# The Next Generation Parallel Architecture: Multiple Executing Threads

W.K. Giloi and W.Schroeder-Preikschat

GMD Research Institute for Computer Architecture and Software Technology
Rudower Chaussee 5, 12489 Berlin, Germany
e-mail: <w.giloi(a)computer.org> , <wosch@first.gmd.de>

## Abstract

Future computers will work with parallel engines built with several processors on one chip. Such computers will be scalable from workstations with a few processors to massively parallel computers with a large number of processors. The overall architecture will be multi-threaded, yet because of the multiprocessor node several threads may be active at the same time. We call such an architecture Multiple Executing Thread Architecture (META). In META, application programming will be facilitated by parallelizing compilers. The paper presents the rationale for and the concepts of META. The specific features that make META an advancement of the usual multi-threaded architecture (MTA) are discussed. The realization of a high-performance META node architecture is presented, and it is shown how META is supported by a modular, parallel operating system kernel that leads to maximum latency hiding and the efficient continuation of thread execution by reactive objects.

## 1. Introduction

The newest development in parallel computer architecture is the multi-threaded architecture (MTA). MTAs are potentially massively parallel and exhibit two important properties that facilitate the task of a parallelizing compiler. (i) The execution of the threads is data driven, i.e., performed by the runtime system. Thus, the compiler need not schedule them. (ii) MTAs lead to a very effectively latency hiding. Consequently, expensive hardware facilities for latency minimization are not necessary, and the need for optimal data distribution is less stringent.

MTAs are characterized by a large number of lightweight processes called threads. The execution of a thread is triggered by some firing rule semantics similar to data flow. Applied to communicating threads, this means that any request for a communication, e.g., access of an object residing in another node, leads automatically to the suspension of the executing thread, to continue only after the communication has been completed. Hence, a thread extends from a communication activity to the next one. While a thread is suspended, other threads that are ready for execution may be processed.

The programming model of MTAs is seen primarily as that of shared-memory. However, MTAs need not be associated with the shared-memory paradigm. Since the shared memory model leads to fine-grained threads, it results in a large number of threads and, thus, context switches. Employing a coarser thread granularity reduces the number of context switches as well as the cost of communication. Coarser threads are obtained by combining the MTA principle with message-passing.

In this paper we present the concepts of a novel multi-threaded message-passing architecture called META (multiple executing thread architecture). Chapter 2 discusses related work in MTA research and presents the rationale for META. Chapter 3 deals with a basic feature of MTAs, the separation of computation and communication/synchronization. Chapter 4 discusses the architecture of the META node. Chapter 5 outlines the unique operating system and runtime system architecture which is the focal point of the META architecture.

## 2. Related Work in Multi-threaded Architecture

### 2.1 The Basic Concepts of Multi-Threaded Architectures

Multi-threading is a twenty year old concept [Smi 78] characterized by the following features:

* *Latency hiding by split transactions*: When a thread performs a load from the shared memory, it is suspended and rescheduled for execution after the load has been completed.
* *Low context switching overhead*: Since multi-threading with split transactions leads to frequent context switches, measures must be taken to minimize the time for it. This can be achieved by providing multiple register sets [Smi 78], however, the need for using "stock processors", leaves only the approach of reducing the cost of context switching by having threads with minimal context.

Current MTA research has been fertilized by over a decade of research of dynamic (tagged-token) data flow architectures [AaK 81], [PaC 90]. The main objective of all MTA designs is to provide effective latency hiding. This is achieved by the split transaction mechanism that takes place whenever a communication through the interconnect occurs. Consequently, a thread is the piece of code between two consecutive communications.

### 2.2 Overlapping of Computation and Synchronization

Current MTA concepts apply the principle of overlapping computation and synchronization by having two processors in the node: (1) the *data processor*, to perform the computations of the application program, and (2) the *synchronization processor* [NPA 92]. The main advantage of providing a dedicated resource for synchronization is that it avoids interrupting the program execution. The synchronization processor performs only a few simple operations. Therefore, it usually is envisioned as a custom design.

The idea of a separate communication processor (to use a more general term) is not new either. It was applied for the first time in the UPPER computer [GaB 81], an early distributed memory parallel architecture developed in 1979-81 at the Technical University of Berlin. The UPPER node had two MC68000 processors, one working as node CPU, and the other handling the message-passing protocols.

### 2.3 Related Work: *T

The probably best known work in MTA so far has been the conceptual development conducted at MIT under the name *T [NPA 92]. *T is the concept of a fine grain multi-threaded architecture, designed to provide a global address space while effectively hiding the latency of remote memory accesses. The physical *T machine is envisioned as a distributed shared memory architecture. Hence, there are loads to local memory and loads to the memory of other nodes (called *rload*). During execution, each *rload* leads to the creation of a thread that performs it. This happens through a *fork* operation. When all concurrent *rloads* have been performed, a corresponding *join* continues the suspended computation in the manner of a barrier synchronization. The context under which execution will be resumed, in short called "continuation", is sent to the remote memory and comes back from there together with the value to be loaded. The memories for shared variables are I-structure stores [AaT 80], thus allowing a *read-before write*. The thread management functions as well as the *send* and **receive** operations are performed by the synchronization processor. For remote memory accesses, the *T node contains a specific "Remote Memory Request Coprocessor."

The fine-grain MTA concept as pursued in *T has the disadvantage of a large number of context switches [AAD 94]. This is a price to be paid for the support of the dataflow language Id [NPA 92]. In contrast, META is designed to support a high-level, procedural programming paradigm which is based on the exploitation of object parallelism with a high degree of spatial and temporal homogeneity [GaS 95]. For such a programming model the large-grain MTA approach is perfectly adequate.

## 3. Multiple Executing Thread Architecture (META)

### 3.1 Prerequisite for Efficient Latency Hiding

The separation of computation and communication is an issue of node architecture, as it requires two processors, the *application processor* (AP) and the *communication processor* (CP) [GaS 89]. However, experiences with the AP-CP scheme have taught us that it is equally important to have the appropriate operating system architecture [BGS 94]. In that respect, all commercial parallel computers that use that scheme fail to utilize it to its full potential. This is evidenced by Table 1 which lists the latency of a single communication activity (send, receive) measured on a number of machines. With the latency hiding mechanisms provided by MANNA's two-processor node architecture [Gil 95] with two i860XP processors and the PEACE operating system, the effective latency is about 1...4 microseconds, ie., as low as if communication were performed by naked hardware. The latency of all other machines is at least an order of magnitude higher.

| Machine | Operating System | Latency [μs] |
|---|---|---|
| Paragon XP/S | OSF/1 | 240 |
| nCUBE/2 | PUMA | 110 |
| iPSC860 | NX | 100 |
| CM-5 | CMOST | 65 |
| Paragon* | PUMA | 50 |
| nCUBE/2 | active messages | 32 |
| Paragon** | PUMA | 30 |
| MANNA* | PEACE | 23 |
| MANNA** | PEACE | 4...1 |

**Table 1** Message-passing latency of parallel machines

PEACE differs from the microkernel operating systems (most operating systems listed in Table 1 are of that type) by two unique features [Sch 91], [Sch 95]. (i) It provides a family of kernels which optimally support specific user demands and operating modes. Thus, the user pays only for the system functions he wants. This is achieved by having a simple communication kernel as minimal basis that may be augmented for the specific use by minimal function extensions. (ii) Most family members differ only in a few components that concern the user interface or the runtime behavior; therefore, object-orientation is the natural approach for implementing the kernel family.

This principle not only leads to extremely lightweight operating system kernels but also to their further modularization—in contrast to the microkernel operating systems whose kernel is monolithic. The PEACE kernel consists of a hierarchy of three modules [Sch91]: NICE (Network Independent Communication Environment), COSY (Communication System), and POD (Port Drivers). NICE takes care of the inter-thread communication policies, COSY handles the communication protocols, and POD is the glue between the operating system nucleus and the communication hardware (the network).

On the MANNA two-processor node the modularization of the PEACE kernel allows for an optimal distribution of operating system functions over the two processors as illustrated in Figure 1. The application processor (AP) executes only NICE functions, while the complete NICE-COSY-POD suite is executed only on the communication processor (CP). On the CP this may take 23 microseconds, yet the AP "sees" only the couple of microseconds needed to put a communication request to the CP. Compared to the microkernel approach, the advantages of this innovative concept are (I) an order of magnitude less code, (ii) an order of magnitude lower start-up time, and, consequently (iii) very efficient latency hiding.

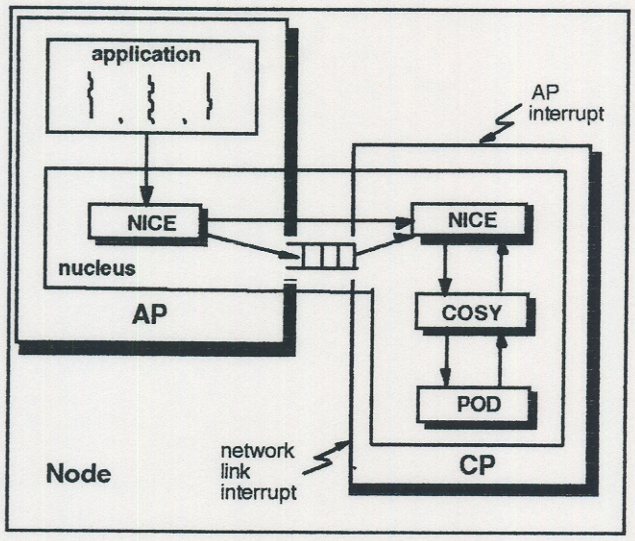

**Figure 1** Distribution of kernel functions over AP and CP

### 3.2 Multiple Executing Threads

In the "classical" multi-threaded architecture many threads may be ready for execution, but only one of them can be worked on at any point in time. At best some synchronization activities are conducted in parallel to the one active computation thread. Yet the scheme discussed in the previous section, the separation of computation and communication, need not be restricted to only two threads. Rather, more threads can be executed in parallel, provided the node has more than two processors. In that case, the runtime system may determine which threads do computation and which communication.

## 4. META Hardware Architecture

### 4.1 The Mandatory Use of Stock Processors

It was mentioned above that almost all concepts for fine-grain multi-threaded architectures envision a dedicated, custom-designed *synchronization processor* (SP), preferably on one chip with the *data processor* (DP). Custom-designing the node hardware of a distributed memory architecture may allow for an innovative architecture but will hardly lead to a competitive product because the cost of design and continuing upgrades of a complex superscalar processor of the highest possible performance are so exorbitant that they could not be amortized by the relatively small volume going into parallel computers.

Processor lines must reach a production volume of millions per year in order to be profitable, a volume that comes from PCs, workstations, and embedded systems. Using a stock processor offers the additional advantage that the design is automatically on any future upgrade path of that device. On the other hand, there is no necessity to have the processor and the additional communication and synchronization hardware on one chip, for that additional functionality may as well be designed into supplementary ASICs.

## 4.2 META Node Architecture

Figure 2 shows a block diagram of the 4-processor-node of the basic building block for a META system. The core of the design is the Data Path Switch (DPS) realized by 3 ASICs (gate arrays). The DPS provides the following functions and facilities:

* crossbar interconnection between the four processors and the four memory banks;
* all needed send and receive queues, including queue management based on the monitoring of full-empty bits;
* snooping of the data traffic between processors and memory needed to maintain the coherence of the processor caches.

The node memory consists of 128 Mbytes of DRAM; the memory word has the width of a cache line. Using EDRAM results in a 4-1-1-1 access pattern. The interconnection network interface (INI) provides two bi-directional links into the interconnect or other components of the system. The INI works synchronously with the bus clock of the processors, i.e., at about 75...100 MHz. As the INI is byte-wide, this amounts to transmission rate of 150...200 Mbytes per second per bi-directional link. For interconnection network we will use the same hierarchy-of-crossbar topologies as in the MANNA computer [Gil 95]. The processors of the META node share the local node memory. Since shared memory communication has a lower latency than communication via the interconnect, the sustained performance of the overall system increases over-linearly with the numbers of processors in the node.

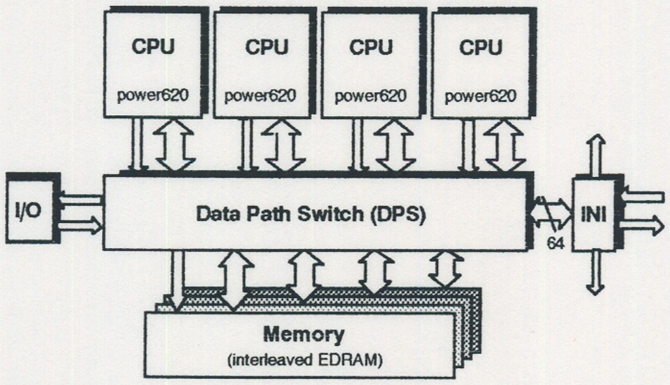

**Figure 2** Block diagram of the META node

## 4.3 Interconnection Networks

In principle, any interconnection topology could be employed. Because of the outstanding latency hiding behavior of META, one need not necessarily resort to expensive latency minimization measures. However, in a general-purpose computer it is not advisable to have an interconnect such as the mesh that works well only for applications that exhibit a strongly local access pattern. The *hierarchy of crossbar interconnect* of MANNA [Mon 89] provides here "the best of all worlds" by combining high global connectivity, low blocking probability, and minimal hardware cost.

The basic building block of this unique, scalable topology is a byte-wide, bi-directional 16x16-crossbar switch realized as a single-chip ASIC. The transmission rate is the same as that of the INI. Groups of 10 nodes are interconnected via the 16x16 crossbar, to form *clusters*, each cluster thus having 32 processors. This leaves 6 "free" links of the cluster crossbar for interconnecting with other clusters. Applying this principle recursively allows for realizing massively parallel machines of almost any system size.

## 5. Operating System Support for META

### 5.1 Inter Thread Communication (ITC)

In META a thread reaches from a blocking communication construct to the next one, while non-blocking communication activities do not interrupt a thread. In order to minimize the number of context switches, it is advisable to employ a "no-wait send" ITC protocol. Such a protocol normally requires buffering to ensure data consistency. However, the *synchronized no-wait send* (SNWS) scheme of META avoids the need for buffering. Moreover, SNWS together with the complementary *prefetch blocking receive* (PBR) scheme maximizes the overlap of computation and communication in the multiprocessor node [BGS 94].

### 5.3 Scheduling Threads

Threads are the units of execution. Each thread possesses its individual runtime context defined as the state of the registers of the executing CPU. The context specifies the current focus of control (instruction pointer register, IP) and activation (stack pointer register, SP). The other CPU registers are used as either temporary or global data stores.

Switching between threads means exchanging contents of CPU registers. This is done by the *resume()* primitive. The overhead for a thread switch is largely determined by the number of CPU registers to be exchanged. C or C++ compilers usually distinguish between two (logical) CPU register sets: non-volatile and volatile registers. The non-volatile register set contains the IP, SP and all the CPU registers used to store global data and must be kept consistent across procedure calls. Thus, if *resume()* is called, only the non-volatile register set must be saved and restored to perform the thread switch. Only an interrupt could destroy the contents of volatile registers. Whenever the compiler needs a register storage, e.g., for intermediate results or function returns, the volatile register bank is used. However, the same may happen when compiling interrupt handling code. The compiler cannot know in advance when an interrupt will occur and, hence, is unable to keep the volatile register set consistent. Therefore, the *interrupt handler* must save or restore the volatile registers. This separation of concern (non-volatile vs. volatile register set) guarantees a low-overhead context switch.

In META, thread scheduling is the means of multiplexing the CPUs of the multiprocessor node between several ready-to-run threads so that latency hiding works at its best. A thread issuing a send or receive operation on an empty mailbox remains unscheduled in favor of the ready-to-run thread that is next in line. Threads may wait for external events (e.g., the receiving of a message sent by a remote thread) or for internal events (e.g., the finishing of a send operation). The occurrence of these events may cause threads to switch from blocked to ready-to-run. A special kind, called *reactive thread*, is employed to promptly serve the event, even in the case that all CPUs of a node are busy processing threads. Reactive threads will be started immediately after the blocked state has ceased to exist.

### 5.4 Continuation by Reactive Objects

The combination of signalling message-receive events with starting reactive threads in META constitutes an important variation of the concept of *active messages* [Eea 92]. The active message approach allocates a user-level *message handler* for the processing of a communication interrupt. The aim is to receive directly data values via a communication channel and assign them without intermediate buffering to certain program variables. Similarly, when requested by a remote task the contents of some program variable should be directly transferred via a communication channel. This model implies that a message handler either receives or sends data concurrently with the execution of some thread. Therefore, the computation thread is not required to explicitly issue communication requests to the operating system kernel; rather, data are automatically placed into the program variables. A message handler can be viewed as a special communication thread whose sole function is to keep the computation thread running.

The distinction of computation thread and message handler in the active message scheme has the following consequence. Firstly, it suggests to give the message handler its own execution context by making it a separate thread. Secondly, this thread becomes activated only when a communication interrupt occurs. The communication thread will then request from the scheduler exclusion from normal scheduling, thus enabling interrupt-driven activation. Communication thread processing is therefore concerned with executing the message handler and accepting the next communication interrupt.

Rather than taking the active message approach—to have a message handler assigned to an active object (a thread) which, in turn, activates a computation thread—one can employ a more efficient mechanism, namely the activation of a computation thread directly by making it a *reactive object*. A reactive object is a thread that provides a runtime context for a user-level interrupt handler. The reactive object encapsulates the message handler and acts as the user-level continuation of a communication interrupt.

The resumption of a reactive object simply requires the activation of the non-volatile register set of the thread. No other state information need be saved or restored. The thread implementing the reactive object returns from the system and executes the message handler. In contrast to the active message approach, no stack frame need be explicitly set up to perform the upcall. Rather, the stack frame is automatically established whenever the reactive object accepts (i.e., synchronizes on) the next communication interrupt, storing an activation record that establishes the proper return path.

In this scheme a synchronization call is just a request to poll for interrupts. To this end, the reactive object provides a runtime context that enables the message handler to do the polling. Consequently, the upcall to the message handler executes faster than in the active message approach. In both cases is it the responsibility of the runtime environment of the message handler to enable user-level interrupt handling. However, the reactive object scheme does not require the dynamic creation of a runtime context (stack frame) for the message handler. Rather, the system-level interrupt handler activates merely an already existing thread with an existing context.

## Conclusion

Compared to the common MTAs, META offers the following advantages.

* META's runtime system can determine which of the parallel threads perform computation and which communication, to balance computation and communication optimally.
* META provides an extremely efficient latency hiding scheme.
* META is a concept designed for the use of stock processors; consequently, all communication and scheduling activities are performed at the periphery of the processors jointly by supplementary hardware (the DPS) as well as the operating system kernel.
* The novel kernel family concept of META's operating system allows it to perform those tasks almost with the speed of pure hardware.

* Since META is based on the message-passing rather than the shared-memory model, threads are of coarser grain than in most MTA concepts, thus reducing the frequency of context switches as well as the cost of communication.
* META allows for new kinds of compiler optimizations and supports strongly automatic parallelization by the compiler.
* META will optimally utilize the 4-processor chips which will come to exist in a the near future and provide the power of four processors almost for the price of one.

## References

[AAD 94] Ang B.S., Arvind, Chiou D.: StarT the Next Generation: Integrating Global Caches and Dataflow Architecture, MIT Laboratory for Computer Science, Computational Structures Group Memo 354 (Feb. 1994)

[AaK 81] Arvind, Kathail V.: A Multiple Processor Dataflow Machine That Supports Generalised Procedures, *Proc. 8th Annual Symp. on Computer Architecture* (1981), 291-302

[AaT 80] Arvind, Thomas R.E.: I-Structures: An Efficient Data Type for Functional Languages, MIT Laboratory for Computer Science, Computational Structures Group Memo 178

[BGS 94] Bruening U., Giloi W.K., Schroeder-Preikschat W.: Latency Hiding in Message-Passing Architecture, Proc. IPPS '94, IEEE-CS Press 1994

[Eea 92]von Eicken T., Culler D.E., Goldstein S.C., Schuster K.E.: Active Messages : A Mechanism for Integrated Communication and Computation, Tech. Report UCB/CSD 92/675, UC Berkeley 1992

[GaB 81] Giloi W.K., Behr P.: *An IPC Protocol and Its Hardware Realization For High-Speed Distributed Multi-computer System*, *Proc. 8th Internat. Sympos. on Computer Architecture*, IEEE Cat. no. 81CH1593-3, 481-494

[GaS 89] Giloi W.K., Schroeder W.: Very High Speed Communication in Large MIMD Supercomputers, *Proc. ICS '89*, ACM order no. 415891, 313-321

[GaS 93] Giloi W.K., Schramm A.: PROMOTER — An Application-oriented Programming Model for Massive Parallelism, in Giloi W.K., Jaehnichen S., Shriver B.D.(eds.): Massively Parallel Programming Models, Proc. Internat. MPPM Conf., 1993, IEEE-CS Press, order no. 4900-02, 198-205

[Gil 95] Giloi W.K.: Towards the Next Generation Parallel Computers: MANNA and META, Proc. ZEUS '95, Linkoeping, Sweden, 1995

[Mon 89] Montenegro S.: Kommunikationsstrukturen fuer verteilte Rechnersysteme, PhD thesis, Technical University of Berlin 1989

[NPA 92] Nikhil R.S., Papadopoulos G.M., Arvind: *T: A Multithreaded Massively Parallel Architecture, *Proc. 19th Annual Internat. Sympos. on Computer Architecture* (1992),

[PaC 90] Papadopoulos G.M., Culler D.E.: Monsoon: An Explicit Token Store Dataflow Architecture, *Proc. 17th lAnnual nternat. Sympos. on Computer Architecture* (1990), IEEE Computer Society order no. 2047, 82-91

[Sch 91] Schroeder-Preikschat W.: Overcoming the Startup Time Problem in Distributed Memory Architectures, in Milutiniovic V., Shriver B.(eds.): *Proc. 24th Hawaii Internat. Conf. on System Sciences*, vol.1, IEEE Society Press 1991, IEEE order no. 91TH0350-9, 551-559

[Sch 94] Schroeder-Preikschat W.: *The Logical Design of Parallel Operating Systems*, Prentice-Hall, Englewood Cliffs NJ 1994, ISBN 0-13-125709-9

[Smi 78] Smith B.J.: A Pipelined, Shared Resource MIMD Computer, *Proc. 1978 Internat. Conf. on Parallel Processing*,