November 1990

Report No. STAN-CS-90- 1344

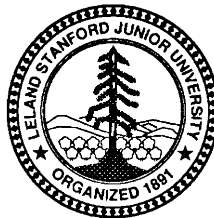
ParaDiGM: A Highly Scalable Shared-Memory Multi-Computer Architecture

by

David R. Cheriton, Hendrik A. Goosen and Patrick D. Boyle*

Department of Computer Science

Stanford University
Stanford, California 94305



REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1990	Technical

4. TITLE AND SUBTITLE
ParaDiGM: A Highly Scalable Shared-Memory Multi-Computer Architecture

5. FUNDING NUMBERS
C-N00014-88-K-0619

6. AUTHOR(S)
David R. Cheriton, Hendrik A. Goosen and Patrick D. Boyle*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Department of Computer Science
Stanford University

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release: distribution unlimited

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
ParaDiGM is a highly scalable shared-memory multi-computer architecture. It is being developed to demonstrate the feasibility of building a relatively low-cost shared-memory parallel computer that scales to large configurations, and yet provides sequential programs with performance comparable to a high-end microprocessor. A key problem is building a scalable memory hierarchy. In this paper we describe the ParaDiGm architecture, highlighting the innovations of our approach and presenting results of our evaluation of the design. We envision that scalable shared-memory multiprocessors like ParaDiGM will soon become the dominant form of parallel Processing, even for very large-scale computation, providing a uniform platform for parallel programming systems and applications.

14. SUBJECT TERMS
Multiprocessor architecture, parallel processing, shared multi-level caches, distributed operating systems

15. NUMBER OF PAGES
19

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

ParaDiGM:
A Highly Scalable Shared-Memory Multi-Computer Architecture

David R. Cheriton, Hendrik A. Goosen and Patrick D. Boyle*
Computer Science Department
Stanford University

Abstract

ParaDiGM is a highly scalable shared-memory multi-computer architecture. It is being developed to demonstrate the feasibility of building a relatively low-cost shared-memory parallel computer that scales to large configurations, and yet provides sequential programs with performance comparable to a high-end microprocessor. A key problem is building a scalable memory hierarchy. In this paper we describe the ParaDiGM architecture, highlighting the innovations of our approach and presenting results of our evaluation of the design. We envision that scalable shared-memory multiprocessors like ParaDiGM will soon become the dominant form of parallel processing, even for very large-scale computation, providing a uniform platform for parallel programming systems and applications.

1 Introduction

ParaDiGM (PARAllel DIstributed Global Memory)[1] is a highly scalable shared-memory multi-computer architecture. By *multi-computer,* we mean a system interconnected by both bus and network technology. By *shared-memory,* we mean an architecture that allows a parallel application program to execute any of its tasks on any processor in the machine, with all the tasks in a single address space. By *scalable,* we mean a design that is cost-effective from a small number of processors to a very large number of processors.

Currently, there is a wide diversity of non-shared-memory parallel machine architectures, some of which have been created in response to the perceived difficulty in building scalable high-performance shared-memory multiprocessors. The diversity of architectures has fragmented the efforts to develop parallel applications, languages, and operating systems support, limiting the portability of the resulting software and generally impeding the development of a strong base of parallel software. We believe that parallel software efforts should be focused on shared memory architectures, because of their ease of programming, compatibility with standard programming languages and systems, and because shared memory multiprocessors are clearly becoming the dominant architecture for small-scale parallel computation.

Current shared-memory architectures were not designed to scale beyond a few tens of processors because of the memory bandwidth requirements, prompting a wide-spread belief that they cannot be designed to scale. ParaDiGM is an architecture we have developed to demonstrate the feasibility of a shared-memory parallel computer system that scales from a small number of processors to hundreds of processors or more, providing cost-effective performance over that entire range, while running the same software on all configurations. The availability of high-performance low-cost microprocessors makes this scaling feasible from the standpoint of raw processing power. The problem lies in the interconnection.

A switching network is required for interconnecting a scalable machine, since the bandwidth required for interprocessor communication, memory traffic, and I/O must grow as processors are added. However, such a network has high latency because of the switching and queueing delays. While this is not a problem for I/O, where transfer units are large enough to amortize the latency, memory accesses and interprocessor communication usually involve much smaller transfer units, and are therefore seriously affected by high latency. Specialized interconnection networks such as crossbar or shuffle-exchange networks are not the

*Now with Digital Equipment Corporation's Western Research Laboratory

[1] An earlier version of this work used the name VMP-MC, indicating an extension of the original VMP [4, 5] work.

solution, since they are not mainstream products, and are therefore expensive (and also suffer from the bandwidth/latency tradeoff). Our solution is to cluster processors together in nodes, and provide each node with an optimized high-speed shared bus/cache hierarchy. This allows low-latency data transfers within the node, optimized for memory access and interprocessor communication, while still providing scalable bandwidth for the machine as a whole.

ParaDiGM achieves high-performance scalability by exploiting distributed operating system techniques for very large-scale, widely distributed operation, and parallel application structuring techniques that maximize locality and minimize contention. This paper describes the design of ParaDiGM, focusing on the novel techniques which support scalability. We identify the key performance issues with this design and summarize some results from our work to date [2] and experience with the VMP architecture [4, 5] design. We argue that ParaDiGM provides a promising approach to a highly scalable architecture.

The next section describes the ParaDiGM system model. Section 3 describes the "building block" components used to assemble a ParaDiGM system. Section 4 evaluates the benefits of a shared cache and bus interconnection for local clusters of processors. Section 5 describes our approach to structuring large-scale applications for ParaDiGM, using a PDE solver as an illustrative example. Section 6 describes the current status of the ParaDiGM hardware and software. Section 7 compares our work to other relevant projects. We close with a summary of our results: identification of the significant open issues, and our plans for the future.

2 ParaDiGM System Model

ParaDiGM is a software/hardware architecture realized by a combination of the operating system, hardware, and firmware-like components such as (software) cache management modules. The operating system interface provides multiple shared virtual address spaces for containing separate applications and multiple processes or tasks per address space for executing parallel applications, similar to features available in multiprocessor Unix[2] systems. However, the kernel implementation differs significantly, particularly in the scalable shared-memory mechanism. Our approach is to build the system on top of a distributed file cache mechanism that provides efficient I/O, simplifies the design, and provides additional features such as fault-tolerance.

Each address space is composed by mapping files to address ranges in the space, typically including the program binary file and a backing *file* for the data segment. Mapped files are implemented by a local file cache and *server* modules that handle file reads, writes and consistency management. The mapped file cache serves in part the role of the virtual memory page pool in conventional operating systems. The operations on a file are communicated from the client file cache to the server using a remote procedure call (RPC)-like communication facility, which provides network-transparent access to servers. Thus, a virtual memory reference in the application can result in a reference to a file page in the file cache, which in the worst case can also miss, resulting in an RPC communication to the associated server module to retrieve the missing page. This system organization and memory reference path is depicted in Figure 1. The server module may

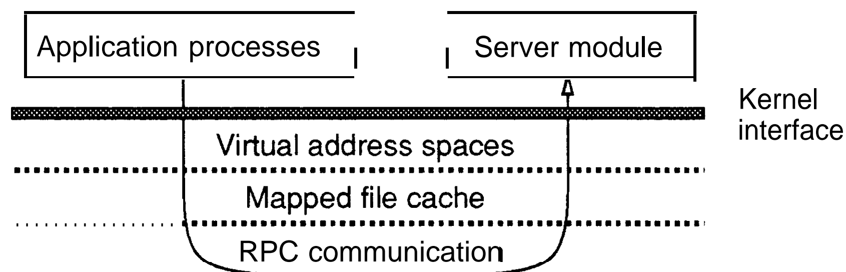


Figure 1: ParaDiGM System Layers and Memory Reference Path

be located on a separate network node or the same network node, transparent to all layers except the RPC layer. The extensive use of caching in ParaDiGM means that most memory references are satisfied without incurring this overhead.

[2] Unix is a trademark of AT&T Bell Laboratories.

ParaDiGM supports multiple layers of cache, multiple caches at each layers, and two types of interconnect, as shown in Figure 2. Caches are connected by network technology for scalability. In addition, caches located

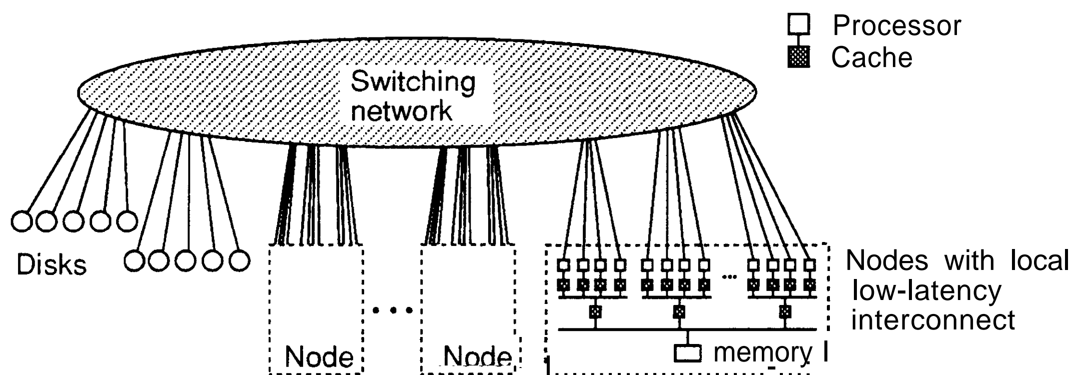


Figure 2: ParaDiGM Interconnection

within a node are interconnected by a shared bus and cache hierarchy. Cache consistency is based on the file consistency mechanism, which uses an ownership-based protocol between the client file caches and the file server. In particular, a write reference to a datum requires getting an up-to-date copy of the page containing the datum and exclusive ownership of this page from the file server before the write operation can complete. Bus-connected caches are further optimized to use physical addressing and hardware consistency mechanisms among themselves to avoid the overhead of the software consistency mechanism. By using a small unit of consistency (the cache block) and the hardware optimizations for consistency within a node, the performance of the ParaDiGM shared memory system is expected to be significantly better than that of virtual shared memory systems relying exclusively on network interconnection with consistency maintained on virtual pages.

In addition to its (distributed) shared memory, ParaDiGM provides application access to the efficient RPC facility upon which shared memory is based. RPC allows *function shipping* [7] in addition to the *data shipping* supported by shared memory. Function shipping is preferred when it takes fewer data transfers to ship the function to the data than to ship the data to the function. (Normally the code is available at the remote site so only the parameters for the call invocation are actually shipped.) For example, a work allocator module that hands out tasks may be accessed in round-robin order by the processors. It is less expensive to execute this module on one processor, with the other processors invoking it by RPC, than for every processor to incur the cost of fetching the data required to execute the module directly every time it needs to allocate more work. Even with very large caches, references to the volatile data are expected to cause a cache miss on each invocation because the data would be modified by other processors executing the module since the previous invocation by the current processor. Direct invocation would require at least $2P$ packets if P packets must be trapped in individually. In contrast, RPC takes two packets in the expected case (assuming no retransmissions and that all parameters fit in a single packet).

Building on the experience with the V distributed system [1], ParaDiGM also supports multicast communication and process groups as an extension of the communication facilities. Multicast can be used to inform a group of processes of an update or event, mimicking the broadcast-like behavior of updating a datum in shared memory. It can also be used to query and efficiently synchronize across a group of processes.

This use of distributed operating system technology to provide scalable (distributed) shared memory and RPC contrasts with the conventional hardware-intensive approach to large-scale parallel architectures that often appear as incremental extensions of small hardware architectures. Several aspects of the design illustrate the resulting differences. First, the virtual memory system is built on top of (shared) files which is built on top of a communication facility. This approach avoids the complexities of building a separate file caching and consistency mechanism that parallels the virtual memory system, or building files on top of- virtual memory support (we argue that files are the more general facility). Second, the provision of RPC with extensions for multicast at the application layer allow applications to choose between function and data shipping depending on data access patterns and latency between processors sharing the data.

This contrasts with distributed memory architectures that only provide interprocessor communication, and with conventional shared memory multiprocessors that provide little or no support for efficient interprocessor communication. Finally, ParaDiGM achieves good performance by careful optimization of the shared memory and communication facilities within local clusters of processors, and by careful structuring and optimization of parallel applications to exploit locality. These optimizations eliminate the overhead of the general model for cluster-local operations. In the case of remote operations, we expect this overhead to be dominated by network switching and propagation delay.

3 ParaDiGM Building Blocks

The three key hardware building blocks of ParaDiGM are the *Memory Module* (MM), the *Multiple Processor Module* (MPM), and the *Inter-bus Caching Module* (ICM). The MM provides backing memory for higher level[3] caches with its space managed as a file page frame pool. It also contains a directory supporting cache consistency, message exchange, and locking between the higher levels caches. The MPM is a single board containing multiple (2-8) microprocessors with on-chip caches sharing a board-level cache. The MPM contains a high-speed network connection, for connecting to other MPMs as well as to outside network services. It also includes a bus interface to lower level caches and memories. The ICM is a bus interconnection as well as a data cache and directory, allowing a group of MPMs on a single bus to be connected to another bus, appearing as a single MPM on this bus, as shown in Figure 3. This structure can be extended recursively

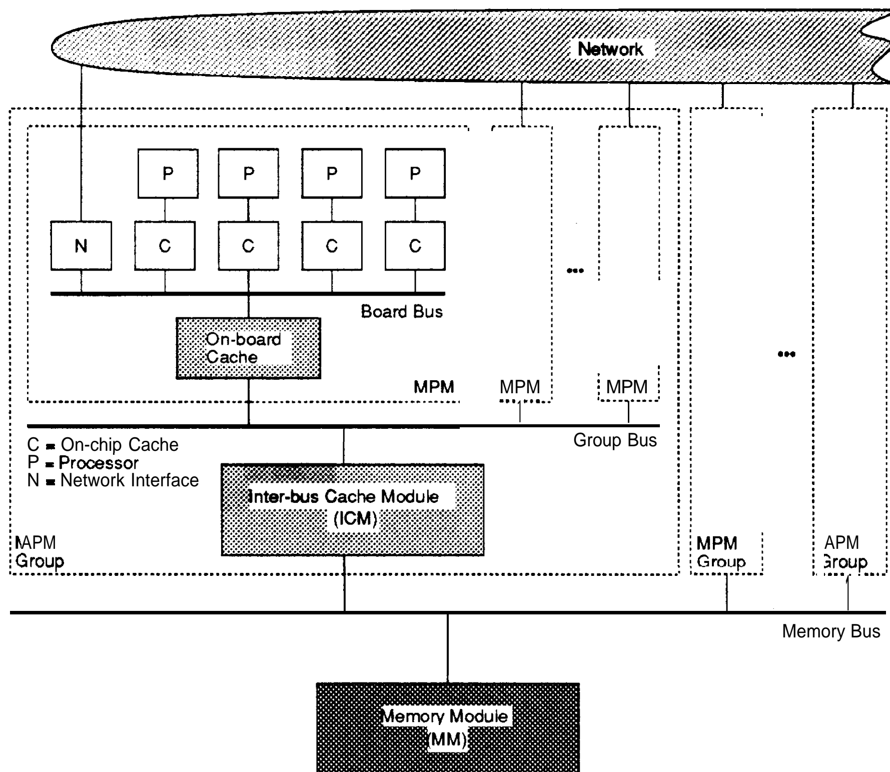


Figure 3: ParaDiGM node with Multi-Processor Module groups (MPM groups)

to additional levels by using ICMs to connect MPM groups to a lower level bus. The shared bus and cache interconnection provided on the MPM as well as by the ICMs allows low latency interaction between clusters of processors within a node. Determining the appropriate configuration of the shared bus and cache hierarchy is a key aspect of our research, and is discussed further in Section 4.

The following sections describe these modules and their interaction; additional detail is available in [2]. The MM is described first to present the bus data transfer and consistency protocols.

[3] We will refer to modules closer to the processor as "higher level" modules.

3.1 Memory Module (MM)

The memory module (MM) is physically-addressed and provides the bulk memory for the system. It includes a directory, the Memory *Module Directory* (MMD), that records the consistency state of each *cache block*[4] stored in the MM. Rapid data exchanges with the MPMs are achieved by block transfers using a sequential access bus protocol and interleaved fast-page mode DRAMS.

For each cache block of memory in the MM, the directory has a 16-bit entry indicating the block's state:

cc	L	P_{12}	P_{11}	...	P_0

where CC is a two bit code, and L is the LOCK bit used for locking and message exchange (described below). Each P_i bit corresponds to one MPM or ICM, allowing up to 13 MPMs and ICMs to share this memory board.[5] If the P_i are all clear, then the block is neither cached nor in use for message exchange. The CC can be in one of four states: undefined, shared, private, or requestnotification.

Directory entries can be written and read directly, but they are normally modified as a side effect of bus operations. The directory is designed to support the implementation of consistent cached shared memory, a memory-based multicast message facility, and memory-based locking, as described below.

3.1.1 Consistent Shared-Memory Mode

The consistency protocol follows an *ownership* protocol, ensuring either a single writable *(private)* copy or multiple read-only *(shared)* copies of a block.

If the block is uncached, the P field of its directory entry contains zeros. A read-shared (or read-private) operation by module i on an uncached block returns the block of data, sets P_i, and changes the CC state to shared (or private). Subsequent read-shared operations on a shared block return the data and set P_i. A read-private operation on a shared block requires all the cached copies to be invalidated. The P bits in the directory allow the MM to send invalidation requests only to modules which have copies of the block, rather than broadcasting it. This attribute of the design is important to its scalability. When a block is private, the MM aborts read-shared and read-private operations, and interrupts the owner to write back the block.

3.1.2 Memory-Based Message Exchange Protocol

The MM implements a message exchange protocol that allows cache blocks of shared memory to be used as message buffers. When a cache block is in the requestnotif ication state, a block write to that block causes every processor which has registered as a receiver for the block to be interrupted. The interrupted processors can then read the block, receiving the transmitted data. Processors can therefore use the data transfer facilities of the memory system like a network interconnection.

This facility requires minimal extra mechanism because it builds on the memory coherence mechanism. It is implemented using a simple extension to the cache directory (two extra bits in each directory entry), and one additional bus operation, Not if y. The `Not if y` bus operation by module i on a given block sets the CC state for the block to requestnotification, and sets P_i. A subsequent write-back to that block sets the L bit and interrupts every module specified in the P field. The L bit shows that the block has been written, but not yet read. A read-shared operation clears the L bit and returns the data.

The memory-based message exchange facility significantly reduces the cost of an interprocessor message compared to a conventional implementation using shared memory, allowing an interprocessor call to be performed in two bus transactions (assuming the parameters and return values fit in a cache block). In contrast, it would take at least five bus transactions to transmit a message on top of conventional shared memory, or ten bus transactions for a full call and return.[6] The benefit of the hardware support is magnified further when the sending and receiving processors are separated by several levels of the hierarchy or when the message is multicast.

In addition to the use of the message exchange facility for efficient application RPC, it is also used for interprocessor communication and synchronization. Each processor has one or more message buffers for which it requests notification. A kernel operation on one processor that affects a process on another processor

[4] A cache block is an aligned N-byte unit of memory. The value of N is a key parameter in the design; values in the range 32 to 256 are our current focus.

[5] An MPM and an ICM appear identical to the MM on the memory bus. We use the term *module* to refer to either.

[6] The extra transactions are due to cache misses to allocate a message buffer, write the message buffer, signal the receiving processor, dequeue the buffer at the receiver, and read the buffer. The extra cache misses occur because the semantics of message exchange differ in two key aspects from that of consistent shared memory: (1) a receiving processor wants to be notified after a cache block (message buffer) has been written, not before it is read, as in consistent shared-memory mode, and (2) a sending processor wants to write a cache block without having to read it in first.

sends a message to that processor by simply writing to one of its message buffers. A specific multicast use of the message exchange facility is to notify processors of memory mapping changes.

The message support makes the intra-node RPC substantially faster and obviates the need for a separate interprocessor communication facility. It also reduces the load on the buses and makes the performance more competitive with the network for very large hierarchies.

3.1.3 Memory-Based Locking

The MM implements a locking protocol; the unit of locking is the cache block. A lock bus operation by module i on an unlocked block (the L bit in the directory entry is clear) succeeds and sets the L bit and P;. If the L bit was set, the bus operation fails and P_i is set. An unlock bus operation by module i clears the directory entry's lock bit, and all modules j for which P_j is set, where $j \neq$ i, are signaled that the lock has been released. This mechanism allows the lock to be cleared by a processor other than the one which set it, as is required by some applications.

One can view the lock mechanism as a single-bit RPC to the memory directory system. As such, the lock mechanism is just a further optimization over the message exchange protocol, recognizing the small grain of the data unit (1 bit) and the simplicity of the call, which can be easily handled by the simple state machine managing the directory.

The provision of locking as part of the consistency mechanism provides several optimizations over a conventional lock mechanism implemented using test-and-set operations. In a conventional write-invalidate design, a spinning contender for a lock may steal the block containing the lock away from the lock holder, forcing the holder to regain ownership to release the lock. In addition, if the lock is in the same block as the locked data, processors spinning on locks contend with the lock holder while it is updating data in the same cache block. Thus, the conventional memory-based locking slows down the execution of the lock holder when there is contention for the lock, resulting in longer lock hold times and even more blocking on the lock. In our scheme, the locking mechanism serves *as contention control* on data structures. A processor needing to acquire a lock must wait until the lock is unlocked, consuming no bus bandwidth (after one lock request) and not interfering with other processors. Used in combination with read operations that specify locking, it also allows a processor to acquire both the lock and the data in one bus operation.

We estimated the performance effect of this mechanism by identifying memory locations used for locking, and ignoring these references in our simulations. Taking this approach, we observed a 40% reduction in bus traffic when locks were ignored in the trace [4]. This reduction in the traffic supports the notion of a specialized locking mechanism to reduce memory contention for locks. Further evaluation of the benefits of the ParaDiGM locking mechanism requires designing applications to take advantage of this facility.

The locking mechanism could be extended to provide queuing of lock contenders to ensure some degree of fairness in lock access. In our opinion, the required hardware complexity outweighs the possible benefits. We subscribe to the operating system "folk theorem" which states that queues in a well-designed system generally have zero or one entries, so the queuing mechanism would have little benefit in many cases. Queuing can also produce convoying of processors through a series of locks, as has been observed in database systems and previous operating systems work. Moreover, fairness and priority as implemented at the operating system level may not map easily onto these hardware queues. In general, it seems better to achieve the appropriate fairness and priority by processor scheduling rather than lock scheduling. Finally, queuing locks or semaphores can be implemented in software in the cases in which they are needed, where the queuing cost is only incurred when the lock requester is blocked, and is thus generally dominated by context switch cost.

The three protocols described above are recursively implemented at each level in the memory hierarchy, by associating a directory with each shared cache (i.e., with each ICM and MPM board cache). The directory information is therefore distributed among various modules, as described in the following sections.

3.2 Multiple Processor Module (MPM)

The Multiple Processor Module (MPM) occupies a single printed circuit board, and is shown in Figure 4. Multiple CPUs (microprocessors) are attached by an on-board bus to a large cache and a small amount of local memory. The cache blocks are large, and the cache is managed under software control, as in VMP [5]. The local memory contains cache management code and data structures used by a processor incurring an on-board cache miss. A FIFO buffer queues requests from the memory bus for actions required to maintain cache consistency, and to support the locking and message exchange protocols. One of the processors is interrupted to handle each such request as it arrives.

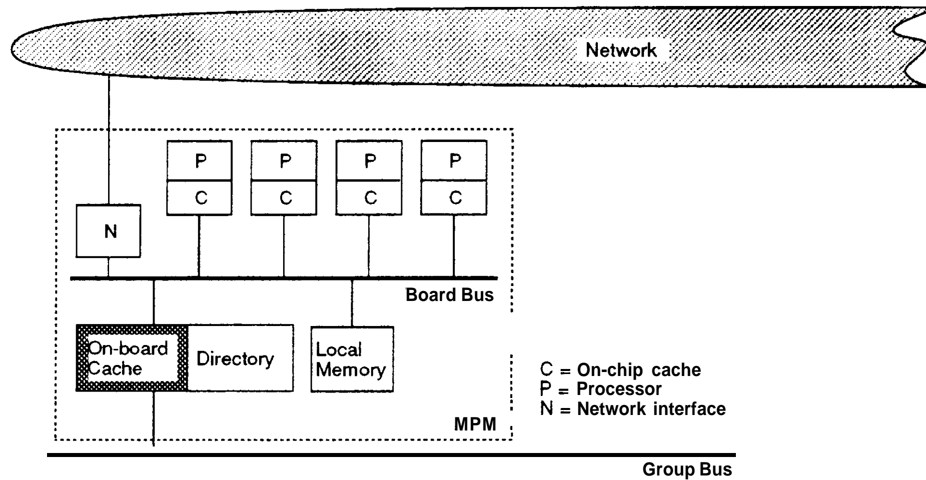


Figure 4: MPM Board Layout

3.2.1 Processor and on-chip cache

The target CPU is a high-speed RISC processor (100 MIPS) with a large (16K or more) virtually addressed on-chip cache with a moderate cache block size (32 bytes). Each on-chip cache block contains flags supporting the locking protocol, in addition to the usual access rights and cache block state flags (including one indicating that the block contains kernel data, eliminating the need to flush the cache on return from a kernel call). The processor has a lock and an unlock instruction,[7] each specifying an address aligned to a cache block. A lock can be owned by the on-chip cache, in which case the lock and unlock instructions execute locally (i.e., lock acquisition is done entirely in the cache with low latency). A cache block can also indicate that the lock has been requested from another cache, in which case further lock requests are ignored until the lock is granted. Finally, the on-chip cache may own a lock that another cache has requested. In this case, the requester is notified when the lock is released.

Cache blocks are transferred between the on-chip cache and the on-board cache by a wide per-processor holding register that transfers the block to the on-chip cache in burst-mode, thereby reducing interference between processors for on-board cache access.

3.2.2 MPM On-board Cache

The on-board cache implements the same consistency, locking and message exchange protocols as the MM. The directory also has an `exclusively_held` bit that indicates whether or not the cache holds exclusive ownership of the block. This allows a block to be shared by processors within the MPM, while it is exclusively owned by the MPM relative to the rest of the system. Finally, there are bits supporting the same locking functions as the on-chip cache.

When an on-board cache miss occurs, the faulting processor takes the necessary steps to transfer the missing block into the cache. Cache access from other processors may proceed concurrently with miss handling except when an actual bus transfer is taking place. Following the VMP design, the complexity of cache management is handled largely in software.

Sharing the on-board cache has three major advantages. First, it results in a higher on-board cache hit ratio due to the sharing of code and data in the on-board cache and by localizing access to some shared data to the on-board cache. Compared to per-processor on-board caches, the sharing reduces the total bus traffic imposed by the processors, which contributes to scalability. Second, sharing the on-board cache reduces the total hardware cost for supporting N processors, since only N/P MPM boards (and on-board caches) are required if P processors share each on-board cache. Finally, the increased hit ratio of the on-board cache reduces the average memory access time of the processor, resulting in a higher instruction execution rate.

[7]This is the ideal case. Processors without these instructions will require extra off-chip logic to implement the locking functions. In a high-performance implementation using current ECL microprocessors, this can be provided as part of the (off-chip) first-level cache.

The on-board cache exploits a. number of ideas from the VMP design. First, the cache is virtually addressed, so there is no address translation required between the on-chip cache and the on-board cache. The complexity of virtual-to-physical mapping is placed (in software) between the MPM and the group bus, simplifying both the processor chip and the on-board logic, and reducing the translation frequency by orders of magnitude [4]. **Also,** the cache miss software uses compact data structures to replace conventional page tables, thereby reducing the memory space overhead of the virtual memory implementation.

The on-board cache minimizes replacements and flushing by using set-associative mapping and an address space identifier as part of the virtually addressed cache mechanism. Thus, the cache can hold data from multiple address spaces and need not be flushed on context switch. The on-board cache provides one address space identifier register per processor. Each off-chip reference by a processor (cache miss) is presented to the on-board cache prepended with the address space identifier. Thus, the on-board cache knows about separate address spaces but the processor chip need not.

The design of the MPM exploits the large size of on-chip caches on new microprocessors. The large block size of the on-board cache is compatible with increasing on-chip block sizes. The inclusion of the lock bits in both the on-chip and on-board caches effectively improves the cache and bus behavior by reducing latency, coherence interference, and contention. These bits impose a modest space overhead that decreases with increasing cache block size. The large cache block size also makes it feasible for the on-board cache to be quite large (i.e., 512 kbytes or more), reducing replacement interference and thereby permitting multiple processors to share the on-board cache even when running programs in separate address spaces.

Although the MPM design presented here assumes that the processor has a virtually addressed on-chip cache, the MPM can be realized using either virtually or physically addressed caches. If the processor has on-chip address translation, the MPM board-level cache can be physically addressed. The only requirement is that the processor must be able to handle an exception on a cache miss even though the address translation succeeded.

3.2.3 MPM Network Connection

The network interface is an extremely simple programmed I/O device, addressed as a set of registers in the MPM physical address space. The processor performs all the transport-layer processing to minimize the latency, given that most communication is expected to be very small packets as a result of the dominance of contention traffic. On transmission, the processor copies the packet directly into the transmit register. On reception, one of the MPM processors is interrupted to copy the packet out of the receive register. For both cases, the software is optimized to handle packet generation and processing "on-the-fly", minimizing the communication latency. As a further optimization, the requesting processor may busy-wait on a network reception of the response to its request if there is no other action to perform, thereby eliminating the cost of a context switch, at the expense of increased bus bandwidth. Having a processor copy the data between the network interface and the on-board cache avoids the complexity of interfacing a DMA controller to the memory system and the extra memory copy that typically arises because of the rigidity of the DMA data copy, especially on reception.

The network connection on the MPM board provides its processors with direct, access to other processors in the network with contention for the network interface limited to the few processors on the board. It also allows external communication to be directed to the portion of the node best suited to handle the request, provided that this information is available to the source. For example, the virtual shared memory consistency mechanism maintains a hint to the location of pages that were recently invalidated by a remote MPM group, allowing it to directly address the remote MPM that stole a page (with high probability) when this page is referenced again locally. The page is then directly returned to the requesting MPM using the local network connection. Another advantage of this structure arises in the timesharing or file server environment where remote users (clients) can directly connect to the MPM that is serving them. At the other extreme, the local MPM network interface makes the MPM an attractive component to use in a minimal (diskless) workstation, avoiding the cost and complexity of a separate network interface board.

The networks of interest range from 100-150 megabit networks being developed to support video to multi-gigabit networks for supercomputer interconnection. In general, we assume that the network connection is the fastest I/O connection available to the processors, in fact subsuming others, as described in the next section.

3.2.4 Other I/O Connections

Other I/O in ParaDiGM is accommodated as specialized nodes on the network. For example, a. disk controller would interface to the network rather than to a local ParaDiGM bus. This approach avoids cluttering and complicating memory busses with I/O controllers, makes I/O devices available subject only to the network and the I/O controller availability and yet has no significant performance disadvantage because of the high performance of the network.

We note that the network data. rates we assume for ParaDiGM far exceed the transfer rates of single disks available now and in the foreseeable future. Disk rotational and seek latency also dominate the network latency for most configurations. Since the network connection interfaces directly to the processor chips and on-board caches, disk data transfers do not load lower level busses, as they would if the disks were connected at these lower levels.

Attaching disks to the network also allows the disk capacity to scale with the scaling of the machine, given that the network must support scalability in any case.

The disk controller depends on one or more managing server modules running on other ParaDiGM nodes to implement higher-level file systems functions. The file cache consistency mechanism provides data transfers between caches to satisfy misses and consistency as well as direct transfer from the disk controller node to the requesting client when the data is not cached nearby.

We see this approach to I/O as the natural step beyond that spawned by local area networks in the 1980's, that allowed terminals to be connected to terminal concentrator network nodes rather than directly to host machines. In both cases, separating the base I/O services from the computation engines simplifies the I/O nodes and computation engine configurations resulting in greater reliability and provides greater flexibility of interconnection.

3.3 Inter-bus Cache Module (ICM)

The inter-bus cache module (ICM) is a cache, shared by the MPMs on an inter-MPM bus (an MPM group), which connects such an MPM group to a next level bus. It appears as an MPM on the memory bus and an MM on the group bus. It caches memory blocks from the MMs, implementing the same consistency, locking and message exchange protocols as the MPMs. These blocks are cached in response to requests from MPMs on its inter-MPM bus. The directory entry per block in the ICM is the same as that of the MPM onboard cache.

Several merits of the ICM are of note. First, as a shared cache, the ICM makes commonly shared blocks, such as operating system code and application code available to an MPM group without repeated access across the memory bus. The ICM shared cache is important for scalability for the same reasons identified for the MPM on-board cache. Second, the ICM supports hierarchical directory-based consistency, providing a complete record of cache page residency, thereby minimizing consistency bus traffic and interprocessor interrupt overhead. Finally, because the ICM appears the same as an MPM, one can mix MPMs and ICMs on the memory bus without change to the MMs.

An alternative ICM design is to use a simple "data path" interconnect that contains the directory but not the data cache. Cache misses on data present in other caches on the same bus could be satisfied by direct cache-to-cache transfer. We chose to make the ICM a full cache for several reasons. First, the effective size of the ICM cache is expected to be substantially larger than the sum of the higher level caches because the latter would contain mostly the same (hot) code and data when running a large parallel application. Second, the ICM can respond to misses on shared blocks even if the block is contained in another cache, thereby offloading that cache. Finally, the ICM design allows a simpler design of MPM and ICM interconnection because direct cache-to-cache transfer is not required.

The ICM allows one to (largely) isolate a computation in a node, which can function as the compute-server portion of an extended workstation. It shares the MM, and possibly local disks with the workstation, but with only slightly greater loading than a single additional processor. For example, an engineer might add such an expansion module to his multiprocessor workstation, allowing him to run compute-intensive simulations on the ICM-connected module while running normal CAD software on the rest of the machine with relatively little interference from the simulation.

3.4 Operating System Software Structure

ParaDiGM uses the V distributed operating system [1] which is structured as a minimal kernel plus a set of service modules that execute on top of the kernel.

The kernel implements virtual memory and file caching, lightweight processes and interprocess communication, exploiting the facilities and structure of the ParaDiGM hardware architecture. The kernel handles cache misses out of the MPM, using virtual memory binding data. structures to map the virtual address to a file page and the file caching information to locate the appropriate page frame in physical memory. If the data is missing from physical memory, it resorts to the RPC facility to retrieve the data from the server or, as an optimization, from another peer node. The kernel also handles inter-node operations to invalidate or write back data, as required for cache consistency. In contrast to conventional virtual memory and file caching systems, it uses the cache block as the unit of consistency and the virtual memory page as the unit of mapping, meaning that a portion of a page may be invalid because of contention with another ParaDiGM node. This finer-grain consistency, say 64 bytes versus 4 kilobytes, is critical for good parallel application performance.

The kernel maps process-level RPC onto either network transmission, or block writes to interprocessor message buffers, depending on routing information about the specified server. It also maps message exchange interrupts to communication with the associated processes. We are exploring techniques to minimize the kernel intervention required for interprocess communication. For example, message buffers can be mapped directly into the application address space so the message data can be written and transmitted without kernel intervention. For reception, the cache management module on each MPM can directly deliver the message to the local processes in the common case, using cached information about the binding of processes to message buffers.

Most operating system services, including the file system and wide-area networking services, are provided as server modules executing outside the kernel. These server modules are easier to write than conventional parallel operating systems, because they need not support the same degree of concurrency as the kernel, relying instead on replication of the servers to increase parallelism. For example, there can be an instantiation of the file system per disk channel with each executing independently in parallel. Thus, with n disk channels and a file load spread evenly across all disk channels, each file server module must handle l/n-th of the concurrent disk access. Because the file caching mechanism is integrated with the virtual memory system in the kernel, this partitioning does not fragment the file buffering or limit the degree of concurrency allowed for buffered data. This approach results in simpler code than using very fine grain locking in order to minimize contention. It also divides up (and therefore cools off) the contention hotspots.

Using this structure, the kernel is relatively small and simple, making it feasible to hand-tune it to execute efficiently on the ParaDiGM. Kernel data structures such as dispatch queues are partitioned across MPMs to minimize interprocessor interference and to exploit the efficient sharing feasible using the shared MPM cache. Also, access to kernel data structures is synchronized using the cache locks, and the data structures themselves are structured to minimize cache miss and contention behavior. For example, descriptors are aligned to cache block boundaries and matched to cache block sizes whenever possible. The process group and multicast mechanism in V is used to accommodate multiple process management module instances within one network node as well as multiple instances executing across the network. Invocation of these facilities local to one node is optimized to make this case significantly more efficient. For example, migration of a process between processors within a node requires moving the process from one dispatch queue to another but not explicitly copying the process state, as is required with network process migration.

The ParaDiGM software design contrasts with the conventional multiprocessor operating system design, which is generally just a version of a single processor operating system such as the standard Unix kernel, modified to run multi-threaded. To accommodate industry standards and existing software bases, ParaDiGM includes support for running Unix programs under emulation with no significant loss in performance [6]. In essence an emulation segment is appended to each emulated Unix process, translating Unix system calls to RPC calls to ParaDiGM system services. Because the emulation segment is per-Unix process, manages data that is largely private to the process, and can be executed in parallel itself, it can support even highly parallel Unix programs.

3.5 ParaDiGM Configurations

The ParaDiGM building blocks described above allow configuration of a range of machines from a personal multiprocessor workstation to a large compute server. The personal workstation would be realized with a single MPM, optionally augmented with video framebuffer, using the MPM network interface to connect to other services. A large server would use network technology to interconnect many MPMs, with clusters of MPMs structured as ParaDiGM nodes using one or more layers of shared busses, interconnected by ICMs. For example, a 2048-processor ParaDiGM machine might be configured as 512 4-processor MPMs clustered into 8 independent network nodes, each containing 8 ICMs with 8 MPMs sharing each ICM. This configuration exceeds the processing power of current high-end supercomputers. However, the performance of large parallel applications on such a machine would be highly dependent on the amount of intercache communication required by the application and the degree to which the machine interconnection matched this communication, both in data rates and traffic topology. The next two sections discuss the use of shared bus and caches to augment network capacity and reduce memory latency, and the structuring of programs to take advantage of the shared bus and cache hierarchy.

4 Shared Bus/Cache Hierarchy Configuration

The shared bus and cache hierarchy in ParaDiGM is designed to augment the intercache and interprocessor communication provided by the network technology. While a large ParaDiGM system could be configured with no hierarchy beyond that provided by the MPM, as illustrated in Figure 5, the network latency would

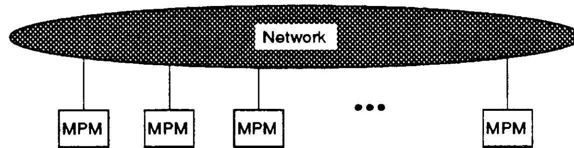


Figure 5: ParaDiGM with one MPM per Network Node

limit performance for a significant number of applications. Network latency is of concern because a goal of ParaDiGM is to use future industry-standard network technology, particularly broadband ISDN (Integrated Services Digital Network) technology such as ATM (Asynchronous Transfer Mode) fast packet switching. This technology is being developed to provide the high data rates required for video and high-performance data transfer. The extremely large customer base also dictates extendable switching capacity, low cost, and high reliability. Because minimizing latency is not of primary concern and is expensive to provide, the switching delay of a single switch is expected to be multiple microseconds.

Bus technology offers lower latency communication between two components directly connected to the bus than this network technology because bus arbitration time is less than the switch routing and store-and-forward times. Shared caches further reduce the latency in a bus hierarchy by increasing the cache hit ratios, thereby providing the faulting processor with faster cache miss response and also reducing the traffic load on other portions of the bus hierarchy. However, a bus can interconnect only a limited number of nodes, and a multi-level bus hierarchy is limited by the bus capacity near the root of the hierarchy, and by increasing latency as its height increases with scale.

The following subsections explore the effect of hierarchy height, shared cache contention and bus loading in limiting the use of shared bus and cache hierarchies in ParaDiGM.

4.1 Hierarchical Latency

Hierarchical latency, the cost of data transfer through the bus hierarchy, must compete with network latency. Each additional layer in a bus hierarchy adds to the data transfer time the latency for a data transfer through two additional ICMs, as well as the queuing delay and transfer time of two additional busses. Adding a switch to a network adds a single additional switching delay because the network need not be hierarchical. Thus, at some scale of bus hierarchy, one would expect the network connection to provide lower latency between some MPMs than the bus hierarchy, negating the benefits of the bus hierarchy. This point is estimated below.

For comparison, consider a 1 Gb/s network and a minimum size packet (large enough to hold a cache block). The transmission time is thus roughly 1 microsecond. Assuming a single switch between nodes introducing one store-and-forward delay and at most another microsecond of propagation time, hardware

setup time, etc., the basic hardware penalty is 3 microseconds. If highly optimized network software can handle packet transmission or reception in roughly 200 instructions, facilitated by the network interface chip doing checksumming, the software delay is roughly 4 microseconds using 100 MIPS processors. Thus, the total roundtrip delay for requesting a cache block on which there is active contention (or to send an interprocessor message) is roughly 14 microseconds, or 1400 processor cycles. Different network parameters result in different latency. For example, a 150 Mb/s network, as provided by the Sunshine switch [8], would have a latency of 32 microseconds, or 3200 cycles.

The latency of the bus hierarchy depends on the cycle time, data width and queuing delay due to load, the number of hierarchies of busses, and the delay through each ICM. Figure 6 shows the cache miss latency

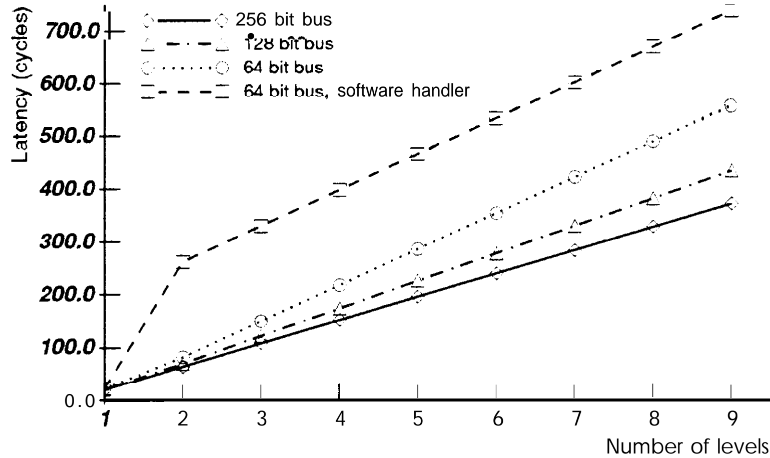


Figure 6: Hierarchical latency

when a processor accesses a block that is privately owned by another processor. This is the worst-case (no-load) latency which is incurred when the requests and the block have to propagate through the entire hierarchy from the requester to the memory module to the block owner and back. The cost model for this calculation is given in Figure 7.

Description of action	Cost for level		
	1	2	3+
Cache signals miss or write-back, and arbitrates bus	4	8	12
Cache handles miss or write-back	1	10	6
Hit or write-back access	1	6	8

Figure 7: Cost Model (in processor cycles)

Block transfer time depends on the block size, bus width, and bus speed. The parameters of the proposed Futurebus+ (cycle time of 10 ns and a maximum data width of 256 bits) are used in this estimate. The on-chip cache has a block size of 32 bytes, and all other caches have a block size of 128 bytes. The block transfer time for 128 bytes over a 256-bit bus is 4 cycles. Memory access time is taken as 12 cycles (120 ns). The miss handle time in the MPM cache (level 2 here) is taken to be 10 cycles. This assumes common-case hardware assist for the software handler*. Also shown in Figure 6 is the time for a full software handler, which takes 100 cycles to execute.[8]

Using these estimates, the worst-case latency for a nine level hierarchy is still substantially less than that of the 1 Gb/s interconnection network. Queuing delays have been ignored in this estimate. However, if system utilization is kept below 50%, the average worst-case latency should be within a factor of two of that shown in Figure 6, assuming that each shared resource can be modeled as an M/M/l queue. Doubling the

*This number is estimated from our experience with a similar mechanism in the VMP machine.

latency for the bus hierarchy to allow for queuing delays means that at least 4 levels of bus hierarchy are still justified. We conclude that very large machines (over 200 processors) can be built as a single ParaDiGM node, relying on the shared bus and cache hierarchy for most intercache and interprocessor communication. However, this conclusion is predicated on modest loading of the shared caches and busses, an issue we explore further in the next section.

4.2 Contention Latency of Shared Caches

Shared caches are used as an interconnection mechanism between different levels of the bus hierarchy, reducing the load on the next level bus, thereby allowing more modules per bus, and thus greater scalability. A shared cache also has a higher hit ratio than private caches if there is any significant amount of sharing between processors, which is the case in shared-memory parallel computations. Previously, we measured a 50% reduction in miss ratio when caches were shared by eight processors [2]. Thus, a shared cache has a lower effective response time because of fewer misses, which are each equivalent in cost to a large number of cache hit times. This benefit is reduced to a modest degree by interprocessor *replacement interference,* where useful blocks are replaced to make space for new blocks. Replacement interference is easily reduced by using a larger cache. In fact, shared caches end up being oversized naturally as a result of making them wide (and therefore using many memory chips) to achieve fast response. Replacement interference is also reduced by increasing the associativity of a cache.

However, sharing caches causes contention among the sharing processors, resulting in slower access to the shared caches. We argue that this contention is not significant in ParaDiGM. First, the number of processors (or MPMs or ICMs) sharing a cache is limited to a number that results in a low enough utilization, so that the queuing delay is not significant. As long as the utilization is below 50%, the average queuing time is less than the average service time, which is approximately the hit access time of the shared cache, assuming the shared cache has a low miss ratio and it can be modeled as an M/M/l queuing system. Second, a shared cache whose cost is amortized over several processors can reasonably be made wider to provide faster response to cache misses. The faster response in providing a full cache block size (of 32 bytes or more) compensates for any queuing delay.

4.3 Bus Loading Latency

The capacity of a bus is fixed, in contrast to the capacity of a point-to-point network which grows by adding more links and switches. For example, the 256-bit wide Futurebus+ has a maximum bandwidth of 3.2 GB/s. The aggregate bandwidth of a point-to-point network with 1 Gb/s links and 50 or more nodes exceeds that of a 256-bit Futurebus+. For a 100 Mb/s network, this crossover occurs when the machine has 500 processors. (Of course, the bandwidth between a given pair of processors cannot exceed the bandwidth of a single link.)

When the load on a bus becomes a significant portion of the capacity, the latency is significantly increased by queuing delay, leading to poorer processor performance. In the shared cache hierarchy, a bus has to support a fraction of the interprocessor communication of all the processors connected by that bus. This fraction, and the interprocessor communication pattern in general, depend on the application behavior.

The importance of application structuring can be illustrated by considering a pessimistic scenario for the architecture, namely one in which all the references to a read/write shared block are writes, and where all the processors are equally likely to reference the block, in other words, there is no locality in the reference pattern.

We quantify the benefit of shared busses and caches as the bus load *ratio* R, defined as the ratio between T_{shared}, the memory bus traffic in a system with shared caches, and T_{private}, the memory bus traffic in a system with private caches. Let p processors be connected to a shared cache, with n processors in the system. There are $m = n/p$ shared caches in the system. Assume that a certain reference τ to a cache block b is by a processor which is connected to shared cache S_i. Then a block transfer is required if the previous reference to b (reference $\tau - 1$) was by a processor which is not connected to S_i. The probability of this is $1 - p/n$, which is the miss ratio of the shared cache under these assumptions. The bus load ratio R is therefore given by

$$R = \frac{1 - p/n}{1 - 1/n} = \frac{n - p}{n - 1} \quad (1)$$

For this type of read./write sharing, the maximum scalability is obtained when p is maximized. This happens when the processors are divided into two groups of equal size, which gives $p = n/2$. Thus, with applications

exhibiting this random memory reference pattern, a. shared bus interconnecting one level of shared cache would saturate, precluding extending the bus hierarchy further.

In summary, a shared bus and cache hierarchy appears to provide significant performance benefits in augmenting network interconnection, provided that large parallel applications exhibit a sufficient degree of locality in communication and memory references. Application structuring to minimize bus load and maximize performance is considered next.

5 Application Structuring

Large-scale applications must be structured to map well onto the shared cache and bus hierarchy to realize its full potential and to avoid rather poor performance that can result in the worst case. The application must have a high degree of locality of reference to shared data in each level of the hierarchy. We illustrate this type of application structuring by considering the problem of programming a partial differential equation (PDE) solver for ParaDiGM, using the successive over-relaxation (SOR) algorithm to calculate a numerical solution. The PDE is approximated by a finite difference equation over a discrete grid.

The SOR algorithm calculates a new value for each grid element by reading the values of the four *nearest neighbors* and the grid element itself. The computation has good locality-each element only affects its four nearest neighbors. Grid elements are divided into two groups, called *red* and *black*, based on the sum of their indices. Each iteration of the SOR algorithm consists of two sweeps, one for updating each color. All the red (black) elements can be updated in parallel, since each red (black) element only depends on the value of black (red) elements, and its own value. Assuming a large number of elements, the algorithm allows a high degree of parallelism. For good performance, the elements must be mapped onto processors and caches in a way that maximizes locality and minimizes contention.

First, we partition the grid into subgrids containing multiple elements and assign each subgrid to a separate processor. We assume for simplicity that the grid is a square of size N x N, with the number of processors n = $(n')^2$ also a perfect square. The number of elements in a subgrid $N_0 =$ N/n' is assumed to be an integer. An example of this partitioning is shown in figure 8 for N = 16 and n = 16. Only those grid

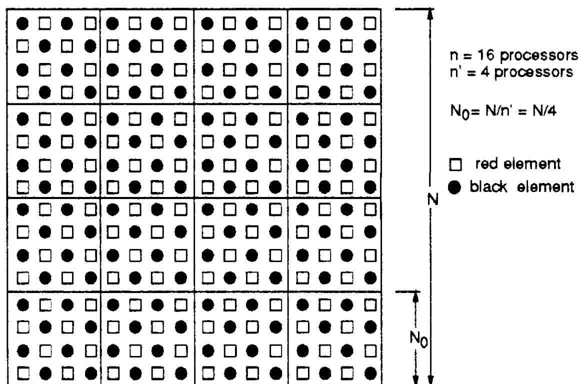


Figure 8: SOR algorithm

elements that form the border of each subgrid have to be communicated between processors.[9] Increasing the subgrid dimension increases the amount of work per processor quadratically, but only increases the boundary size, and thus the interprocessor communication, linearly. Therefore, it is advantageous to pick the subgrid size as large as possible, subject to the subgrid fitting in the processor's cache.

Interprocessor communication traffic on a next level bus is minimized by allocating adjacent subgrids to processors sharing a common cache. For example, the right hand side of Figure 9 shows *supersubgrids* of 4 adjacent subgrids each being allocated to sets of 4 processors sharing a cache.[10] The subgrid labels indicate the assignment of subgrids to processors. The shaded area on each grid shows the data that has to

[9] Simulation shows that read/write sharing required for this interprocessor communication is the performance-limiting factor in large-scale multiprocessors.

[10] This shared cache is intended to be a second-level cache as on the MPM. The first level private cache is not show or discussed for simplicity and brevity.

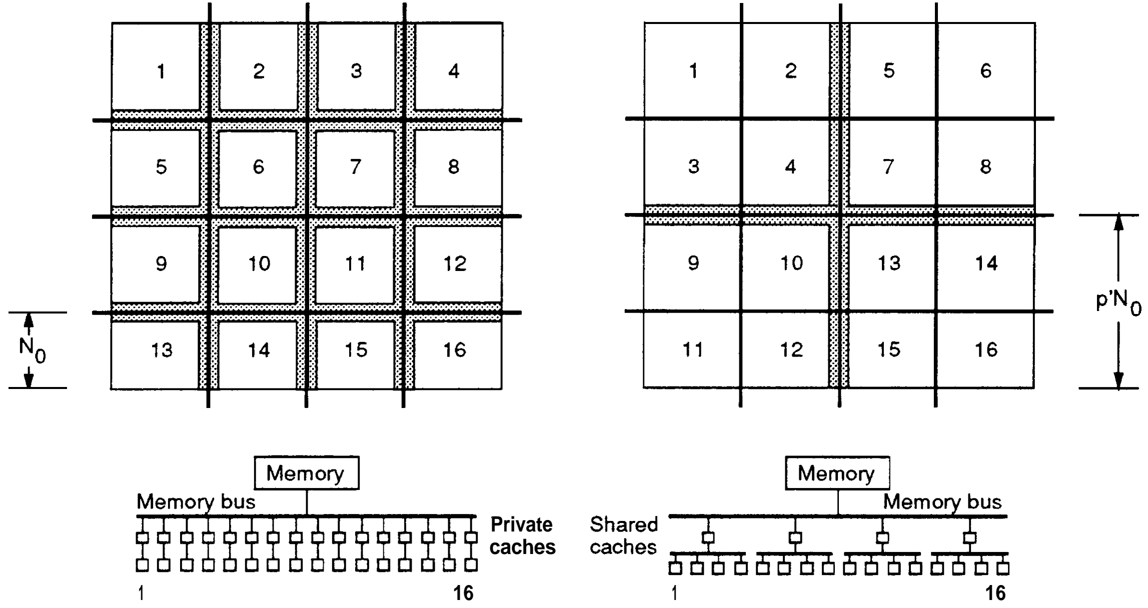


Figure 9: Bus traffic comparison

be communicated over the memory bus during each iteration. Only those elements which form the outside border of the collection of subgrids have to be communicated on the memory bus. The left hand side of the figure shows that all border areas have to be communicated over the memory bus when each processor has a private cache. This bus load also arises if adjacent subgrids are not allocated to processors sharing a common cache.

The reduced traffic on the memory bus in the shared case allows more processors to be connected to the bus, quantified as the bus load ratio, as defined in Section *4.3.* We denote the number of groups of processors sharing a cache by m, and the number of processors per group as $p = n/m$. Then p' $= \sqrt{p}$, and m' $= \sqrt{m}$ (m and p are assumed to be perfect squares).

Referring again to Figure 9, the memory bus traffic is proportional to the number of shaded regions if N is large. In that case, $T_{\text{shared}} \propto 2(m' - 1)$, and $T_{\text{private}} \propto 2(n' - 1)$, which gives

$$R = \frac{T_{\text{shared}}}{T_{\text{private}}} \approx \frac{m' - 1}{n' - 1}. \tag{2}$$

According to Equation 2, R is minimized by small values of m', in other words, by having as few shared caches as possible on the memory bus, with as many processors per shared cache as possible. As m' and n' increase, R approaches l/p' from below. The smallest value of m' consistent with our assumptions is 2 (4 shared caches), which gives R = l/n' $= 1/\sqrt{n}$. Therefore

$$1/\sqrt{n} \leq R < 1/\sqrt{p}.$$

If there are four shared caches (m' = 2), equation 2 shows that the traffic on the memory bus scales as the square root of the number of processors in the system. Therefore, if a system with private caches has sufficient memory bus bandwidth to support x processors, the memory bus on the system with four shared caches has enough bandwidth for x^2 processors. For example, a bus capable of supporting 16 processors with private caches could support the communication between shared caches for 256 processors, at least with the application structure we propose for the PDE solver. Thus, with this particular application, the program can be structured to make good use of a deep shared cache and bus hierarchy, providing significantly greater scalability and performance than using pure network interconnection, which is equivalent to the private cache model.

Many applications of interest can be viewed as simulations operating in a finite-dimensional space, and thus can be structured similarly to the PDE solver. In particular, a subspace can be assigned to each

processor, and most of the inter-processor communication is then between processors operating on adjacent subspaces, reflecting the physics of the simulated system. These adjacent subspaces can be assigned to processors in the same shared cache as done with the above example. As one supporting data point, we have restructured a sparse gas particle simulation along these lines [3] and measured under simulation a reduction in cache miss traffic by a factor of 30 (to less than 1 percent) running on a. ParaDiGM architecture. A digital circuit simulator is another example that is amenable to this structuring because each circuit element has limited fan-out. Output changes of an element only influence a fraction of the elements in the circuit. The data in the simulator can be arranged to reflect this spatial locality. Network simulation provides similar opportunities to achieve locality and minimal contention.

We conclude that a significant number of important applications can be programmed to work well on this architecture. Although this structuring is similar to that required for distributed memory machines, our structuring requirements are weaker. In particular, the shared data does not need to be strictly partitioned between separate memories, only partitioned so that *statistically* the references are local. In contrast to distributed memory architectures, the shared-memory architecture of ParaDiGM allows its use for timesharing, transaction processing and other general-purpose uses. We hypothesize that appropriately structured applications will result in such low loads on the bus hierarchy that the size of this hierarchy is likely to be constrained by reasons other than performance, including packaging, power and cooling, unit of failure, configuration balance, and administration.

6 Status

The ParaDiGM represents (and requires) the culmination and focus of several projects with the V software and VMP hardware. We are progressing incrementally in the development, evaluation and construction of hardware because of the magnitude of building a full-scale ParaDiGM configuration.

The MM has been built and tested. The transfer speed in the prototype (using the VME bus) is approximately 40 megabytes per second. (Our board utilizes a two-edge handshake protocol, not the VME standard block transfer protocol.) We plan to use existing VMP processor boards initially because they require only minor modifications to work with the MM. The MPM is still in design as we evaluate candidate microprocessors. The ICM, combining the logic of the MM and MPM, is still at the initial design stage.

The V distributed system has been ported and runs on the original VMP processor modules. We are currently reworking the V kernel to provide cleaner and faster parallel execution within the kernel. In related work on distributed operating systems, we have been investigating a distributed virtual memory system that provides memory consistency of virtual memory segments shared across a cluster of networked workstations.

7 Other Scalable Parallel Machines

Scalable parallel machines can be classified by whether the machine provides no shared memory, partial shared memory or full shared memory. The Cosmic Cube [13] is an example of the first category. Each processor has a local memory and connections to all its neighbors in the hypercube. Programming this machine requires partitioning the application across all the nodes, which is currently done by hand. Although some applications partition easily and run well on this machine, the lack of shared memory is a compromise favoring ease of (hardware) implementation over ease of programming. There has been some work to implement shared virtual memory on top of the cube architecture using distributed operating system techniques [12]. However, the performance of this extension suffers from a lack of memory management hardware.

The Connection Machine is another example of a machine in this class. Significant concessions were made in the nature of its processors to facilitate large-scale parallelism. At 0.1 MIPS per CM-2 processor, a fully configured 64000-processor connection machine is roughly equivalent to a 65-processor ParaDiGM (using 100 MIPS processors) or a 325-processor ParaDiGM built from 20 MIPS processors. However, some applications that map well to the connection machine architecture would significantly tax a ParaDiGM architecture.

Partially shared-memory machines generally provide non-cached access to shared memory, with program code and private data stored in local memory, avoiding the need for a cache consistency mechanism. Examples include the BBN Butterfly and IBM RP3. In these architectures, shared data access is significantly more expensive than local access. Shared data access is also more expensive than on a fully shared-memory machine with caches, unless the amount of contention on the data is very high. With the high cost of shared memory access on these machines, the application must be carefully partitioned to limit references to shared data. Shared data that is read frequently but written infrequently is particularly a problem because it must

reside in the shared memory area to allow for updates, but incurs a significant read performance penalty because of the cost of shared memory access. In some cases, the shared memory is simply used to implement a message system between processors, simulating a. hypercube architecture. In general, these architectures force a more static partitioning of the application data than is necessary with ParaDiGM, where the caches automatically redistribute data in response to changing access patterns. The benefits of the non-cached shared-memory architectures do not appear to compensate for the difficulty of programming them to achieve good parallel program performance.

The ParaDiGM is more directly comparable to other fully shared-memory systems, such as DASH [11], the Wisconsin Multi-cube [9] and the Gigamax [14]. DASH is focused on interconnecting existing multiprocessors by an interconnection network to provide a modest scale machine. Because of the more limited scale and existing software (Unix), DASH only uses the interconnection network for implementing a shared physically addressed memory, in contrast to the more general file-based shared-memory approach taken by ParaDiGM. DASH will provide a testbed system to investigate parallel programming and interconnection networks, and is thus constrained by the need to get a system running quickly.

The Wisconsin Multicube implements a large-scale shared-memory multiprocessor. However, it does not use shared caches and relies on a two- or three-dimensional mesh to interconnect caches. To date, the work on the Multicube has been largely focused on novel cache consistency mechanisms.

The ParaDiGM resembles most closely the Gigamax [14] being developed at Encore. The Gigamax uses a shared board-level cache among four processors with a similar directory scheme to keep track of copies. Major differences at this level are their use of physically addressed caches, the absence of locking and message support, and the hardware implementation of cache miss handling. Finally, the Gigamax work has not, to our knowledge, used distributed systems technology to achieve high scalability. In particular, they have not considered network interconnection, only bus interconnection. However, their more focused scope has put the Gigamax much closer to being a product than ParaDiGM.

Finally, one might question how a ParaDiGM configuration would compete with a very fast (500-1000 MIPS) uniprocessor [10]. Second and third level caches are required in such a uniprocessor to maximize performance, given the large access time of bulk memory. We hypothesize that the benefits of shared caches in the second and third levels of the memory hierarchy may in fact finally enable multiprocessors to triumph over uniprocessors, even given arbitrarily fast CPUs. While these caches can be made larger than the first level cache, the miss ratio is significant, since a cache miss at these levels costs on the order of 200 CPU cycles. The miss penalty is therefore large enough to significantly degrade processor performance, yet too short to perform a context switch, as is done with page faults (which take thousands of cycles). Thus, with a uniprocessor machine, the entire machine will be idle during these misses. In contrast, a multiprocessor that shares these caches has additional processors to utilize the caches during a miss by one processor. This approach is preferable over fast context switching support in the single processor for several reasons. First, the multiprocessor approach provides more parallel cycles when all processors are executing out of their local caches, which is most of the time. Second, first level caches are too small to support multiple contexts, so a fast context switch would incur the cost of flushing and reloading the cache, both for the new context and on return to the old one. Finally, commercial microprocessors do not, and are not likely to, support fast context switching (with the required multiple register sets, etc.) because there are many other demands on the design efforts and VLSI real estate, such as a larger on-chip cache and better floating point support.

8 Concluding Remarks

ParaDiGM is a scalable general-purpose shared-memory parallel architecture. Using a building block technology, machines can be constructed with two to hundreds of powerful processors. The same applications and programming systems can run on all these configurations providing they are structured to maximize locality, minimize contention and take advantage of key ParaDiGM facilities such as locking and message exchange.

Several aspects of the ParaDiGM design are of particular interest. First, ParaDiGM is based on a distributed systems model of processor/memory nodes connected by a network with the operating system supporting RPC, shared memory, process groups, multicast, and a file-based virtual memory which is extendable with file replication and atomic transaction support to support fault tolerance. This approach has also lead to a highly partitioned and replicated structure for the operating system, including facilities such as process migration. These facilities, which are natural to consider in the loosely coupled distributed systems

environment, are critical for scalability and yet can be highly optimized within a network node to match or exceed the efficiency of more conventional techniques.

Second, ParaDiGM exploits shared cache/bus technology for low-latency coupling between local clusters of processors as an optimization over processor interconnection using only network technology. A hierarchy of shared caches and busses is used to maximize the number of processors that can be interconnected with current bus and cache technology. A hierarchical directory-based consistency scheme implements coherence without needing broadcast. Preliminary evaluation of this design shows that sharing reduces the miss ratios and hardware costs of these caches, and reduces the contention between caches. Consequently, the average memory reference cost and the network load are reduced, resulting in better performance than a. non-shared design. The reduced amount of hardware leads to a significant reduction in cost as well as an improvement in reliability.

Several optimizations of the shared cache and bus hierarchy provide further efficiency gains. The memory-based message exchange mechanism supports efficient interprocessor calls. Similarly, a simple memory-based locking facility reduces the memory coherence contention on shared data structures. We provided a preliminary analysis indicating significant benefits from these facilities if used by contention-intensive parallel applications. As another optimization, a network connection is placed at each second-level cache, rather than at the lowest point in the node memory hierarchy. Network traffic therefore does not load the lower (more highly shared) levels of the memory hierarchy, processors have lower latency access to the network, and the network access scales with the number of processors. With appropriate structuring of application software, these hardware optimizations allow the common cases to be handled efficiently (and largely in hardware). Consequently, the generality of the distributed systems model that forms the base for ParaDiGM, which we have argued is a key strategy for scalability, does not result in a significant performance penalty.

Finally, ParaDiGM relies on application structuring that provides a high degree of locality to communication and contention. We have explored the structuring of a few example applications along these lines and argued that a significant class of applications, namely many simulation problems, naturally exhibit such locality because of the physics of the systems they are simulating. We see general techniques developing for mapping this intrinsic application locality onto cache locality in a ParaDiGM-like cache hierarchy. This structuring approach can be reasonably supported by a parallel programming system. We hypothesize that the shared bus and cache hierarchy that augments network interconnection in ParaDiGM in conjunction with this structuring approach will allow simulations to run far more efficiently than on a machine with only network interconnection.

The ParaDiGM design represents our best notion of the architecture that can and should be built. No concessions have been made to what our research project itself can most easily build. In our work to date, we have designed and implemented a few components of ParaDiGM, such as the memory module, as well as provided an initial performance evaluation of the design based on trace-driven simulation. However, considerable work remains.

The technology required for a ParaDiGM implementation (microprocessors, busses, high-speed networks) is rapidly developing, driven by significant market pressures that are largely independent of the needs of large-scale parallel computation. The full development of a ParaDiGM machine may require waiting for this technology to develop, plus pushing to get the details of the critical chip designs to match our requirements. For example, no microprocessors currently support cache-based locking in their on-chip cache, as we have advocated, yet the provision of this facility is relatively simple and inexpensive in VLSI real estate.

More sophisticated parallel programming systems are required to fully exploit this architecture without excessive burden on the application programmers or loss of performance. However, this is true for all parallel architectures. We predict that, in due course, the programming benefits of shared-memory multiprocessors and the feasibility of scaling these machines, as demonstrated here, will result in an application, operating system and programming support base that far exceeds that available for distributed memory parallel machines. Consequently, we expect shared-memory parallel programming to be the dominant form of parallel programming in the future.

9 Acknowledgments

This paper has benefited from comments and criticisms of members of the Distributed Systems Group at Stanford. It was also significantly revised in response to helpful and insightful comments from the referees. Gert Slavenburg contributed significantly to the original VMP design. We are grateful to Anoop Gupta and

Wolf Weber for making the trace data used in this paper available to us. This work was sponsored in part by the Defense Advanced Research Projects Agency under Contract N00014-88-K-0619.

References

[1] D.R. Cheriton. The V distributed operating system. *Communications of the ACM*, 31(2):105–115, February 1988.

[2] D.R. Cheriton, H.A. Goosen, and P.D. Boyle. Multi-level shared caching techniques for scalability in VMP-MC. In *Proc. 16th Int. Symp. on Computer Architecture,* pages 16-24, May 1989.

[3] D.R. Cheriton, H.A. Goosen, and P. Machanick. Restructuring a parallel simulation to improve cache behavior in a shared-memory multiprocessor: A first experience. Submitted for publication.

[4] D.R. Cheriton, A. Gupta, P.D. Boyle, and H.A. Goosen. The VMP multiprocessor: Initial experience, refinements and performance evaluation. In *Proc. 15th Int. Symp. on Computer Architecture,* pages 410-421. ACM SIGARCH, IEEE Computer Society, June 1988.

[5] D.R. Cheriton, G. Slavenburg, and P.D. Boyle. Software-controlled caches in the VMP multiprocessor. In *13th Int. Symp. on Computer Architecture,* pages 366-374. ACM SIGARCH, IEEE Computer Society, June 1986.

[6] D.R. Cheriton, G.R. Whitehead, and E.W Sznyter. Binary emulation of Unix using the V kernel. In *Usenix* Summer Conference. Usenix, June 1990.

[7] B.G. Lindsay et al. Computation and communication in R*: A distributed database manager. *ACM Trans. on Computer Systems,* 2(1):24–28, February 1984.

[8] J.N. Giacopelli, M.Littlewood, and W.D.Sincoskie. Sunshine: A high performance self-routing broadband packet switch architecture. To appear ISS'90, 1989.

[9] J.R. Goodman and P.J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proc. 15th Int. Symp. on Computer Architecture,* pages 422-431. June 1988.

[10] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th Int. Symp. on Computer Architecture,* pages 364-373, May 1990.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. 17th Int. Symp. on Computer Architecture,* pages 148-159, May 1990.

[12] K. Li and R. Schaefer. Shiva: An operating system transforming a hypercube into a shared-memory machine. CS-TR 217-89, Princeton University, 1989.

[13] C.L. Seitz. The Cosmic Cube. *CACM*, 28(1):22–33, January 1985.

[14] P. Woodbury et al. Shared memory multiprocessors: The right approach to parallel processing. In *Spring Compcon,* pages 72-80. IEEE, February 1989.