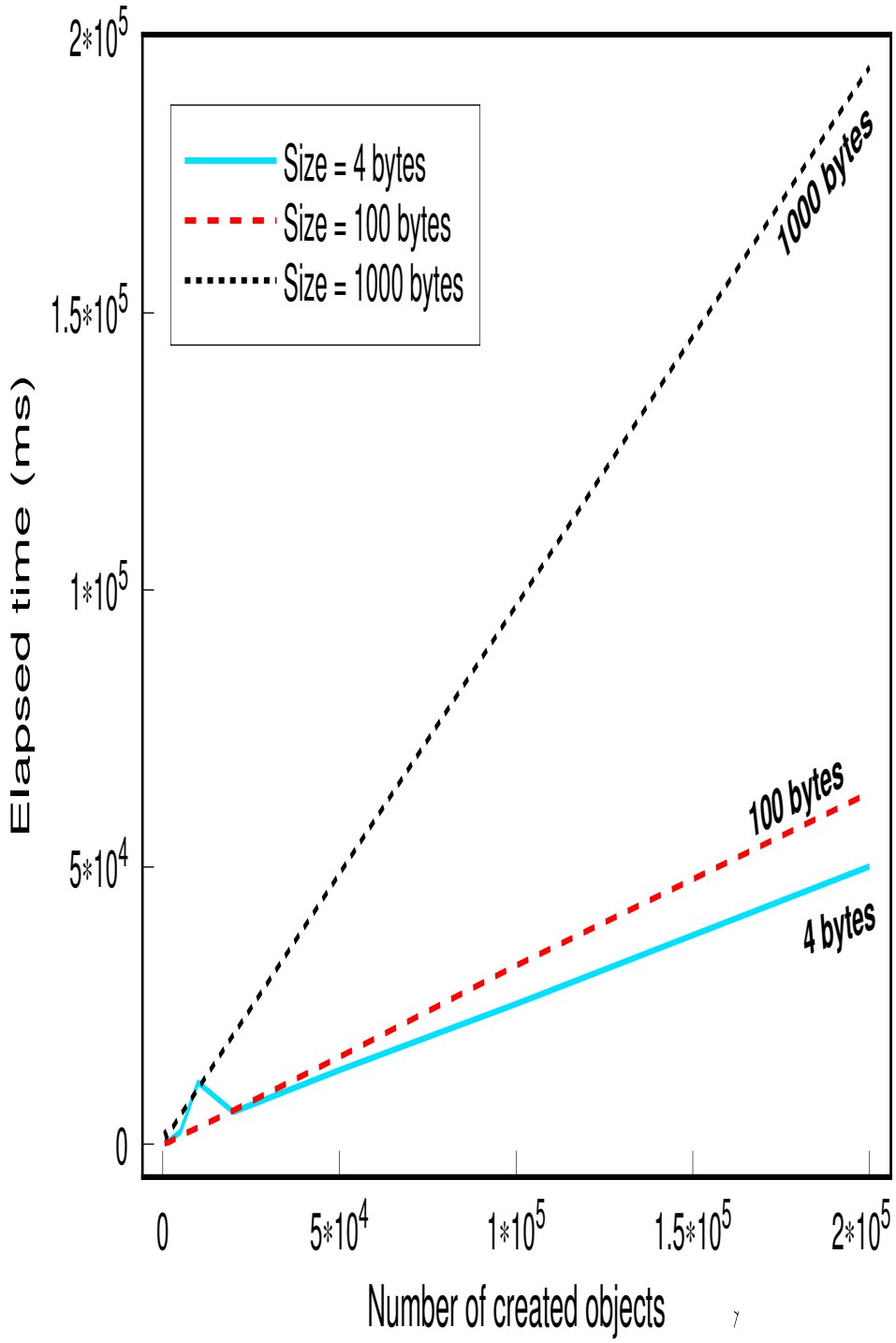


Fig. 1 Local creation cost depending on Address space filling

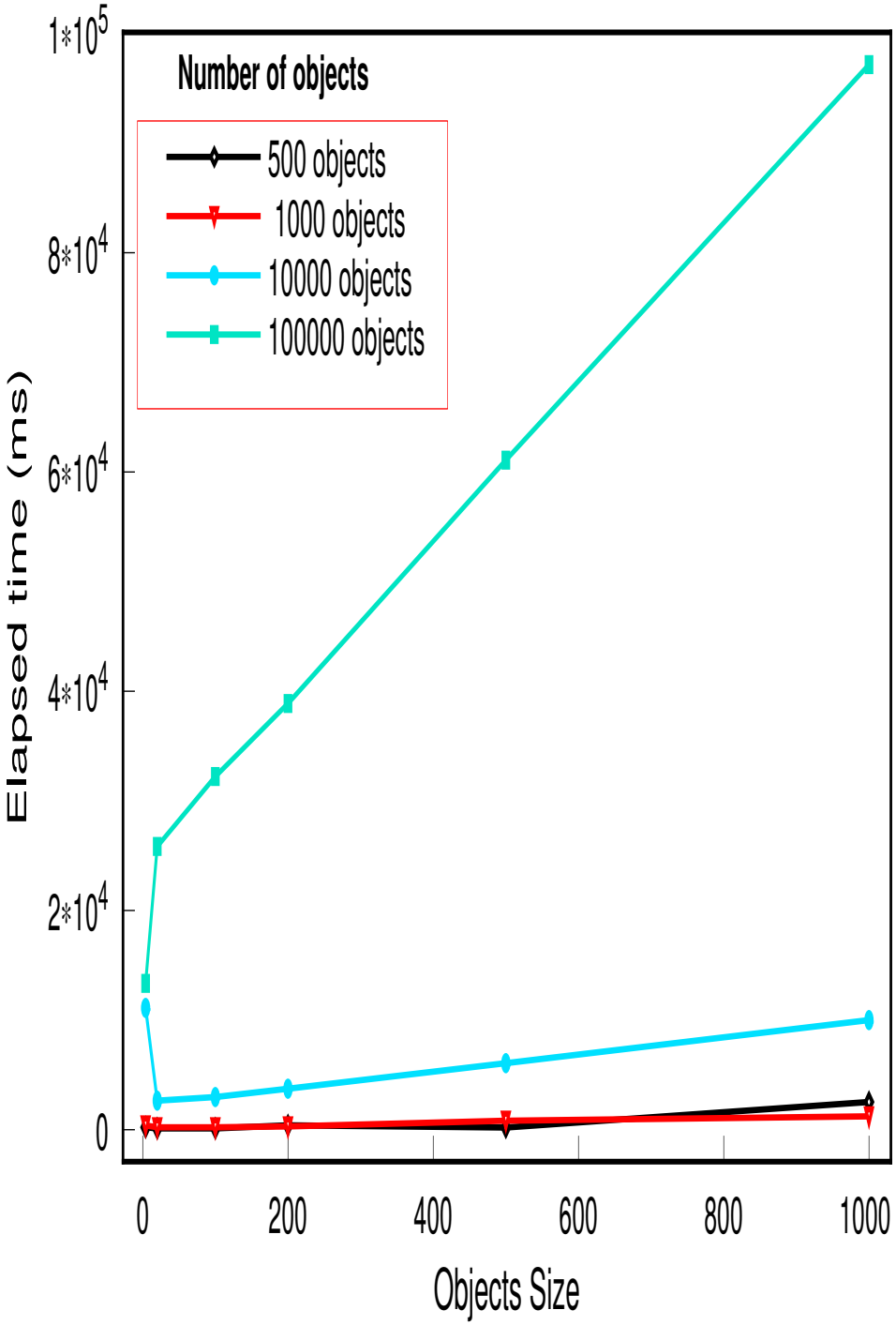


Fig2. Creation cost depending on Object size

**Analysis**

When a *new* operation is issued on a persistent class, an instance of that class is allocated in the current address space and the object is registered in the object's table. However, there is no operation involving access to a disk. The created object is stored at the disk only at the end of the execution of the application when all clusters are unmapped and stored back to the persistent store. Notice that the bulk of the time of object creation is spent in the GRT part.

From figure 1, we notice first that the size fo object is an important paramater for the cost of the creation operation, even if in Amadeus, the created object is not immediately stored on disk until the termination of the creating application. This is why we haven't considered the type of used filesystem as a criterion of this operation. The second important point that can be noticed is that the gradually filling of the context by objects of a given size is linear. This means that the cost of creation does not depend on the number of mapped objects. This is a good asymptotic behavior, since this means that the creation cost is the same for the first as for the last created object. However, for the same size, creating few big objects is more efficient than creating many small objects (see Fig. 2). This is much like, the cache buffer size in filesystems and was expectable, since each created object involves an overhead to process it. To have a rough picture of performance cost of the creation operation it varies from some tens of microseconds (86 us for 4 bytes size) to hundreds of milliseconds (376 ms for 1Mb size). From this benchmark, we should retain :

- The size of an object influences the performance of its creation.
- The cost of the creation operation is uniform for an object type all over the life−time of the job. The creation of the last object costs as much as what the first object cost.
- The organisation of data in terms of objects influences the performance cost of the creation operation. Grouping data in few objects is more efficient than having it sparsed over many objects. Notice that this can go against the oriented−object programming paradigms.

## 6.2 Name service usage

A name service is necessary in any persistent object environment. The humans use string names to identify objects, while the internal system uses unique numbers to identify objects. To translate between the two identifications, the system provides a name service. In Amadeus, the programmer can use the name service to associate a string name to an object identification. It can also use an object name to obtain its identification. Since any persistent application uses this component, it seems to be important, that the checkin and a retrieval operation have an acceptable performance. As for object creation, we have measured the

time needed to create a growing number of objects, to see how the name service component evolves.

There are two operations involved in the usage of the name service; the checkin operation and the lookup operation.

**Checking in objects**

Checking in objects is the operation which consists of associating a string name to a system–wide object identifier. Many names can be associated to an object. We are interested in determining the parameters influencing this operation and studying its behavior. First investigations allowed us to state that this operation doesn't depend on the size of checked in objects, nor on the number of already mapped objects. However, two parameters were found relevant to this operation:

- The filesystem used.
- The location of the checked in object.

Hence we have developed a benchmark that deals with different locations of the object and we ran it on different filesystems. The results gathered are presented in the table below.

| Filesystem | Location | Elapsed time (ms) |
|---|---|---|
| AFS | Local | 420 |
| AFS | Cross–context | 40 |
| NFS | Local | 240 |
| NFS | Cross–Context | 90 |
| UFS | Local | 20 |
| UFS | Cross–Context | 10 |

Fig.3 Performance of Object registration

**Looking up for registered objects**

The purpose of this operation is to obtain a memory pointer to an object given its name. Notice that the object looked up may not be mapped . Hence, the cost of a lookup operation depends on the operations done to get the object ready for use. For example, if we lookup on an object of whose cluster is not mapped yet, the cost of looking up will include the cost of mapping the whole cluster that contains that object and binding that object, because we expect that the lookup operation returns a pointer to an object in the memory. Furthermore, the mapping operation is more or less expensive depending on the number of objects in that

cluster and how they are related to each other, because mapping may need identifying the objects in the cluster and the more we have, the more the operation is expensive. Another issue that should be taken into account when measuring the lookup operation is the number of objects pointed at, by the target object, or what we have called the degree of complexity of that object.(5) Since the user expects a consistent state of the object, the system should also at least map the objects pointed by the target object. These objects may be on different clusters leading to mapping many clusters, which increase considerably the lookup operation.

We were concerned to measure the different schemes that may be involved when looking up an object. To summarize the lookup operation is influenced by the following parameters:

- The degree of complexity of the target object or in general data organisation.
- The location of the object and the objects that it refers.
- The filesystem used.

We have noticed that a lookup operation may vary from few milliseconds to some hundred seconds, depending on the conditions surrounding this operation. To illustrate these cases we have written many benchmarks, each of, representing a specific case. We describe below these benchmarks.

### *6.2.1 Benchmark 1*

Description

This benchmark concerns the lookup on an object of type *array_int.* which contains an array of N references to objects of type *perm_int*. Each object of type *perm_int* is the head of a list of objects of the same type (See Fig. 4). This list contains M elements including the head. Each list is created by a separate application leading to have the list grouped in one or more clusters depending on M. This means that each list is stored on different clusters. The class of objects of type *perm_int* is not global. Hence, if one object of the list is bound, all objects of the list are. This benchmark tries to measure the worst case of looking up an object not mapped anywhere yet and managed by the local site. Indeed, the lookup up of an object of type *array_int*, leads to map the cluster where this object is located plus the clusters where the objects of type *perm_int* are, since our benchmark has stored each list in different clusters. The objects that the object *array_int* points to, are heads of lists. Their binding involve the binding of all the list. So, we obtain the mapping of at least (N + 1) cluster (6)and the binding of at least ( 1 + M*N) objects. The graphics no 5, 6 and 7 gives an idea of the costs of such lookups, depending on the number of elements in the array (N) and the number of objects per cluster (M). M takes respectively the values 10, 100, 1000 in figures 5,

(5) See Appendix A

(6) N clusters correspond to the N lists, each list stored in at least one cluster plus the cluster where the object Array is stored.

6 and 7, while N varies from 1 to 20 in each figure. It was sometimes difficult to accomplish these measurements due to exhaustion of Amadeus resources such as the OT (object table). So the Amadeus runtime was recompiled for higher resource limits in these benchmarks.

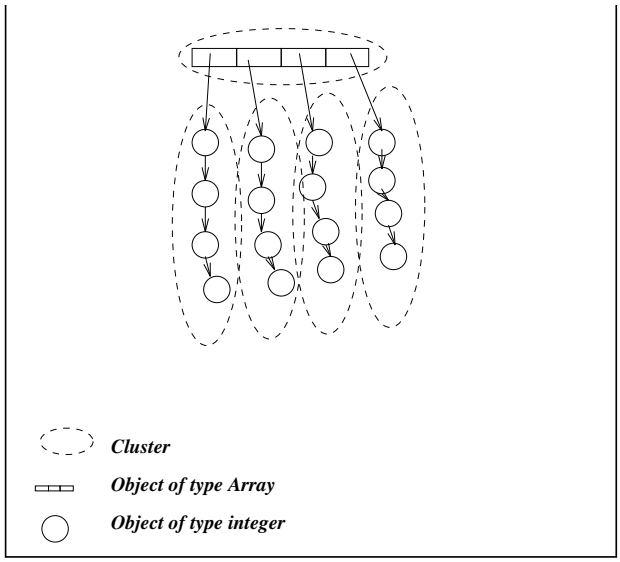


Fig.4 Data organisation

*6.2.2 Benchmark 2*

This benchmark aims at contrasting the influence of the organisation of data on the performance of the lookup operation. This benchmark measures the lookup operation on the most favorite case of data organisation. The looked up object does not contain references to other objects. The lookup operation, hence, involves the mapping of one cluster and the binding of one object (the looked up object) containing that object cost mapping operation in comparison to binding operation. Unlike the previous benchmark, this one does not bind all objects mapped. The cost fo this operation is mainly influenced by disk accesses and location of data (local disk, remote disk). The size of the object is not important in this case since what is mapped is the cluster that contains the object, not the object itself.

| AFS | NFS | UFS |
|---|---|---|
| 90 ms | 96 ms | 35 ms |

Fig. 8 Best case performance for lookup

### *6.2.3 Benchmark 3*

After experimenting the local behavior for the lookup operation, we turned to observe the case of lookups that involves objects stored remotely. In fact, a remote location of an object involves a fixed overhead for all benchmarks. This is why it's measured separately. This benchmark is similar to benchmarkb 2 except that the object is stored in a LC (Logical Container) not managed by the local site. The first lookup on that object, involves an overhead due to the diffusion of that job as we can see in the following table.

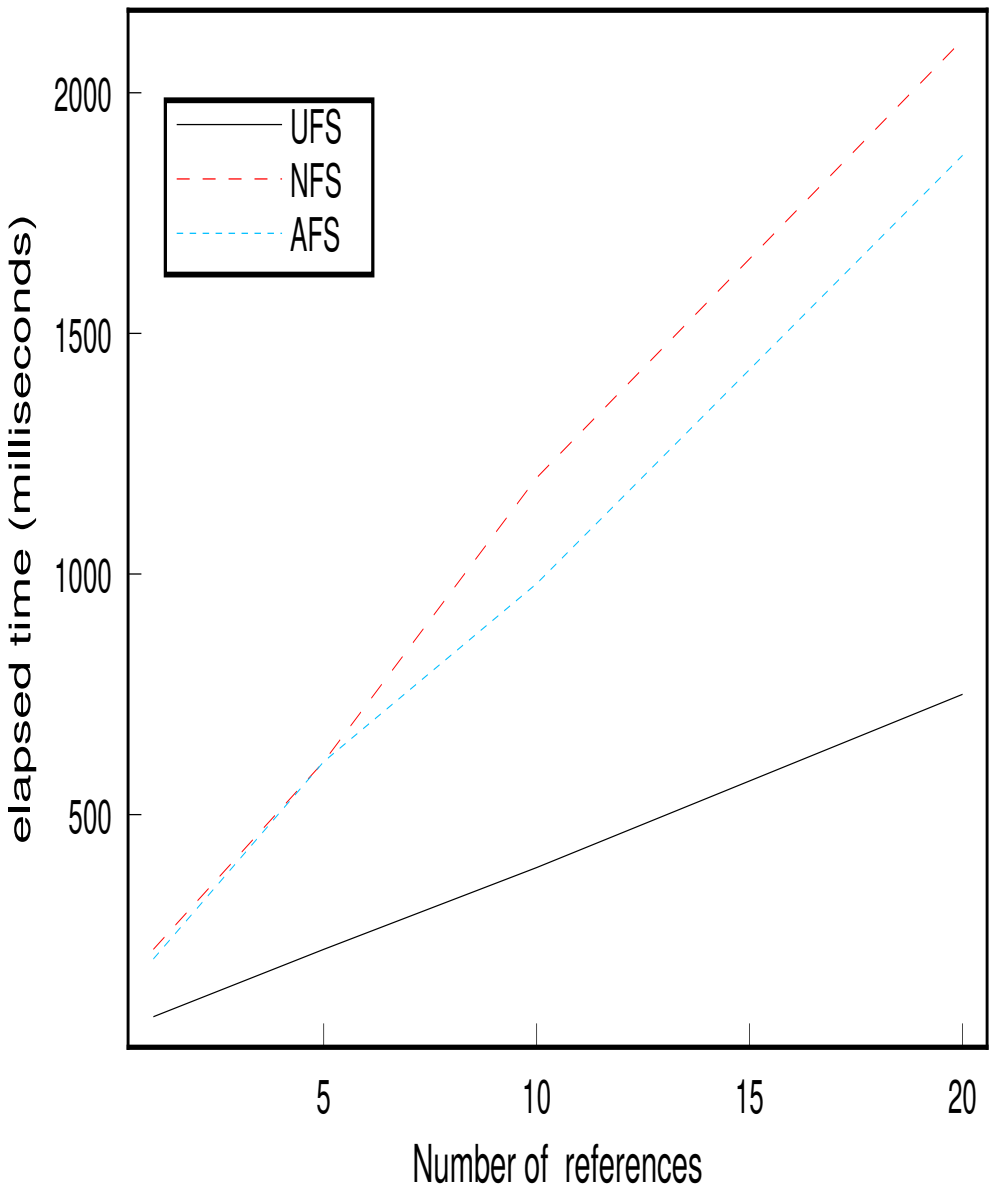


Fig.5 Measuring the cost of the lookup operation for M = 10

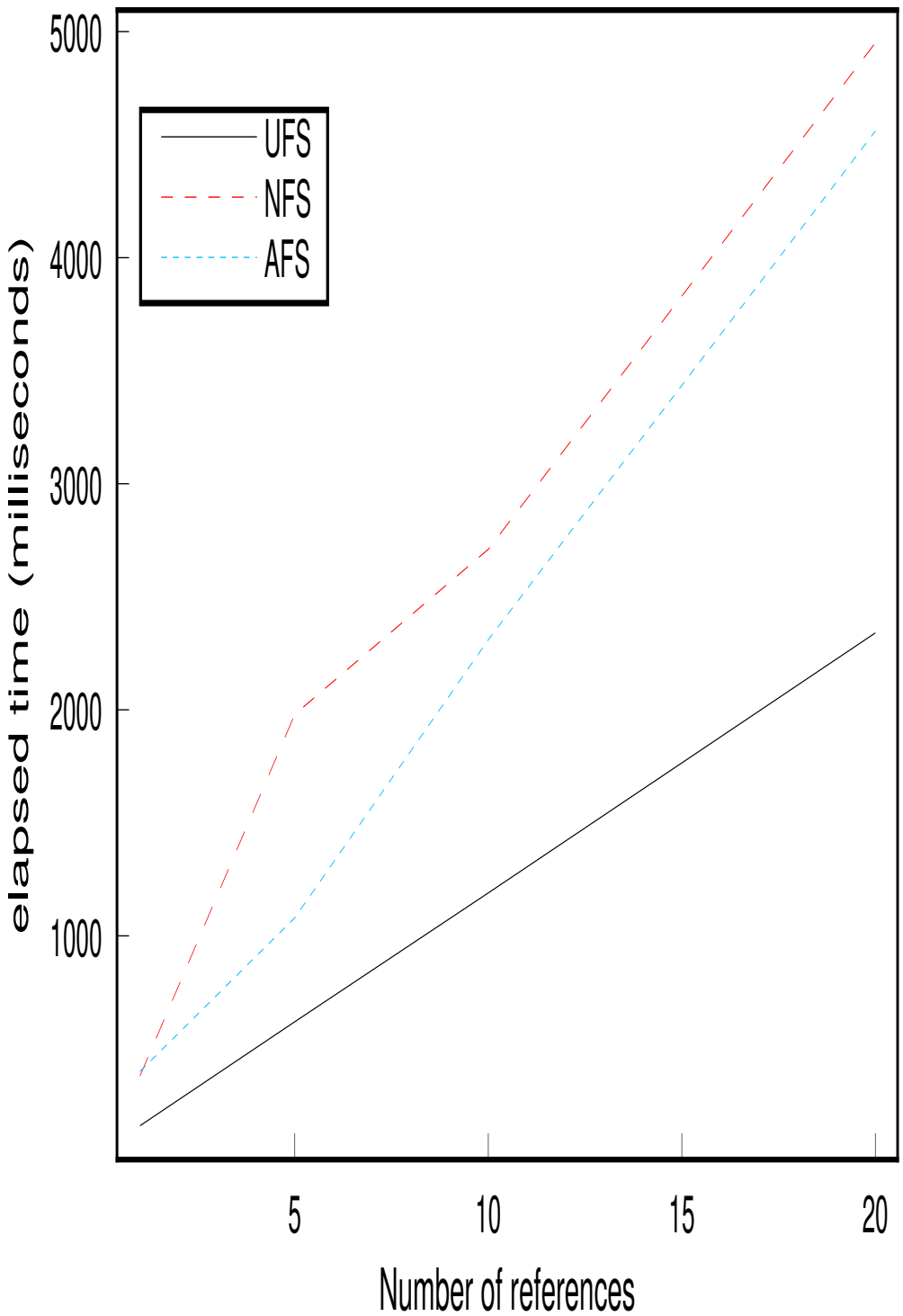


Fig. 6 Measuring the cost of the lookup operation for M = 100

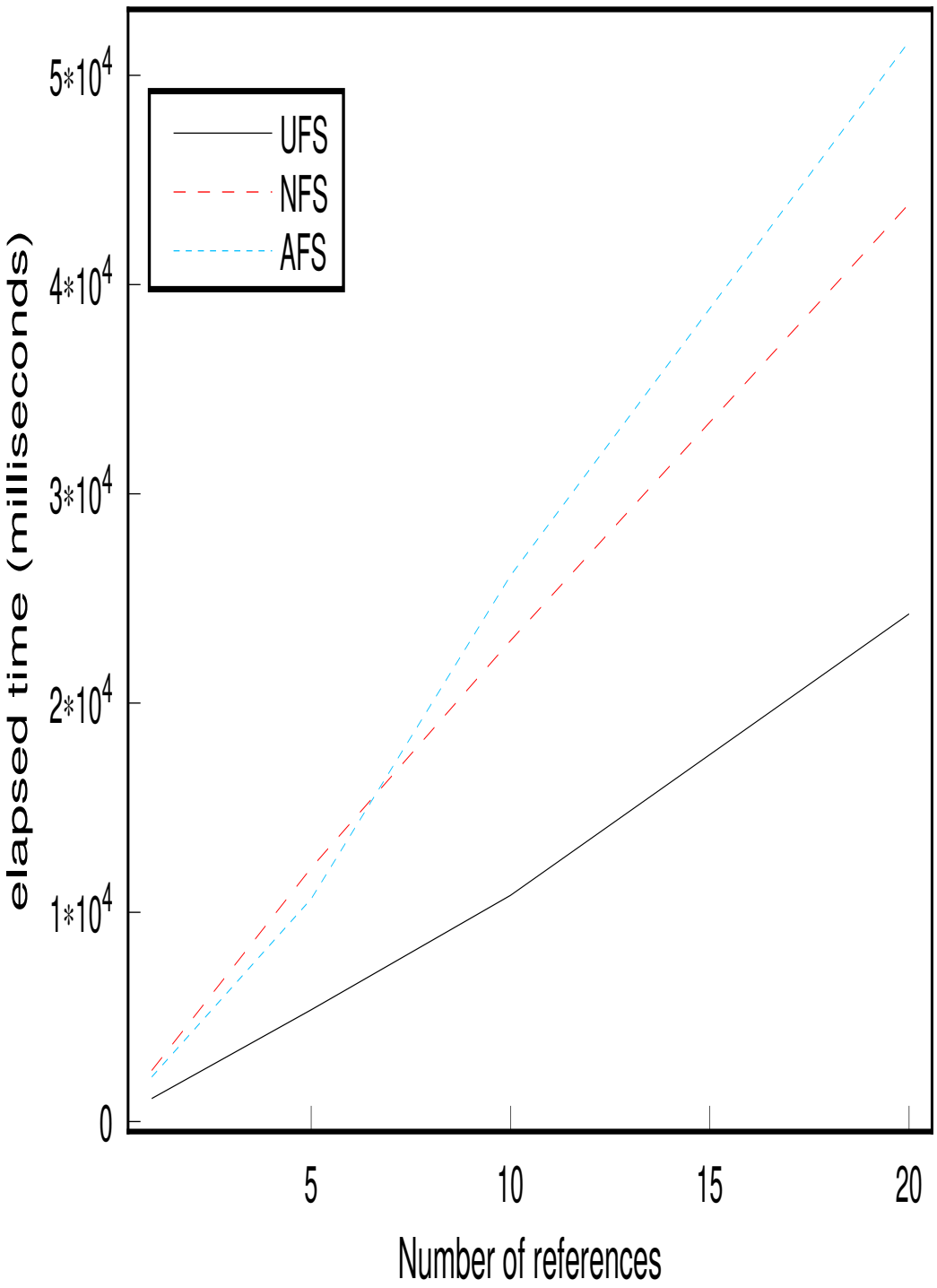


Fig. 7 Measuring the cost of the lookup operation for M = 1000

| | AFS | NFS |
|---|---|---|
| First lookup | 1s 810 ms | 17s 93 ms |
| Subsequent lookups | 30 ms | 70 ms |

Fig 9 Remote lookup performance

**Analysis**

The checkin operation seems to be cheaper when we register an object not located in the current context.. It's 10 times cheaper . When profiling the call in the local and cross−context, we found out cases, that the overhead in the local case is due to updates of information on the storage system related to the mapping of this object, which is not done in the cross−context case.

We can also notice that AFS in the local case has got the worst performance. Profiling got us with the response. In fact as we said, in a local case, Amadeus updates information on the disk about mapping. To do that, it locks the file containing this information using the C library call `flock()` . It seems that locking files under AFS takes about 200 ms. This is confirmed by the cross−context case where updates on disk doesn't happen and AFS in this case is better than NFS.

However the lookup operation is more complex, since it depends on more parameters. It can varies from tens of milliseconds (fig.5) to tens of seconds (fig.7). The cost of the lookup operation is linearly dependent on the number of references that the object contains (fig 5 , 6 and 7). The choice between AFS and NFS seems less important. It appears that the number of elements in the list is relevant to the lookup cost and varies rapidly for higher values of M. This is because bigger lists are stored in more clusters involving more mappings and bindings. However, the most important thing that the benchmark 3 highlights is the fact that AFS is for far better than NFS in the case of a remote lookup, especially for the first lookup.

From this analysis we should retain :

- Checking in objects depends only on the object location. A cross−context checking in is less expensive than a local checking in.
- The lookup operation depends mainly on the degree of complexity of the object and the depth of the list of objects pointed to by our object. The global property of the class to which the object belongs to is important (See 3.2).
- The used filesystem is relevant to the checkin operation. However, for the lookup operation, it's only relevant for remote lookups. In this later case, AFS outperforms NFS.

## 6.3 Object invocation

This operation is at the heart of any object based system and is for far the most frequent one. This is why its performance is critical. In Amadeus, there are many possible schemes to invoke objects. These schemes are related to parallelism and distribution. The parallelism is controlled by the programmer while the distribution is dependent on the location of objects and the programmer cannot predict where the invocation will take place. The parallelism is provided by means of creating jobs and activities, that handle invocations to objects in parallel. The location of an invocation's processing can be in :

- The same address space from where the invocation was issued.
- A different address space on the same node where the object is currently mapped.
- A different address space on another node in the Amadeus cell.

The first invocation must always be distinguished from subsequent invocations on a given object. Because, most of the time, it requires an extra processing, such as loading and mapping. This is when the invocation is to take place locally. For remote invocations, it necessitates the creation of a representant of the invoquing job on the remote node if it's the first time this job extends on that node.

We can sense nine possible combinations of invocation types due to the synchronisation scheme of the invocation or two the location of the invoked object.

- Local Synchronous invocation.
- Local Asynchrnous invocation by creating an activity to handle the invocation.
- Local Asynchronous invocation by creating a separate job to handle the invocation.
- Cross−context Synchronous invocation.
- Cross−context Asynchrnous invocation by creating an activity to handle the invocation.
- Cross−context Asynchrnous invocation by creating a separate job to handle the invocation.
- Remote Synchronous invocation.
- Remote Asynchrnous invocation by creating an activity to handle the invocation.
- Remote Asynchrnous invocation by creating a separate job to handle the invocation.

The figures 9 to 12 show the performance of all the schemes outlined above, now we try to explain and comment these numbers.

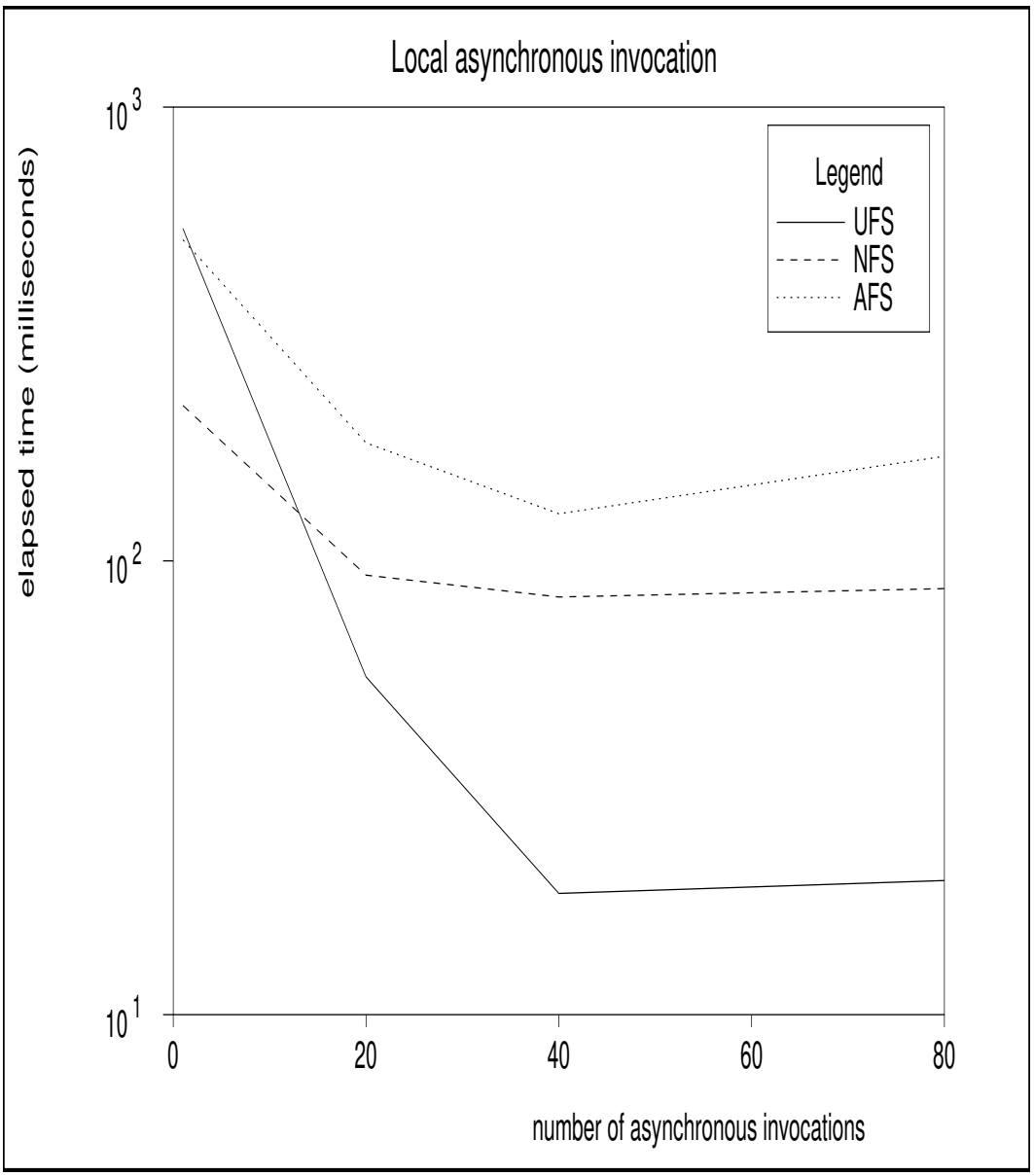


Fig. 10 Asynchronous local invocation

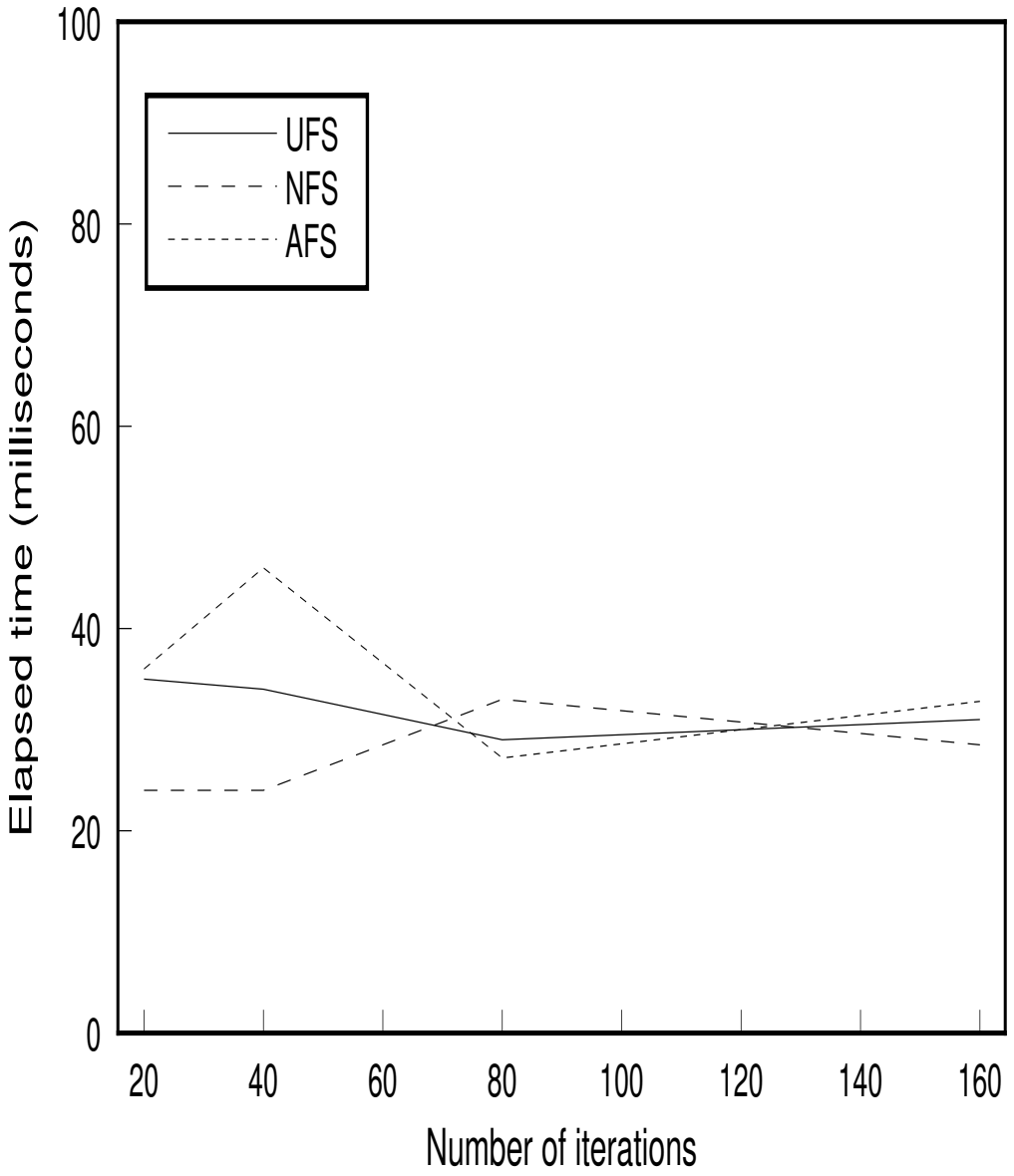


Fig. 11 Synchrnous Cross−Context Invocation

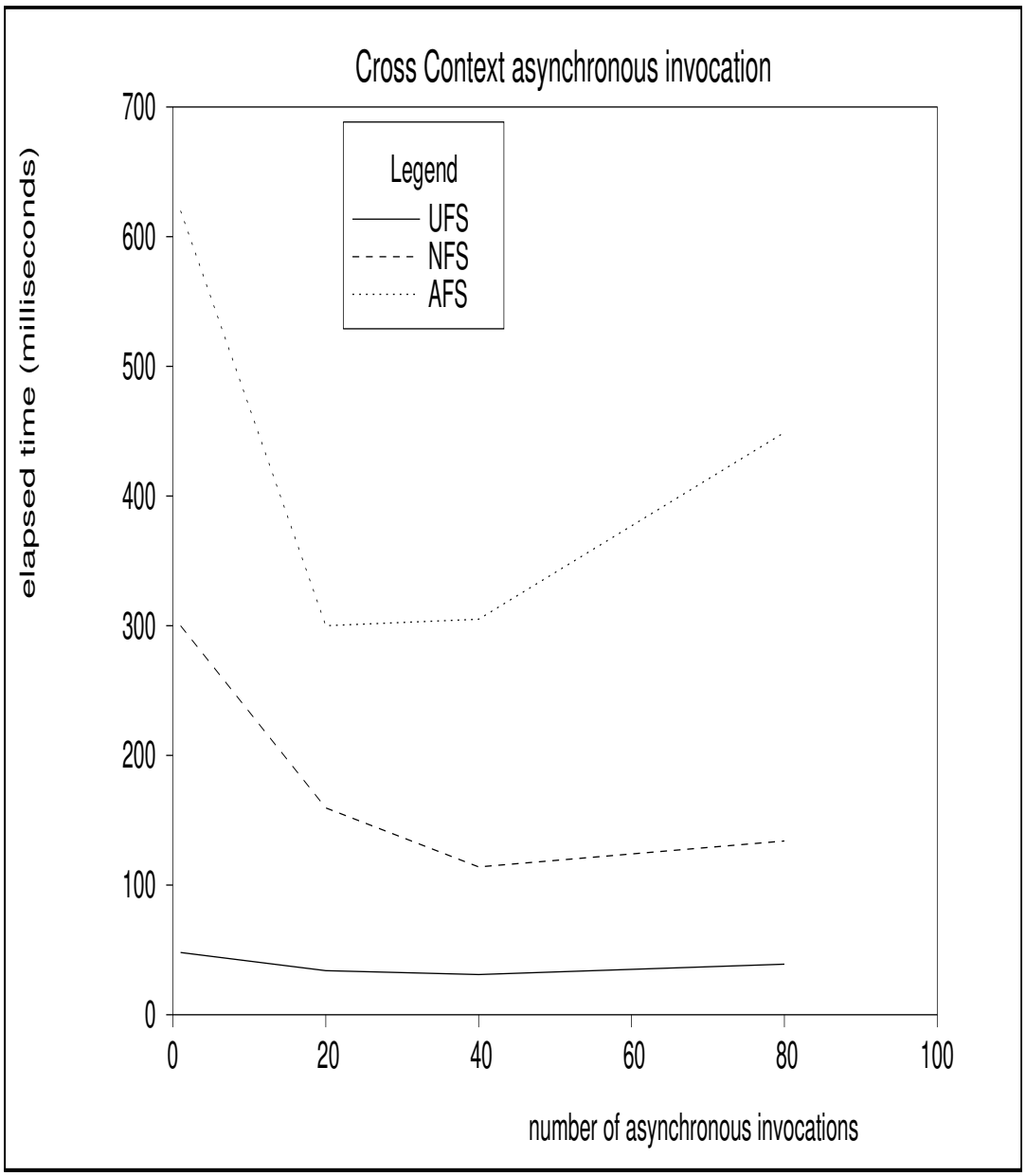


Fig. 12 Asynchronous Cross−Context Invocation

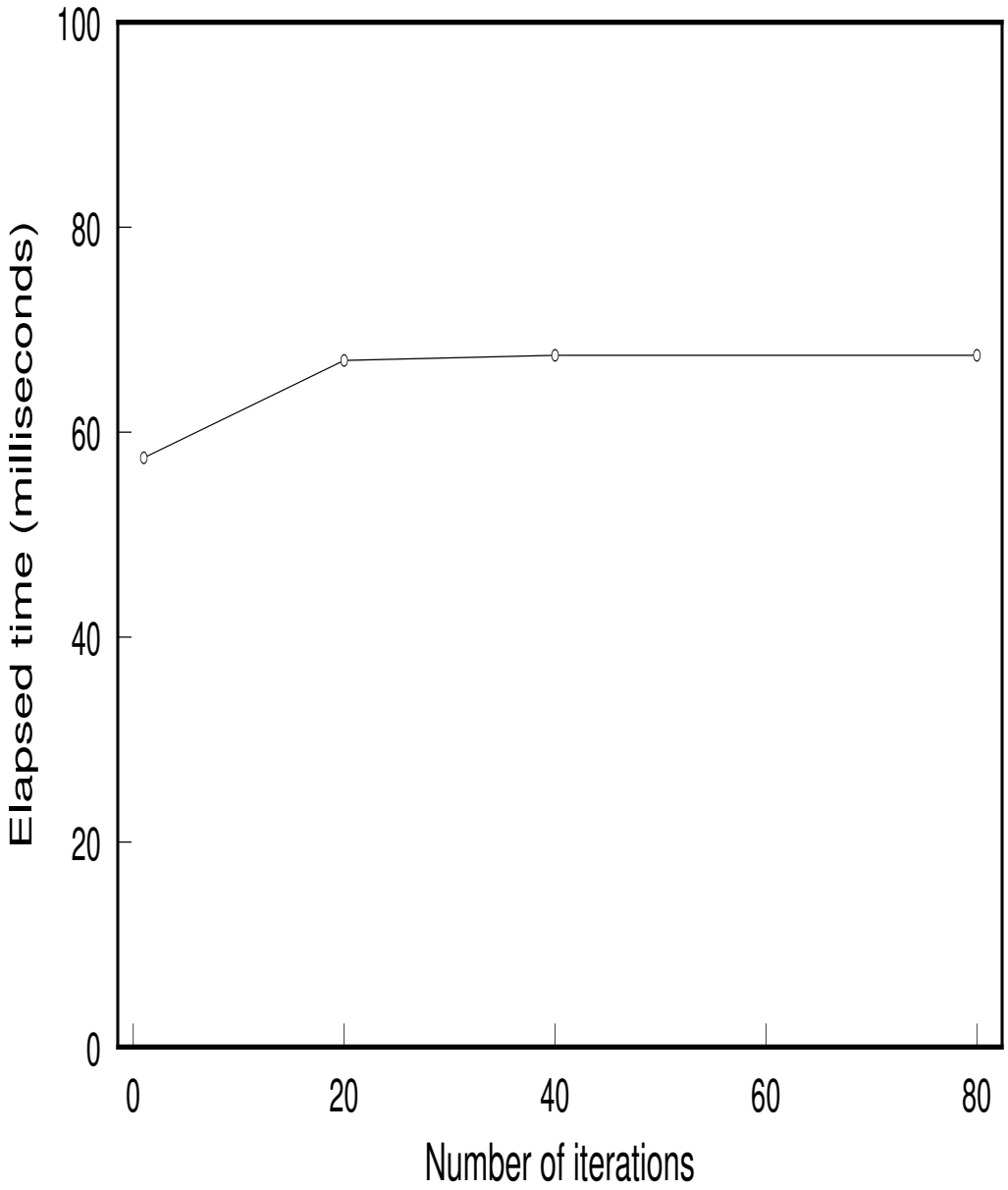


Fig. 13 Remote Synchronous invocation

### *6.3.1 Benchmark 1*

The first benchmark deals with the invocation in a local case. An object is mapped locally and then invoked either synchronously or by creating an activity that handles invocation. The cost of synchronous local invocation, is illustrated by the following table:

| Number of invocations | 12500 | 50000 | 100000 | 200000 |
|---|---|---|---|---|
| mean time | 16 microsec. | 15 microsec | 16 microsec. | 15 microsec. |

Fig.14 Local synchronous invoation timing

However the asynchronous local invocation is illustrated by fig 10, where the benchmark was run on different filesystems.

*6.3.2 Benchmark 2*

This benchmark illustrates the cost of an invocation done on an object mapped in another context on the same node. To create this situation, this benchmark when running is made up of two jobs. The first job maps the target object while the second job invokes synchronously and asynchronously that object. The results gathered are presented in fig.11 and fig.12 for different filesystems.

*6.3.3 Benchmark 3*

This last benchmark deals with remote cases. As for benchmark 2, it involves the running of two jobs. Each one on a different Amadeus node. The first invocation is ignored since it involves an extra cost and is separately measured in benchmark 4. The results for synchrnous and asynchronous cases are represented by figures 13. However, NFS is not represented due to a bug in the NFS code of OSF/1 MK, not yet fixed.

*6.3.4 Benchmark 4*

This benchmark deals with the first invocation overhead. There are many reasons for that overhead. Indeed, the invoked object may be not mapped, bound. It may also be mapped on a remote node.

First we measured the overhead induced by mapping. We distinguished two cases; The invoked object is stored in an LC managed by the local site. In the second case, the LC is managed by a remote site.

| | AFS | NFS | UFS |
|---|---|---|---|
| Managed Locally | 97 ms | 102 ms | 31 ms |
| Managed remotely | 1s 340ms | x(7) | o(8) |

(7) x : Value not available.

(8) o : UFS cannot be used in this case.

Fig.15 Mapping local and remote managed clusters

All the first invocations measured were synchrnous. Now let's see the overhead due to the binding operation. This occurs, when the invoked object is mapped, but not yet bound (registered in the system tables as active).

| Number of iterations | 100 | 1000 | 5000 | 10000 |
|---|---|---|---|---|
| Elapsed time | 200 microsec. | 247 microsec. | 279 microsec. | 234 micorsec. |

Fig.16 Binding a local mapped object performance

Last, we measured the overhead of an invocation on an object mapped on a remote node, which may lead to the diffusion of the job. The following table compares first invocations leading to diffusion with those that don't :

| | AFS | NFS |
|---|---|---|
| First invocation with diffusion | 1s 310ms | 25 s 370ms |
| First invocation without diffusion | 63 ms | 55 ms |

Fig.17 First invocations on remote objects.

**Analysis**

First, we notice that except for the local synchronous invocation, the performance of the other cases is generally bad comparing to other Comandos implementations. The asynchrnous invocation is worse than the synchronous invocation in all cases. After profiling, we deduced that most of the overhead in the asynchronous invocation, is due to filesystem operations. For example, locking a file under AFS itself, is very expensive as we noticed earlier. The bulk of remaining time is due to I/O operations. While, the cost of creating a parallel thread of control to handle the invocation (a Cthread in Amadeus/Mach) takes only about 1.2 ms.

For synchronous calls, the difference between cross−context and remote invocations is due to the use of the network in the last case (about 22 ms overhead due to the network).

The bad performance of asynchronous invocations means that using acctivities to speed up a program should be studied carefully . The filesystem operations are the main responsible of this overhead. Notice that for the first invocation, binding an object costs only 200 to 300

microseconds while mapping a cluster containing an object takes about 97ms under AFS. Thus grouping related objects into common clusters to minimize the number of mapped clusters saves many milliseconds.

# 7 Analysis of Mach3.0 impact

In its implementation under Mach3.0, Amadeus makes the use of two main Mach3.0 features :

- Mach3.0 kernel threads via the Cthread package.
- Mach3.0 shared memory.

However Amadeus, unlike other Comandos implementations on top of Mach3.0 doesn't use the following features :

- Network−wide memory manager to share info. between nodes.
- Mach IPCs for local and remote communications.
- Mach notification mechanism to detect the death of jobs or services.

Amadeus thread package was replaced by the Cthread package that use the Mach3.0 kernel threads. Hence the synchronization mechanism between threads was also replaced by the Cthread synchronization mechanism. We have then measured the cost of the creation of a Cthread wired to a Mach3.0 kernel thread.

| Number of iterations | 100 | 1000 | 5000 | 10000 |
|---|---|---|---|---|
| Elapsed time | 700 microsec. | 630 microsec. | 882 microsec. | 792 microsec. |

Fig.18 Cthread creation operation performance.

Even if we can't compare these results with the native Amadeus thread package since there is no Amadeus / Unix version running on the same hardware, we can at least compare it to process creation [(9)]which takes on top of OSF/1 MK about 30ms, which means that creating a cthread is 42 times cheaper than creating an OSF/1 process.

However, as we noticed in the asynchrnous invocations, the good perfromance of cthread creation is not reflected in the activity creation, which was mainly due to I/O operations on disk. It's necessary to get rid of the use of these I/O operations, to have the activity creation performance approximating the Cthread's one.

The Mach3.0 shared memory is used in Amadeus, to share system tables between the Amadeus server and all the contexts on the same node. Each system table is handled by one Mach3.0 memory object. When an Amadeus server is started, it creates these memory objects. Then, the subsequent clients map them into their address spaces. We are interested

(9) We mean by process creation, only the fork() operation not the exec().

to compare the performance of Mach3.0 shared memory mechanism used by Amadeus on top of Unix. We used OSF/1 MK, as a Unix reference. The comparison concerned two operations on shared memory :

- Creation of a shared memory object and a shared memory segment with the same size.
- Mapping a shared memory object or Unix segment.

The results concerning these operations are respectively represented by  table and

| Size (bytes) | Creation time(us) | Mapping time (us) |
|---|---|---|
| 1 | 2600 | 2200 |
| 50 | 2800 | 2000 |
| 100 | 2600 | 2000 |
| 500 | 2800 | 2000 |
| 1000 | 2600 | 1800 |
| 10000 | 2600 | 2000 |

Fig.19 Unix shared memory performance

| Size (bytes) | Creation time(us) | Mapping time (us) |
|---|---|---|
| 1 | 1500 | 300 |
| 50 | 2000 | 250 |
| 100 | 2000 | 250 |
| 500 | 2000 | 275 |
| 1000 | 2000 | 250 |
| 10000 | 2000 | 250 |

Fig.20 Mach shared memory performance

We notice that in both operations, the Mach shared memory feature is better than the Unix one. The use of Mach shared memory in Amadeus, improved the creation by about 40 % and the mapping by 700 %. The mapping improvement is very important, since the mapping is a more frequent operation.

As we said earlier, Amadeus in its implementation on top of Mach3.0, still use the Unix sockets mechanism for local and network communications. In order to have an idea how much would the improvement be (if there is one) if the Unix socket's mechanism was replaced by the Mach IPC mechanism. Independently from Amadeus, we compared the Unix socket mechanism and Mach IPCs in local and network communications. For network communications we used UDP sockets for Unix and the "Network Message server" for Mach. In both cases (local and network), we measure the message RTT (Round Trip Time).

| Size (bytes) | Local (us) | Network (us) |
|---|---|---|
| 4 | 7470 | 9540 |
| 256 | 8380 | 13990 |
| 1024 | 9700 | 20430 |

FIg.21 Unix sockets performance

| Size (bytes) | Local (us) | Network (us) |
|---|---|---|
| 4 | 550 | 198000 |
| 256 | 950 | 196800 |
| 1024 | 2200 | 198000 |

Fig.22 Mach IPC performance

We notice that the use of Mach IPC for local communications is 13 times cheaper than the Unix sockets. However for network communication. Mach3.0 combined to the NetMsgServer perform very bad. This is because the message is sent first to the NetMsgServer which do the necessary processing, then uses TCP protocol to convey the message over the network to the remote NetMsgServer. However, Mach NORMA(10) IPC, seems to have a more better performance. The RTT on 486 machines is about 3 ms.

However, having performant IPCs does not mean that replacing the use of Amadeus sockets by Mach IPCs improves the cross−context and remote invocation by the same ratio. Currently only 25 % of the cross−context invocation cost is related to sockets. The remaining 75 % are Amadeus stuff which needs improvement to take more benefit from the good performance of Mach IPCs.

---

(10) The version of the microkernel used in Multi−computer and cluster projects.

(11) Recall that a stub is a large object reference, which allows to locate the object and also the contsruct a proxy for the object.

# 8 Conclusions

This report has presented the performance study of Amadeus on top of OSF/1 MK. The performance of different object operations exported by the Amadeus runtime, has been studied and the benchmarks used have been described. The report has also exposed the performance of the Mach3.0 features involved in the implementation of Amadeus on top of Mach3.0 and compared them to the equivalent Unix features.

This study allowed to understand the parameters influencing each operation and the system components involved in each operation. For example, at the start of this study the filesystem used seemed not to be relevant to the performance. However, it seems that it has a major role in the cost of some operations. This study also allowed to identify the critical components, and to figure out a hierarchy of priorities among components that can benefit from improvements added to the system.

Mach3.0 support features have also been assessed together with the way they are used in Amadeus. We noticed that some bottlenecks (e.g activity creation code), prevents Amadeus to take advantage of the use of underlying performant Mach3.0 features (eg. Cthreads). We also measured the impact of extending Amadeus to use more Mach3.0 features (ie. Mach IPC) and what would be the impact on the overall performance.

Besides identifying bottlenecks, we have led to some useful rules that may be used by a programmer under Amadeus, to enhance the performance of its applications.

- Grouping related objects in the same cluster.
- Creating clusters containing related objects in the same logical container.
- Objects belonging to global classes are loaded only on demand, which is not the case of non global objects.
- Start the application on the node that manages its objects or most of its objects. In fact, the Amadeus user has not to know where its objects are stored and who managed what. But if he knows, it might help.
- Programming while performance in mind. We mean that organising the data in objects and the definition of relations between those objects is not a matter of modularity and reusability but also of performance. We saw, for example that the complexity of an object influences the cost of operations performed on it.
- Modularity and performance are often two opposite requirements. The programmer should make its trade−offs to let the user satisfied by the two requirements.

The creation of activities, which is intended to provide parallelism and speedup to the program is currentlly not accomplishing its functionality due to its high cost. Involving I/O operations on files during the activity creation seems not to be a good idea, and its design has to be reviewed.

Other operations such as local synchronous invocation or binding operation seem to have a good performance, and using the clustering technique to group objects together, has a good global effect.

We hope that this study highlighted the major bottlenecks and efficiency problems in Amadeus on OSF/1 MK. The results are valid for other implementations of Amadeus as well. The main conclusion that we can draw from this study is that porting Amadeus on top of Mach is in some aspects not sufficient to provide Amadeus with the performance intended, since most of the architecture was specially built for use on top of Unix features. Thus, taking advantage from microkernels, needs in most case a redesign of part or the whole target system to fit and use extensively the microkernel abstractions. This was the approach used by Guide and IK, other Comandos implementations, and the first performance results seem very encouraging.

# Annex A

# Index

**Context**

A context as a contiguous address space on one machine. Amadeus jobs are made up of many contexts. However, a job is represented only by one context on a node.

**Cluster**

In Amadeus, the objects are grouped into *clusters*. Each cluster is represented by a file. The clusters have always a fixed size of 8KB, even if they contain only one object of one byte.

**Mapping**

We mean by mapping an object, the action that consists to read the object from the backing store into the address space of the mapping context.

**Binding**

After an object has been mapped, it's not in a usable form yet. The action to make an object usable or active is called the *binding*. Hence, an object can be mapped but not yet bound.

**Simple and Complex Objects**

We mean by a simple object, an object which has no pointers to other objects. It's a leaf object in the store hierarchy. Unlike this is a complex object, in which references to one or many other objects are made. We talk about a degree of complexity for designing the number of references an object holds on other objects.

Bibliography

[1]

[2] , , , ( ), pp. , .

# Performance analysis of Amadeus on top of OSF/1 MK


**1 Introduction** . . . 1

**2 Amadeus Overview** . . . 2

2.1 Computational model . . . 3

2.1.1 Jobs . . . 3

2.1.2 Activities . . . 3

2.1.3 Objects . . . 4

2.2 Amadeus Architecture . . . 4

2.2.1 Storage system . . . 5

2.2.2 Generic runtime . . . 5

2.2.3 Remote location service . . . 6

2.2.4 Activity Manager . . . 6

2.2.5 The Communication system . . . 6

**3 Some C** concepts** . . . 6

3.1 Persistence support . . . 7

3.2 Distribution support . . . 7

**4 Methodologies of measuring** . . . 8

**5 Measurement precautions** . . . 9

**6 Measuring Interface primitives** . . . 10

6.1 Object creation . . . 10

6.2 Name service usage . . . 14

6.2.1 Benchmark 1 . . . 16

6.2.2 Benchmark 2 . . . 17

6.2.3 Benchmark 3 . . . 18

6.3 Object invocation . . . 22

6.3.1 Benchmark 1 . . . 26

6.3.2 Benchmark 2 . . . 27

6.3.3 Benchmark 3 . . . 27

6.3.4 Benchmark 4 . . . 27

**7 Analysis of Mach3.0 impact** . . . 29

**8 Conclusions** . . . 32

**Annex A Index**