Toward A Cost-Effective DSM Organization That Exploits Processor-Memory Integration*

Josep Torrellas[†], **Liuxi Yang**[‡], and **Anthony-Trung Nguyen**[†]

[†]University of Illinois, Urbana-Champaign
{torrella,anguyen}@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

[‡]Sun Microsystems
liuxi.yang@eng.sun.com

Abstract

Dramatic increases in the number of transistors that can be integrated on a VLSI chip will soon allow commodity microprocessors to include both processor and a sizable fraction of main memory on chip. Distributed Shared-Memory (DSM) multiprocessors typically use the latest off-the-shelf microprocessors and thus will be affected by the upcoming processor-memory integration. In this paper, we explore how a cache-coherent DSM machine built around Processor-In-Memory (PIM) chips might be cost-effectively organized.

To take advantage of the close coupling between processor and memory, we propose tagging the memory and organizing it as a cache. Furthermore, commercial considerations dictate the use of off-the-shelf hardware largely designed for uniprocessors. Consequently, we keep the directory control off-chip. To keep the multiprocessor cheap and simple, and to allow for reconfigurability, directory control is performed by chips that are identical to the ones used as compute nodes. As a result, the machine hardware can be easily reconfigured for computing or coherence-handling depending on the needs of the application. We also propose a cache coherence protocol that is tailored to our architecture: it uses the memory very efficiently while exploiting the large caching space available. Overall, the resulting machine is simple and inexpensive, and delivers performance that is comparable to, and higher than, the more expensive traditional COMA and CC-NUMA organizations, respectively.

1 Introduction

Vast increases in the number of transistors that can be integrated on a VLSI chip are fueling the trend toward integration of processor and memory on a chip. It is widely expected that off-the-shelf microprocessor designs will exploit this trend to provide low-latency and high-bandwidth communication between processor and memory [1, 4, 6, 9, 12, 16, 19].

Since directory-based, cache-coherent Distributed Shared-Memory (DSM) multiprocessors are typically built around the latest off-the-shelf microprocessors, they will be affected by the trend of progressive processor-memory integration. Currently, the nodes in DSM systems are typically organized as in Figure 1-(a). The processor accesses memory via the memory bus, to which directory controller and memory controller are connected. Processor-memory communication occurs through low-bandwidth and high-latency paths. To access main memory, a processor request has to traverse many levels of caches and use the bus. Bus transactions are slow because they typically require bus arbitration, synchronization between different clock frequencies, and support for cache coherence.

As technology advances, the nodes of DSM systems will get more integrated. For example, they may start looking like Figure 1-(b), where the memory controller is moved on chip and the memory is directly connected to the processor chip via a dedicated point-to-point bus. This type of memory system is enabled, for example, by a Rambus memory interface [5], and has been announced for microprocessors like Alpha 21364 [6] and UltraSPARC III [12]. With this dedicated bus, memory can be accessed with high bandwidth.

As more transistors can be placed on a chip, commodity microprocessors are likely to incorporate a sizable fraction of the main memory on chip. In this case, DSM nodes may be organized like in Figure 1-(c). In this design, part of the main memory is on the processor chip, while part of it is off chip. Eventually, most of the memory may be on chip. In these designs, processor-memory communication is fairly inexpensive.

In the past, there have been efforts trying to use commodity workstations as nodes to build DSM machines [2, 11, 14, 17]. In this paper, we explore how to design a cache-coherent DSM machine around commodity Processor-In-Memory (PIM) chips like the one in Figure 1-(c).

In our opinion, such a design should be guided by several principles. First, given the close coupling between processor and on-chip memory (20-30 ns round-trip latency), the design should exploit physical locality as much as possible. Second, given the relatively modest size of the multiprocessor market, the design should ideally use off-the-shelf PIM chips designed for uniprocessors. Finally, the machine should be able to run many different types of parallel applications with high performance.

In this paper, we propose a design that addresses these issues. Physical locality is exploited by tagging the memory and organizing it as a cache. We expect these on-chip memory tags to also be useful for uniprocessor designs that need off-chip memory expansions [18]. To maximize the use of off-the-shelf, uniprocessor PIM chips, directory control is kept off the processor chip. To keep the multiprocessor cheap and simple, directory control is performed by PIM chips that are

*This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, ASC-9612099, and MIP-9619351, DARPA Contract DABT63-95-C-0097, NASA Contract NAG-1-613, and gifts from IBM and Intel.

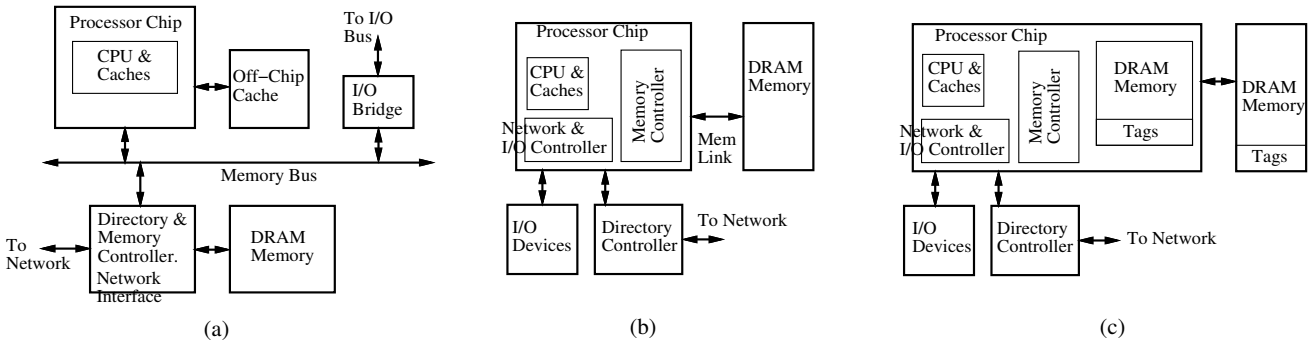

Figure 1: Different organizations of a node in a DSM system.

identical to the ones used as compute nodes. Thanks to this, the machine can now be reconfigured in software to provide more or less computing and coherence-handling capabilities, so as to match the needs of the application. Finally, we also propose a cache coherence protocol that is tailored to the architecture.

Overall, the resulting DSM machine contains a single type of off-the-shelf uniprocessor PIM chip plus some extra DRAM memory to hold large applications. Furthermore, our simulations show that this simple machine delivers performance that is comparable to, and higher than, the more expensive traditional COMA and CC-NUMA organizations, respectively.

This paper is organized in three parts: Section 2 discusses the architectural issues involved and proposes a design, Section 3 describes how we evaluate the design, and Section 4 evaluates it.

2 Architecture Design

The building block of our design is a commodity PIM chip largely designed for uniprocessors. For example, a design that may appear within the next 5 years may include a 1 GHz high-end superscalar with 128 Mbytes of DRAM. Applications running on such a processor chip are likely to need more memory than can fit on chip. Consequently, we use a processor chip like the one in Figure 1-(c), where part of the memory is on chip and part is off chip. To save space, the on- and off-chip DRAM contain exclusive data. Moreover, for reduced memory fragmentation, the transfer of data between on- and off-chip memory occurs at a memory-line grain size. Specifically, when the processor references a line that is found in the off-chip DRAM, that line is moved to the on-chip DRAM, possibly displacing a second line into the off-chip DRAM.

To explore cache-coherent DSM designs based on this chip, we consider three main issues: node design (Section 2.1), system design (Section 2.2), and reconfigurability (Section 2.3).

2.1 Node Design

2.1.1 Exploiting Physical Locality

It is harder to exploit tight processor-memory integration in traditional DSM machines than in uniprocessors. Indeed, consider the operations that follow a cache miss. In a uniprocessor, a cache miss is followed by a memory access, which takes advantage of tight processor-memory integration. In a traditional CC-NUMA, however, a cache miss is followed by a local or a remote directory/memory access. If the access is remote, the tight processor-memory integration is wasted.

Unfortunately, accesses to remote directory/memory modules may be frequent. Which directory is accessed depends on the physical address issued by the processor. For an N-node DSM, the local directory contains entries for only $1/N$ of the addresses. While many programs exhibit good locality or can be optimized for locality, others certainly do not.

To reduce the number of remote directory/memory accesses, we propose to augment each line in the local memory (both on- and off-chip) with extra bits that contain the line's state (invalid, shared, or dirty) and address tag. Furthermore, we treat the memory as a cache, allowing lines to migrate to and/or replicate in other nodes. With this support, a processor can directly access its local memory after a cache miss, irrespective of the address of the requested line. If it is reading from a valid line or writing to a dirty line, there is no need to access any directory. Consequently, we are exploiting the processor-memory integration. In addition, using the local memory as a cache enables the processor to keep large working sets within very small round trip latencies. Section 2.2.2 presents a non-COMA cache coherence protocol tailored to this organization.

Adding tags to the local memory has a modest cost and, in addition, is useful for uniprocessors too. For example, assume that the total size of all the pages mapped in the machine is N times the size of the on-chip memory of a node. Furthermore, assume that the latter is A-way set-associative. Each on-chip address tag then needs $log_2(NA)$ bits. If, for instance, N is 16384, A is 4, memory lines are 128 bytes, and the cache coherence protocol has four states, then the overhead of the state and address tag bits in a memory line is 1.8%. The overhead is smaller for the larger off-chip memory.

Adding tags to on-chip memory is also useful for PIM-based uniprocessor machines when they need off-chip memory expansions. Specifically, Saulsbury *et al* [18] have shown how to use on-chip tags to manage, in hardware, uniprocessor memory partitioned into on- and off-chip portions. Consequently, we can use this off-the-shelf PIM uniprocessor hardware and software-reconfigure the memory controller so that the on-chip tags are used differently. In addition, we also need to add tags to the off-chip memory. This we can do while still using only off-the-shelf DRAM chips.

2.1.2 Keeping MP Hardware Off Chip

Since directory controllers are not used by uniprocessors and the latter are likely to dominate the market, there may be little incentive to complicate the latest off-the-shelf microprocessors with on-chip directory controllers. Furthermore, it is quite unclear what functionality a "standard" on-chip directory controller should provide. Consequently, in our node design, we keep the directory controller off-chip.

Note that any performance degradation resulting from separating processor and directory controller in our system may well be quite modest. Indeed, since we organize the large local memory as a cache, there will be few overflow-induced local accesses to the directory. Moreover, the directory accesses that remain after start-up may be mostly remote anyway. This is because they are largely caused by coherence misses, and data with coherence misses is often hard to place optimally in memory.

Including the directory controller on chip may even degrade performance and efficiency in some cases. Indeed, due to the variable directory locality of applications, a sizable fraction of the accesses to the local directory/memory may come from remote processors instead of from the local one. This incoming traffic can consume valuable pin and on-chip bandwidth.

2.2 System Design

2.2.1 Using Few Different Types of Chips

To complete the system, we must provide for protocol processing and, since the memories described so far function as caches, for plain memory to back up the caches. We can certainly provide the traditional solution: a custom-designed protocol engine chip per node and extra memory per node. Using a custom-designed protocol engine, however, eliminates the possibility of reconfiguring the machine for different assignments of resources to computation and coherence-handling (Section 2.3).

To keep the machine as inexpensive and simple as possible, and to support reconfigurability, in this paper we explore implementing the protocol controller as a separate node with the same off-the-shelf processor chip built for uniprocessors that we use in the processing nodes (Figure 1-(c)).

As a result, there are two types of nodes with the same hardware. These nodes we call computing or processing nodes (*P*-nodes), and non-computing or directory nodes (*D*-nodes). Figure 2-(a) shows the resulting multiprocessor. The ratio of *P*- to *D*-nodes varies as we reconfigure the machine for an application. Usually, there are fewer *D*-nodes than *P*-nodes. In addition, as we will discuss in the section on reconfigurability, the *D*-nodes may be "fatter" in memory: they may contain more memory chips attached to the fast link to the processor chip (Figure 2-(b)).

Each *D*-node is home to a fraction of the physical addresses. For that range of addresses, the *D*-node performs two functions. First, it keeps and maintains the directory state. Using a DSM cache-coherence protocol, it sends coherence messages to the *P*-node memories to ensure that the data in that range of addresses is kept coherent everywhere. Second, the *D*-node uses its own memory as the only back up memory in the machine for that range of addresses. Its memory contains, in most of the cases, an up-to-date copy of all the lines in that range of addresses that are not owned by

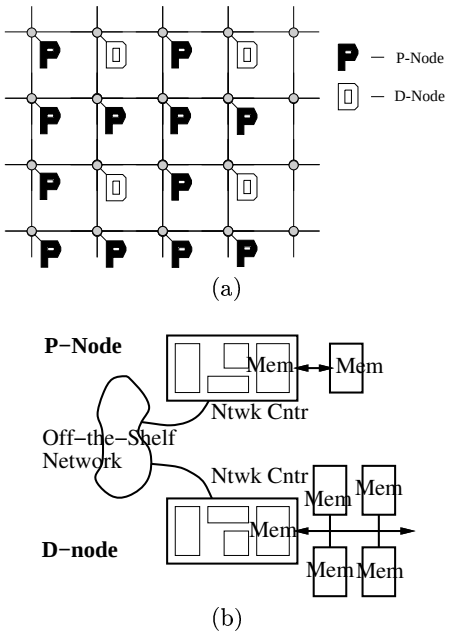

Figure 2: Organization of a DSM machine that uses a single type of PIM chip plus, if necessary, additional memory chips.

any *P*-node. However, we use a software technique to avoid consuming much *D*-node memory (Section 2.2.2). We avoid the COMA approach of not guaranteeing back up memory in the home because it induces injections of memory lines into memory modules when space needs to be found [8]. Injections complicate the coherence protocol and pollute memory modules (Section 4).

Since these *D*-nodes are off-the-shelf PIM chips plus memory, they do not have any special hardware support to perform these two functions. These functions are performed by software handlers running on the processor and accessing software data structures. Section 2.2.2 presents efficient algorithms tailored to this architecture. In addition, we use the on-chip memory tags to manage the off-chip memory extension in hardware as it is done in [18] for uniprocessors. Lines move to the on-chip memory on being referenced and there is no replication of lines between on- and off-chip memory.

Overall, the machine is built out of a single type of off-the-shelf PIM chip designed for uniprocessors, off-the-shelf DRAM chips, and an off-the-shelf network. The design also assumes that the on-chip network and I/O controller is somewhat programmable. Such programmability involves modest complexity and is necessary to make an integrated chip useful in a variety of environments. With such hardware, communication between *P*- and *D*-nodes proceeds as follows. When a *P*-node misses in its local (on- and off-chip) memory, the memory controller informs the network and I/O controller. The latter generates a message directed to the correct destination and injects it into the network. When the message reaches the *D*-node, the processor detects it by polling memory-mapped registers in the network and I/O controller. Later, when the *D*-node is ready to send the reply, the processor simply posts the message in its network and I/O controller. Finally, when a reply or invalidation reaches a *P*-node, the network and I/O controller informs the memory controller. The latter, without involving the

processor, acts like a cache controller. It may write a memory line back to the network, store the incoming line into the local memory, or invalidate a line in the local memory. This memory controller support requires some limited programmability, which is likely to be needed even in uniprocessor systems, to support different memory standards and interfaces.

2.2.2 Coherence Protocol and Storage

We make some changes to a directory-based cache coherence protocol like DASH's [13] to tailor it to our architecture. The goal is to use the memory of *D*-nodes efficiently, while exploiting the large caching space of *P*-nodes. In the following, we first explain the protocol and then its implementation.

Protocol

Directories keep information about which *P*-nodes cache the data and in what state. The large caches of *P*-nodes enable them to keep a large working set for an application without many conflict-induced displacements. Potentially, many of the lines in the working set may be cached in state dirty. It would be a vast waste of memory if the corresponding *D*-node memories kept place holders for the lines that are dirty in *P*-nodes.

To save space, *D*-node memories do not do so. Instead, the place-holder memory location in the *D*-node is reused to hold another line whose home is the local *D*-node and that is not dirty in any *P*-node. For this reuse to work, we organize the memory in each *D*-node in a fully-associative manner in software. The allocation is done at the line level and only for the lines whose home is this node.

Moreover, if necessary, we can reduce the memory requirements further by not keeping in *D*-nodes, copies of lines that are clean in the cache of at least one *P*-node. To support this feature, we add the COMA-inspired *shared-master* state. One of the cached copies of the line is set to the shared-master state. Then, the home *D*-node can free up the location that the line was using in memory if space is needed. A *P*-node read miss on a shared line may now involve 3 node hops because the home may not keep a copy of the line. For this reason, we try to keep shared lines in the home most of the time. Of course, if a shared-master line is displaced from a *P*-node, it must be written back to the home.

With this approach, we expand the perceived capacity of the physical memory of the machine beyond the combined size of the *D*-node memories. We expect that many of the dirty (and, optionally, shared-master) lines in *P*-nodes will remain there without being displaced by conflicts. Their space in the home memory can be reused. Of course, each *D*-node keeps as many directory entries as memory lines exist in all the pages that it has mapped. When a *D*-node receives many line write-backs, we free up space in that *D*-node by paging out pages to disk. Unlike in COMA, we do not support injections of lines to other *D*-node memories or *P*-node caches. We avoid them by using software-based fully-associative *D*-memories and paging out pages if more space is required. This approach simplifies the protocol and prevents memory pollution.

Implementation

We implement the directory and the fully-associative organization of the memory with three software data structures in each *D*-node: the *Directory*, *Data*, and *Pointer* arrays (Figure 3). The Directory array has more entries than the other two arrays, which have the same number of entries. As a program executes, the more frequently-accessed portions of these software structures will naturally reside in the caches of the *D*-node chip.

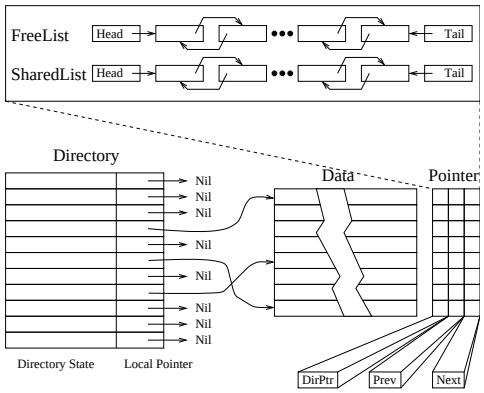

Figure 3: Organization of the memory in a *D*-node.

While many organizations are possible for the Directory array, we use a simple direct-mapped table with one entry reserved for each of the lines whose home is this node. Each entry includes directory state information for the line plus a pointer (*Local Pointer*) to where the line is stored in the Data array. If the line is not in memory, the pointer is nil. With this organization, the memory becomes fully associative: a memory line whose home is this node can be stored anywhere in Data. The whole memory can be utilized.

The Pointer array helps insert and remove lines from memory. Each entry is associated with one entry in Data and contains three pointers. One of them, *DirPtr*, is a back pointer to the Directory. If the Data entry is empty, *DirPtr* is nil. The other two pointers, *Prev* and *Next*, may tie the associated Data entry to one of two doubly-linked lists: *FreeList* and *SharedList*. If Data is empty, the pointers are tied to *FreeList*. If, instead, the line is in state shared non-master, the pointers are tied to *SharedList*. Finally, when the home has the master copy, the pointers are nil.

When we need to save a line in memory, we use the first entry from *FreeList*. Optionally, if *FreeList* is exhausted, we can reuse the first line from *SharedList*. We need to be cautious, however, not to reuse much of *SharedList*: it may result in many transactions requiring 3 node hops instead of 2. Therefore, if *SharedList* falls below a certain threshold, we page out some pages to free up space.

For locality reasons, we manage *SharedList* in a FIFO fashion. A line is inserted at the tail of the list when the first *P*-node reads it, since we give out mastership. When we need space, we pick the line at the head of the list. We hope that, because of locality, it is not actively used.

The actual line insertion and removal proceed as follows. To insert a line, we select the Pointer entry at the head of *FreeList* (or *SharedList*). We store the line in the corresponding Data entry, and set *Local Pointer* and *DirPtr*. If the memory will not keep mastership, *Prev* and *Next* tie the line to the tail of *SharedList*; otherwise, they are set to nil. When a line is removed from memory, we set *Local Pointer* to nil, unlink the Pointer entry from *SharedList* (if it applies), and link it to *FreeList*. Finally, the Pointer array is also updated when memory gains or loses the mastership

of a line: the line is unlinked from or linked to *SharedList* respectively.

Overall, the space and time overhead of this implementation is modest. Consider first the space taken by the Directory and Pointer arrays. Suppose that we can have up to 1024 *P*-nodes and that the directory uses a 3-pointer limited-vector scheme. Furthermore, we use 32 bits for *Local Pointer*, thereby addressing up to 4 Glines of on- or off-chip DRAM per *D*-node. Adding 2 bits for the state of the line, we end up with 64 bits per Directory entry. We use 32 bits for each pointer in the Pointer array. Suppose that the memory lines are 128-bytes long, there are 128 Mbytes of DRAM in each *D*-node PIM chip and that, as we assume in the evaluation section, the Directory array has 50% more entries than the Data array. In this case, it can be easily shown that the Directory and Pointer arrays each take 7.9% of the on-chip DRAM in the *D*-node PIM chip. The remaining on-chip DRAM and the off-chip DRAM are available for Data. Directory and Pointer entries can also be stored off-chip.

While this implementation is not as fast as a custom-designed protocol processor, two effects minimize its time overhead. First, the large *P*-node memories suffer few overflow misses, which reduces *D*-node activity. Secondly, the Directory, Data, and Pointer arrays are naturally cached in the caches of the *D*-nodes, which speeds up protocol transactions. Note also that, in a transaction, both the directory state and the corresponding *Local Pointer* can be accessed in parallel. Since they share the same cache line, at most one miss will occur. Once *Local Pointer* is read, the corresponding Data entry can be accessed.

This indirection through *Local Pointer* can be avoided if *D*-nodes managed their memories in a limited set-associative manner instead of fully-associatively. However, the scheme would then lose the major advantage of not having to support COMA-like bouncing of lines. Indeed, an incoming line arriving at its home could find its set full. We would then have to inject one of the lines into another node.

In our scheme, incoming lines are always taken in by their home memory. When, in a *D*-node, the memory available to take lines in reaches a certain low threshold, the operating system pages out a few pages to disk, therefore freeing up some memory. To page out, the operating system sequentially goes through the corresponding directory entries, recalling the lines that are currently not in the *D*-node memory. While this threshold-based scheme should work well, we can have further support to handle crises. For example if, while paging out to free up space, the available memory in a *D*-node is getting too low, the *D*-node could send a high-priority pause interrupt to all *P*-nodes. The *P*-nodes would then stall until further notice.

2.3 Reconfigurability

Implementing the protocol controller with the same hardware that we use in the processing nodes lends flexibility to our architecture. We are not limited to a fixed partition between computing and coherence-maintenance resources. Instead, we can assign more or fewer resources to each of these two activities depending on the application's needs. If the application can use a large number of computing threads that need little protocol-processing activity, we configure the machine with many *P*-nodes. Alternatively, if the application needs a lot of protocol-processing activity, we configure the machine with many *D*-nodes.

It can be argued, however, that the processing/storage needs of *P*- and *D*-nodes are different. The activity of *D*-nodes is more memory intensive and may require more memory for the same compute power. Consequently, to use the hardware efficiently, while still allowing reconfigurability, we propose to equip the *D*-nodes with extra off-chip memory (Figure 2-(b)). There are, therefore, some "fatter" nodes with several times more memory on which protocol processing is preferred. For example, they can have 4 times as much memory as a regular node. All nodes, however, are fully reconfigurable as *P* or *D*. When a fatter node is used as a *P*-node, some of its memory is unutilized.

With this organization, Figure 4 shows the design space. The figure plots the number of *P*-nodes versus the number of *D*-nodes that participate in the execution of an application. Each straight line at -45 degrees, where *P+D* is constant, corresponds to a fixed machine size. For a given machine size, a highly-parallel application with little data sharing and good data reuse like some numerical applications is likely to run best in *1*, while one with much sharing like some symbolic applications runs best in *2*. As we vary the number of nodes in the machine, the optimal configuration for a given application changes (for example, as curve *OPT*).

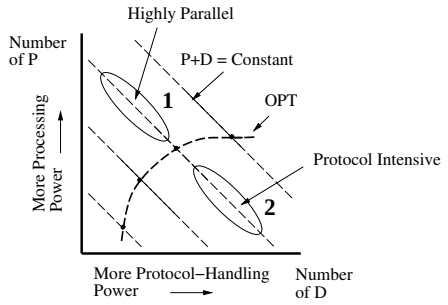

Figure 4: Effect of changing the number of *P*- and *D*-nodes.

We consider two approaches to reconfiguration, namely static and dynamic. In a static approach, we fix the desired number of *P*- and *D*-nodes before the application starts executing. In a dynamic approach, we let the total number and type of nodes change as the application executes.

A third approach would be to dynamically multiplex a *P*- and a *D*-thread on each node. Such an environment has been studied by Falsafi and Wood [3]. As it is, our machine does not support multiplexing because a *P*- and a *D*-thread use the local memory and memory controller differently. Indeed, *P*-threads use the local memory as a fast hardware-managed cache to extract high performance and to exploit physical locality (Section 2.1.1); *D*-threads, instead, manage the local memory in software to keep and use the directory state efficiently, and use the storage efficiently. For these reasons, we cannot exploit multiplexing in our design.

Static reconfiguration incurs no overhead. However, we usually do not know a priori, for a given application and a given number of available nodes, what the optimal partition between *P*- and *D*-nodes is. In practice, we can execute the application for the first time with a wasteful number of *D*-nodes and record the *D*-node processor utilization. The recorded utilization is used as a hint to tune the number of *P*- and *D*-nodes requested in subsequent runs of the application.

Dynamic reconfiguration has overhead. However, it enables a more efficient use of resources. Reconfiguration may

occur when an application enters a new phase of computation and triggers a request for a change in the resource allocation. In addition, the operating system can initiate reconfiguration events based on the current state of the system, including how busy *D*-nodes are, how well utilized the *D*-node memories are, and whether other applications have arrived or finished. There are two classes of reconfigurations, namely moving a node from one application to another while maintaining its type unchanged, and changing the type of a node, either within an application or across applications. We consider each class in turn.

Moving a node across applications while maintaining its type. This form of reconfiguration is simple. A *P*-node moves from one application to another by a simple thread context switch. The new thread will slowly displace the cached lines of the old thread and automatically send them to their home *D*-node. Moving a *D*-node from one application to another simply requires unmapping the pages of the first application from that *D*-node. These pages can be mapped to another *D*-node or sent to disk. Then, we map pages of the second application on the node. In practice, a *D*-node can be shared by several applications if it maps pages from different applications.

Changing the type of a node either within or across applications. This operation also requires operating system involvement. The procedure is the same irrespective of whether the node moves from one application to another or stays within the same application. If we need to transform a *P*-node into a *D*-node, two steps are required. First, the operating system writes back the dirty and shared-master lines from the caches and memory of the *P*-node, sending them to their homes. Then, the operating system reconfigures the node's memory controller to access the DRAM as plain memory and starts mapping pages, thus transforming the node into a *D*-node.

To transform a *D*-node into a *P*-node, the operating system unmaps any pages of the application from that *D*-node. These pages can be mapped to another *D*-node or sent to disk. In all cases, when pages are unmapped or mapped, the page tables and TLBs need to be updated. Then, we reconfigure the memory controller in the node to access the DRAM as a cache and start a new application thread on the node.

2.4 Computation in Memory

Finally, since *D*-nodes include a state-of-the-art general-purpose processor running software protocol handlers, we can add other software handlers that pre-process data before it is sent to the *P*-nodes. Such handlers should work relatively best for operations that both touch a lot of data and do not reuse it much – if the data is reused, *P*-node caches can still work well. For example, the *sequential scan* operation in database queries sequentially traverses a table of records, possibly selecting only a few of the records. This operation may best be done by a *D*-node processor. Only the records that satisfy the desired condition need to be sent to the requesting *P*-processor. Other similar operations exist. Computation in memory is better done on data that is guaranteed not to leave memory. Otherwise, we need to write back the data from the caches in advance. Identifying operations for computation in memory is beyond this paper's scope.

3 Evaluation Environment

We evaluate the proposed architecture with simulations. The architecture, which we label *AGG* for *aggressive*, is compared to Flat COMA and CC-NUMA. We label the latter organizations as *COMA* and *NUMA*, respectively. The processor chips in *COMA* and *NUMA* are the same as in *AGG* (Figure 1-(c)) except that the directory controller is on chip. In a *NUMA* node, the directory access is overlapped with the memory access. As a result, if the transaction is satisfied by the memory, the directory access adds no latency to the transaction.

All the machines that we compare have the same amount of DRAM memory and run 32 application threads. In *AGG*, we vary the ratio of *P*- to *D*-nodes. For example, Figure 5 shows the hardware needed to run 2 application threads for different organizations: *NUMA&COMA*, *AGG* with the same number of *P*- as *D*-nodes, and *AGG* with twice as many *P*- as *D*-nodes. With such a hardware organization, the advantages of *NUMA* and *COMA* over an *AGG* design with the same number of *P*- as *D*-nodes are that, in *NUMA&COMA*, a processor has twice the amount of local DRAM memory, and that there is no network latency for the processor to access the local directory. The disadvantage is the increased contention over on-chip busses and chip interfaces shared by the processor and the directory controller, which we do not model. Finally, because the *AGG* machines have more nodes, we double the bandwidth of the network links in *NUMA* and *COMA*. As a result, the bisection bandwidth is the same for *NUMA*, *COMA*, and an *AGG* design with the same number of *P*- as *D*-nodes.

The architectural parameters used for all the architectures are shown in Table 1. We model 4-issue superscalars that cycle at 1 GHz and can have up to 32 outstanding memory accesses, of which 16 can be loads. The memories in *COMA* and in *P*-nodes of *AGG* are 4-way set-associative. We perform two sets of experiments, namely one where the ratio between the application footprint and the total DRAM in the machine (memory pressure) is 25%, and one where it is 75%. In all the architectures, pages are allocated using the first-touch policy.

Device	Description	Avg. Round Trip Latency (CPU Cycles)
Write Buffer	32-entry fully assoc.	-
Load Buffer	16-entry fully assoc.	-
On-Chip L1	2-bank, direct-mapped, 64 B lines	3 (fully-pipelined)
On-Chip L2	Dir.-mapped, 64 B lines	6
Memory	Bwidth: 32 B / CPU ck. Local mem: on-, off-chip Remote mem: 2-node hop Remote mem: 3-node hop	 37, 57 298 383

Table 1: Architectural parameters. The latency values correspond to a round trip from the processor and do not include contention.

The network modeled is a wormhole-routed 2D-mesh. For *AGG*, the wires are 2-byte wide and cycle at 1 GHz, for a total of 2 Gbytes/s per link per direction. The links in the *NUMA* and *COMA* machines have twice that bandwidth. All contention in the system is modeled. In *COMA*, the replacement policy is such that invalid and non-master lines are replaced first. If a master line is replaced, we use Joe and Hennessy's method [8] of injecting the replaced line to

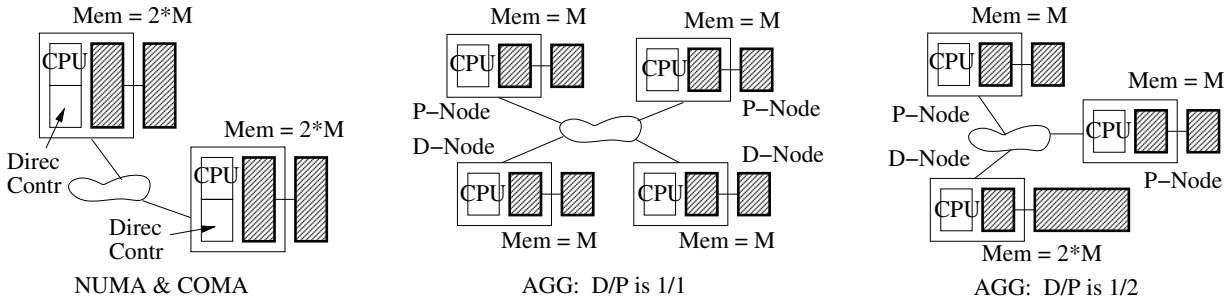

Figure 5: Hardware needed to run 2 application threads for different machine organizations.

the provider.

To estimate the overhead of cache coherence protocol processing in *AGG*, we coded the protocol handlers. We then ran them on an SGI Origin 2000 and used its performance counters to determine how many cycles each handler takes on an R10K processor. We ran each handler several times to eliminate the effects of the cache misses in the Origin. We use the cycles measured in the Origin as the compute component of the handlers. We also estimate a miss component time from an estimation of the misses in the memory hierarchy of *D*-nodes while executing protocol handler code. Adding the compute and miss component times, we get the times in Table 2. The numbers in the table do not include any contention time. We have written the handlers to minimize the latency, even though that may have increased the occupancy. These numbers are similar to those of Michael *et al* [15].

Handler	Latency	Occupancy
Read	40-80	80
Read Exclusive	40-80	80 + 10 per inval.
Acknowledgment	40	40-140
Write Back	40	140

Table 2: Latencies and occupancies in processor cycles for the major types of protocol handlers in *AGG*.

NUMA and *COMA* execute the protocol in hardware, while *AGG* does it in software. Thus, the overhead of the *AGG* protocol is higher. However, protocol processing in *NUMA* and *COMA* requires a custom-designed protocol processor. Designing such a module for clock frequencies as high as those of the fastest commodity microprocessors is challenging. Overall, we assume that the protocol processing overhead in *NUMA* and *COMA* is 70% of that in *AGG* in terms of latency and occupancy.

We run 7 applications on a MINT-based [22] execution-driven simulator of the architectures. The simulator can model aggressive superscalar processor architectures [10]. The applications, shown in Table 3, come from the SPLASH-2 suite [23] (*FFT*, *Radix*, *Ocean*, and *Barnes*), the SPEC95 suite [20] (*Swim* and *Tomcatv*), and the TPC-D workload [21] (*Dbase*, which is Query 3 from TPC-D). The SPEC95 applications have been automatically parallelized by the SUIF compiler [7]. Query 3 from TPC-D runs on a stand-alone system of tables and has been parallelized by hand. Table 3 also shows the problem size and the size of the two on-chip caches. These sizes are chosen based on the input sizes of the applications following the methodology described by Woo *et al* [23]. The primary working sets of the applications fit in the primary caches, while the secondary working sets do not fit in the secondary caches.

Appl.	Description & Problem Size	L1, L2 (KB)
FFT	Complex 1-D FFT with 64K points	8, 32
Radix	Integer radix sort with 1M keys and a 1K radix	8, 32
Ocean	Current simul. with a 258x258 grid	8, 32
Barnes	N-body problem with 16K bodies	8, 32
Swim	Weather predict. with Ref. pbm. size	32, 128
Tomcatv	Fluid dynamics with Ref. pbm. size	64, 256
Dbase	TPC-D query 3 with 1GB dbase	64, 512

Table 3: Applications used in the experiments.

The partition of the per-node memory between on-chip and off-chip memory also depends on the application. We run the application for 75% memory pressure on *AGG* for the same number of *P*- as *D*-nodes. We then select the size of the on-chip memory to be the one that has a 5% local miss rate for *P*-nodes. This is the size of the on-chip memory that we use in all experiments for this application. Overall, however, given that the difference in latency between an on- and off-chip local memory access is small, the fraction of local memory that is on-chip has only a modest impact on execution time.

4 Evaluation

4.1 System Architecture

To evaluate the architecture, we examine the overall performance and the *D*-node memory utilization.

Overall Performance

Figure 6 compares the execution time of the applications on *NUMA*, *COMA*, and *AGG*. Both *AGG* and *COMA* are evaluated with 25% and 75% memory pressure, which is shown with the suffix 25 and 75, respectively. In *AGG*, memory pressure is based on *P+D* memory. For *AGG*, we vary the ratio of *D*- to *P*-nodes while keeping the total memory constant. Specifically: *1/1AGG* means 32 *D*- and 32 *P*-nodes, with the same memory in each node; *1/2AGG* means 16 *D*- and 32 *P*-nodes, with twice as much memory in each *D*-node; and *1/4AGG* means 8 *D*- and 32 *P*-nodes, with four times as much memory in each *D*-node. Each application is run with a 1/1 ratio and with either 1/4 or 1/2: *Barnes*, *Swim*, *Tomcatv*, and *Dbase* are run with 1/4, while *FFT*, *Radix*, and *Ocean* are run with 1/2 because they put relatively more demands on the *D*-nodes. For each application, the bars are normalized to *NUMA* and divided into time when the processor stalls due to memory accesses (*Mem*-

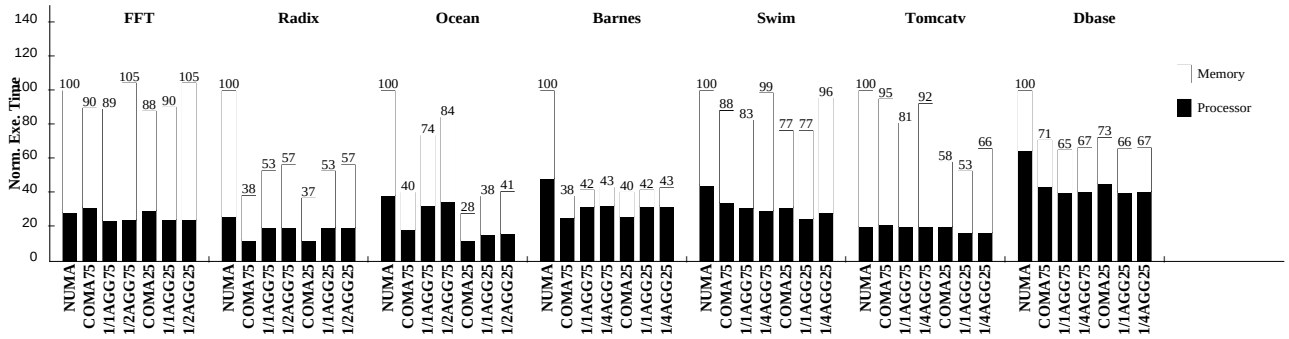

Figure 6: Processing-node activity in the different architectures. For each application, the bars are normalized to *NUMA*. The *Processor* component includes executing useful instructions, spinning for synchronization, and stalling for non-memory pipeline hazards.

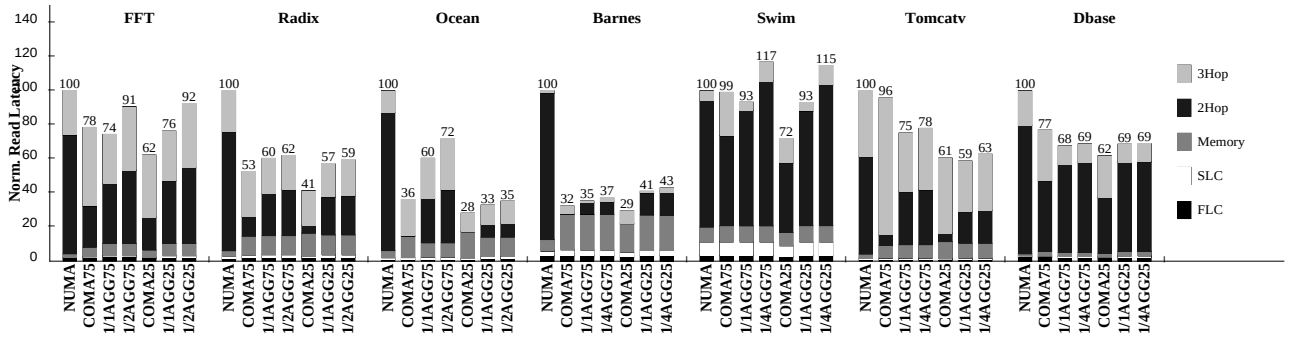

Figure 7: Total read latency of the applications for different architectures. For each application, the bars are normalized to *NUMA*.

ory) and the rest (*Processor*). *Processor* time includes executing useful instructions, spinning for synchronization, and stalling for non-memory pipeline hazards.

The figure shows that *COMA* and *1/1AGG* perform better than the simple *NUMA* design considered. The main reason for their higher performance is their lower *Memory* time: in these architectures, processors attract the remote working set into their large on-chip memories organized as caches. *NUMA*'s lower performance is not surprising: processors are not able to use local memory as a large caching space. If we compare *COMA* to *1/1AGG*, we see that the latter is on average only a bit slower. Both types of architectures have similar performance. They reduce the execution time of *NUMA* by about 30% with 75% memory pressure and by 40% with 25% memory pressure.

Most of the execution time differences between these three architectures are due to the *Memory* time. To explain the trends, Figure 7 adds up the latency of all the reads in each application, irrespective of whether or not the processor was stalled. Consequently, these bars are not exactly the same as the *Memory* category in Figure 6. Reads can be satisfied by the first-level cache (*FLC*), second-level cache (*SLC*), on-chip memory (*Memory*), and remote memory in two (*2Hop*) or three (*3Hop*) node hops. As in the previous figure, the bars are normalized to *NUMA*.

Two first-order effects explain the trends. The first one is that, in both *1/1AGG* and *COMA*, data migrates to local memories, transforming many of the *2Hop* transactions in *NUMA* to the cheaper *Memory* transactions. In general, this effect is more marked in *COMA* than in *1/1AGG* since, in the former, threads have twice as much on-chip memory. The second effect is that, in *1/1AGG*, home memories retain copies of shared lines more often than in *COMA*. In *1/1AGG*, home displacements are unlikely because of the full associativity of the *D*-node memory and the fact that we discourage reusing *SharedList*; in *COMA*, lines can be displaced from their home due to conflicts. The result is that some *2Hop* transactions in *1/1AGG* become *3Hop* ones in *COMA*, increasing overall latency. Overall, *1/1AGG* performs well under the two effects mentioned.

However, *1/1AGG* may be wasting *D*-node processing time. Indeed, the *1/4AGG* (or *1/2AGG*) bars in Figure 6 show that, by reducing by a factor of 4 (or 2) the number of *D*-nodes, applications are only slowed down slightly. Specifically, for both memory pressures, the average application takes only 12% longer to run on a reduced *D*-node machine than on *1/1AGG*. The slowdown comes from a larger *Memory* time, as shown in Figure 6. The reason for the larger *Memory* time is that accesses to the fewer *D*-nodes take longer: the *D*-nodes suffer more contention and, in addition, they are physically farther away in the network from some *P*-nodes. The result of these effects is visible in Figure 7, were the *2Hop* and *3Hop* times are higher in *1/4AGG* (and *1/2AGG*) than in *1/1AGG*.

Overall, *AGG* designs with only a fraction of *D*-nodes per *P*-node and fatter *D*-nodes are attractive platforms. We have used a 1 to 4 ratio or, for applications with higher traffic, a 1 to 2 ratio. In these platforms, applications take only slightly longer to run than on *1/1AGG*, and much less than on *NUMA*. *1/4AGG* (and *1/2AGG*) need less PIM chip hardware than plain *1/1AGG* and, if the PIM chips described indeed become off-the-shelf, these machines will be much cheaper than *NUMA* and *COMA*.

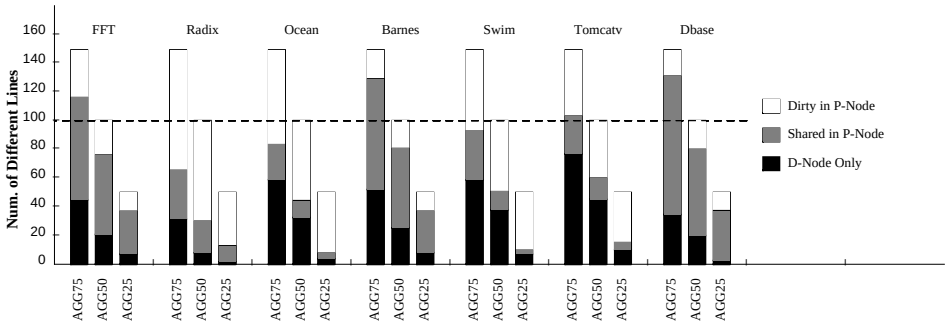

Figure 8: State of the *different* memory lines in the system. The horizontal dotted line represents the total number of *D*-node storage locations in the machine.

D-Node Memory Utilization

To assess *D*-node memory utilization, we classify the state of the *different* memory lines in the machine. These lines can be either *Dirty in P-Node*, *Shared in P-Node*, or *D-Node Only*. *Dirty in P-Node* means that there is only one valid copy of the line, and it is in state *Dirty* in a *P*-node. *Shared in P-Node* implies that at least one *P*-node caches the line in state *Shared*. In this case, the line is also present in its home *D*-node, unless the *D*-node dropped the line from the *SharedList* list to free up space. Finally, *D-Node Only* means that the home *D*-node has the only copy of the line. For this experiment, the actual ratio of *D*- to *P*-nodes is not very significant as long as the total *D*-memory and the number of *P*-nodes do not change. Consequently, we use the *1/1AGG* configuration.

Figure 8 shows the data for each application. The horizontal dotted line represents the total number of *D*-node storage locations in the machine (normalized to 100). For each application, *AGG75*, *AGG50*, and *AGG25* represent 75%, 50%, and 25% (D+P) memory pressures, respectively. For example, at 75% memory pressure, there are 150 different memory lines in the system per 200 (D+P) storage locations, or per 100 *D*-node storage locations. Recall that the total memory in the system is twice the combined size of all the *D*-memory.

Figure 8 shows that organizing the coherence protocol and *D*-node memories in the manner described in Section 2.2.2 allows good memory use. We can see this in two ways. First, the number of *D-Node Only* lines is significantly lower than the dotted line in Figure 8, which is the size of the total storage available in *D*-nodes. *D-Node Only* lines are, strictly speaking, the only ones that the *D*-nodes are required to keep in their memory at all times. The figure shows that, in *AGG75*, they account for an average of only 50% of the *D*-node memory. In *AGG50*, which is the case when there are as many different memory lines in the system as *D*-node storage locations, *D-Node Only* lines account for only 25% of the *D*-node memory. The corresponding fraction in *AGG25* is a tiny number. Overall, therefore, we can reduce the amount of *D*-node memory or, equivalently, run with higher memory pressures.

Second, a large fraction of the remaining lines are *Dirty in P-Node*. *D*-nodes are not even allowed to keep a copy of them. With so many *Dirty in P-Node* lines, it is unlikely that we need to reuse the shared lines in *SharedList*. Indeed, the figure shows that the dotted line, which is the size of the storage available in *D*-nodes, is usually higher than the combined number of *D-Node Only* and *Shared in P-Node* lines. The difference is the number of unused *D*-node memory locations. When the difference is negative, *SharedList* lines are reused. From the figure, we see that, in *AGG75*, *SharedList* lines are reused to some extent in only three applications, namely *FFT*, *Barnes*, and *Dbase*. Two applications even have plenty of unused *D*-memory space (*Radix* and *Ocean*). For *AGG50*, which is the case when there are as many different memory lines in the system as *D*-node storage locations, an average of 40% of the memory in *D*-nodes is unused. For *AGG25*, this figure is 75%. With the use of the fully-associative memories described, *AGG* can use up any unused memory without suffering conflicts. It can, therefore, run high memory pressures.

4.2 Reconfigurability

In this section, we evaluate static and dynamic reconfigurability. The evaluation is constrained by the fact that SPLASH-2 applications need a power-of-two number of threads to run efficiently.

To make the case for static reconfigurability, we show that applications have significantly different computing and protocol processing requirements. Consequently, a non-reconfigurable machine is bound to run many applications sub-optimally. This can be deduced from Figure 9, which plots the execution time of the applications (*Z* axis) under different numbers of *P*- and *D*-nodes. The *Z=0* plane is like Figure 4 except that the chart is rotated so that the area with few *P*- and *D*-nodes is in the far side. The *X* and *Y* axes are in logarithmic scale because the SPLASH-2 applications need a power-of-two number of *P*-nodes. In each application, the bars are normalized to the execution time for 2-*P* and 2-*D* nodes, and are divided into *Memory* and *Processor* time. The experiments use *AGG75* for 2-*P* and 2-*D* nodes, and keep the problem size and total *D*-memory size fixed as more nodes are added.

With more *P*-nodes, *Memory* tends to decrease because processors issue memory accesses in parallel and, in addition, there is more *P*-node caching space. However, it may also increase as in *Radix* if there is more coherence activity or if memory accesses take significantly longer. The latter may be due to higher contention or to longer average distance between a *P*- and a *D*-node. With more *P*-nodes, *Processor* may decrease or increase. It decreases if the application's operations can be executed in parallel, while it may increase due to load imbalance. Finally, as the number of *D*-nodes increases, the execution time tends to decrease or stay constant as in *Swim*, depending on the amount of protocol handling activity present.

Overall, the optimal combination of *P*- and *D*-nodes varies

across applications. *Dbase* runs best with a high number of *P*s and *D*s, while *Radix* runs most cost-effectively with a medium number of *P*s and *D*s. *Swim* and *Tomcatv* run most cost-effectively with high *P* and low *D*. Finally, the other applications run most cost-effectively with high *P* and medium *D*. Unlike a non-reconfigurable machine, a statically-reconfigurable machine can potentially run each application efficiently.

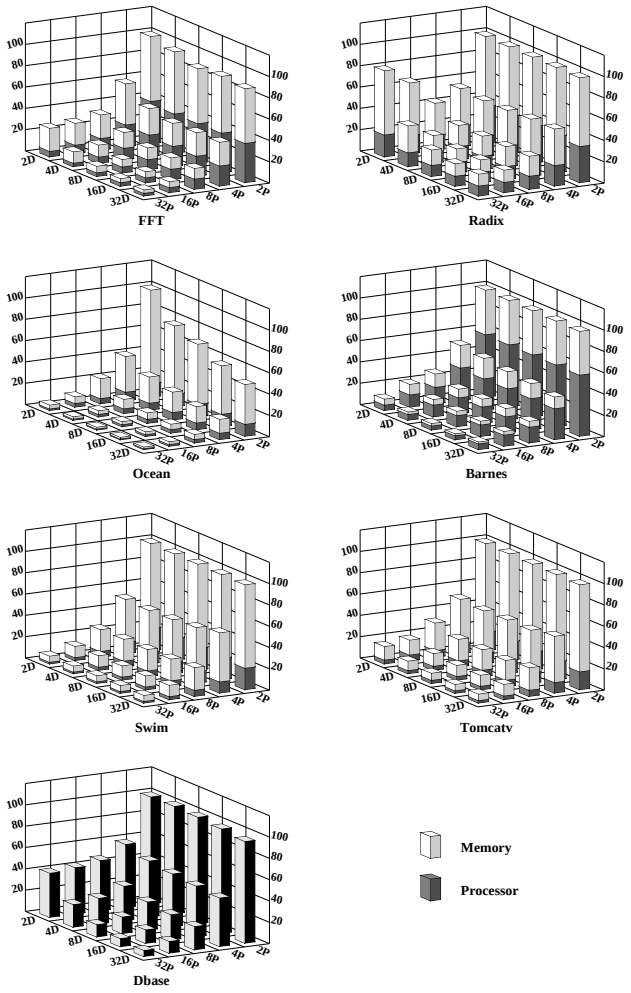

Figure 9: Total execution time of the applications for different numbers of *P*- and *D*-nodes. The time breakdown in *Dbase* is not available.

We now turn to dynamic reconfigurability, which we find harder to exploit than static reconfigurability. One application that can exploit it is a database query. Queries often go through different phases, and the query optimizer often knows a priori the approximate sizes of the tables and the access patterns. The optimizer can then choose the number of *P*- and *D*-nodes for each phase. Dynamic reconfiguration, however, is less beneficial for SPLASH-2 and SPEC95 applications because they need fixed numbers of threads that iterate over the same tasks repeatedly.

Figure 10-(a) shows the execution time of *Dbase* with different combinations of *P*- and *D*-nodes (shown as P&D), keeping the total number of nodes fixed at 32. Our parallel *Dbase* goes through two major phases: hash and join. The hash phase is *D*-node intensive. Indeed, all threads read chunks of a large table without data reuse and build a shared hash table. They create *D*-node transactions by missing repeatedly in *P*-node memories and synchronizing often. In the join phase, each *P*-node performs two joins using two tables and the hash table. One of the tables is divided into chunks, which are given to *P*-nodes. Once a *P*-node brings a chunk into its cache, it can reuse it to some extent. Since caches have some use and there is a lot of parallelism, we can benefit from many *P*-nodes.

The figure shows two static configurations: 16&16 and 28&4. 16&16 works well in the hash phase and poorly in the join phase, while 28&4 has the opposite behavior. The third bar in the figure corresponds to a system that runs with 16&16 for the hash phase and is reconfigured to 28&4 before the join phase starts. It therefore captures the best of both worlds. However, it suffers the reconfiguration overhead (*Reconf*). We model the *Reconf* overhead in a *D*-node as a base cost of 100,000 cycles for setup, synchronization, and decision making, plus the additional cost of collecting and migrating each memory line. In addition, for every 10 pages moved, the *D*-node incurs a 1,000-cycle overhead to update the page mappings. If, as a result, a *P*-node processor has to update its TLB, the *P*-node processor incurs a 1,000-cycle overhead. Overall, the figure shows that, with dynamic reconfigurability, the execution time of the best static case is further reduced by 14%. Consequently, dynamic reconfigurability is attractive in some cases.

4.3 Computation in Memory

Even with the best *P*- and *D*-node combination, the applications suffer from much memory stall time. To mitigate this problem, we can off-load application operations to our *D*-node processors. Since determining what to off-load is in general a hard programming problem, we only do it for the simple case of *Dbase*. Instead of having *P*-nodes traverse tables to identify records that satisfy the select condition, *D*-nodes do it. Once the *D*-node finds a selectable record, it returns a pointer to the *P*-node. The latter then performs the join and invokes the *D*-node again. Figure 10-(b) compares the execution time of *Plain* (*P*-nodes traverse) to *Opt* (*D*-nodes traverse) for different *P*- and *D*-node combinations. As in Figure 9, the experiments use *AGG75* for 2P&2D and keep the problem size and total *D*-memory size fixed as more nodes are added. The results show that this optimization reduces the execution time of *Dbase* by about 70% for different *P*- and *D*-node configurations.

5 Concluding Remarks

This paper has explored how a cache-coherent DSM machine built around off-the-shelf, largely uniprocessor PIM chips might be cost-effectively organized. To exploit processor-memory integration, we propose tagging the memory in the compute nodes and organizing it as a cache. We keep directory controllers off-chip to use potentially off-the-shelf uniprocessor hardware. To keep the machine cheap and simple, and to enable reconfigurability, directory control is performed by nodes that have identical hardware as compute nodes. The only difference is that directory control nodes can have more memory. We also propose a cache coherence protocol that is tailored to our system: it uses the memory very efficiently while exploiting the large caching space

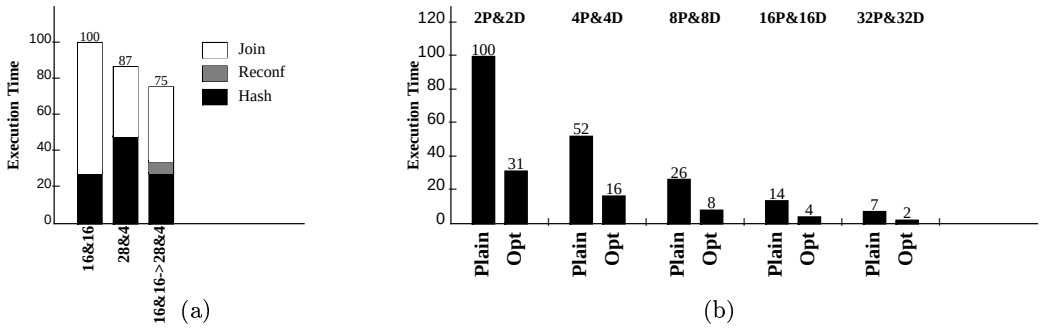

Figure 10: Two different ways to speed up *Dbase*: dynamic reconfiguration (a) and performing computation in the *D*-nodes (b).

available. The resulting hardware delivers performance that is comparable to, and better than, the more expensive traditional COMA and CC-NUMA organizations, respectively. In addition, static reconfigurability gives great versatility to the machine.

It is conceivable that future off-the-shelf PIM chips may incorporate on chip some logic that could be used as a protocol controller. However, it is far from clear what functionality a "standard" design of an on-chip protocol controller should provide. Even if its hardware were somewhat programmable, there are too many different issues to address and functionalities to provide, to satisfy all application domains and uses. Consequently, in this paper, we have assumed a very simple PIM chip as a basic block, so that we can build a highly-versatile and most inexpensive DSM organization.

References

[1] D. Burger, S. Kaxiras, and J. Goodman. DataScalar Architectures. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 338–349, June 1997.

[2] K. Ekanadham, B.-H. Lim, P. Pattnaik, and M. Snir. PRISM: An Integrated Architecture for Scalable Shared Memory. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 140–151, February 1998.

[3] B. Falsafi and D. Wood. Scheduling Communication on an SMP Node Parallel Machine. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 128–138, February 1997.

[4] M. Fillo, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee. The M-Machine Multicomputer. In *28th International Symposium on Microarchitecture*, 1995.

[5] J. A. Gasbarro. The Rambus Memory System. In *International Workshop on Memory Technology, Design and Testing*, pages 94–96, 1995.

[6] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, pages 12–15, October 1998.

[7] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[8] T. Joe and J. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 82–93, April 1994.

[9] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies. In *Frontiers'96*, pages 88–97, October 1996.

[10] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[11] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[12] G. Lauterbach. UltraSPARC III: A 600 MHz Superscalar Processor for 1000-way Scalable Systems. In *Hot Chips X Symposium Record*, page 1.2, August 1998.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[14] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[15] M. Michael, A. Nanda, B.-H. Lim, and M. Scott. Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 219–228, June 1997.

[16] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Tomas, and K. Yelick. A Case for Intelligent DRAM. In *IEEE Micro*, pages 33–44, March/April 1997.

[17] S. Reinhardt, R. Pfile, and D. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 34–43, May 1996.

[18] A. Saulsbury, S. Huang, and F. Dahlgren. Efficient Management of Memory Hierarchies in Embedded DRAM Systems. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 464–473, June 1999.

[19] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 90–101, May 1996.

[20] The Standard Performance Evaluation Corporation. The SPEC95fp Suite. URL: http://www.specbench.org.

[21] Transaction Processing Performance Council. *TPC Benchmark D (Decision Support) Standard Specification Revision 1.1*, December 1995.

[22] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *MASCOTS'94*, pages 201–207, January 1994.

[23] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.