PRISMA Contributions

PRISMA, a platform for experiments with parallelism

P.M.G. Apers
University of Twente
L.O. Hertzberger
University of Amsterdam
B.J.A. Hulshof, A.C.M. Oerlemans
Philips Research Laboratories Eindhoven
M.L Kersten
Centre for Mathematics and Computer Science

Abstract

Using a large multiprocessor, consisting of 100 processing nodes, in the area of data and knowledge processing poses challenging research and engineering questions. The PRISMA project presented here has addressed many of the problems encountered by the design and the construction of such a system. Among the results obtained are an implementation of a parallel object-oriented language, a hardware platform with efficient communication, and a distributed main-memory relational database system. Their combination forms a platform for further experimental research in several areas of distributed processing.

1 Introduction

The PRISMA project is a large-scale research effort in the design and implementation of a highly parallel machine for data and knowledge processing. It is organized as a nationwide Dutch research activity with combined forces from four universities, a governmental research institute, and Philips Research Laboratories. It ran from 1986 until end of 1990 and was manned with thirty persons. In this paper we present an overview of the PRISMA project.

The key research issue of the PRISMA project is to study the interaction between a non-trivial application, an object-oriented language, and a multiprocessor machine architecture. For this purpose an experimentation platform has been designed, which enables us to experiment with coarse grain parallelism for applications at many levels of detail. Such as its effect at the hardware level, the basic software such as the distributed operating system, language design and its compilation, and, finally, the construction of applications that exploit the computing power provided by a large multiprocessor system. Our effort has resulted in a prototype system of 100 processing nodes.

Since a multiprocessor system offers many opportunities to improve the performance of database systems, the prime target application selected as a testcase for the PRISMA platform was the construction of a new relational database management system. The DBMS architecture is based on two novel aspects. First, the PRISMA machine is equipped with about 1.5 Gigabyte of direct store, which provided the hardware foundation to focus on a main-memory DBMS architecture. Second, the multiprocessor system leads to a decomposition of the DBMS into functional components and the design of an extensible distributed system architecture. Thus, the research issues focus on exploitation of main-memory as the prime database store and the effective exploitation of the potential parallelism offered by the platform.

The PRISMA project is supported by the Dutch "Stimuleringsprojectteam Informaticaonderzoek" (SPIN)

One way to exploit a multiprocessor system is to augment a traditional implementation language with communication primitives, such as remote procedure calls, and process management. Its potential drawback is that application programs then tend to be bulky with hardwired code that manages process allocation and intra-process communication. To alleviate these problems we developed a Parallel Object-Oriented Language (POOL) that allows the programmer to express the inherent parallelism in an algorithm without concern on its mapping to the underlying system. The experimentation platform then provides the application and the hardware boundaries to study implementation techniques for POOL that show performance improvement over a wide range of multiprocessor system configurations without altering the program source.

The key issues in the design of the multiprocessor system are to choose the level of potential parallelism and the interconnect. To obtain a viable experimentation platform, we aimed for a large system, consisting of about 100 processing nodes equipped with sufficient main-memory to support large applications. Moreover, by focusing on an experimentation platform we needed an architecture that scales without showing step function behavior during performance measurements.

As mentioned before, the outcome of this project is a platform that supports a wide range of experiments to improve our understanding on parallel (symbolic) processing. Instead of focusing on toy problems, we have chosen a non-trivial application that provides an abundance of optimization problems for the language and the system implementation, while at the same time it would benefit from hiding the multiprocessor details. Moreover, by actually building such an integrated platform, we obtained a tool to calibrate performance models in a wide range of research areas.

In addition to the main-stream research and development within the PRISMA project, we fostered exploratory research in various directions. To illustrate, we looked at various techniques to speedup expert-systems, to exploit parallelism in information retrieval [Aalbersberg1, Aalbersberg2], and parallel term rewritting [Rodenburg].

The remainder of this paper is organized as follows. Section 2 describes in more detail the assumptions, questions, and methodology to construct the PRISMA system. Following, in Section 3 we focus on the realization of the platform from three perspectives; the database management system, the object-oriented language and runtime system, and the machine architecture. For each we describe how the component was realized and what lessons we learned in the course of its construction. We conclude with summary and future research in Section 4.

2 Assumptions,Questions, and Methodology

Research on the construction of systems is always characterized by having externally given assumptions and additional assumptions to focus the research. In the following subsections these assumptions and related questions are made explicit for the various layers in the system. Moreover, the assumptions are related to the state of the art in the various fields.

The externally given assumptions stem from the vision we have on the field. First, we expect that homogeneity and scalability are important factors for the production of hardware. Second, by chosing an object-oriented platform we want to prove that this platform provides the right kind of abstraction to easily express parallelism available in the application and to efficiently implement this platform. To prove these points, we have taken a non-trivial application, a database management system, and started experiments for the object-oriented platform and the database management system to gain experience in expressing and implementing parallelism.

2.1 Database Management System

The design and construction of database machines to improve non-numeric processing has attracted many researchers during the last two decades. At one end of the spectrum they have reduced the amount of data to be manipulated by filtering records as they are transferred from disk to main-memory [Ozkarahan] and using several functionally specialized computers linked into a network

[DeWitt1, Gardarin, Leland]. The other end of the spectrum is characterized by attempts to harness the processing power and storage capacity offered by large scale integration [Shaw, Katuka].

An observation is that no single database management system will efficiently support all database applications in the future. There arises a need for a variety of facilities, providing the proper level of abstraction, user interface, and functionality. The consequence is that future database management systems should be designed with functional distribution in mind. That is, a database management system comprises a set of data managers, each providing part of the required overall functionality and coordinated by a distribution manager. Structuring of the database management system as a collection of such data managers (operating on local data) controlled by a distribution manager fits well with the architecture of a (shared-nothing) *multiprocessor system.* Another observation is that the cost for main memory drops rapidly. Therefore, it becomes feasible and cost-effective to keep a major portion of the database in main memory[DeWitt2, Molina]. Our multiprocessor system is equipped with sufficient *main memory* to directly store and manipulate a medium-sized database, and thus avoid the I/O bottle-neck to disk. Two questions have to be answered here. First of all, can "massive" parallelism provided by a 100-node system (or even more) effectively be used in processing queries and, secondly, what are the effects of a main memory approach for data storage. Related to the main-memory approach is the question whether the system can be made reliable enough.

Database management systems are often viewed as interpreters of "unformatted" data on disk. The interpretation problem can be overcome by introducing the concept of a One-Fragment Manager (OFM). Such an OFM can be regarded as a relational algebra machine that handles relations or fragments of one specific tuple type (intuitively, one specific relation schema). This makes it possible to *compile* selections submitted to an OFM into code directly working on the data structures used by the OFM. The question to be answered here is whether the overhead of compilation is small enough to make a performance increase feasible.

One externally given assumption was that the database management system should be implemented in the *object-oriented language* called POOL [America1, America2] [America3]. In a previous project [Bronnenberg] in the area of distributed systems the language POOL (Parallel Object-Oriented Language) was developed. This language was slightly extended to make it suitable as implementation language for a database management system. This language almost completely shields the distribution aspect of the multiprocessor system from the database management system. The question related to this assumption is whether the language POOL provides the right level of abstraction to implement a database management system and whether this implementation can be made efficient. Section 2.2 will elaborate more on this.

To specify distributed processing that is inherent to a database management system it was set up in a strictly *modular* way. At the top level this modular design led to components such as parsers for SQL and PRISMAlog (see below), Query Optimizer, Data Dictionary, Transaction Manager, One-Fragment Manager, and a Data Allocation Manager. All of these components can be assigned to processors rather independently of each other. It is our conviction that fine-grain parallelism is not suitable to implement a database management system; none of the above components has any parallelism in it. The question is whether this *coarse-grain* approach is the right one and what the optimal size is of a relation or fragment to be managed by a One-Fragment Manager.

In Section 3.1 we will come back to these assumptions and corresponding questions.

2.2 The Object-Oriented platform

The main purpose for the platform is to offer a programming paradigm for writing parallel symbolic applications. The DBMS system being a testcase for this platform. The main programming paradigm chosen for the platform was object-orientation. Existing object-oriented languages, like Smalltalk [Goldberg], C++ [Stroustrup], and Eiffel [Meyer1], have shown that the approach is promising for constructing large software systems. Key issues mentioned here are reusability, maintainability and modularity [Brooks, Cox, Meyer2].

Several ways to introduce parallelism into an object-oriented language have been proposed; in-

troduce parallelism as an orthogonal concept, as e.g. in Trellis/Owl [Moss] or allow asynchronous communication between objects, as in actor languages [Hewitt]. The approach taken for our object-oriented platform, called POOL, is that each object is essentially an independent process.

Furthermore, the following features are offered by the platform. The POOL language offers primitives for (synchronous and asynchronous) message passing between objects, however, since each object is an active entity, it determines itself when to answer the messages. In general, messages can have any size. Also, objects can be created dynamically. Since objects can be referred to from all over the system, it is hard to recognize, at POOL programming level, whether it has become redundant. Therefore, a garbage collector is present which will automatically eliminate most of these redundant objects. As a consequence no language features are offered to remove objects explicitly.

In response to the requirements analysis of the DBMS, some facilities have been added to the object-oriented platform. The most important ones are; support for exception handling and error propagation between objects, support for tuples, and support for stable storage. The platform only provides a stable storage medium, where data written to this is guaranteed to survive system crashes. The responsibility to exploit the stable storage facility such that it can be used to recover the state of an application subsequent to a failure, lies with the application itself.

The major question to be answered is whether the language features of POOL can be implemented efficiently enough. In particular, the uniform use of the object from integers, simple record like structures to complicated processes makes it hard to find optimal implementations over the whole range. The key research issues have been efficient scheduling, message passing, garbage collection and optimal allocation control.

2.3 Machine architecture

Two basic assumptions were given for the architecture of the machine. First, it should provide a large amount of memory and processor power. Second, we wanted to investigate parallelism. Therefore we chose a multiprocessor system. In designing such a system, there is constant trade-off between three parameters. These parameters can be used to describe the properties of processor and memory subsystems as is illustrated in Table 1 (including the chosen prototype configuration). Clearly there

Processors	Memory	Prototype
number	number of banks	100 processors, 100 banks
power	size of banks	2 MIPS, 16 Mbyte
connectivity	connectivity	point to point network, non shared memory

Table 1: Parameters of memory and processor system

is an interrelation between choices made in either of the first two parts of the table. As an example the connection machine [Hillis] has chosen for a large number of simple processing elements each element connected to a limited size memory bank, whereas the connectivity of the network between processing nodes is high. In this situation a point to point network exist between processing nodes, therefore, these are called direct connection or distributed memory machine. On the other side of the spectrum we see the development in current super and mini super computers where a small amount (less than 20) of powerful processors are connected towards a small number (less than 20) of memory banks each of considerable size and connected via a low connectivity network. In this example the network is shared between processors and memory elements and, therefore, they are called shared connection or shared memory machine.

Another important issue in designing multiprocessor systems is the flexibility of the system. This is determined by such parameters as:

Extensibility: indicating that it should be easy to add processing power and memory to the system.

Scalability: indicating that an increase in processing and memory should lead to a smooth addition of total system performance.

Modularity: indicating that adding processing or memory should not lead to changes in system homogeneity.

If we take these criteria into account we can observe that shared memory systems score less on the flexibility scale than direct connection machines. Adding processing power to shared memory machines could give rise to step function behavior when the shared resources are exhausted. It is not argued that the extension problem is not present in direct connection machines at all, but because of the point to point connections it is possible to add an almost unlimited number of processing nodes. In practice, the maximum number (degree of the network) is decided during design time. However, this is possible to the cost of a more complex communication problem if processing nodes a non-neighbor.

In case of the PRISMA architecture the following boundary conditions had to be taken into account:

- Optimal support of a virtual Pool machine consisting of a large number of (coarse grain) objects with explicit message passing based communication.
- A strong demand for flexibility in particular homogeneous scalability because we wanted to build a machine with a large (order of 100) number of processing nodes and memory banks. Moreover, the addition of processing power and memory should be smooth.

This leaded us to the conclusion that the architecture for our machine should be a direct connection machine.

2.4 Experimentation

In the previous sections we have formulated the research goals and the boundary conditions of our project. In our research we have chosen for the methodology of performing experiments to verify the correctness of a particular model and to change the implementation dependent on the results obtained. Because of this choice for experimentation we first had to realize the instrumentation of our experimental set-up. Implementing the DBMS on top of the object oriented platform which is built on top of a multiprocessor system was not enough instrumentation to be able to do our experiments. To realize a flexible experimental system, extra facilities were needed at all levels, the DBMS as well as the language implementation and the operating system. Among others they include:

- A more powerful profiler to study the behavior of various software components, such as the objects and message traffic.
- The ability to pass-on object scheduling information from the application towards the O.S., via so-called pragma's offered on the object oriented platform.

With the set of facilities described here it now becomes possible to study the research questions mentioned in Section 2 and give quantitative answers which are a measure for the quality of a particular solution chosen. These results can be the incentive to change part of the implementations or even try another model. In the remainder of the article, we will describe the instrumentation of our experimentation platform, and some preliminary results.

3 Realization and design decisions

Preliminary results and some experiences gained during the construction of our experimental platform are given in the following sections for each of the three layers.

3.1 Database Management System

This subsection discusses the current status of the implementation of the database management system and the preliminary answers to the questions posed in Section 2.1.

3.1.1 Realization

Currently the first version of the database management system, called PRISMA/DB0, runs on the multiprocessor system consisting of 100 nodes. It consist of 35.000 lines of POOL code and is fully documented with a WEB-like documentation system. It consists of the following components: SQL and PRISMAlog parsers, Query Optimizer, Transaction Manager, Concurrency Controller, One-Fragment Manager, Data Dictionary, and Data Allocation Manager (see Fig. 1). For more details on DB0 we refer to [Kersten, Apers1].

The forms of *parallelism* used in the database management system are multi-tasking, pipe-lining, and task spreading. Multi-tasking is used to execute concurrent queries in a parallel fashion if data access allows that. The execution of a query is done in the form of a directed-acyclic graph, where the nodes represent OFMs and the directed edges represent data transmissions. This is done in a pipe-lining manner. As soon as intermediate results are produced, they are sent to the next OFM where they are used as input. Task spreading occurs because the data of the entire database is fragmented and allocated to many OFMs. So, the traditional relational operators can be computed in a distributed manner. It turns out that some of these operators, like sort or join, prevent or diminish the effect of pipe-lining. So, special care has been taken to avoid these operators or to come up with an implementation to improve pipe-lining. In [Wilschut] a hash-based join algorithm is proposed that hashes tuples from both operands as they come in and that computes the output tuples at the same time. It turns out that this join algorithm is slightly more expensive than the traditional hash join algorithm but that it produces its output much earlier, thereby enhancing pipe-lining.

The database management system is responsible for its own *recovery* after hardware crashes using stable storage, a service provided by the object-oriented platform. This means that after a crash the whole database has to be reloaded from disk. The expectation is that the recovery time after a failure is not more than a few minutes. Since disks are spread over the system most of the data can be read from disk into the OFMs in parallel. Currently, various main-memory data structures are studied that make writing to stable storage efficient.

PRISMAlog, a Datalog-like language, is provided as an interface allowing for recursive queries. For the class of *regular recursive queries*, containing the class of linear recursive queries, it was shown that it could be rewritten to a traditional relational algebra program with the transitive closure as an additional operator [Apers2]. Optimization techniques to process the transitive closure of a fragmented relation efficiently have been proposed [Houtsma].

3.1.2 Experience

The prime method to visualize the performance of a database management system is to run a well-known benchmark, such as the Wisconsin benchmark [DeWitt3]. The first series of runs support most of our ideas but they also uncovered several weaknesses in the various layers of the system, which are being corrected now.

Our experiences with the POOL language are that it gave us sufficient facilities to design the database management system in a *modular* way. Furthermore, shielding low level communication from the implementors was one of the most attractive points of POOL. It gave a lot of flexibility in specifying the various forms of parallelism and determining the actual parallelism by allocation pragma's. However, the performance of local computations of the first POOL implementation is rather poor, because every user-defined object is implemented like a process.

First experiments show that the *coarse-grain* approach is the appropriate one. As soon as the system is tuned more extensive experiments will be done to determine the "optimal" fragment size

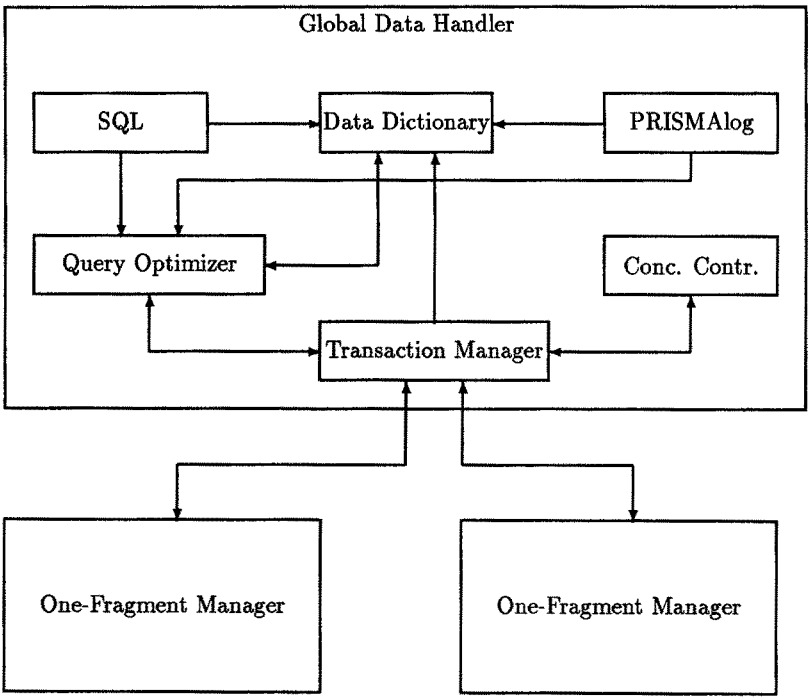

Figure 1: The PRISMA Database Architecture

to gain as much performance as possible given a pool of 100 processors.

In an early stage of the project a C implementation of the OFM has been constructed. This experiment showed that even for small fragments and relatively simple selections the cost incurred by the *compilation* is small compared to the execution time, and almost always results in an improvement [vdBerg].

3.2 The object-oriented platform

This subsection discusses the current state of the implementation of the object oriented platform and the experiences obtained.

3.2.1 Realization

Currently a running implementation is available of the object oriented platform, consisting of a compiler, runtime support, target operating system, and some performance measurements tools.

The POOL compiler is structured in a traditional way; it consists of a frontend, which translates POOL into an intermediate language, and a backend which takes care of the translation into assembly code. The compiler supports separate compilation. The intermediate language is chosen at a relative high level and can be characterized as a declarative typed stack machine combined with procedure, send, answer and create primitives in which it resembles the POOL language itself. In the backend the send, answer, and create primitives are mapped to operating system primitives, possibly via the run time support.

The target operating system [Brandsma] is split into two parts, the nucleus which is POOL independent, and the POOL specific support. The nucleus takes care of the resource management and offers a multi tasking environment as well as some (parameterizable) transport primitives over the network. The operating system itself is also implemented as a collection of communicating processes. All (POOL and operating system) processes share the same address space allowing to implement these as light weight processes and allowing fast exchange of data. The POOL language offers sufficient protection to prevent malicious usage. The memory management is fully implemented in software to allow experimentation with various strategies and to obtain insight into the "typical" allocation behavior of POOL programs. For stable storage the data will be stored twice on different disks, furthermore all disks are attached to different nodes, so no data is lost in case of a single node (disk) crash. The garbage collector which is implemented is a distributed on-the-fly mark and sweep garbage collector [Augusteijn].

3.2.2 Experience

Preliminary experiments with prototype implementations of the object-oriented platform have shown that it is not a trivial task to recognize optimizable objects. A main problem appears to be clustering of objects to increase the granularity of parallelism. Due to the generality of the object and communication constructs, this clustering cannot be done efficiently enough by the compiler for the object-oriented programming language. Therefore the object-oriented platform is extended to support pragma's, to identify objects with a straightforward behavior, such as record like structures. Using these pragma's the compiler can cluster objects and subsequently generate efficient code to be executed on the platform.

The implementation of the clustering pragma's is still under development. Therefore overall performance measurements are not possible, due to the slow local computations.

3.3 Machine architecture

This subsection describes the realized architecture and discusses some results obtained.

3.3.1 Realization

The PRISMA architecture has been realized in four prototype machines. Two smaller prototypes of 4 nodes, one with 8 nodes and one big prototype with 100 nodes. Each node consist of an off the shelf processor board and memory board, connected with the other nodes by a communication processor developed by the DOOM project [Bronnenberg]. The small prototypes are used by the different research institutes involved in PRISMA.

The Communication Processor (CP) takes care of deadlock free routing of packets through the communication network without intervention from the dataprocessor. Each CP has four bi-directional links connected to other CP's, the speed of these (serial) links is 20 Mbit/s in both directions. In addition it has a parallel port to the dataprocessor. The dataprocessor communicates to the network by writing 256 bit packets, containing the destination node, into the CP and by reading the packet at the destination. The CP is capable of adaptive routing, multiple paths to the same destination can be chosen to avoid network congestion areas. This has the consequence that packets can arrive in a different order than they were send. Furthermore the CP is designed for general purpose network routing, it is fully deadlock free, can handle any network with a diameter of up to 8 and connect maximal 4096 nodes.

In the smaller prototypes, the topology of the network (the connections between the CP) is hardwired. The large machine has a 400x400 switch, enabling us to partition machine in one large or more smaller machines. Each machine can be interconnected in a requested topology. The set of topologies that can be chosen is limited to: mesh, torus, chordal ring and an optimal extended chordal ring.

For communication with the host one on every five nodes is equipped with an ethernet board running the standard ethernet protocol layers.

To support fault tolerance, the machine is equipped with disk based permanent storage where the database relations can be stored to survive crashes. The disks are distributed over the machine, half of the nodes each have one 300 Mbyte disk.

An address tracer is constructed to measure the locality behavior of POOL programs as input for future cache support.

3.3.2 Experience

The distributed memory architecture has proven to be a good choice. It has been rather simple to built, because all nodes are equal (except form discs and ethernet) and are connected by an asynchronous network.

The switch gives us the possibility to use the machine optimal (many persons can use small parts to test software), but also to do experiments with network topology, and to measure the effect of the number of processors on the behavior of an application. Each node has 16 Mbyte of memory which allows enough room for experimentation and measurements for a wide range of applications.

The CP is successful although it was more difficult to built than was expected. The adaptive routing capability is currently not used because of the expected overhead at the processor side when packets arrive out of order. Experiments with different topologies have shown no significant differences in performance. For a small 1 packet POOL message only 15% of the time is used by the physical transport layer in the CP.

The optimization of the physical transport layer has been fruitful, but we neglected the interface between the data-processor and the CP itself. The mapping from the POOL messages to the hardware should be implemented more efficient. A separate administration processor between the CP and the data processor is under study [Muller]. Several interface levels are considered, but no choice has been made. One possibility is to make the CP interface stream oriented instead of packet oriented. Another option is that the administration processor takes care of the conversion from POOL messages to packets and vice versa. Still, we also have to keep the flexibility of the system in mind. Would the administration processor prescribe the exact format of messages, it will reduce the possibilities for implementation experiments at the compiler side of the interface. Moreover

the interface would become much more detailed, because it has to include a description of memory management, scheduling, and parts of the system that must be accessed atomically such as message queues.

At this moment, we get some feeling about the behavior of POOL and "typical" POOL programs. We are now able to study their behavior further, and start to design a less flexible, more POOL optimized machine. From the experiments we hope to get the required insight in the communication strategy, caching (memory management) support and possibly dataprocessor support to use.

4 Summary

In this paper we have presented an overview of the PRISMA project. The major research question addressed in this national project is to design, to construct, and to experiment with a highly parallel machine for data and knowledge processing. Among the results obtained are an implementation of a parallel object-oriented language, a hardware platform with efficient communication, and a distributed main-memory relational database system.

A prototype PRISMA/DB system has been implemented and is currently subjected to various detailed performance studies. In particular, new query processing strategies, concurrency control techniques, and implementations of data managers are being developed.

The parallel object-oriented language POOL-X has proven to be a decisive factor in this project. Although the construction of a new programming language (and support environment) requires a lot of manpower, it greatly simplified the construction of parallel programs. In particular, the database system software obtained is highly modular and benefits from the computational and typing model offered by the language.

Since, automatic parallelization of programs is beyond the current horizon of technology, we focused on ways to obtain advisory information from the programmer to arrive at an efficient application program without revealing too much of the language runtime detail. This has resulted in a few orthogonal language features that provide the necessary resource control information to the runtime system.

Lastly, we designed and constructed a shared nothing multiprocessor system. A novel aspect included is a separate communication processor to improve the performance of the interconnect.

The hardware and software platform obtained in the PRISMA project provides a basis for experimental research in many areas of research in distributed processing. Its existence can provide the necessary feedback to analytical and theoretical work in this field.

5 Acknowledgement

The following persons have contributed to the PRISMA project. Carel van den Berg, Marc Bezem, Anton Eliens, Martin Kersten, Louis Kossen, Peter Lucas, Kees van de Meer, Hans Rukkers, Jan Willem Spee, Nanda Verbrugge, and Leonie van de Voort, from Centre for Mathematics and Computer Science. Peter Apers, Herman Balsters, Maurice Houtsma, Jan Flokstra, Paul Grefen, Erik van Kuijk, Rob van der Weg and Annita Wilschut, from University of Twente. Marcel Beemster, Maarten Carels, Sun Chengzheng, Boudewijn Pijlgroms, Bob Hertzberger, Sjaak Koot, Henk Muller and Arthur Veen, from University of Amsterdam. IJsbrand Jan Aalbersberg, Pierre America, Ewout Brandsma, Bert de Brock, Huib Eggenhuisen, Henk van Essen, Herman ter Horst, Ben Hulshof, Jan Martin Jansen, Wouter Jan Lippmann, Sean Morrison, Hans Oerlemans, Juul van der Spek, Marc Vauclair and Marnix Vlot, from Philips Research Laboratories. Piet Rodenburg and Jos Vrancken, from University of Utrecht. George Leih, from University of Leiden.

References

[Aalbersberg1] IJ.J. Aalbersberg, *A Parallel Full-Text Document Retrieval System*, Workshop on Object-Oriented Document Manipulation, Rennes, France, pp. 268–279, May 1989.

[Aalbersberg2] IJ.J. Aalbersberg, F. Sijstermans, *InfoGuide: A Full-Text Document Retrieval System*, International Conference on Database and Expert Systems Applications DEXA 90, Vienna, Austria, Springer Verlag, pp. 12–21, August 1990.

[America1] P. America, *POOL-T — A parallel object-oriented language*, In: Akinori Yonezawa, Mario Tokoro (eds.): Object-Oriented Concurrent Programming, MIT Press, 1987, pp. 199–220, 1987.

[America2] P. America, *Issues in the design of a parallel object oriented language*, Formal Aspects of Computing, Volume 1, number 4, pp. 366–411, 1989.

[America3] P. America, *Language definition of POOL-X*, PRISMA Doc. 350, Philips Research Laboratories, Eindhoven, the Netherlands, November 1989.

[Apers1] P.M.G. Apers, M.L. Kersten, and H.C.M. Oerlemans, *PRISMA Database Machine: A Distributed, Main-Memory Approach*, Proc. Int. Conf. on Extending Database Technology; Venice, 1988.

[Apers2] P.M.G. Apers, M.A.W. Houtsma, and F. Brandse, *Processing Recursive Queries in Relational Algebra*, Proc. IFIP TC2 Working conf. on Knowledge and Data, 1986.

[Augusteijn] L. Augusteijn, *Garbage collection in a distributed environment*, Proc. of Parallel Architectures and Languages in Europe, Eindhoven, the Netherlands, 1987.

[vdBerg] C. v.d. Berg et al., *A Comparison of scanning algorithms*, Proc. Parbase90, Florida, 1990.

[Brandsma] E. Brandsma, Sun Chengzheng, B.J.A. Hulshof, L.O. Hertzberger, A.C.M. Oerlemans, *Overview of the PRISMA Operating System*, Int. Conference on New Generation Computer Systems, 1989.

[Bronnenberg] W. Bronnenberg, L. Nijman, A. Odijk, R. v. Twist, *DOOM: A Decentralized Object-Oriented Machine*, IEEE Micro, October 1987.

[Brooks] F. Brooks, Jr., *No silver bullet - essence and accidents of software engineering*, IEEE Computer, April 1987.

[Cox] B.J. Cox, *Object-Oriented programming; an evolutionary approach*, Addison-Wesley, 1986.

[DeWitt1] D.J. DeWitt, *DIRECT - A Multiprocessor organization for Supporting Relational Database Management*, IEEE Transactions on Computers, Volume C-28, number 6, pp. 395–406, June 1979.

[DeWitt2] D.J. DeWitt, R.H. Katz, K. Olken, L.D. Shapiro, M.R. Stonebraker and D. Wood, *Implementation Techniques for Main Memory Database Systems*, Proc. ACM SIGMOD 1984, pp. 1–8, 1984.

[DeWitt3] D. Bitton, D. DeWitt, and Turbyfill, *Benchmarking Database Systems. A Systematic Approach*, Proc. 9th VLDB, Florence, 1983.

[Moss] J.E. Moss, B. Moss, W. Kohler, *Concurrency features for the Trellis/Owl language*, Proc. ECOOP'87, June 1987.

[Gamma] D.J. DeWitt, *A Performance Analysis of the Gamma Database Machine*, Proc. SIGMOD 1988; Chicago, 1988.

[Gardarin] G. Gardarin, P. Bernadat, N. Temmerman, P. Valduriez and Y. Viemont, *Design of a Multiprocessor Relational Database System*, IFIP World Congress, Paris, Sep. 1983.

[Goldberg] A. Goldberg, D. Robson, *Smalltalk-80, The language and its implementation*, Addison-Wesley, 1983.

[Hewitt] C. Hewitt, *Viewing control structures as patterns of passing messages*, Artificial intelligence, 8:323-364, 1977.

[Hillis] W.D. Hillis, *The connection machine*, MIT Press, 1985.

[Houtsma] M.A.W. Houtsma, P.M.G. Apers, and S. Ceri, *Parallel Computation of Transitive Closure Queries on Fragmented Databases*, submitted for publication, 1989.

[Katuka] T. Katuka, N. Miyazaki, S. Shibayama, H. Yokota, and K. Murakami, *The Design and Implementation of Relational Database Machine Delta*, Proc. of the 4-th Int. Workshop on Database Machines, editors D.J. DeWitt and H. Boral, Springer Verlag, page 13-34, 1985.

[Kersten] M.L. Kersten et al., *A Distributed Main-Memory Database Machine*, Proc. 5th IWDM; Japan, 1987.

[Leland] M.D.P. Leland and W.D. Roome, *The Silicon Database Machine*, in Proc. of the 4-th Int. Workshop on Database Machines, editors D.J. DeWitt and H. Boral, Springer Verlag, page 169-189, 1985.

[Meyer1] B. Meyer, *Eiffel: A language and environment for software engineering*, Report TR-EI-2/BR, Interactive Software Engineering Inc., 1987.

[Meyer2] B. Meyer, *Object-oriented software construction*, Prentice-Hall, 1988.

[Molina] H. Garcia-Molina, R.J. Lipton and P. Honeyman, *A Massive Memory Database System*, Techn. Report 314, Dep. of Comp Sci. Princeton Univ., Sep 1983.

[Muller] H. Muller, *Mixed level simulation of the POOMA architecture*, Submitted for publication, March 1990.

[Ozkarahan] E.A. Ozkarahan, S.A. Schuster and K.C. Smith, *RAP- An Associative Processor for Database Management*, Proceedings of the National Computer Conference, Volume 45, page 379-387, 1975.

[Rodenburg] P. Rodenburg, J. Vrancken, *Parallel Object-Oriented Term Rewriting: The Booleans*, , 1988.

[Shaw] D. Shaw, *Knowledge-Based Retrieval on a Relational Database Machine*, Ph.D. Department of Computer Science, Stanford University, 1980.

[Stroustrup] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[Wilschut] A.N. Wilschut and P.M.G. Apers, *Pipelining in Query Execution*, Proc. Parbase, 1990.