PiSMA: A Parallel VSM Architecture

by Dimitris Lioupis, Andreas Pipis, Maria Smirli, and Michael Stefanidakis

Introduction

There is an increased interest in large scale multiprocessors composed of a large number of processing elements. The two main models of multiprocessor architectures, the shared memory model [10] and the distributed memory model [1, 2, 3, 4, 11], suffer from serious limitations because of their nature. Distributed memory architectures promise better scalability and increased performance but are more difficult to program efficiently. Shared memory architectures, on the other hand, have a better programming model. Their disadvantage, however, is the limited bandwidth of the shared bus, which restricts scalability.

This paper presents a parallel computer architecture, called PiSMA (Parallel vIrtually Shared Memory Architecture), introduced in [5, 6]. PiSMA architecture combines the benefits of both multiprocessor models mentioned above, resulting in an efficient parallel architecture. The simplicity of the underlying hardware permits the scaling of PiSMA, utilizing new generations of microprocessors, without the need to redesign the whole system. PiSMA can be used with or without a global communication network depending on the type of applications it is used for, as described in [9]. In the following sections a detailed description of the PiSMA architecture, its programming model, the **Virtual Memory management** and **message passing** support mechanisms are presented.

Description of the PiSMA Architecture

The PiSMA architecture forms an expandable toroidal grid with alternating processors and memories: each processor is connected to four memories and each memory to four processors as shown in Figure 1.

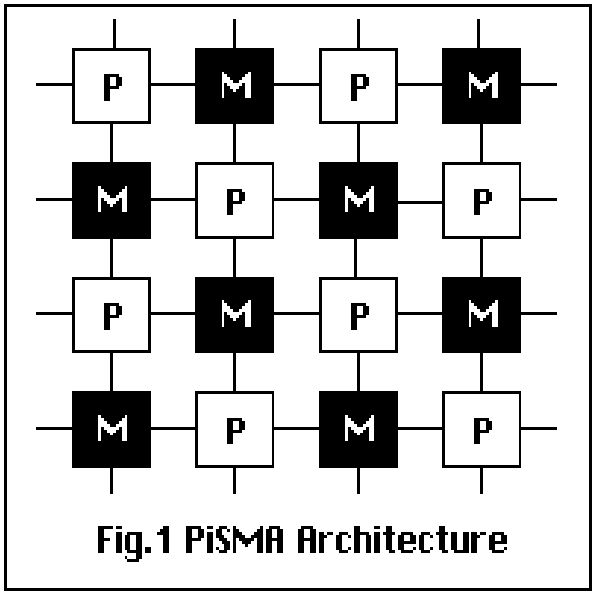

Fig.1 PiSMA Architecture

This structure enables each processor to communicate directly (through a common memory) with up to eight neighboring processors as shown in Figure 2.

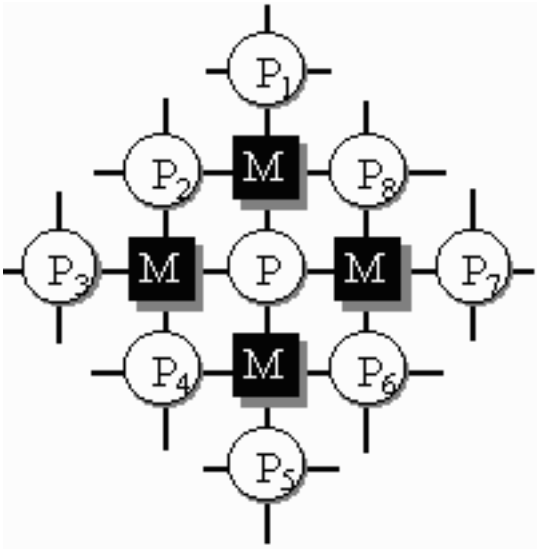

Fig.2 A processor and its eight neighbors

Communication with any other processor beyond these eight processors on the grid is performed by *message passing* which is transparent to the user. The four ports to memory are implemented as four caches connected on the same bus with the memory module as depicted in Figure 3.

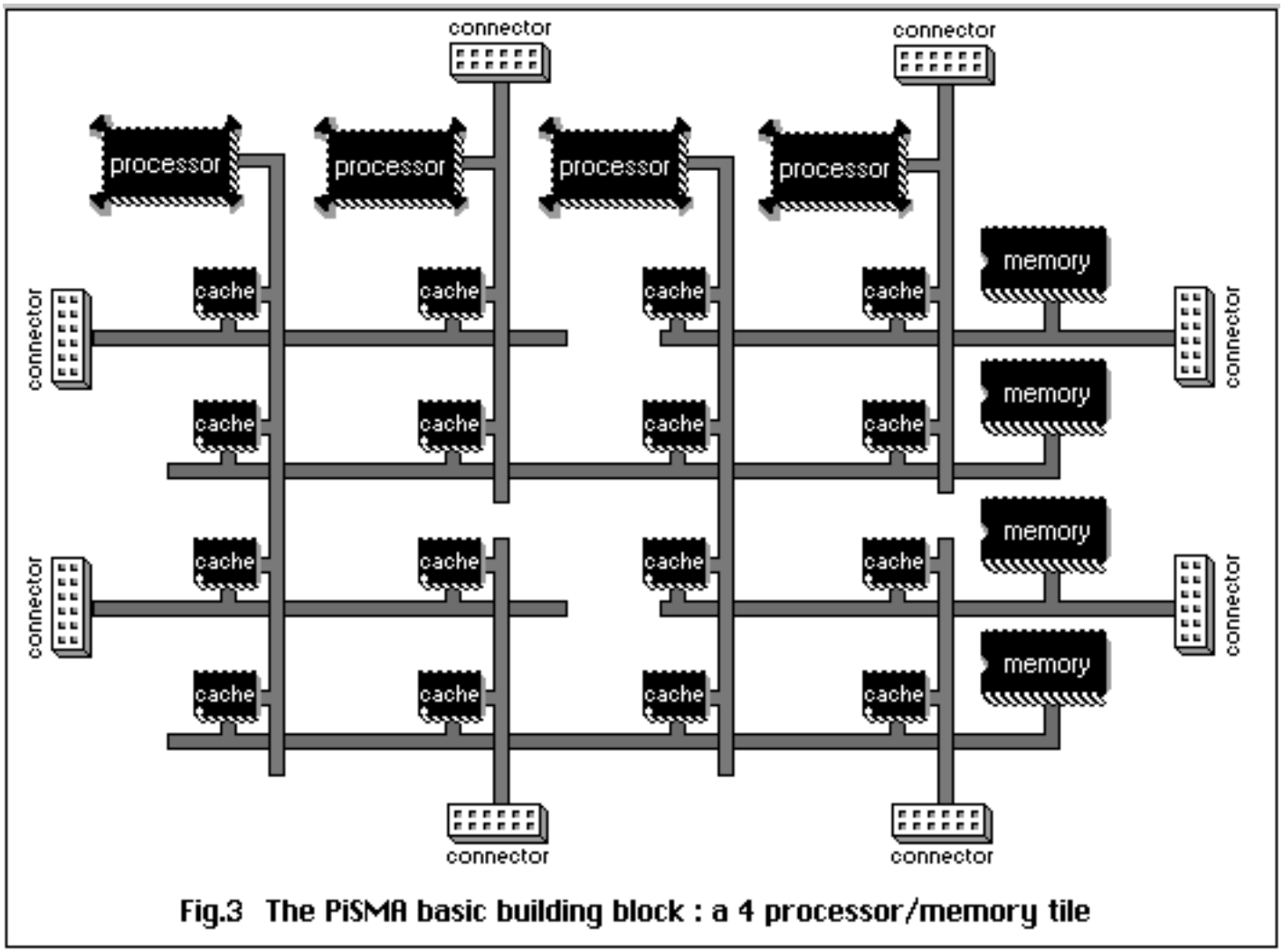

Fig.3 The PiSMA basic building block : a 4 processor/memory tile

The processor-memory cluster consists of two basic boards, namely the processor board and the memory board. The processor board has two versions implementing the desired connectivity and allowing for easy expansion. The memory board contains four memory modules. By connecting the top and the bottom connector and the left and the right connector, the smallest configuration possible (four processors and four memories) is formed. The physical implementation of the architecture is described in detail in [5]. PiSMA is easily scalable, and bigger configurations can be easily constructed by repeatedly connecting together the basic building blocks shown in Figure 3.

A very important aspect of the basic PiSMA architecture is that no network for intra-processor communication exists. Memory modules are used as a communication media instead, with messages being transferred to their destinations via consecutive memory-to-memory copying. This copy operation is performed by processors common to each pair of memories. It was shown in [9] that this approach works well for a wide range of parallel applications and that the addition of a global communication network does not improve performance significantly. Moreover, the absence of a network has the advantage of easier scalability. In the case of applications with heavy global communication, however, the presence of an intra-processor communication network does improve the system's performance. In order to satisfy application demands, a network can be easily incorporated into the basic PiSMA hardware, as shown in [7], where PiSMA was used to execute special purpose applications such as

Video on Demand.

Fault tolerance is another important feature of PiSMA architecture; this capability comes from its structure shown in Figure 3, that provides alternative paths that can be used when a component (processor or memory) fails, as described in [9].

Programming model

Programs are executed on PiSMA by mapping the execution tree of a parallel program onto the grid of processors. The code is loaded in a memory module and the root of the tree (which is the first call to the main procedure) is randomly assigned to one of the adjacent processors. This processor starts executing the code and spreads work out dynamically to its neighbors, through its four adjacent memories. The work distribution is performed by the **dynamic load balancer**, described in detail in [8]. Distributing work consists of copying a work granule into a processor's adjacent memory and inserting an entry into its task queue. A work granule is a self-contained piece of work (typically 50-500 lines of code). Each work granule consists of:

1. a header, indicating the resources required for execution of this particular work granule
2. the body, which contains the code to be executed.

The dynamic load balancing algorithm is a key element in our architecture. This algorithm attempts to evenly distribute the work granules on the processing surface, avoiding, as much as possible, data transfers and creation of long communication paths in shared data access. In order to achieve these conflicting goals, the load balancing algorithm draws on information included in the header of each work granule (such as code size and shared variable dependencies) to decide which neighboring processor is best suited to be assigned a piece of work. This decision is also based on data dependencies and each processor's load.

Virtual Memory Management

The virtual memory management mechanism of the PiSMA architecture supports the mixed addressing mode (direct access and message passing) of the architecture. The selection of the appropriate addressing mode is completely dynamic and transparent to the application programmer. The dynamic nature of this mechanism imposes the fact that resources' physical locations on PiSMA are unknown until runtime.

The PiSMA programming model distinguishes between shared and private data. Private data are expected to reside in one of the four memories adjacent to the executing processor and are referenced relatively through an offset. Referencing private variables is straightforward; the processor presents the real address on its bus and reads from the appropriate memory. There is no need for special hardware to perform this transaction; the processor sees its four neighboring memories as a uniform and contiguous

memory space. Any compiler that references data locally (as the majority of compilers do) into the data segment can be used to handle this kind of data.

Shared data may or may not be in an adjacent memory and are always referenced through **indirection pointers**. When the application program executes, shared data may be directly accessible (if they reside in an adjacent memory to the executing processor memory), or they may be remotely accessed (via message passing). Therefore it is not possible to directly reference this kind of data in an absolute manner, since its location in the system before runtime is unknown.

The application compiler generates the appropriate code to support this accessing scheme. For each shared resource in the application program the respective indirection pointers are allocated. After that, shared resources are referenced through these pointers.

The contents of indirection pointers are undefined at compile time. At runtime (specifically, during the mapping of the work onto the PiSMA grid), the location of shared variables is established when they are used for the first time and the associated indirection pointers are initialized.

When a shared variable is directly accessible by a processor, the corresponding indirection pointer contains its physical address in one of the four adjacent memories. The executing processor simply executes an indirect access to its physically connected memory space. In this way there is no special hardware or software intervention; shared data can be accessed as local data.

This is not the case when the shared data resources are remote to the executing processor. The value contained in the indirection pointer in this case is a special memory address, designated to generate an OS trap. The indirection pointer contains additional information about the shared data location in the system, such as the physical memory module, the address and usage status of this particular shared resource. The generated OS trap service routine translates the request into a remote-data requesting message, utilizing the resource's location information from the indirection pointer.

It should be clear that the PiSMA architecture does not utilize a pure global virtual memory address space scheme. That is, the addresses generated describe uniquely a physical location in the whole virtual memory space and a translation hardware mechanism exists to map these addresses into physical space. Instead, PiSMA uses physical addresses when accessing data directly, and relies on a software translation mechanism to generate messages for remote data requests.

PiSMA does require that its data have a fixed physical location in the system's memory when it is used, as it is a **non-Cache Only Memory architecture** [11]. The fixed location of data resources in use avoids data coherency problems, and requires no global data invalidation messages.

Message Passing Support

Messages generated from the operating system are forwarded to their destination through system memory. Each processor has a message queue defined in each of its four adjacent memories. In these queues the eight neighboring processors store the messages, whose recipient is this particular processor. During the processor's normal operation, an interrupt driven software message handler queries the queues and services the pending messages.

Messages are forwarded to their destination by **packet switching** from processor to processor. Each message is equipped with a header describing its type, its destination processor's location on the grid, and its originator (if this is necessary). The message route through the processor grid is not pre-established when the message is generated. Each processor, upon receiving a message, decides to forward it to one of its eight neighboring processors, or to consume it itself. The forwarding direction is found by comparing the destination processor's location, from the message header, with the forwarding processor's own location (it is assumed that each processor has the knowledge of its own location).

Message passing always imposes high latency to the application's execution. Messages kept in main memory are uncachable because of their nature, and message reading or writing operations take many execution cycles. The time spent for message handling on each processor is another source of overhead, delaying the application execution. The PiSMA architecture tries to minimize the message passing latency by avoiding it or hiding it. In the first case, the advantage of the physical communication through common adjacent memories is used, avoiding the generation of messages. Processors working with shared resources on a common memory can execute very quickly, without message intervention. It is the responsibility of the diffusion algorithm to assign work granules with common resources to neighboring processors, but efficient distribution of processes is not always possible. Sometimes the diffusion algorithm has insufficient information to perform an optimal job distribution, and at other times the parallel application does not allow the distribution of its data set in different system memories. The result is an increase in message traffic (which could be excessive). In order to minimize the impact of such a situation, the operating system of PiSMA is designed to swap out processes waiting for remote data. In this way the processor can execute something useful, thus hiding wait time.

Conclusion

PiSMA is a Virtually Shared Memory Architecture that combines both the benefits of the shared and distributed models. A detailed description of the PiSMA Architecture, its Virtual Memory Management and Message Passing support was presented and it was shown that parallel programming in PiSMA is an easy task, without the need for knowledge of the underlying hardware. All these details are transparent to the programmer since the operating system kernel of PiSMA takes care of them. The encapsulation of intelligent dynamic algorithms in the operating system kernel, such as the diffusion algorithm and the dynamic memory management, makes PiSMA a powerful parallel distributed memory computer with an easy programming interface compared to that of the pure shared memory architectures.

References

1

A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, D. Yeung, The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Int'l Symp. on Computer Architectures*, Los Alamitos, California 1995

2

S. Borkar et al. iWARP: An Integrated Solution to High Speed Parallel Computing. In *Proceedings of Supercomputing'88*, Nov. 1988.

3

J.T. Kuehn, B.J. Smith, The HORIZON Supercomputing System: Architecture and Software. In *Proceedings of Supercomputing'88*, Nov. 1988.

4

D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowit, and M.S. Lam. The Stanford DASH multiprocessor. *IEEE Computer J.*, 25(3), pp 63-79, March 1992.

5

D. Lioupis, N. Kanellopoulos, CHESS Multiprocessor: A Processor-Memory Grid for Parallel Programming. In *Cache and Interconnect Architectures in Multiprocessors*, edited by M. Dubois & T. Thakard, Kluwer Academic Publishers, pp245-257, June 1989.

6

D. Lioupis, N. Kanellopoulos, M. Stefanidakis, The Memory Hierarchy of the CHESS Computer. *Microprocessing and Microprogramming (JSA)* (vol 38), pp 99-107, Sept 1993.

7

D. Lioupis, A. Pipis, M. Smirli, N. Kanellopoulos, Architecture of a VSM Parallel Video-on-demand server. Appeared in the *special session on Communication and Computing for Distributed Multimedia Systems of the 10th International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, Nevada, October 28-31, 1998

8

D. Lioupis, M. Stefanidakis, Dynamic Load Balancing on a Virtually-Shared Memory Parallel Computer System. In *Proceedings of the 6th International PARLE Conference*, pp813-819, Athens, July 1994.

9

D. Lioupis, A. Pipis, M. Stefanidakis, PiSMA: An Upgradeable Fault Tolerant Approach to Parallel Processing. In *Proceedings of the 4th IEEE International Conference on High Performance Computing*, pp277-283, Bangalore India, Dec 1997.

10

C. Thaker, L. Stewart, Firefly: A Multiprocessor Workstation. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, October 1987

11

D. H. D. Warren and S. Haridi, The Data Diffusion Machine - A Scalable Shared Virtual Memory Multiprocessor, In *Proceedings of the 1988 Int. Conf. on Fifth Generation Computer Systems*, pp 943-952, Tokyo, Japan , Dec. 1988

Dimitris Lioupis (lioupis@cti.gr) is an Assistant Professor at the Department of Computer Engineering

and Informatics, University of Patras, Greece. Andreas Pipis (pipis@cti.gr), Maria Smirli (smirli@cti.gr) and Michael Stefanidakis (mistral@cti.gr) are PhD students at the Department of Computer Engineering and Informatics, University of Patras, Greece.