How Processor-Memory Integration Affects the Design of DSMs[1]

Liuxi Yang, **Anthony-Trung Nguyen** and **Josep Torrellas**

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801
l-yang3,anguyen,torrella@cs.uiuc.edu
http://iacoma.cs.uiuc.edu/iacoma/

May 1997

Abstract

The progressive integration of processor and memory has unexpected implications for the design of DSM systems. To exploit this integration best, we claim that we need to redesign the nodes of DSM systems and then reorganize the whole machine. In this paper, we propose a new DSM organization where processor nodes have their on-chip memories configured as caches and their directory controllers have been moved off-chip. The directory controllers are built with the same processor-memory chip as the computing nodes and, therefore, can be considered as non-computing nodes. The function of these non-computing nodes is to support the cache coherence protocol and to backup the application's data in their memories. Because off-the-shelf processors are so fast, these non-computing nodes manage the coherence operations and the storage of data in their memory in software. A high-level evaluation of the proposed architecture shows that it is significantly better than idealized versions of the traditional COMA and NUMA organizations.

1 Introduction

With vast increases in the on-chip transistor count expected in the near future, a major trend in microprocessor design is the progressive integration of processor and memory to relieve the memory access bottleneck [4, 5, 6]. Traditional node organizations, where the processor is connected to the memory via a slow memory bus are expected to be replaced by more integrated organizations. Eventually, processor and main memory are likely to share the same chip. Such an organization promises to provide low latency and high bandwidth communication between processor and memory.

Directory-based Distributed Shared Memory (DSM) systems are typically built using the latest microprocessor technology. Consequently, they will be affected by the trend of progressive integration of processor and memory. Currently, the nodes in DSM systems are usually organized as in Figure 1-(a). The processor has one or two layers of on-chip caches and/or a large off-chip cache to reduce memory latency. The processor accesses memory via the memory bus. The directory controller and the memory controller are connected to this memory bus. The off-chip cache does not necessarily have to be connected to the memory bus. Instead, it may be a back-door cache directly connected to the processor chip. In any case, however, its tags have to snoop on the bus. Overall, memory accesses in this design suffer from relatively low bandwidth and high latency. This is because, to access main memory, a request has to traverse many levels of caches and, in addition, use the bus. Bus transactions are necessarily slow because of the need to perform bus arbitration, synchronize different clock frequencies, and support snooping for data coherence.

As integration technology advances, the nodes of future DSM systems are likely to look like Figure 1-(b) and (c). In a first stage, shown in Figure 1-(b), the memory controller is moved on chip and the memory is directly connected to the processor chip via a dedicated point-to-point bus. With such an organization, accesses to main memory have lower latency and higher bandwidth than with a traditional memory bus. For example, main memory can be accessed 10-30 times faster than the 300-800 ns that are required for a round-trip to the main memory in the node of Figure 1-(a). Finally, as more transistors can be placed on chip, the final node design is likely to include the DRAM memory on chip (Figure 1-(c)). This further reduces the memory access bottleneck.

To benefit the most from these more advanced designs, we need to exploit the close integration of processor and memory as much as possible. The subject of this paper is how to do this for DSM machines built out of off-the-shelf microprocessors. Exploiting this integration is much harder in traditional DSM machines than in uniprocessors. The reason is that uniprocessors and nodes in DSM machines exhibit different geographical locality when a cache miss occurs. In uniprocessors, after a cache miss, the memory is accessed. In traditional NUMA DSM systems, however, a cache miss is followed by a local or a remote directory access and then by a local or a remote memory or cache access. This poor geographical locality clearly hampers exploiting the processor-memory integration. In this paper, we address this problem.

This paper is organized as follows. The poor geographical locality in DSMs forces us to reorganize the node (Section 2). This node reorganization in turn impacts the whole DSM system organization (Section 3). Then, we discuss a way to further enhance the processor and memory integration (Section 4). Finally, we use simulations to evaluate the proposed system (Section 5).

2 Node Issues

We would like to exploit the upcoming processor-memory integration to decrease the execution time. This is much more challenging to do in DSM machines than in uniprocessors. The reason, as indicated before, is the type of operations that follow a cache miss. In a uniprocessor, a cache miss is followed by a memory access, which takes advantage of the

[1]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 Intel Corporation.

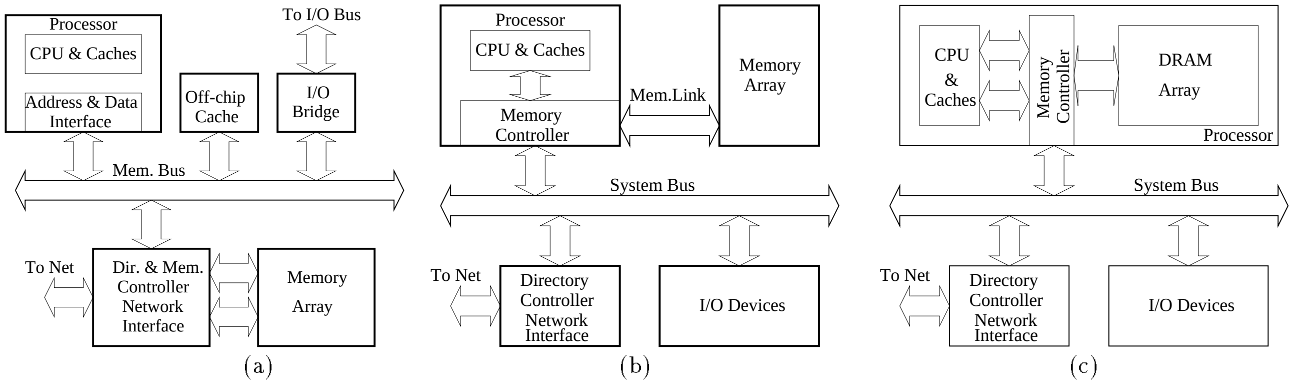

Figure 1: Different organizations of a node in a DSM system.

processor-memory integration. In traditional NUMA DSM machines, instead, a cache miss is followed by a local or a remote directory access and then by a local or a remote memory or cache access. If either of these two accesses is remote, the processor-memory integration is wasted.

Unfortunately, such accesses are likely to be remote. Consider first the directory access. Which directory is accessed depends on the physical address issued by the processor. For a N-node DSM, the local directory will contain entries for only $1/N$ of the addresses. Consequently, many cache misses are likely to induce a remote directory access. It can be argued that the programmer, compiler, or operating system can allocate the data accessed by a processor in the local memory of the processor, thereby reducing the fraction of remote directory accesses. While this is easy to do for private data, it is unclear how to do it effectively for shared data in non-trivial applications.

Furthermore, accesses to remote memories or caches can also be frequent. Such accesses can be due to three reasons, namely start-up accesses to data allocated in remote memories, remote data overflow from the local caches, and coherence activity. All these remote accesses dilute the advantages of having a close processor-memory integration.

To eliminate many of these remote accesses, we propose to add to each line in the on-chip memory its state (invalid, shared, or exclusive) and its address tag. Furthermore, we treat the memory as a cache, allowing data to replicate and migrate between memories. Finally, we move the directory controller off-chip to get a node configuration like the one in Figure 1-(c).

By adding the state to each memory line, we can skip the directory check after a cache miss. Instead, the local memory is accessed first. If we are reading from a non-invalid line or writing to an exclusive line, there is no need to access the directory at all. In addition, by treating the memory as a cache and allowing memory lines to migrate and replicate across memories, we can eliminate many remote memory accesses resulting from overflow of the local caches. This is because on-chip memories are so large that overflow will be minimal. Of course, we need to keep the lines in the on-chip memories of the different nodes coherent with a coherence protocol. In the next section, we will see how this is done and how the resulting architecture is different from the more costly COMA organization.

There are a few reasons why the directory controller should be off-chip. These reasons also apply to NUMA DSM systems built out of processors integrated with memory. The first reason is that most of the accesses to a directory are likely to come from remote processors instead of from the local processor. This is due to the fact that, as indicated before, in a machine with N nodes, many of the requests issued by a processor are likely to access the $(N\text{-}1)/N$ fraction of memory locations that have remote directory entries. Having the directory on-chip will likely mean accepting intolerable incoming remote traffic. A second, possibly less important reason, is that adding a directory controller to a chip makes it more expensive and no more useful if the chip is to be used in uniprocessor systems. Since uniprocessor systems may still account for most of the sales in the near future, there may be little incentive to complicate the chip design to include the directory controller.

Finally, another reason to decouple processor and directory controller is that, for a given number of processors, we can then vary the number of directory controllers depending on the application. Such a capability is interesting because, with more directory controllers, we can exploit more parallelism in cache coherence protocol processing. Different applications may be more demanding than others in terms of protocol processing. We are actively researching this issue.

It can be argued that, in any case, we do not lose that much by separating processor and directory controller. Indeed, because the on-chip DRAM is so large, if we organize it as a cache, there will be few directory accesses anyway. Moreover, the directory accesses that remain will result mostly from coherence misses. Data with coherence misses is hard to place optimally in memory and, therefore, these misses will largely induce remote directory accesses anyway.

It is clear that the addition of the tags to the on-chip memory involves some cost. However, uniprocessor systems can also take advantage of such support. Indeed, consider a uniprocessor system that runs applications that overflow the on-chip memory. If we want to expand the memory with some off-chip memory, then it is very convenient to have the ability to reconfigure the on-chip memory as a cache of the off-chip memory. Overall, therefore, we envision a node like Figure 1-(c) where the memory controller can easily reconfigure the on-chip memory as a memory or as a cache.

Finally, to handle private data best, we propose to set aside a partition in the DRAM memory that is addressed as a regular memory instead of as a cache. This will prevent the displacement of private data.

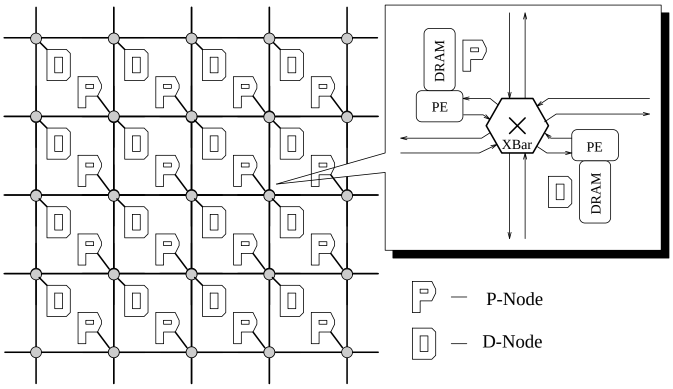

Figure 2: Organization of a DSM machine that uses processors with the memory on-chip and the directory controller off-chip.

3 Systems Issues

This new node organization in turn affects how we organize the DSM machine. We have argued that, for several reasons, we want to separate the directory controller from the processor. One overriding goal, however, is to keep the multiprocessor simple and cheap. This implies using off-the-shelf chips and as few different types of chips as possible. For this reason, we propose to implement the directory controller as a separate node, with the same processor-memory chip that we use for the ordinary computing nodes. Consequently, we reduce the number of different types of chips to a processor-memory chip and one or more network chips. There are two types of nodes, namely computing or processing nodes (*P*-nodes), and non-computing or directory nodes (*D*-nodes). Figure 2 shows the resulting multiprocessor. Note that there is no need for the ratio between the number of *P*-nodes and *D*-nodes to be 1. In fact, as we will see later, one *D*-node is likely to handle several *P*-nodes.

The function of the *D*-nodes is twofold. On the one hand, they keep the distributed directory state and perform all the directory transactions. They send the right messages to the *P*-nodes to ensure that the data in their cache-configured memories is kept coherent. On the other hand, their own memory is used as the only main memory for shared data in the machine. Their memories contain copies of all the lines of shared data that are not held in exclusive mode in any of the *P*-nodes. A plain DSM directory-based cache coherence protocol ensures that the lines in the *P*- and *D*-node memories are kept coherent.

Note that the machine proposed is neither a traditional NUMA nor a COMA. In a NUMA, the only caching space is the one provided by the L1 and L2 caches, while in a COMA, all the storage in the system is cache. In the proposed machine, the memories of the *P*-nodes are caches, while the memories of the *D*-nodes are not. By using the *P*-node memories as caches, we exploit the geographical locality that a NUMA cannot exploit. By having plain memory in the *D*-nodes, we do not need to worry about dropping the last shared copy of a line from a cache, unlike in the COMA protocol. This simplifies the cache coherence protocol. We keep, therefore, the best of the NUMA and COMA organizations. We will see later how to reduce the storage requirements in the *D*-node memories.

We expect that, in many applications, *D*-nodes will have relatively little to do beyond the start-up period. Indeed, consider the *D*-nodes' two functions, namely directory handling and data storage. We expect that there will be only a modest number of directory accesses. This is because, due to the large size of the *P*-node caches, misses resulting from overflow are likely to be minimal. Directory accesses will be caused only by coherence transactions. Similarly, we also expect that there will be only a modest number of line insertions and extractions from the *D*-node memories. Again, because of the large size of the *P*-node caches, very few exclusive lines will be displaced from the *P*-node caches and into the *D*-node memories. Consequently, only coherence transactions will cause lines to be inserted, supplied or invalidated from the *D*-node memories.

These *D*-nodes are built out of very fast off-the-shelf processors. To keep them busy, we propose that the *D*-nodes perform their functions in software. Specifically, we propose that they perform the management of the directory state and of the data stored in their memories in software.

The management of the directory state is done with software handlers executed by the processor. These routines prompt the *D*-node to send messages to *P*-nodes to keep the distributed data coherent. The directory itself is stored in the *D*-node memory like any other data structure. It is accessed by the software protocol handlers like plain data. Consequently, we do not use any custom-designed hardware at all.

We also use software to manage the *D*-node memory. Note that not all the memory lines whose home is that *D*-node have valid copies stored in the *D*-node's memory. Specifically, the lines that are in state exclusive in any of the P-nodes do not have a valid copy in the D-node memory. Consequently, we do not reserve a location in the *D*-node memory for each line in the system whose home is that node. Otherwise, some of the *D*-node memory would be wasted with stale data. Instead, when a line needs to be stored in the *D*-node, we save the line in *any* free location in the memory. The corresponding directory entry is updated to point to the location where the line is saved. With this scheme, the DRAM memory of *D*-nodes acts as a fully associative memory. We can compact the data in the *D*-node memories much more. As a result, we need fewer *D*-nodes and the ratio between the number of *D*- and *P*-nodes decreases.

Figure 3 shows the high-level organization of the directory and memory of the *D*-nodes. Each directory entry has a pointer that points to a memory location or to nil if the corresponding line is exclusive in one *P*-node. The memory locations are also linked to each other with several pointers to allow for the fast identification of a free location. When

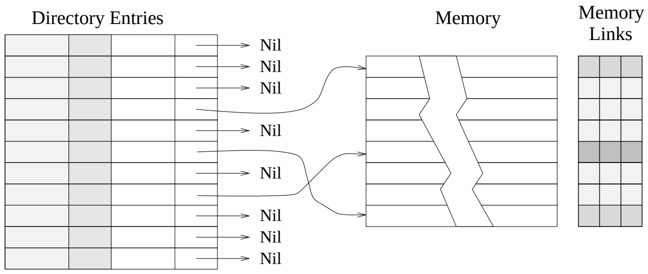

Figure 3: Organization of the directory and memory in the D-nodes.

a P-node requests a line in state exclusive, that line is extracted from the D-node memory and the location is freed up. When a line needs to be inserted in the D-node memory, we use the linked list to find an empty location. In all cases, accesses to data in memory are made faster by the fact that both directory and memory data are automatically cached in the D-node caches.

We can further reduce the number of D-nodes required if we do not store in the D-node memories the lines that are in state shared in at least one P-node. If these lines are not stored, there is more space available in the D-node memories. In addition, there is likely to be less overhead inserting and extracting lines from memory. As a result, fewer D-nodes per P-node are required and the system may be cheaper. The disadvantage, however, is twofold. First, when a D-node receives a request for a shared line, it will have to forward the request to a P-node with a copy of the data. These transactions will involve three network hops instead of just two and, therefore, will be slower. In addition, we need an extra state in the cache coherence protocol, namely that of *master*. This is to avoid losing the last copy of a line. One of the sharing P-nodes has the master copy. If that line is displaced, it needs to be sent back to memory. The other shared copies can be discarded when they are displaced. Overall, we do not use this optimization because it is not clear that it is beneficial.

4 Further Optimization: The Row Tag Buffer

Even if the memory is on-chip, accessing a large DRAM involves a considerable latency. For this reason, we propose a very simple enhancement to the on-chip memory hierarchy that reduces DRAM access latency. This enhancement, called the *Row Tag Buffer*, is enabled by the integration of the memory controller on chip. This optimization can improve the performance of both uniprocessors and DSM systems. However, it has a higher impact in uniprocessors because on-chip memory access latencies have a relatively higher impact on the execution time of uniprocessors.

The idea is to eliminate unnecessary RAS accesses to memory. We add a buffer to the on-chip memory controller to keep the numbers (tags) of the rows currently loaded in the row buffers in memory. Every time that the second-level cache is accessed, this row tag buffer is also accessed. Because the row tag buffer is very small, it determines whether or not a hit occurs before the second-level cache does. If the access to the second-level cache finally results in a miss, we use the outcome of the access to the row tag buffer: if it was a miss, we issue a RAS and a CAS to memory; if it was a hit, we issue only a CAS to memory. In the latter case, we have saved the latency and bandwidth consumed by a RAS access. No action is taken if the second-level cache records a hit. In that case, while we could issue a RAS cycle if the row tag buffer missed, such a RAS access could prove of no use in the future and would consume processor-memory bandwidth anyway.

The row tag buffer is small. If we assume a DRAM with 128 Mbytes and banks of 2 Mbytes, it needs to keep only 64 entries. Furthermore, each entry uses little space, since it stores only a row number and a valid bit. Figure 4-(a) shows a block diagram of how the row tag buffer is connected to the other on-chip modules of an Alpha-based microarchitecture. From the diagram, we see that the address lines used to access the second-level cache (*addr* lines in the upper left side of the diagram) are also fed into the row tag buffer. If a secondary cache miss is finally detected, the *miss* signal is routed to the row tag buffer. Then, the row tag buffer issues a CAS+RAS cycle to memory or just a CAS cycle.

A logic diagram of the row tag buffer is shown in Figure 4-(b). The Tag Buffer is the table with the row numbers. Since processors support several outstanding memory accesses, several RAS signals for different banks may be in progress at a time. For this reason, we use the Pending Buffer. The Pending Buffer is a FIFO buffer that stores, for each of the RAS accesses currently in progress, the row number it refers to and the time when the RAS will be completed. At that time, a CAS cycle will be issued. To know when the RAS accesses is completed, we use a comparator and a free-running counter A. Counter A is incremented every cycle and can count up to the number of clock cycles in a RAS cycle. When an entry is allocated in the Pending Buffer, counter A is copied to the Time field. Counter A keeps incrementing every cycle. Later, after a period of time equal to a RAS cycle, A will have wrapped around and will match exactly the entry in the Time field. The comparator will then assert the *CAS* line. In the figure, counters B and C point to the head and the tail entries in the Pending Buffer respectively.

5 Evaluation

We now use simulations to evaluate the proposed architecture from a high-level point of view. In the following, we first discuss the simulation environment and then present the results.

5.1 Simulation Environment

We call the multiprocessor proposed in Sections 2 and 3 *AGG* for *aggressive*. We compare it to idealized versions of

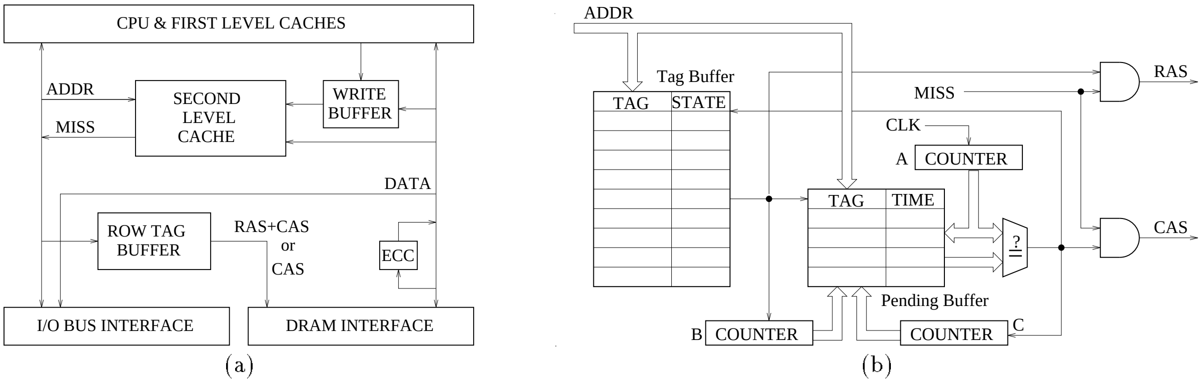

Figure 4: Connection of the row tag buffer to the other on-chip modules (Chart (a)), and logic diagram of the row tag buffer (Chart (b)).

Table 1: Architectural parameters.

Device	Description	Round-trip Access Latency (CPU Cycles)
Write Buffer	6-entry fully associative	-
Load Buffer	6-entry fully associative	-
On-Chip L1	2-bank, direct-mapped, 64-byte lines	1 (3-deep pipe)
On-Chip L2	Direct-mapped, 64-byte lines	6
Memory	Bandwidth: 32 bytes per CPU clock	
	On-chip memory	37
	Remote memory (2 hops)	350
	Remote memory (3 hops)	450
Network	Wormhole routed	-

two traditional systems, namely Flat COMA and NUMA. The latter two organizations have the same node structure (except that the directory controller is on chip) and interconnection network as *AGG*.

By keeping the directory controller on chip in COMA and NUMA, we are giving an advantage to these organizations. Specifically, their processor chips would be more expensive because they would include a custom-designed directory controller and a more complicated interface, and be larger. In addition, in our simulations, we do not model the resulting increase in contention in the internal busses and interface. We do all this to minimize the number of parameters that we vary at a time. We model, however, the higher inbound and outbound traffic in the chip.

The simulations are performed using 32 execution threads. This means that *AGG* runs with 32 *P*-nodes and 32 *D*-nodes, while *NUMA* and *COMA* run with 32 processors each. However, the amount of DRAM memory is the same in all the machines. In our evaluation, we perform two sets of experiments, one where the footprint of the application is much smaller than the total DRAM memory (the ratio between the footprint and the memory size, or *memory pressure* is 15%), and one where memory is tighter (the memory pressure is 75%). By default, the memories in *COMA* and the *P*-node memories in *AGG* are 4-way set-associative. However, we also simulate an *AGG* system with direct-mapped *P*-node memories that we call *D_AGG*.

The architectural parameters used for all the architectures are shown in Table 1. We model statically-scheduled 4-issue superscalars that cycle at 1 GHz. As shown in the table, the on-chip memory is accessed in a 37-cycle round trip, while the memory in a remote node is accessed in a 350-450 cycle round trip.

To estimate the overhead of cache coherence protocol processing in *AGG*, we wrote the protocol handlers. We then ran them on an SGI Origin 2000 [3] and used the performance counters to determine how many cycles each handler takes on a 4-issue processor. We ran each handler many times to eliminate the effects of the cache misses, therefore giving a lower bound in the execution time. The overheads obtained in cycles for some of the most important handlers are shown in Table 2. It is hard to argue whether *NUMA/COMA* or *AGG* should have higher protocol processing overheads. *AGG* will execute more operations than *NUMA/COMA* for a given handler because the execution is done in software. However, protocol processing in *AGG* can use more chip resources than in *NUMA/COMA*. In the latter, the protocol controller has to share the chip with many other modules. For example, the caches used by the protocol controller in *NUMA/COMA* will be smaller and the cycle time probably larger. For these reasons, and to change as few parameters as possible across architectures, we use the same overheads for protocol processing in *NUMA/COMA* and *AGG*.

Table 2: Latencies and occupancies for the major types of protocol handlers.

Handler	Latency	Occupancy
Read	19-21	29-39
Read Exclusive	19-21	47-75
Acknowledge	—	15-128
Write Back	24	128

For the experiments, we run 4 applications on a MINT-based [7] execution-driven simulator of the architectures. The applications, shown in Table 3, come from the Splash2 [8] and Spec95 [2] suites. They run with 32 threads. The Spec95 applications have been automatically paral-

Table 3: Applications used in the experiments.

Application	Description / Suite	Problem Size	Footprint (MB)	On-Chip L1, L2 (KB)
FFT	Complex 1-D fft / Splash2	64K points	3.4	2, 8
Radix	Integer radix sort / Splash2	256K keys, radix 1024	3.6	2, 8
102.swim	Weather prediction / Spec95	Reference	13.2	8, 64
101.tomcatv	Fluid dynamics / Spec95	Reference	14.7	8, 64

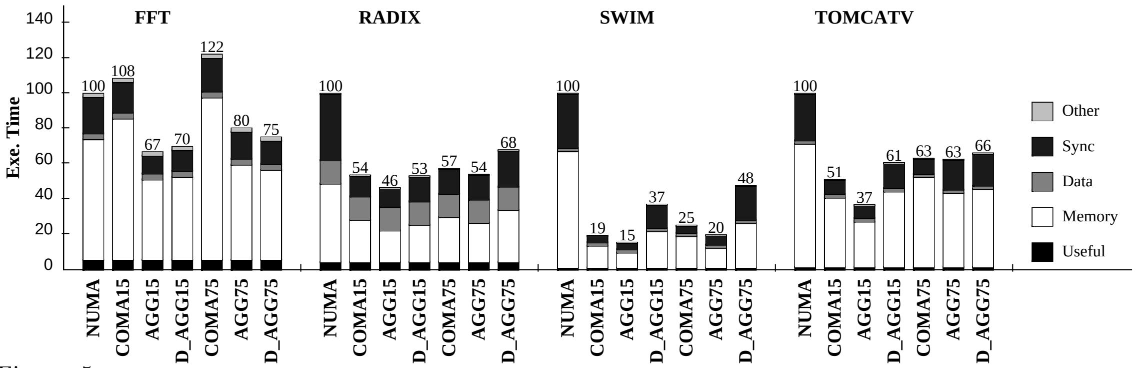

Figure 5: Execution time of the applications for different architectures. For each application, the bars are normalized to *NUMA*.

lelized by the Polaris compiler [1]. The table also shows the problem size and footprint of the applications, as well as the size of the two on-chip caches used in the experiments. Because the working sets of each of the threads of the applications are small, we use small caches. As indicated before, we use 15% and 75% memory pressure.

5.2 System Evaluation

Figure 5 compares the performance of our proposed system (*AGG*) to advanced *NUMA* and *COMA* organizations. For each application, the figure shows the execution time under *NUMA*, *COMA*, *AGG*, and *D_AGG*. The latter three organizations are evaluated with 15% and 75% memory pressure, shown with the suffix 15 and 75 respectively. For each application, the bars are normalized to *NUMA* and divided into *Useful* (instruction execution), *Memory* (stall due to memory accesses), *Data* (stall due to data hazards), *Sync* (stall due to synchronization), and *Other*.

From the figure, we can see that the proposed system is faster than the traditional ones. It is faster than *COMA* and much faster than *NUMA* because it has lower *Memory* time. This result holds across the different applications and memory pressures. *D_AGG* is a bit slower than *AGG* and, sometimes, even *COMA*. This implies that it is a good idea to give some associativity to the *P*-node memory.

To understand the lower execution time of *AGG*, we look at the time it takes to access memory in all these architectures. Figure 6 adds up the latency of all the loads in the application, irrespective of whether or not the processor is stalled. Loads 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*) network hops. As usual, the bars are normalized to *NUMA*.

The *AGG* system has three advantages. First, like the *COMA* system, it has the ability to replicate and migrate data to the processors that use them. This effect is responsible for transforming much of the *2Hop* time in *NUMA* to the cheaper *Memory* time in the other architectures. This effect is particularly strong in *COMA*, where all memory is cache. The second advantage is that, unlike in *COMA*, a line that is in state shared in *AGG* is guaranteed to be present in its home memory. In *COMA*, shared copies may get displaced from their home. As a result, many transactions that need three hops in *COMA* can be satisfied in two hops in *AGG*, cutting down the latency significantly. A dramatic example is *FFT*. Finally, the third advantage of *AGG* is that, by moving the directory controller off-chip, we reduce the contention inside the processor-memory chip, reduce the traffic in its pins, and give more resources to protocol processing. This effect, which is not obvious from the figure, should not have much impact on our results. This is because we chose to favor the traditional architectures by modeling neither the extra contention inside the processor-memory chip nor the fewer resources available to protocol processing.

These favorable results for *AGG* are despite the fact that the *COMA* and *NUMA* systems are more expensive. Indeed, the latter systems have the directory controller on chip, which increases the cost of the design. In addition, their chips are larger than *AGG*'s chips. Recall that, in *AGG*, we have split one processor-memory chip into two smaller, identical chips to be used in a *P* and a *D* node. Finally, the *COMA* system is more expensive because its cache coherence protocol is more complicated. The protocol has to ensure that last copies of data are not lost. All these reasons make the *AGG* system more attractive.

5.3 Row Tag Buffer Evaluation

We finish this section by evaluating the impact of the row tag buffer. The row tag buffer reduces the time taken by the accesses to local memory. However, in the multiprocessors considered, the latencies of the remote memory accesses are so large that the combined weight of all the processor stalls due to local memory accesses is relatively unimportant. This can be seen in Figure 6, where the *Memory* category is often relatively small. Consequently, the row tag buffer has a negligible impact on the multiprocessor systems considered and we do not show the performance data.

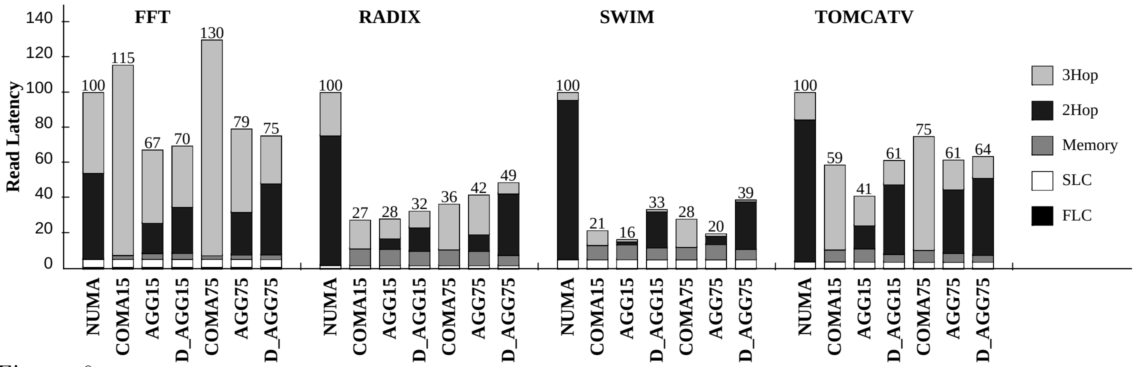

Figure 6: Total read latency of the applications for different architectures. For each application, the bars are normalized to *NUMA*.

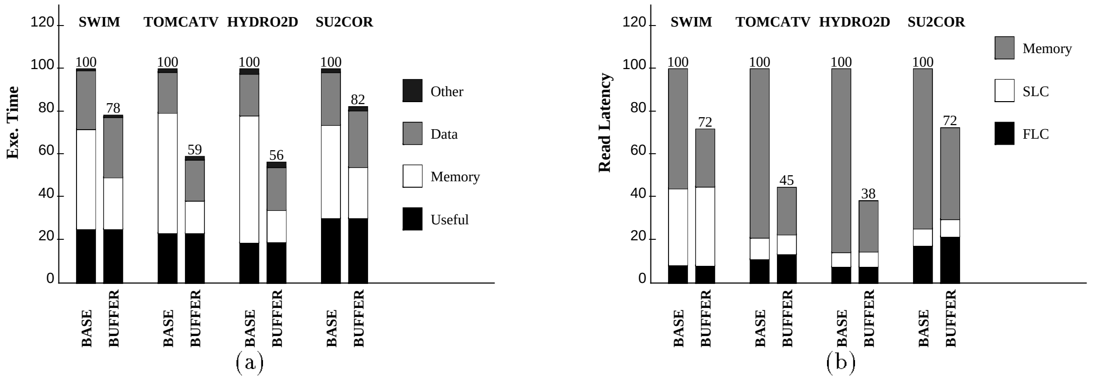

Figure 7: Execution time (Chart (a)) and total read latency (Chart (b)) of the applications with and without the row tag buffer.

In uniprocessor systems, however, the row tag buffer has a good impact. For this reason, we evaluate very briefly the row tag buffer in a uniprocessor system with a processor-memory chip like the one in Figure 1-(c). For the evaluation, we can only use the two uniprocessor applications from Table 3, namely *102.swim* and *101.tomcatv*. Consequently, we add two more Spec95 applications, namely *104.hydro2d* and *103.su2cor*, both with the reference problem size. The architectural parameters used in the evaluation are shown in Table 4. Note that we increase the size of the on-chip caches. This is because the applications are now single threaded and, therefore, threads now have larger working sets.

Table 4: Architectural parameters.

Device	Parameters
On-Chip L1	32 Kbyte size
On-Chip L2	256 Kbyte size
Memory	64 Mbyte size, 32 banks, 2 Kbytes per row Memory access time: 6 cycles without RAS; 30 cycles with RAS

Figure 7-(a) shows the impact of the row tag buffer on the execution time of the applications. For each application, the plot shows the execution time for a system without the buffer (*BASE*) and with the buffer (*BUFFER*). Each bar is divided into the same categories as Figure 5 except *Sync*. The figure shows that the row tag buffer uniformly reduces the execution time of all the applications. On average, the applications run about 30% faster.

The reason for the speedup can be seen in Figure 7-(b). The figure, similar to Figure 6, shows the aggregate latency of the loads in the application. In the architecture considered, loads satisfied by the on-chip memory (*Memory*) account for the largest fraction of the total latency. The figure shows that the row tag buffer reduces this component of the latency. It can be shown that, with the row tag buffer, about 80% of the accesses to memory skip the RAS cycle. The row tag buffer, therefore, is quite useful.

6 Conclusions

This paper has argued that the progressive integration of processor and memory has significant implications on the design of DSM systems. To exploit this integration best, we claimed that we need to redesign the nodes and reorganize the whole machine. We proposed nodes where memories are configured as caches and directory controllers are moved off-chip. The directory controllers are built out of the same hardware as the computing nodes and, therefore, can be considered non-computing nodes. The function of these non-computing nodes is to support the cache coherence protocol and to backup the application's data in their memories. Because off-the-shelf processors are so fast, these non-computing nodes manage the coherence operations and the storage of data in memory in software. A high-level evaluation of the proposed architecture shows that it is significantly better than idealized versions of the traditional COMA and NUMA organizations.

Acknowledgments

We thank David Koufaty, Venkata Krishnan, Jovan Mitrevski and the rest of the students in the I-ACOMA group for their comments on the paper. We thank Venkata Krishnan for developing the processor simulator. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Effective Automatic Parallelization with Polaris. *International Journal of Parallel Programming*, May 1995.

[2] The Standard Performance Evaluation Corporation. The Spec95fp Suite. URL: http://www.specbench.org.

[3] Silicon Graphics Inc. Origin Servers. URL: http://www.sgi.com/Products/hardware/servers.

[4] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, pages 88–97, October 1996.

[5] 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.

[6] 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.

[7] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

[8] 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.