Stardust: an Environment for Parallel Programming on Networks of Heterogeneous Workstations

Gilbert Cabillic

IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cédex, FRANCE

Isabelle Puaut

INSA/IRISA, Campus Universitaire de Beaulieu,

35042 Rennes Cédex, FRANCE

Proposed abbreviated title

Stardust: parallel programming on heterogeneous hosts

Abstract

This paper describes Stardust, an environment for parallel programming on networks of heterogeneous machines. Stardust runs on distributed memory multicomputers and networks of workstations. Applications using Stardust can communicate both through message-passing and distributed shared memory. Stardust includes a mechanism for application reconfiguration. This mechanism is used for balancing the load of the machines hosting the application, as well as for tolerating machine restarts (anticipated or not). At reconfiguration time, application processes can migrate between heterogeneous machines, and the number of application processes can vary (increase or decrease) depending on the available resources. Stardust is currently implemented on an heterogeneous system including an Intel Paragon running Mach/OSF1 and a set of Pentiums running Chorus/classiX. The paper details the design and implementation of Stardust, as well as its performance.

Contact author

Isabelle Puaut
IRISA, Campus Universitaire de Beaulieu
35042 RENNES Cédex, FRANCE

Phone: (+33) 99 84 73 10
FAX: (+33) 99 84 71 71
e-mail: puaut@irisa.fr

1 Introduction

The proliferation of inexpensive and powerful workstations has continued to increase at a rapid rate in the last few years. This increase of machine performance is likely to continue for several years, with faster processors and multiprocessor machines. Studies have shown that for a large percentage of their lifetime, the machines are used for small tasks (reading *e-mail*, editing files), thus demonstrating an average idle percentage of at least 90% even during peak hours [4]. One possible use of these idle cycles is to run parallel applications on networks of workstations. The network of workstations can then be considered as a parallel computer, or *hypercomputer* whose performance is similar to the one of a parallel machine for a much lower cost.

Numerous research activities have tried to exploit the computing power of networks of workstations, like PVM [10], based on the message-passing paradigm, and Mermaid [24], based on the shared-memory paradigm. To be fully usable, an environment for parallel computing on networks of workstations should include the following facilities:

- *Support for multiple programming paradigms:* two classes of programming paradigms for parallel applications can be distinguished: the *message-passing* programming paradigm, in which processes communicate through message exchanges, and the *shared-memory* paradigm, in which processes communicate by reading and writing shared data items. Although all applications can be written using both paradigms, some studies have shown that application performance can be increased by the use of both message-passing and shared memory [12].

- *Support for heterogeneity:* existing computing environments often include a number of computers with various architectures and performance (parallel machines, workstations) and different operating systems; more computing power is then available to environments supporting heterogeneous machines.

- *Support for load balancing and application reconfiguration:* the concept of ownership is frequently present when using workstations for executing parallel applications. Workstation owners do not want their machine to be overloaded by the execution of parallel applications, or simply want exclusive access to their machine when they are working. Reconfiguration mechanisms are thus required to balance the machines loads, and to allow parallel computations to coexist with other applications.

- *Support for fault tolerance:* as the number of nodes achieving a parallel computation increases, possibility of a node failure also increases. The failure can be a hardware failure or may be a reboot done by the owner of the workstation. Without provision for tolerance to such failures, it is necessary to restart the entire computation.

Stardust is an environment that provides all these facilities. Processes executing on Stardust can communicate both through message-passing and (page-based) distributed shared memory (DSM). Library calls are provided for sending/receiving a message, as well as for creating a shared memory region and mapping it onto a process address space. Stardust library calls are currently available for application programs written in C. The base DSM of Stardust is very close to the fixed distributed page-based DSM of K. Li [13]. The consistency model used is sequential consistency, and is implemented by a write-invalidate protocol (a page may be replicated only if it is protected against writes; when a process attempts to write on a protected page, the privilege violation is detected by the memory management hardware and all the page replicas are invalidated before permitting the write access). Information about pages is maintained using K. Li's fixed distributed scheme [13]. Every host is given a predetermined subset of the pages to manage. A fixed distribution function is applied to obtain the manager of a page, which keeps the current access privileges of the page as well as the list of nodes having a copy.

Stardust executes on an heterogeneous computing environment composed of a parallel machine (the Intel Paragon) and PCs connected by a 10 Mb/s Ethernet network. Programs developed with Stardust follow the *Single Program Multiple Data* (SPMD) programming paradigm. All processes execute the same code, and each process can retrieve its identification (process number within the application) using a call to the Stardust execution library. Stardust includes transparent mechanisms for converting data between different types of hosts. These mechanisms are used for converting both the contents of communication buffers and the contents of shared virtual memory regions. Moreover, in order to cope with the processors different instruction formats, each program developed with Stardust is compiled for every type of architecture.

In addition to its support for heterogeneity, Stardust includes a reconfiguration mechanism used both for checkpointing and for load balancing. The basic principle of this mechanism is to consider an application state only when all the application processes have reached a synchronization point (synchronization barrier) in their main program (C function *main*). The application state can then be saved onto disk to support node failures, or moved to other nodes for balancing the system load. An architecture-independent standard representation of data is used for saving an application state. No architecture-dependent data (e.g., threads stacks) is saved onto disk (respectively transferred to other nodes). Hence, migration of applications between heterogeneous hosts is made possible. Moreover, the application state saved onto disk does not contain any information that depends on the number of application processes. Thus, the number of processes of an application can change (increase or decrease) at application reconfiguration time.

The remainder of the paper is organized as follows. Section 2 first presents Stardust support for heterogeneity. Section 3 then presents the technique used for checkpointing and load balancing. Section 4 gives the code of an application developed with Stardust. Section 5 details the implementation of Stardust and gives performance data. Stardust is compared with related work in section 6. Finally, we conclude in section 7.

2 Management of Heterogeneity

This section deals with the management of heterogeneity in Stardust. At first, the different sources of heterogeneity in distributed systems are briefly enumerated. The techniques implemented in Stardust for managing heterogeneity are then described: paragraph 2.2 details the base mechanisms used for data conversion and data typing; paragraph 2.3 then describes higher level mechanisms used for sharing virtual memory regions across hosts of distinct types. A summary of the benefits and limitations of the management of heterogeneity in Stardust concludes this section.

2.1 Issues Related to Heterogeneity

The different sources of heterogeneity in distributed systems (hardware, operating system and programming languages) as well as the resulting issues to be considered, are discussed below.

2.1.1 Physical representation of data

Data items may be represented differently on various types of machines due to differences in machine architectures, programming languages for the applications and compilers. Even for basic data types such as integers, the size in bytes, as well as the order of the bytes in memory can be different. For instance, integers on a Pentium processor can be represented by 16 or 32 bits, while on a 64-bit Sparc V9 processor they can be represented by 32 or 64 bits. For floating point values, the length of the mantissa and exponent fields, as well as their position can differ from one architecture to another. For higher level data structures such as records, the alignment and order of the components in the data structure can differ between hosts due to the differences in their architectures and programming language compilers. In addition, processors can have different instruction sets and different binary representations.

Sharing data between heterogeneous hosts requires that the physical representation of data is converted when data is transferred between hosts of different types. Note that data conversion not only incurs run-time overhead, but also may result in information loss while converting data between architectures with different sizes of data representations (e.g. conversion of a 64-bit integer into a 32 bits integer).

2.1.2 Page size

The unit of data managed and transferred by a page-based DSM is the virtual memory page. In a heterogeneous DSM system, the hosts may have different sizes of virtual memory pages. In addition, due to the different data representations, a virtual memory page on a given architecture may include a number of data items (e.g. integers) different from the other architectures. In order to extend a page-based DSM to an heterogeneous environment a new unit of data management and data transfer between architectures must be chosen.

2.1.3 Operating system

Operating systems running on the different hosts need not be the same. They can include different support for virtual memory management. For instance, micro-kernel based operating systems include a facility for writing user-level code for virtual memory management, while monolithic kernels do not provide such a facility.

DSM systems usually provide a thread system that allows multiple threads to share the same address space. The facilities for thread management (thread creation, scheduling, synchronization) may be different on various types of hosts. In addition, migrating a thread from one host to another in a homogeneous DSM system is usually easy, since minimal context is kept for the threads, and since the same data representation is used for the threads context. In heterogeneous systems, thread migration is much more difficult. The binary images of the programs are different, so it is difficult to identify equivalent points of executions in the binaries (the hosts have different instruction representations). Similarly, the format of the threads' stacks are likely to be different due to hardware, programming language and compiler differences. Therefore, converting thread stacks at migration time may be hard and time consuming.

The various operating systems may implement different communication protocols. The realization of a programming environment on an heterogeneous architecture requires the existence of a common communication protocol between the different types of hosts involved in the computation. The availability of standard protocols like TCP/IP on most systems makes inter-host communication increasingly feasible. However, one must be careful about the communication protocol used for communication between hosts of the same type. Communication protocols specific to a given hardware or operating system (e.g. the NX communication library on the Intel Paragon) can be much more efficient than standard communication protocols like TCP/IP.

2.2 Mechanisms for data conversion and data typing

Two approaches can be taken for converting data representations between different types of hosts. The former one consists in sending messages in a standard architecture-independent format. The latter approach consists in sending the message in the sender's physical format. The second approach requires a single conversion of the message contents, while the first one requires two data conversions: one from the sender's format to the standard format, and another one from the standard format to the receiver's format. Nevertheless, the first approach is implemented in Stardust because a new type of architecture can be integrated more easily: only routines for converting data between the standard data representation and the one of the new type of architecture have to be developed. The standard data format used for data transfer is SUN eXternal Data Representation (XDR) format

[21]; it was chosen because of its availability on a wide range of architectures and operating systems. The XDR library, linked with the Stardust environment, provides routines for converting basic data types (chars, integers, floating point values) to and from the XDR format.

In order to apply the XDR conversion routines when a piece of data is transferred between hosts of different types, it must be possible to have the type of each data structure. One approach consists in analyzing the source language (for instance by modifying the C compiler). Another approach, taken in [24] is to exploit information generated by compilers in object files, and intended to be used by symbolic debuggers (Symbol TABle, or *stab*). The *stab* entries include information about the type, size and alignment of data structures. With such an approach, the user only has to associate a string to each data type, so the system is able to locate information about the corresponding data structure in the object file. In Stardust, we did not want to modify the compiler or make assumptions on the kind of run-time information it generates in object files. Consequently, the type of every data structure is given explicitly by the application programmer when the data structure is allocated (creation of a shared virtual memory region, allocation of a communication buffer). This is done by using a simple language, taken from [17] which is analyzed when the data structure is transferred between hosts of different types. A type is of the form *(TypeString,* N_1*,* N_2*, ... ,* N_n*)*, where *TypeString* is the string identifying the data type (see syntax below), and N_i is the number of elements of the ith nested subtype of string *TypeString*. For example, the type *("{C{D}}",10,20)* corresponds to an array of 10 structures, each structure being composed of a character and 20 double precision floating point values.

Type	:	'{' (BasicType | Type)* '}'	
BasicType	:	'C' |	(char)
		'I' |	(short integer)
		'L' |	(long integer)
		'F' |	(single precision float)
		'D'	(double precision float)

Let us call *base element* the first nested subtype of a data structure (in the above example, the base element is the structure composed of one character and 20 floating point values). Note that due to the method used to associate a type to a data structure, the conversion algorithm can only apply to a multiple number of base elements. Note also that the base element can have a different size on distinct hosts, since the sizes of the basic data types can differ from one host to another.

The data conversion routines are called by Stardust each time data (communication buffer or virtual memory page) is transferred between nodes of different types. Calling the conversion routines when processes communicate through message-passing is straightforward. Additional information concerning communication through DSM is given below.

2.3 Mechanisms for sharing virtual memory regions across heterogeneous architectures

Issues other than data conversion have to be dealt with for processes to communicate through DSM (see [24] for an overview of these issues). Mainly two issues have to be addressed. Firstly, different hosts can have distinct page sizes. Secondly, depending on the architecture, the number of base elements in a page can vary, because of the differences in the physical representation of data.

Let us consider for example an application running on an Intel Paragon and on a Dec Alpha-based architecture. Assume that the application processes are sharing a vector whose type is *("{LC}",8192)*. Due to the characteristics of the two architectures and due to the compiler alignment constraints, the size of the base element on the Paragon is 6 bytes (4 for the long integer, 1 for the character and an extra-byte for the alignment constraints), while its size is 10 bytes on the DEC alpha, for which long integers take 8 bytes. Even if the page sizes on the two architectures are identical (8 Kbytes), they do not contain the same number of base elements. On this example, they do not even contain an integral number of base elements. To the best of our knowledge, no heterogeneous DSM fully addresses all issues arising from heterogeneity. In [24], it is assumed that all basic data types have the same size on all architectures; the problem of a base element crossing a page boundary is solved by adding an additional alignment constraint so that this problem does not occur.

In Stardust, the issues of different data representations and page sizes are addressed by choosing a common unit for data conversion and data transfer between different types of hosts, called *heterogeneous page*. Details concerning the management of heterogeneous pages are given in the following paragraph.

Fragmentation of regions into heterogeneous pages

Data conversion and data transfers are done in Stardust using the granularity of an *heterogeneous page*. The size of an heterogeneous page is a multiple of the size of a base element, so that the same data item is never on the main memory of the two hosts. In addition, the size of an heterogeneous page is a multiple of the page size of every architecture, so that an action on an heterogeneous page can always be mapped on a set of actions on pages of every architecture. In the above example, an heterogeneous page contains 4096 base elements, and corresponds to 5 Paragon pages and 3 Dec pages. Note that the decomposition of a shared region into heterogeneous pages not only depends on the page size of all the machines hosting the application; it is also dependent on the type of data structure contained in the region. If the region taken as example above had contained characters instead of a more complex data structure, the size of the heterogeneous page would have been different (a single virtual memory page for both architectures).

A region is fragmented into heterogeneous pages at region creation time. At this time, all the types of architectures on which the application is running are known, as well as the type of the data structure contained into the region. The fragmentation algorithm finds out, for every architecture, a page size for which no base element crosses a page boundary, and then takes the least common multiple (LCM) of theses values, thus obtaining the size of the heterogeneous page for the region.

Consistency protocol

Consistency of shared data is managed by a 2-level system (see figure 1). The *intra-architecture* level manages data consistency for a subset of homogeneous hosts. It uses a *Homogeneous Memory Manager* (HoMM) per node (on the figure, each of the three nodes a_1, b_1 and b_2 hosts a HoMM). HoMMs manage data consistency within the associated architecture. They use the architecture's virtual memory page as a unit of data transfer between machines. Pages are transferred using the most efficient communication protocol that exists on the architecture. In addition, pages are transferred in the architecture's physical data representation, without conversion into an architecture-independent format.

In the *inter-architecture* level, data consistency is managed by an *Heterogeneous Memory Manager* (HeMM) per group of nodes of the same type (2 HeMMs are depicted on the figure, one for each type of architecture). The unit of data transfer between HeMMs for a given region is the heterogeneous page that corresponds to the region's type. Transfers are achieved via a common communication protocol (TCP) and data is transferred in the XDR architecture-independent format.

An heterogeneous page can be in READ-WRITE mode on a single group of machines of the same type at a time. When an heterogeneous page is not present on a type of architecture (respectively if the process does not have enough access privilege on the page), this is noted in the HoMM's data structures. The page fault (respectively access privilege violation) is then sent to the HeMM of the architecture. Figure 1 shows the resolution of a page fault on an architecture of type A on a region whose heterogeneous page size is exactly A's page size and twice B's page size. Message exchanges are represented in the figure by arrows. In the figure, since the HoMM of machine a_1 is not the current owner of the page, the page fault message is then redirected to the HeMM of architecture A (step 1). A's HeMM then forwards the message to the HeMM of architecture B, since the page is owned by processes running this architecture (step 2). B's HeMM requests the page to the HoMMs of machines b_1 and b_2, which each own half of the heterogeneous page's contents (steps 3 and 4). The heterogeneous page is then transferred to its requester (machine a_1) through B's HeMM, A's HeMM and a_1's HoMM (steps 5, 6, 7 and 8).

Note that with such a structure of heterogeneous DSM there is no time overhead due to the management of heterogeneity when an application is running on hosts of the same type.

2.4 Benefits and limitations of heterogeneity management

Although we found Stardust sufficient for many practical applications, our solution has a number of limitations. We discuss them below.

Some transparency is lost by requiring the programmer to specify the type of data being allocated. This small annoyance only comes from our desire not to modify the compiler used for writing applications and not to make any assumption about the exact format of the object code generated by the compiler. In addition, in order to be used for a wide range of applications, the syntax of types given in § 2.2 could be extended to support pointers and recursive types.

Basic data items can lose precision when being converted to and from the XDR standard format. This problem is already present in all RPC systems using an architecture-independent format. It is especially dangerous in DSM systems for floating point values, since the application programmer does not have direct control about how many times a page has migrated and hence has been converted [24]. While one may object about the accuracy of results when a page containing floating point values has a high migration rate, we do not consider this is a problem for practical applications. Too high a page migration rate is a direct consequence of false sharing, which must absolutely be avoided in DSM applications to obtain acceptable performance.

In our approach, entire heterogeneous pages are converted even though only a small portion of these pages are accessed. However, the cost of a page conversion has proved to be small compared to the overall transfer cost (see section 3.2.1). Again, applications that access only a few data items of a page between page migrations will perform poorly both in homogeneous and heterogeneous page-based DSMs because of false sharing.

The granularity of transfer in Stardust DSM (heterogeneous page) may be larger than every architecture page size. This may increase false sharing on regions that contain complex data types. In order to overcome this limitation, the user can obtain from Stardust the size in bytes of the heterogeneous page for each region. Hence, the application programmer can reorganize the data structures contained in the shared regions so as to limit the size of heterogeneous pages. This can be done with the same techniques used for reducing false sharing in standard (homogeneous) DSM systems (e.g. data reorganization within shared virtual memory regions).

3 Support for Checkpointing and Load Balancing

In addition to its support for heterogeneity, Stardust includes a mechanism for capturing an application state. This mechanism is used for checkpointing the state of applications onto disk to support node crashes, as well as to move the state of applications from one set of machines to another to balance the system load. The following paragraphs present the basic principle of the mechanism used for capturing the state of applications, and then detail its use for checkpointing and load balancing.

3.1 A common mechanism for load balancing and checkpointing

To be widely useful, environments for parallel programming on networks of heterogeneous workstations should support machine crashes and reboots. In addition, as workstation owners generally do not want their machine to be overloaded by parallel jobs, a mechanism for balancing the system load is also required. Both facilities require a way to capture the state of applications (union of state of all application processes). To support machine crashes, the captured state must be saved onto a crash-proof storage device. To balance the system load, the application state has to be moved from overloaded nodes to underloaded ones.

Capturing the state of an application composed of multiple processes communicating through message-passing or DSM raises a number of issues. First, it must be ensured that the states of all application processes are mutually consistent [6]. This can be achieved by using two classes of techniques: *consistent* and *independent* capture techniques. The first class of techniques consists in synchronizing all processes so that the application global state is consistent. These techniques require the capture of a single state for each application. Independent capture techniques capture the state of each process without synchronization with the others. Several states per process are kept. When the application state is needed (e.g. for restarting an application after a failure) a set of mutually consistent process states has to be found. This requires the checkpointing mechanism to keep track of all interprocess communications.

Our mechanism is based on a coordinated capture technique. It takes advantage of the behavior of many parallel applications in which processes regularly synchronize through barriers, thus obtaining a consistent state. By ensuring that the state of an application is always captured within barriers, we use these *natural* consistent states and hence avoid the overhead of computing new ones. In addition, the selected technique does not need to log messages exchanged between processes. This means that time overhead is incurred only while capturing the state of an application (i.e. while migrating an application or when saving its state onto disk), and not during normal operation. Finally, compared to independent capture techniques, only a single application state has to be saved.

The other issues to be dealt with while saving the state of a parallel application are related to the management of heterogeneity. Our objective is to be able to migrate an application from one type of architecture to another (*heterogeneous* migration). To migrate processes between heterogeneous hosts, each component of the process state must be saved in an architecture-independent format:

- process data (global data plus process stack),
- point of execution in the process binary.

To transform data between heterogeneous hosts, the type of each data item must be known. Conversion of the processes' stacks at migration time is too time-consuming in our opinion. Consequently, we deliberately have chosen not to capture the processes' stacks: the state of an application can be captured only in synchronization barriers of the application main program (C function *main*). This restricts the number of places when an application can be stopped/migrated. Nevertheless, it avoids compiler modifications to keep track of stack frame internal structure, and does not require time-consuming operations when capturing the state of applications. The address space of every application process in Stardust is divided into two parts: a shared part, which contains shared virtual memory regions, and a private part, which contains the process private data (code, heap and stack). For the sake of simplicity, only the contents of the shared virtual memory regions is saved. This way, only shared virtual memory regions have to be typed by the programmer. The only restriction for application programmers is to init non-shared variables (e.g. loop variables) when an application is restarted/migrated (see section 4 for an example).

As the private data of the application processes is not saved, the saved state does not include any information which depends on the number of machines on which the application is running. Thus, at reconfiguration time, the number of machines used to execute the application can be changed (be extended or restricted). This allows an efficient use of all available resources.

Due to the choices made for capturing the state of an application, obtaining an architecture-independent point of execution in the processes binaries is straightforward. It simply consists in assigning a label, defining the program counter of the application, to every synchronization barrier of the main program.

3.2 Checkpointing mechanism

The checkpointing mechanism included in Stardust relies on the capture mechanism given in the previous paragraph. The state of an application is stored onto *stable* (crash-proof) storage. This is done by a distinguished process, called *Capture Server* (CS). There is at any time a single *permanent* consistent checkpoint, which is identified by an increasing *Consistent Checkpoint Number* (CCN). Several implementations strategies have been evaluated for implementing the capture server. These preliminary experiments, conducted on a 56 nodes Intel Paragon, and fully described in [3] are given in the following paragraphs.

3.2.1 Preliminary experiments

Three implementation alternatives of the capture server (*basic*, *incremental* and *non-blocking*) have been compared. For all of them, the capture server stores checkpoints in files. There is also an *identification file* per node, which stores the current value of CCN, and is used to identify the permanent checkpoint. It thus permits to discard a checkpoint under construction (if any). Identification files are written atomically by the capture server. A checkpoint under construction becomes permanent if and only if the identification files of all nodes have been written to disk.

Basic checkpointing algorithm

The computation of a consistent checkpoint is performed by every process within a synchronization barrier. During the checkpointing protocol, each process saves its participation to a *tentative* checkpoint. The checkpointing protocol proceeds as follows:

- Application processes synchronize with each other at a synchronization barrier;
- Each process increments its local value of CCN, and then requests the saving of its part of the tentative checkpoint to the capture server. The data saved by a process P contains the pages currently owned by P (even if they have not been modified since the last checkpoint). The application process is blocked until the capture server has written data to disk.
- Each process requests the writing of its identification file by sending a message containing the current value of CCN to the capture server.
- Processes synchronize again to avoid any change to DSM pages before the end of the checkpointing protocol.

Optimizing the basic checkpoint algorithm can be performed using two strategies: (i) by reducing the amount of data saved in a checkpoint (*incremental* scheme), and (ii) by reducing the delay during which processes are blocked (*non-blocking* scheme). These optimizations are described below.

Incremental checkpointing

The basic checkpointing algorithm saves all shared pages onto disk. A first optimization consists in saving only the pages that have been modified since the last checkpoint. This reduces the amount of data written to disk, and thus decreases the checkpointing overhead. The implementation of this scheme requires to identify which pages have been modified since the last checkpoint. As the target operating system used for these preliminary experiments does not provide a primitive for consulting the pages *dirty bits*, two mechanisms (*accurate* and *estimate*) where experimented for detecting modified pages. Both schemes add a flag DBS (*Dirty Bit Set*) to each page descriptor that indicates if the page is modified.

In the *accurate* scheme, the DBS bit of a page indicates (accurately) if the page has been modified since the last checkpoint. The implementation of this scheme relies on the operating system ability to trap access privilege violations. When a process takes a checkpoint, only the pages with their DBS bit set are stored onto disk. Their DBS bit is then reset and their access privilege is changed to READ-ONLY. The DBS bit of the page will be set again if a privilege violation occurs later. The main drawbacks of this scheme are the overhead due to the system calls required for restricting the access privilege on each page written to disk, as well as the cost of detecting privilege violations.

The *estimate* scheme has been designed to avoid performance drawbacks of the accurate scheme. When taking a checkpoint, a process saves all its owned READ-WRITE pages, but only the owned READ-ONLY pages with their bit DBS set. The DBS bit of a page is set when the owner of the page changes, and is reset when the page has a READ-ONLY access privilege and is written to disk. Thus, the set of pages saved during the checkpointing protocol is a superset of the pages modified since the last checkpoint.

Non-blocking page flushing

In the basic checkpointing protocol, an application process resumes only when the pages it owns have been written to disk. Two solutions, *pre-copying* and *copy-on-write*, were considered to reduce the delay during which application processes are blocked.

In the *pre-copying* scheme, an application process is not blocked until data (DSM pages) is transferred to disk. It does not even wait for an acknowledgment of the messages' receipt. The checkpoint is sent in a message to the capture server, which writes it onto disk concurrently with the program execution. A simple implementation of message-passing, in which data is immediately copied across address spaces is used for communicating between application processes and the capture server.

The *copy-on-write* scheme differs from the pre-copying scheme in the way data is transferred between the application processes and the capture server. Instead of using a simple implementation of message-passing, this scheme uses an implementation of message-passing which relies on the *copy-on-write* mechanism. On the sender process side, the pages to be transferred are write-protected until the receiver physically reads them; if the sender attempts to modify a page, the kernel makes a copy of the page permitting the sender to continue its execution. This scheme is similar to the one used in [14].

In order to guarantee the consistency of checkpoint files, for both pre-copying and copy-on-write schemes, the capture server is modified to ensure that identification files are written to disk only when the DSM pages have been flushed to disk.

Performance

The experiments were done using sixteen Paragon nodes running Paragon-OSF/1, with a single application process per node. The performance of the checkpointing protocol was measured on several parallel applications [3]. The results for two of them, Mp3d and Matmult are given below. Mp3d is an application from the SPLASH benchmark [19] that solves a problem in rarefied fluid flow simulation. The main shared data structures of the application are two large arrays; one for storing the state information for each particle and the other for storing the properties of the space where particles move. The experiment was run for a system of 40000 particles for 11 iterations. False sharing occurs when accessing the array of particles. Matmult consists of a multiplication of two 512x512 matrices of double precision floating point values, iterated 25 times. There is no false sharing in this application; each node fills exactly sixteen pages of the result matrix. A summary of the applications' characteristics is given in table 1 (the applications' running time is the running time of the slower of the 16 application processes):

Table 2 gives the checkpointing overhead (percentage of the application running time required to save checkpoints) for the different variations of the checkpointing protocol. The first column of table 2 gives the checkpointing overhead for the basic checkpointing protocol. The checkpointing time includes: (i) the cost of network transmission to the capture server, (ii) the cost of saving checkpoints onto disk and (iii) the cost of communication required between processes to ensure that a consistent state is recorded. The results show that even with a non optimized checkpointing protocol, the checkpointing overhead is reasonably low (8% for Matmult and 11% for Mp3d). It is higher for

Matmult than for Mp3d as Matmult uses more shared memory than Mp3d.

The time overhead of the two incremental checkpointing protocols, estimate and accurate, is given in the second and third columns of table 2. Compared with the basic checkpointing protocol, the estimate scheme reduces the checkpointing overhead from 8% to 7% for Mp3d and from 11% to 6% for Matmult. The performance gain for the estimate scheme is higher for Matmult, for which the introduction of incremental checkpointing avoids saving the two source matrices (2/3 of the application data). The performance gain is lower for Mp3d, for which only three shared pages are not modified between two successive checkpoints. For both applications, the estimate scheme gives better results than the accurate scheme. This is due to the memory access patterns of the two applications, which modify their whole working space (except the source matrices for Matmult and three pages for Mp3d) between two consecutive checkpoints. The two optimizations detect the same set of pages as being modified. The additional overhead in the accurate scheme comes from the protection violations needed to accurately keep track of modified pages.

The last two columns of table 2 give the performance of the two non-blocking protocols: pre-copying and copy-on-write. These two schemes are implemented within the estimate incremental scheme. The cost of checkpointing for these two optimizations only includes: (i) the cost of sending messages to the capture server and (ii) the cost of synchronization (compared to the basic checkpointing scheme, the cost of network transmission and disk access are not included). The results show that an important reduction of the checkpointing overhead is obtained by both the non-blocking schemes. The checkpointing overhead is divided by an average factor of 35 for Mp3d and 55 for Matmult compared to the blocking incremental checkpointing protocol. This shows that substantial performance gains are obtained by both non-blocking protocols.

3.2.2 Integration of checkpointing in Stardust

The preliminary experiments have shown that a careful optimization of the capture server reduces the time overhead of checkpointing for failure-free executions by an important factor. Checkpointing in Stardust relies on a set of *Capture Servers* (CSs), one per type of architecture (see paragraph 5.1 for more details about the internal structure of Stardust). Each CS implements an incremental (estimate) non-blocking (pre-copying) scheme. A non-blocking pre-copying scheme was preferred over a copy-on-write non-blocking scheme because it was easier to implement on top of Chorus, one of the two operating systems on which Stardust is implemented.

During the capture protocol, each homogeneous memory manager (HoMM) running on an architecture A communicates with the corresponding capture server. It sends to the CS all (dirty) DSM pages it owns in A's physical format. When a page is needed at application restart time, it is requested to the CS having a copy of the page.

3.3 Load balancing

It is frequently observed that in clusters of workstations, there is a high probability that some hosts are heavily loaded, while others are almost idle. In addition, the instantaneous load of a given machine is likely to fluctuate. This suggests that performance gains may be achieved by transferring application processes from the currently heavily loaded hosts to the lightly loaded ones (*load balancing*). A load balancing algorithm consists of three components [23]: (i) The *information policy* specifies the nature and amount of load information used for deciding where to place processes, and the way by which the information is distributed; (ii) The *transfer policy* determines the eligibility of a process for load balancing based on the load of the nodes; (iii) The *placement policy* decides, for eligible processes, the hosts to which the processes should be moved. A description of these three policies in Stardust is given in the following paragraphs.

3.3.1 Information policy

Information policy definition in an heterogeneous environment requires selection of an architecture-independent measure for the machine loads. In Stardust, we use an integer ranging from 0 to 99. On the Paragon machine, as our configuration only supports a single process per node, a node load can only take the values 0 or 99. On the Pentiums, every load value between 0 and 99 can be taken; the load of a given machine depends on the number of processes running on the machine and on the machine memory usage.

There is one *Load Information Manager* (LIM) per different type of architecture. It maintains the current load of every machine of this type of architecture. Each LIM updates its data structures by periodically requesting each machine of the architecture its current load value.

3.3.2 Transfer policy

The transfer policy used in Stardust relies on the assignment of static priorities to applications. In the case of an overloaded system, only applications with a high priority are allowed to execute; applications with lower priorities are suspended until the system load gets low enough. The transfer policy does not consider each process of an application separately; all processes of a given application are considered as a whole (at application reconfiguration time, every application process is restarted). The transfer policy is activated in three situations:

- When a new application is started: every process of the application is then selected by the transfer policy.
- When an application terminates: all other application processes are selected by the transfer policy, beginning with the processes having the higher priority.
- When a machine becomes overloaded: the transfer policy then selects all processes belonging to an application which runs on the currently overloaded machine. A machine is considered to be overloaded if its load value exceeds a static threshold. All architecture types, except the Paragon, on which each node hosts at most one process per node, obey this simple threshold strategy.

When an application with a high priority is started, its execution can cause applications with a lower priority to be migrated, or in the worst case suspended if the new application overloads the system.

3.3.3 Placement policy

The choice of the hosts on which application processes should be transferred takes into account both the machines loads and the user preferences. For each application, the programmer specifies in a configuration file an ordered list of *topologies*. A topology describes the resources needed by an application:

- number of hosts on which the application must be executed;
- type of host needed for every process (or the constant ANYWHERE to specify that the process can be assigned to whatever type of architecture). The user can also assign a process to a specific machine by giving its Internet address.

An example of configuration file is given below:

```
4
PENTIUM     astro1.irisa.fr
PENTIUM
PENTIUM
PENTIUM
4
ANYWHERE
ANYWHERE
ANYWHERE
ANYWHERE
```

The programmer specifies with this configuration file that its application should preferably be executed on 4 PCs, the first process being assigned to machine named *astro1.irisa.fr*. If these machines have crashed or are overloaded, 4 nodes of any type can be selected.

When the processes of an application are selected by the transfer policy, the ordered list of topologies is scanned until one topology can be mapped on the available machines. If different machines make the mapping of an application on the available machines possible, the less loaded ones are selected. If none of the topologies described in the configuration file can be selected, the application is suspended until its processes are made eligible again by the transfer policy.

Note that the number of processes of an application is chosen by the application programmer. This is particularly important in an environment supporting DSM since the number of processes can be fixed to reduce false sharing.

4 A Complete Application

The use of the checkpointing and load balancing mechanisms of Stardust is illustrated by giving the complete code of a toy application. The application, whose code is given below, is a sequence of products of two 256x256 matrices. The three matrices (the two source matrices and the result matrix) are stored in shared virtual memory regions. Each process computes a set of lines of the result matrix.

```
#define NE              256                     /* Matrix number of rows and columns */
#define MATSIZ          (NE*NE*sizeof(double))  /* Matrix size */
StardustPinkerton Stype;                        /* Pinkerton type of the shared matrices */
long i1,i2,ires;                                /* Region identifiers for the matrices */
long nbn;                                       /* Number of nodes */
long me;                                        /* Number of the current node */
double *m1,*m2,*mres;                           /* Addresses of the matrices */
/* Matrix product */
product()
{
long i,j,k;             /* Loop variables */
long lbeg,lend;         /* First and last line computed by the current processor */
long lperproc;          /* Number of lines computed per processor */
        lperproc = NE / nbn; lbeg = me * lperproc; lend = lbeg + lperproc;
        for (i=lbeg;i<lend;i++) {
                for (j=0;j<NE;j++) {
                        mres[i*NE+j] = 0;
                        for (k=0;k<NE;k++) {
                                mres[i*NE+j] += m1[i*NE+k] * m2[k*NE+j];
                        }
                }
        }
}
/* Main program */
main()
{
        nbn = svm_numnodes();   /* Get the number of nodes of the current topology */
        me = svm_mynode();      /* Get the number of the current node */

        /* Create, type the matrices, and map them onto the process address space */
        svmMakePinkertonType(&Stype,"{{D}}",NE,NE);
        svm_create_region(MATSIZ,&i1,(char **)&m1,&Stype );
        svm_create_region(MATSIZ,&i2,(char **)&m2,&Stype );
        svm_create_region(MATSIZ,&ires,(char **)&mres,&Stype );

        /* Start the application from its last saved state */
```

```
    svm_go();

    /* Fill the matrices */
    /* ... */

    svm_gsyncCheck();        /* Save the application state */
    product();               /* Make a matrix product */
    svm_gsyncCheck();        /* Save the application state */
    product();               /* Make a matrix product */
    svm_gsyncCheck();        /* Save the application state */
    product();               /* Make a matrix product */
}
```

The main program includes calls to functions *svm_numnodes* and *svm_mynode*, that give the application knowledge about the current application topology. The main program of the application includes two calls to function *svm_gsyncCheck*, which synchronize all application processes and causes the application state to be saved. This saved state can be used for restarting the application after a failure, and for migrating the application on a different set of nodes. Note that the programmer does not explicitly request the migration of his/her application. He/she only gives the places where the application can be migrated (calls to *svm_gsyncCheck*). The decision of actually migrating the application is taken by the load balancing strategy (transfer and placement policies).

Function *svm_go* is used for restarting an application from its last saved state after a checkpoint or a migration. It includes a communication with the capture server, which returns a barrier identifier where the application must be restarted. Pre-processing is used so that the application restarts at the synchronization barrier whose identifier is returned by *svm_go*.

Note that this application supports dynamic changes of its number of processes: each time the application is started (or restarted after a failure or a migration) functions *svm_numnodes* and *svm_mynode* are called to obtain the current application topology. In addition, all non-shared variables are set at application restart/migration time (code before *svm_go*).

5 Implementation and Performance of Stardust

Stardust is currently implemented on an Intel Paragon running a Paragon-OSF/1 kernel and on a 10 Mb/s Ethernet network of Pentium PC machines running the Chorus/classiX operating system [18]. This section gives more details about the internal structure of the Stardust software and gives its performance.

5.1 Software structure of Stardust

Figure 2 shows the structure of the Stardust software in a system with 3 Paragon nodes and 3 PCs. Each square represents a process.

There is a single homogeneous memory manager (HoMM) per node which hosts an application process (on the figures, hosts 1 and 2 on both architectures). There are also on each architecture three processes: HeMM (*Heterogeneous Memory Manager*), CS (*Capture Server*) and LIM (*Load Information Manager*). In the figure, for the sake of simplicity, the three processes have been placed on the same machine (host 3 on both architectures). We use a LIM and a CS per architecture instead of using a centralized LIM and CS in order to make Stardust scalable to a large number of architectures and machines.

5.1.1 Implementation of the homogeneous and heterogeneous memory managers

In order to ease portability, homogeneous and heterogeneous memory managers (HoMMs and HeMMs) have exactly the same structure, which is depicted in figure 3. An important part of the code of HoMMs and HeMMs is architecture and operating system independent. This way, we expect Stardust to be easily portable to other machines and/or operating systems.

The *consistency manager* maintains shared data consistency using a write-invalidate protocol. Its code is completely independent from the target operating system and communication network. It is used both by HoMMs and HeMMs.

HoMMs and HeMMs communicate with each other through interactions with the *network manager*, which offers primitives for exchanging messages. The code of the network manager depends on the types of architecture on which the message sender and receiver are running. If they are executing on the same type of host, the most efficient communication protocol for the type of hosts is used

(NX communication library on the Intel Paragon, Chorus IPC on the Pentium PCs). If both the message sender and receiver run on different types of hosts, TCP is used via sockets, and the message is converted into the XDR format before being sent.

The *OS interaction manager* is responsible for the communication with the underlying operating system concerning memory management and thread management. Its interface includes primitives for (a) requesting a page; (b) modifying the access privileges on a page; (c) invalidating a page. The implementation of this module for HoMMs uses the operating system virtual memory interface for completing the requested action on the page. Both types of operating systems on which Stardust is implemented are based on the micro-kernel technology. The implementation of most of the routines of the OS interaction manager are then straightforward, as they can directly be mapped onto a kernel call. The implementation of the OS interaction manager of an HeMM simply consists in redirecting each request related to an heterogeneous page to the concerned HoMMs. For instance, let us assume that an heterogeneous page *Hp* is present on an architecture of type A and corresponds to two pages p_1 and p_2 located on two machines m_1 and m_2 of type A. A request for page *Hp* to A's OS interaction manager is translated into two requests for pages p_1 and p_2 to m_1 and m_2's HoMMs. Concerning thread management, the OS interaction management offers a common interface for thread creation and synchronization. Implementation of the thread management interface on the Paragon relies on the *pthread* library; the Chorus kernel calls are directly used on the PCs.

5.1.2 Implementation of the processes managing checkpointing and load balancing

The checkpointing and load balancing facilities heavily use the *Capture Servers* (CS). These processes are responsible for capturing the state of applications to stable storage. In addition, the CSs keep a cache of the applications state in main memory. Stable storage is implemented using files (PFS on the Paragon and NFS on the Pentiums). The state capture protocol for an application proceeds in two steps:

1. On each architecture, every HoMM sends the (dirty) pages it owns to the corresponding CS. The CS saves the page onto disk using the architecture's physical format.
2. CSs of all architectures synchronize in order to validate the application state.

The same mechanism is used for migrating an application and for restarting an application after a failure. New application processes are started. All pages that have been modified since the last state capture are set to a new state. Every attempt to access a page that is in this new state triggers a page fault which is redirected to the capture server owning the page, which provides it to the requester. If the page's requester and the CS having the copy of the page belong to different architectures, communication uses TCP and data is exchanged using the XDR format. The region's type (see paragraph 2.2) is used for converting the page to and from the XDR format. In the current implementation, a simple error detection mechanism, based on the periodic use of watchdog timers, is used to detect host failures.

The information policy of the load balancing strategy is implemented by the set of *Load Information Managers* (LIMs). Periodically, the LIM of a given architecture requests the current load of all nodes of the architecture. The placement policy is implemented by one process, running on whatever type of architecture (in the current implementation on a Pentium node). This process is invoked at start and end of each application, and when a node overload is detected by one LIM.

5.2 Performance measures

Performance of Stardust was measured on a 56-node Paragon and a 10 Mb/s Ethernet network of Pentiums. Each node of the Paragon runs Paragon-OSF/1, based on the Mach micro-kernel. Each node is equipped with 16 Mbytes of memory, of which nearly 8 Mbytes are consumed by the operating system. The page size on the Paragon is 8 Kbytes, and the measured transfer rate between nodes is 60 Mbytes/s. The Pentiums run the Chorus/classiX operating system, also based on the micro-kernel technology. Each machine has 32 Mbytes of memory, of which Chorus only leaves 8 Mbytes free for the paging activity. The size of pages on the Pentiums is 4 Kbytes. For all the performance measures given in this paragraph, the size of heterogeneous pages is 8 Kbytes.

5.2.1 Basic performance measures

The basic cost of handling a page fault in Stardust is shown in table 3. It includes the detection of a page fault (READ or WRITE), the time overhead due to the consistency protocol, the request transmission time and the time needed to transfer and convert (if required) the page contents. Figures were obtained by doing a large number of page faults in a sequence, and measuring the total elapsed time.

The first column shows the cost of a page fault when the application entirely runs on the Paragon. The second column gives the figure when the application runs on the Pentiums. Finally, the third column shows the cost of page faults when the page owner and the page requester run on different architectures. The first line shows the time needed to create a page (first page fault on a page). The second line gives the cost of a read page fault on a node, when a READ-WRITE copy of the page exists on another node. Finally, the third and fourth lines give the cost of a write page fault, when 1 (respectively 7) copies of the page exist and have to be invalidated. The figures given in table 3 show that an extra overhead (for instance 4.3 ms for a read page fault) is incurred in dealing with heterogeneity. This is due to (i) the cost of type analysis, (ii) the transformation of data to and from the XDR format, and (iii) the data transfer in the XDR format, which takes more memory space than each architecture's physical format. A detailed analysis of the timing of a read page fault in the heterogeneous case shows that 5.9 % of the total page handling cost is required for data conversion. 89.4 % is required for communication. The remaining 4.7 % is due to the consistency protocol and to the interactions with the operating system.

Figure 4 shows the performance of two applications using distributed shared memory: Povray (a) and MGS (b). Povray [22] is a freeware ray-tracing application. It reads standard ASCII text files that describe the shapes, colors, textures and lighting in a scene, and mathematically simulates the rays of light moving through the scene to produce a photo-realistic image. The base ray-tracing program was modified to store the result image in a shared virtual memory region. Measures for this application were done for a 1024x728 image, each pixel being represented by a 16-bit value. MGS (Modified Gram-Schmidt) is an algorithm which produces, from a set of vectors, an orthonormal basis of the space generated by these vectors. The problem size for this application is 256x1024 double precision floats. As dealing with false sharing was not our main concern when designing Stardust, we deliberately chose here two applications that do not exhibit false sharing.

Part (a) of figure 4 shows the time required to execute Povray successively on 1, 2, 8, 16, 32 and 56 Paragon nodes (curve labelled *Paragon*), and then on an heterogeneous system of 56 Paragon nodes and 8 PCs (curve labelled *Stardust*). Part (b) of figure 4 shows the elapsed time for MGS when it runs on (i) a set of Paragon nodes, (ii) a set of Pentium nodes, and (iii) an heterogeneous system with the same number of nodes of both architectures. Measurements for MGS were done on a system starting with 1 and up to 8 nodes.

The curves show that the application executed on an homogeneous set of Pentium nodes always exhibits better performance compared to the same application running on Paragon nodes or on an heterogeneous system containing both Paragon and Pentium nodes. This is due to the difference of computing power between the processors. In addition, even if the time required for resolving a page fault, involving nodes of different architectures, is longer than the time needed to resolve a page fault in an homogeneous environment, applications still exhibit a good speedup when the number of nodes increases. Finally, compared to the execution time for the Paragon machine, only a small percentage of the total execution time is needed to deal with heterogeneity.

5.2.2 Performance measures of the reconfiguration mechanisms

Table 4 shows the cost of capturing and restoring the state of a matrix product application (*Matmult)* on a set of Paragon nodes (first set of column) and on a set of Pentium nodes (second set of column). Matmult is made of a sequence of multiplications of matrices of 256x256 double precision floats, the input and output matrices being stored in shared virtual memory regions. Column S of figure 4 gives the time required to capture the application state (the time includes the cost of communicating with the CSs and saving the application state onto disk). Column R gives the cost of restarting Matmult on the same architecture and with the same number of processes as before. The cost only includes the time required for re-starting all application processes. Finally, column P gives the time needed to load on demand all pages from the capture server. All figures in table 4 are given in ms.

Saving the state of the application takes from 1.8 to 2.0 s on the Paragon, and from 1.7 to 1.9 s on the Pentiums. The time required to save the state of the application varies almost linearly with the number of processes of the application. Restarting application processes takes approximately 2 s on the Paragon. This figure does not vary significantly with the number of processes. On the Pentiums, restarting the application takes from 2 to 4 s depending on the number of application processes. For both architectures, large variations of the restart times have been observed. Finally, the time required for loading shared pages from the capture server takes from 1 to 4 s depending on the target architecture and the number of application processes. More time is needed on the Pentiums than in the Paragon due to the latency of the communication network.

Table 5 shows the performance gains that can be obtained using Stardust load balancing facility, for three applications. The first one, named MatMultHetero4_4 is made of a sequence of 10 matrix products between 256x256 matrices. The first 5 products are executed on 4 nodes of the Paragon and the remaining 5 products on 4 Pentiums. MatMultHomo2_4_8 is again the matrix product, in which the first 3 products are computed on 2 Pentiums, then 3 products are computed on 4 Pentiums and the remaining 4 products are executed on 8 Pentiums (the execution includes 2 application migrations). Finally, application PovrayHetero2_8_64 is the Povray application introduced before. It includes two process migrations (one for moving the application from 2 to 8 Pentium nodes and the other to move it from 8 Pentium nodes to an heterogeneous system of 56 Paragon nodes and 8 PCs). 200 lines of the output image are computed between two successive migrations. For the three applications, the migrations were triggered manually; they do not result from load variations. Obviously, additional experiments are needed to show the behavior of Stardust in the event of dynamic load variations.

The measures given in table 5 show that important performance gains can be obtained by the load balancing strategy. 6% of the execution time is gained on the Matmult application when it is moved at the middle of its execution to another architecture with enhanced performance. 32% is gained for the same application when the number of application processes is dynamically extended from 2 to 8. For Povray, 71% of the execution time is gained by dynamically changing the application topology. A higher gain is obtained for Povray than for Matmult, since ray-tracing is much more compute-intensive than matrix multiplication.

6 Related Work

Many environments for parallel programming on networks of heterogeneous workstations have been designed and implemented in the last ten years. A key difference between Stardust and most of these environments is that Stardust runs both on parallel machines and networks of workstations. In addition, unlike most environments, Stardust supports both the shared-memory and message-passing paradigm, and includes support for both load balancing and checkpointing.

Most environments for parallel programming on networks of workstations are based on the message-passing paradigm. PVM [10] is a representative example of such environments. Both PVM and Stardust use XDR as a standard data representation. PVM provides a richer set of primitives than

Stardust; in particular, Stardust does not support dynamic process creation. Another difference between Stardust and PVM is the way data items are packed into communication buffers. In Stardust, a single routine call is required to associate a type to a message buffer, while in PVM each basic data item has to be packed into the message buffer by a separate routine call. This makes code for message-passing in Stardust a little bit more concise than in PVM. From an implementation point of view, we have tried in Stardust to use the most efficient communication protocol of a given architecture for communications between hosts of this architecture. In comparison, most implementations of PVM use TCP for all communications, which can be less efficient than architecture-specific communication protocols. An extension of PVM integrating dynamic process migration, called MPVM [5] has been designed. Unlike in Stardust, process migration in MPVM is requested explicitly by application programmers, and a process executing on a host can only migrate to a host of the same type.

Few environments based on DSM have been designed for clusters of heterogeneous workstations. To the best of our knowledge, all DSM systems, except Mairmaid [24] are designed for homogeneous environments. One difference between the two systems is the way types are associated to shared memory regions. In Mairmaid, the user only gives to the system the name of the C data type contained in the region; a description of the data type is then found at run-time in the program object file within the symbol table generated in Unix for symbolic debuggers (*stab*). In contrast, a Stardust user has to give a complete description of the contents of a region, but the implementation of Stardust is independent of the format of object files. Another difference between the two systems is that in the Mairmaid prototype, it is assumed that basic data types have the same size on all architectures. This simplifying assumption is not done in the Stardust prototype, where different sizes of data items are dealt with thanks to the use of heterogeneous pages as a common unit for data transfer and data consistency. Let us also note that in opposition to Mairmaid, every page fault does not lead to a conversion into standard data representation; this occurs only when the page resides in a different architecture type. Finally, Mairmaid does not include mechanisms for checkpointing and load balancing.

Much of the previous work in checkpointing was targeted for message-based systems. Fewer studies concentrate on DSM systems (see [16] for a survey of these works). Much of the past work on consistent checkpointing has focused on minimizing the number processes that must participate in taking a consistent checkpoint or in rolling back. Another issue that has been received attention is how to reduce the number of messages required to synchronize the consistent checkpoint. In this paper, we take the approach adopted in [8] and focus instead on minimizing the checkpointing overhead on the failure-free running time of programs. Most optimizations presented in this paper as similar to the ones given in [8], the difference being that [8] only considers message-based applications while we consider both message-based and DSM applications. Like [9], we adopt consistent checkpointing instead of independent checkpointing, as it has a lower overhead during failure-free execution of programs (messages exchanges need not be logged). This is especially important in DSM systems where the message traffic can be important and is not under the user's control. Like [7], we try to reduce the amount of data in checkpoints. In our proposition, this is achieved by only saving DSM regions (the processes' stacks and heaps are not saved). In contrast, in [7], the programming paradigm used for writing applications (coarse-grained dataflow) is used to keep the memory overhead for checkpoints low.

Studies on application reconfiguration and heterogeneous process migrations are described in [11] and [20]. A method for automatically saving and restoring the application stack is presented in [11]. Similarly to our work, the programmer explicitly specifies the points were the application can be reconfigured (e.g. moved to another machine). The source program is then transformed so as to keep track of the structure of the application stack. In contrast to the work described in [11], application reconfiguration in Stardust is expected to be more efficient, as no stack transformation is required. Moreover, the study presented in [11] only deals with sequential programs, while we consider parallel applications. A system for object and native code thread mobility among heterogeneous computers is presented in [20]. This system is based on the concept of *bus stops*, which are machine-independent representations of program points as represented by program counter values. Unlike our system, their is no need for any programmer intervention in [20] for identifying bus stops, but important modifications of the programs' compilation process seem to be required.

Although numerous studies related to load balancing have been conducted in the last decade, few practical systems have been built. The most popular systems are Utopia [25], Condor [15], Mosix [1], and Calypso [2]. Utopia [25], built at the university of Toronto, is a system for automatic load sharing in large, heterogeneous distributed systems. Utopia, unlike Stardust, does not support process migration; it only supports initial process placement. The Condor system, at the university of Wisconsin provides load sharing of sequential batch jobs [15]. Condor aims at making use of idle workstations. In contrast, Stardust supports both batch and interactive jobs and makes use of not only idle workstations. Like Stardust, Condor supports process migration, but only on homogeneous hosts. Mosix [1] is a distributed operating system built at Hebrew University of Jerusalem. It is an enhancement of Unix with network transparency and automatic dynamic load balancing. Since load balancing is implemented inside the Mosix kernel, there is no problem while migrating OS-dependent data structures. Unlike Stardust, Mosix does not include support for heterogeneity. Calypso [2] is a system for fault-tolerant parallel processing on distributed platforms. Calypso automatically distributes the work-load among the machines. Programs in Calypso are written in a specific language (CSL, for Calypso Source Language). Programs are described as a sequence of parallel steps (threads executing in parallel on distinct hosts) and sequential steps. Threads communicate through shared-variables. Unlike Stardust, load balancing in Calypso does not rely on migration; rather, it is based on an initial placement of the threads (once assigned to a given host, a thread does not migrate).

7 Concluding Remarks

This paper has described the design and implementation of Stardust, an environment for parallel programming on networks of heterogeneous machines, including both workstations and parallel computers. The key distinctions of Stardust compared to related environments are the following: (i) *Support for multiple programming paradigms*: parallel and distributed applications built with Stardust can communicate both through message-passing and page-based distributed shared memory; (ii) *Support for checkpointing and load balancing*: the state of an application can easily be saved onto disk. The application can then be restarted on other machines, either to support machine crashes or to balance the machines loads; and (iii) *Support for heterogeneity*: Stardust includes mechanisms for automatically converting data between architectures of different types. Application programmers only have to specify the type of the data structures they use. In addition, unlike most related environments, Stardust implements heterogeneous process migration.

Performance measures have shown that the load balancing mechanism included in Stardust can lead to an substantial reduction in the applications execution time. The execution time of the applications taken in examples in this paper has been reduced by an average factor of 37%. From an implementation point of view, Stardust is structured to ease its extension to new operating systems and new types of architectures. We expect its extension to a Unix platform to be straightforward. Our current work consists in studying the scalability of Stardust when the number of hosts increases.

Acknowledgements

The authors would like to thank Michel Banâtre, Thierry Priol, Stéphane Billiart, Akhil Sahai and the anonymous referees for their comments on earlier versions of this paper. Their comments greatly improved both the technical content and the presentation of this work.

References

[1] Barak, A., Guday, S., and Wheeler, R. *The MOSIX Distributed Operating System, Load Balancing For Unix.* No. 672 in Lecture Notes in Computer Science. Springer Verlag, 1993.

[2] Baratloo, A., Dasgupta, P., and Kedem, Z. M. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *4th IEEE Synposium on High Performance Distributed Computing* (1995).

[3] Cabillic, G., Muller, G., and Puaut, I. The performance of consistent checkpointing in distributed shared memory systems. In *Proc. of the 14th Symposium on Reliable Distributed Systems* (Bad Neuenahr, Germany, September 1995), pp. 96–105.

[4] Cap, C. H., and Strumpen, V. Efficient parallel computing in distributed workstation environments. *Parallel Computing 19* (1993), 1221–1234.

[5] Casas, J., Clark, D., Kunuru, K., Otto, S., Prouty, R., and Walpole, J. MPVM: A migration transparent version of PVM. Technical Report CSE-95-002, Oregon Graduate Institute of Science and Technology, February 1995.

[6] Chandy, K. M., and Lamport, L. Distributed snapshots : Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (February 1985), 63–75.

[7] Cummings, D., and Alkalaj, L. Checkpoint/rollback in a distributed system using coarse-grained dataflow. In *Proc. of 24th International Symposium on Fault-Tolerant Computing Systems* (Austin, Texas, June 1994), pp. 424–433.

[8] Elnozahy, E. L., Johnson, D. B., and Zwaenepoel, W. The performance of consistent checkpointing. In *Proc. of the 11th Symposium on Reliable Distributed Systems* (October 1992), pp. 39–47.

[9] Elnozahy, E. N., and Zwaenepoel, W. On the use and implementation of message logging. In *Proc. of 24th International Symposium on Fault-Tolerant Computing Systems* (Austin, Texas, June 1994), pp. 298–307.

[10] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[11] Hofmeister, C., and Purtilo, J. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *Proc. of 13th International Conference on Distributed Computing Systems* (Pittsburgh, Pennsylvania, May 1993), pp. 101–110.

[12] Kranz, D., Johnson, K., Agarwal, A., Kubiatowicz, J., and Lim, B.-H. Integrating message-passing and shared-memory: Early experience. In *Proc. of the Fourth SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1993), pp. 54–63.

[13] Li, K., and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst. 7*, 4 (November 1989), 321–357.

[14] Li, K., Naughton, J., and Plank, J. Real-time concurrent checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice Parallel Programming (PPOPP), SIGPLAN notices* (1990), vol. 25, pp. 79–88.

[15] Litzkow, M., Livny, M., and Mutka, M. W. Condor - a hunter of idle workstations. In *Proc. of 8th International Conference on Distributed Computing Systems* (San Jose, CA, January 1988), pp. 104–111.

[16] Morin, C., and Puaut, I. A survey of recoverable distributed shared memory systems. Tech. Rep. 975, IRISA, December 1995.

[17] Pinkerton, C. A heterogeneous distributed file system. In *Proc. of 10th International Conference on Distributed Computing Systems* (May 1990).

[18] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Léonard, P., Langlois, S., and Neuhauser, W. The Chorus distributed operating system. *Computing Systems 1*, 4 (1988), 305–370.

[19] Singh, J., Weber, W., and Gupta, A. SPLASH : Stanford parallel applications for shared-memory. Tech. Rep. CSL-TR-91-469, Computer Systems Laboratory, Stanford University, April 1991.

[20] Steensgaard, B., and Jul, E. Object and native code thread mobility among heterogeneous computers. In *Proc. of 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, Colorado, December 1995), pp. 232–242.

[21] Sun. *Network Programming Guide - External Data Representation Standard: Protocol Specification*, 1990.

[22] Young, C., and Wells, D. *Ray Tracing Creations, Second Edition.* Waite Group Press, November 1994.

[23] Zhou, S. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering 14*, 8 (September 1988), 1327–1341.

[24] Zhou, S., Stumm, M., Li, K., and Wortman, D. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems 3*, 5 (September 1992), 540–554.

[25] Zhou, S., X, Z., Wang, J., and Delisle, P. Utopia: A load sharing facility for large, heterogeneous distributed computing systems. *Software Practice and Experience 23*, 12 (December 1993), 1305–1336.

Biographies

GILBERT CABILLIC received his Diplôme d'Études Approfondies (DEA, french equivalent of Masters) in 1992. He his currently finishing his Ph.D. at the University of Rennes. During his Ph.D, he has developed a shared virtual memory system on an Intel Paragon machine. His research interests include massively parallel architectures, consistency protocols, operating systems and high performance computing.

ISABELLE PUAUT received her engineer diploma from the french engineering school INSA (Institut National des Sciences Appliquées) of Rennes in 1989. She received her Ph.D. in 1993 from the University of Rennes. She is currently lecturer at INSA and contributes to the research activities of the IRISA research center (Institut de Recherche en Informatique et Systèmes Aléatoires). Her research interests include distributed systems, heterogeneous computing, parallel programming and operating systems.

Program name	**Running time** (s)	**Shared memory** (Kbytes)	**Checkpointing interval** (s)
Mp3d	1487	1640	370
Matmult	1314	6144	438

Table 1: Summary of the applications' characteristics

Program name	**Basic** (%)	**Estimate** (%)	**Accurate** (%)	**Pre-copying** (%)	**Copy-on-write** (%)
Mp3d	8.14	7.33	8.71	0.21	0.18
Matmult	11.04	6.42	6.52	0.11	0.04

Table 2: Performance of the different variations of the checkpointing protocol

Machine type	**Paragon**	**Pentiums**	**Heterogeneous**
Page creation	2.5	2.5	4.9
Read page fault	4.9	12.6	16.9
Protection violation (1 invals)	3.7	3.6	7.9
Protection violation (7 invals)	6.0	8.8	11.1

Table 3: Basic cost of handling page faults in Stardust (ms)

Machine type	**Paragon**			**Pentiums**		
	S	R	P	S	R	P
2 processes	1843	$\simeq$ 2000	1224	1780	$\simeq$ 2000	1520
4 processes	1975	$\simeq$ 2000	1863	1909	$\simeq$ 3000	3430
8 processes	2046	$\simeq$ 2000	1866	1660	$\simeq$ 4000	4320

Table 4: Basic cost of saving and restoring the state of Matmult (ms)

	No Migration	**Migration**	**Gain** (%)
MatMultHetero4_4	65187 ms	61410 ms	5.8
MatMultHomo2_4_8	69640 ms	46910 ms	32.6
PovrayHetero2_8_64	3945 s	1125 s	71.5

Table 5: Performance gains of the load balancing strategy

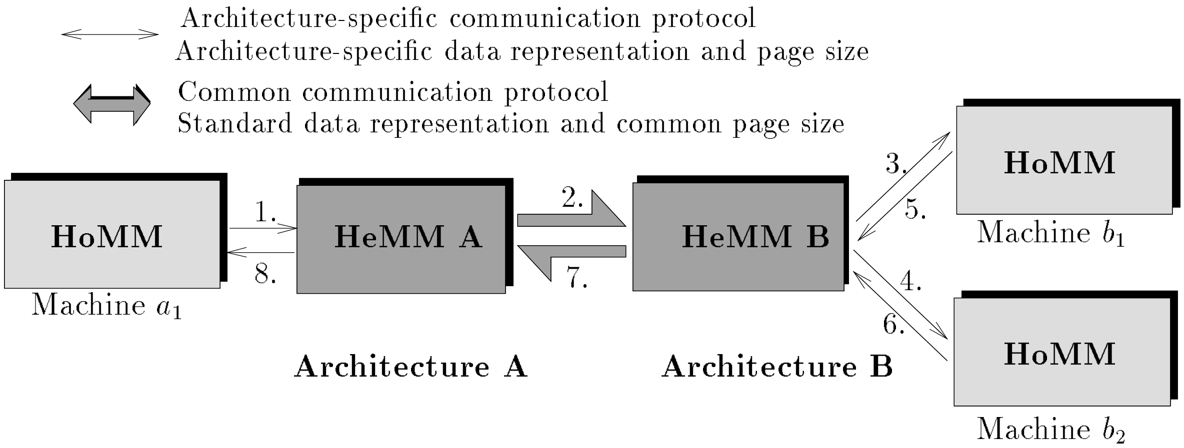

Figure 1: Resolution of a page fault in Stardust

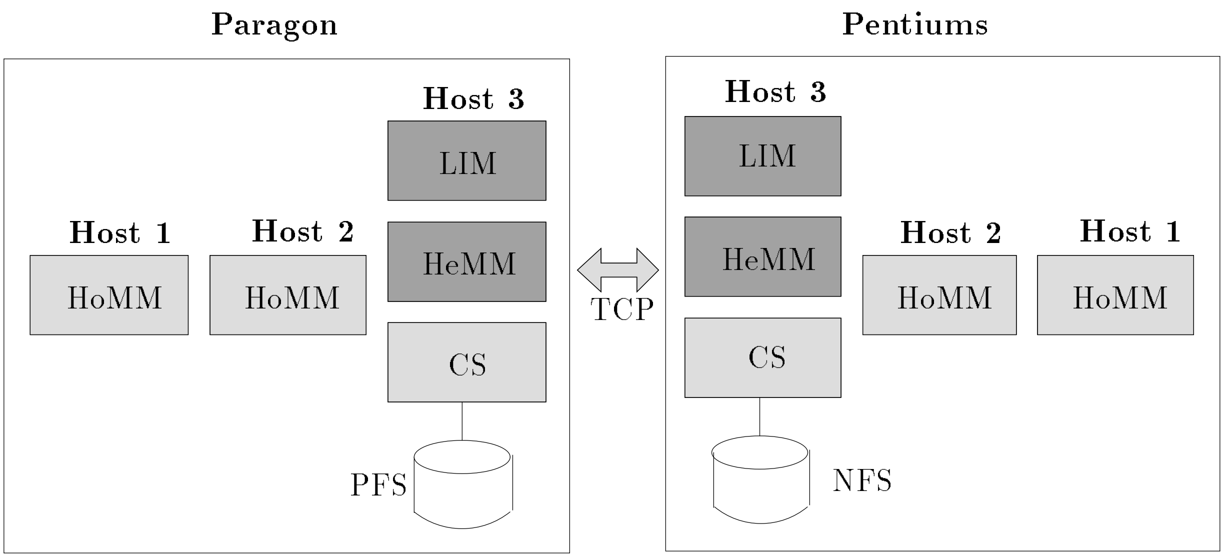

Figure 2: Structure of the Stardust software

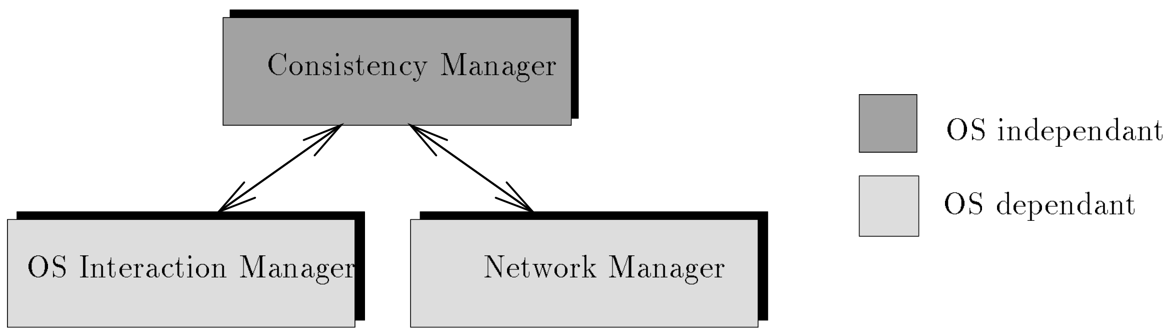

Figure 3: Internal structure of the memory managers

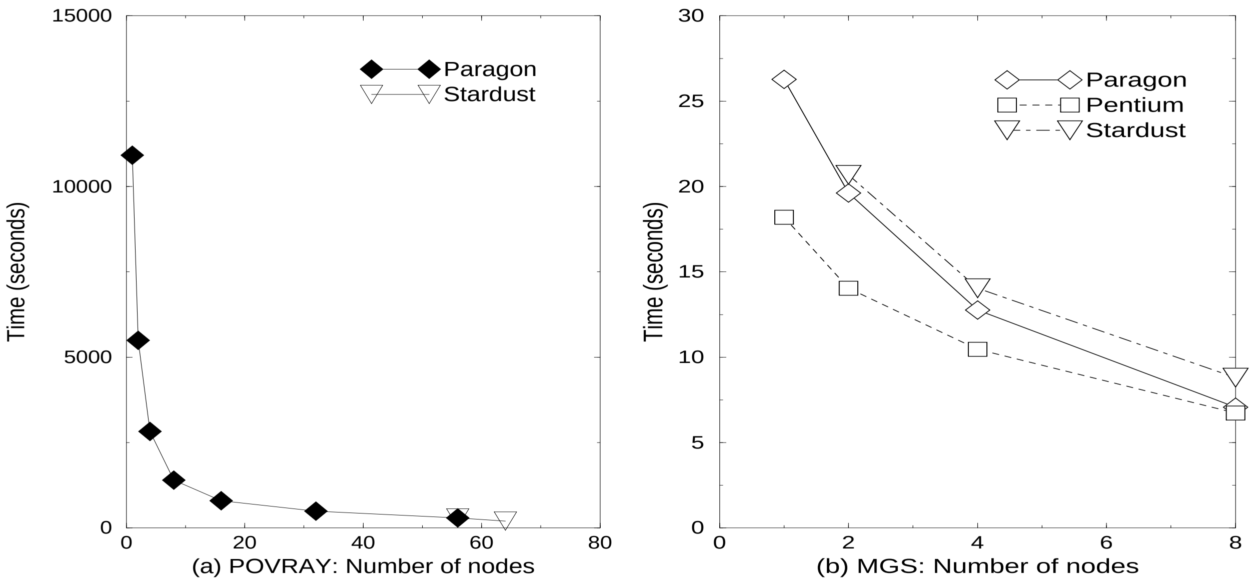

Figure 4: Execution time of the Matmult and MGS applications