Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA[1]

Zheng Zhang and **Josep Torrellas**

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801
zzhang,torrella@csrd.uiuc.edu
http://www.csrd.uiuc.edu/iacoma/

Abstract

Many future applications for scalable shared-memory multiprocessors are likely to have large working sets that overflow secondary or tertiary caches. Two possible solutions to this problem are to add a very large cache called *remote cache* that caches remote data (NUMA-RC), or organize the machine as a cache-only memory architecture (COMA). This paper tries to determine which solution is best.

To compare the performance of the two organizations for the same amount of total memory, we introduce a model of data sharing. The model uses three data sharing patterns: replication, read-mostly migration, and read-write migration. Replication data is accessed in read-mostly mode by several processors, while migration data is accessed largely by one processor at a time. For large working sets, the weight of the migration data largely determines whether COMA outperforms NUMA-RC. Ideally, COMA only needs to fit the replication data in its extra memory; the migration data will simply be swapped between attraction memories. The remote cache of NUMA-RC, instead, needs to house both the replication and the migration data. However, simulations of seven Splash2 applications show that COMA does not outperform NUMA-RC. This is due to two reasons. First, the extra memory added has more associativity in NUMA-RC than in COMA and, therefore, can be utilized better by the working set in NUMA-RC. Second, COMA memory accesses are more expensive. Of course, our results are affected by the applications used, which have been optimized for a cache-coherent NUMA machine. Overall, since NUMA-RC is cheaper, NUMA-RC is more cost-effective for these applications.

1 Introduction

The performance of scalable shared-memory multiprocessors is often determined by how well the applications exploit their memory hierarchy. Indeed, for an application to run effectively, caches must intercept most of the memory accesses. Unfortunately, many future applications of shared-memory multiprocessors may have very large working sets that overflow secondary or tertiary caches. In such cases, processors may need to issue many remote accesses and, as a result, the application may run slowly.

At least two solutions have been proposed for this problem. The first one is to add to the memory hierarchy of each node a very large DRAM cache called *remote cache*. The remote cache caches the data in the working set that is allocated in remote memory modules and, therefore, minimizes the cost of cache conflict misses. The operation of the remote cache is simple: when a secondary cache miss occurs and the physical address of the requested data is not within the range assigned to the memory module local to the node, the remote cache is checked to determine if it contains the requested data. If so, the data is provided to the secondary cache. Otherwise, the data is requested from the corresponding node and eventually stored in the remote cache before passing it to the processor. The remote cache is kept coherent at all times. This approach is used in current machines like Sun's S3.mp [6] and Sequent's STiNG [5]. In this paper, we call this machine organization NUMA-RC, for non-uniform memory access with remote cache (Figure 4-(b)).

A second approach is to use a Cache-Only Memory Architecture (COMA) organization (Figure 4-(c)). COMA machines [2, 4, 7, 8, 9] organize the distributed memories as caches called *attraction memories*. Like caches, attraction memories contain data and tags. A tag keeps information on the memory line that is currently stored in this location: enough of its physical address to identify the line, and its sharing state, for example shared, invalid, or exclusive. In addition, to reduce the number of conflicts, attraction memories are usually set-associative, often 4- to 16-ways.

In a COMA machine, when a processor misses on a line, it brings the line to both its cache and its attraction memory. As a result, memory lines migrate and replicate automatically in the different attraction memories of the machine, moving close to the processors that use them. A consequence of this is that cache conflict misses become cheap. When a processor suffers a secondary cache miss on a line, it first accesses its local attraction memory to see if the line is there. In the cases where the line was displaced from the secondary cache due to conflicts, chances are that the attraction memory still has a copy and can therefore supply it quickly to the processor. Examples of COMA machines, also called *All-cache*, are the Swedish Institute of Computer Science's DDM1 [2] and Kendall Square Research's KSR1 [7].

One of the overheads of COMA machines is that the attraction memory needs to be larger than the size of the application data to allow for the latter to replicate. However, even in this case, it is possible that, when a processor replicates a line by loading a fresh copy of it into its attraction memory, no space is available in the corresponding set. A space is then forced by overwriting one of the lines already in the set. This is fine unless the overwritten line is the last copy in all attraction memories. If so, or we suspect that it might be so, the victim line is sent to another attraction memory for it to keep. The lines that must be sent to another attraction memory are the *master* lines. Master lines include those in state exclusive and one of the copies of the lines in state shared. The algorithm that determines how to assign mastership in the case of a line that is shared by more than one processor is called the *data mastership* assignment algorithm.

The COMA organization that we consider in this paper is the Flat-COMA [3, 9] one. In this organization, each individual memory line has a directory entry in an agreed-upon node. This node is chosen based on the physical address of the line and is called the home of the line. When a processor requests a memory line that can not be found in its local attraction memory, a message is sent to the home of the line. The home, however, does not have a repository for the data. It may or may not have a copy of the data. Therefore, in the general case, the message checks the directory and is then forwarded

[1] This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436 and RIA MIP 93-08098, DARPA Contract No. DABT63-95-C-0097, NASA Contract No. NAG-1-613, and Intel Corporation.

to a third node that has a copy of the data.

In this paper, we address the issue of which of the two organizations, namely NUMA-RC and COMA, is the most cost-effective one for Splash2-like applications [10] with a large working set. To our knowledge, the only prior work directly related to ours is a comparison of NUMA to COMA architectures by Stenström *et al* [9]. Such work, however, used more primitive NUMA and COMA organizations and did not dimension the caches so that the working set overflows them. The conclusion of the work was that each architecture performs best for a subset of the applications.

In our analysis, we use seven Splash2 applications [10] running on simulated 32-node architectures. The cache sizes are such that the working sets of most of the applications overflow them. To make the comparison fair, the NUMA-RC and COMA organizations examined have the same total amount of memory space. To perform our analysis, we design a new model of data sharing that classifies the data sharing patterns into replication, read-mostly migration, and read-write migration. We show that it is hard for COMA to outperform NUMA-RC for these applications for two reasons. First, the extra memory added to the two machines has more associativity in NUMA-RC than in COMA and, therefore, can be utilized better by the working set in NUMA-RC. Second, COMA memory accesses are more expensive. Of course, our results are affected by the applications used, which have been optimized for a cache-coherent NUMA machine. Overall, since NUMA-RC is cheaper, NUMA-RC is more cost-effective for these applications.

This paper is organized as follows: Section 2 describes the model of data sharing; Section 3 describes the methodology used for the evaluation; Section 4 presents the evaluation; Section 5 analyzes some architectural variations; and Section 6 concludes.

2 Modeling the Effect of Memory Accesses

To understand the differences in performance between the NUMA-RC and COMA organizations, we introduce a simple model of the effect of memory accesses. In the following discussion, we assume that the two architectures have the same primary and secondary caches. Furthermore, the size of the remote cache in NUMA-RC is as large as the extra memory given to the COMA machine. This extra memory and the remote cache we call *MemOvhd*. We feel that this is a reasonably fair way of comparing NUMA-RC to COMA even though we are neglecting the cost of the COMA tags. This is because DRAM tags add very little extra memory overhead. Finally, in our timing analyses, we assume that an access to the remote cache in NUMA-RC and to the COMA attraction memory take the same time. In the following, we first look at the impact of the different types of accesses and then focus on the effect of remote conflict misses.

2.1 Impact of the Different Accesses

We can gain insight into the differences in performance between the NUMA-RC and COMA organizations by classifying the memory accesses of an application into *local*, *remote-cold*, *remote-coh*, and *remote-conf*. *Local* accesses are those satisfied by the local memory hierarchy. For NUMA-RC, they include those satisfied by the two levels of cache, the local memory, and the remote cache. For COMA, they include those satisfied by the two levels of cache and the local attraction memory. *Remote-cold* are misses in the local memory hierarchy that occur when a processor accesses a memory line for the processor's first time. *Remote-coh* are misses in the local memory hierarchy due to coherence effects. Finally, *remote-conf* are misses in the local memory hierarchy due to overflow of the local caching space.

For a given memory line size, the number of *remote-cold* and *remote-coh* accesses is largely intrinsic to the application and, therefore, is not expected to change much across architectures. The number of *remote-conf* accesses, however, varies across architectures. Indeed, NUMA-RC has fewer *remote-conf* accesses than plain NUMA thanks to the remote cache. It is not clear, however, whether COMA has fewer *remote-conf* accesses than NUMA-RC or vice-versa. Finally, since the total number of memory accesses cannot vary across architectures, any decrease in the number of *remote-conf* accesses will necessarily appear as an increase in the number of *local* accesses. As a result, NUMA has fewer local accesses than the other architectures. The complete picture is shown in a graphical manner in Figure 1-(a). The figure is not drawn to scale and assumes that COMA has fewer *remote-conf* accesses than NUMA-RC.

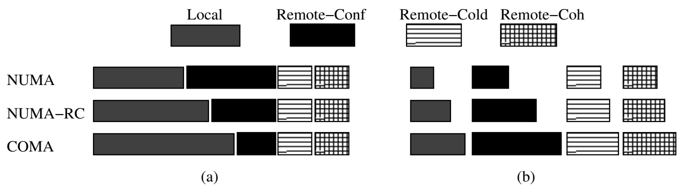

Figure 1: Qualitative impact of the different types of memory accesses. Chart (a) shows a breakdown of the total number of memory accesses while Chart (b) shows the average processor stall time per memory access. The figure is not drawn to scale.

Each of the groups of memory accesses in Figure 1-(a) causes a certain stall time to the processor. The exact amount of stall time, however, depends on the architecture. While the total amount of stall time is hard to predict, we can reason about the average stall time per access in each group. For example, the average stall time caused by a *local* access increases from NUMA to NUMA-RC. This is because the *local* accesses in NUMA-RC include those in NUMA plus those that hit in the remote cache. The latter accesses drive the average stall time up. Suppose that, as it is shown in Figure 1-(a), COMA has fewer *remote-conf* accesses than NUMA-RC. Then, COMA also has a higher average stall time due to *local* accesses than NUMA-RC. This is because more accesses are intercepted by the slow attraction memory.

The average stall time due to *remote-cold*, *remote-conf*, and *remote-coh* accesses increases from NUMA to NUMA-RC to COMA. The reason for this is threefold. First, any remote access in NUMA-RC or COMA will be slower than in NUMA because it involves a failed lookup access in the remote cache or attraction memory respectively. Second, while *remote-cold* and *remote-conf* accesses are always satisfied in two network hops in NUMA and NUMA-RC, they may involve three hops in COMA. The latter may occur when the home node needs to make space and displaces the master copy of a line. Any subsequent request for the line that reaches the home will be forwarded to a third node and, therefore, involve three network hops. Finally, for *remote-coh* accesses, two more effects result in longer stall times in COMA relative to NUMA-RC, and in NUMA-RC relative to NUMA. One effect is the displacement of dirty data from a node's caches or attraction memory. A displacement sends the line to the home memory and, therefore, makes it possible for a future request to the line to access the line in only two network hops. Note that the number of such displacements is lower in NUMA than in the other two architectures. This is because NUMA-RC and COMA have a larger local caching space and, therefore, suffer fewer displacements. The second effect is that, under NUMA-RC and NUMA, in a read miss transaction on data that is exclusive in some other processor, the home ends up being updated with the latest copy of the data. Again, this reduces the number of network hops required in future trans-

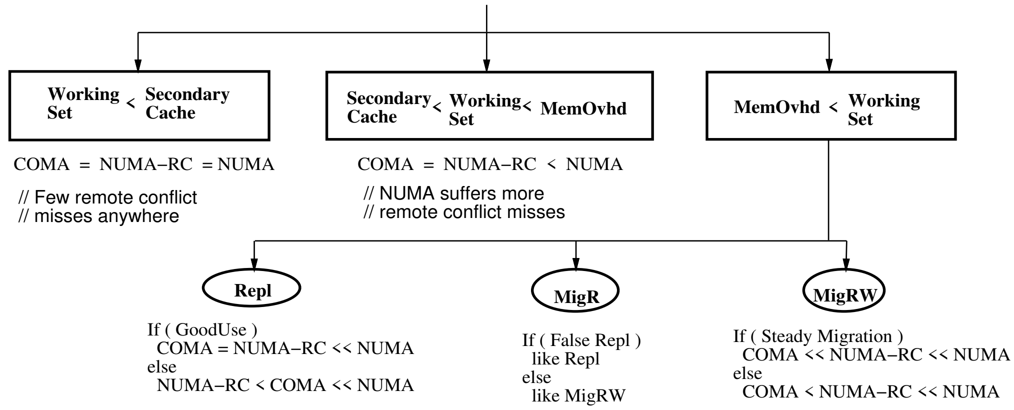

Figure 2: A simplified model for remote conflict misses in NUMA, NUMA-RC, and COMA architectures. All comparisons between architectures in the figure refer to the number of remote conflict misses.

actions to two.

Figure 1-(b) qualitatively shows how the average processor stall time per memory access varies across architectures. The impact of the stall time due to memory accesses in the different architectures depends on the breakdown of the access types and on the average stall times.

2.2 A Model for Remote Conflict Misses

Of all the types of memory accesses discussed above, this paper focuses on the *remote-conf* ones. The importance of such accesses depends on how the sizes of the caches and the *MemOvhd* in the machine compare to the size of the working set of the application. Recall that *MemOvhd* is the size of the remote cache or the extra memory added to COMA. Such comparison can be used to gain insight into the differences between NUMA, NUMA-RC, and COMA. Indeed, as shown in Figure 2, there are three interesting cases depending on whether the working set is smaller than the secondary cache, larger than the secondary cache but smaller than *MemOvhd*, or larger than *MemOvhd*. In our discussion, we assume a round-robin memory allocation scheme among the nodes and, therefore, that the data in a processor's working set is largely allocated in remote memory modules. This is likely to be true unless the data layout is carefully optimized by the programmer.

If the working set is smaller than the secondary cache, the number of *remote-conf* accesses is small in all three architectures. If, instead, the size of the working set is between the secondary cache size and *MemOvhd*, the number of *remote-conf* accesses is likely to be high in NUMA and small in both NUMA-RC and COMA. The interesting case is when the working set size is larger than *MemOvhd*. In this case, NUMA again has the highest number of *remote-conf* accesses. However, NUMA-RC may have more or fewer *remote-conf* accesses than COMA.

To understand why, we identify several major sharing patterns that give insight into how COMA machines work. The first sharing pattern, which we call *replication* (*Repl*), occurs when a data structure is being accessed by several processors in a read-mostly manner. In a COMA machine, the data structure replicates itself in the attraction memories of the referencing processors. The second sharing pattern occurs when a data structure is accessed largely by one processor at a time. This pattern includes private data and data that is accessed by several processors in the course of the application. To include the most general case, we call this pattern *migration* (*Mig*). We distinguish the case where the data structure is read-mostly (*MigR*) or read-write (*MigRW*). In a COMA machine, *MigRW* data will tend to reside in only one attraction memory at a time. However, *MigR* data, while being used by only one processor, may linger on in several attraction memories.

To predict the relative number of remote conflict misses in NUMA-RC and COMA organizations with the same *MemOvhd*, it is important to identify what sharing pattern is dominant in the application's working set. The weight of the *Mig* data in the working set largely determines whether COMA has fewer remote conflict misses than NUMA-RC. Ideally, COMA needs to fit only the *Repl* data in its *MemOvhd*, since the *Mig* data will simply use up in one attraction memory the same space that it frees up in another attraction memory. The *MemOvhd* of a NUMA-RC machine, however, needs to house both the *Repl* and the *Mig* data of the working set. As a result, COMA machines can potentially tolerate larger working sets than NUMA-RC machines without inducing as many *remote-conf* accesses. In the ideal case, if the size of the *Repl* data in the working set is no larger than *MemOvhd*, a COMA may have no *remote-conf* accesses. This is irrespective of the size of the *Mig* data.

This ideal scenario is distorted by two implementation-induced effects. The first one is that, for a fixed amount of *MemOvhd*, NUMA-RC can utilize the *MemOvhd* memory better than COMA. This is a result of the limited associativity of attraction memories. Indeed, consider 4-way set-associative attraction memories, 4-way set-associative remote caches, and a *MemOvhd* of 25% of all memory. In COMA, each attraction memory set has one free line (Figure 3). When a processor references a remote memory line and has to store the line in its attraction memory (in COMA) or remote cache (in NUMA-RC), the *MemOvhd* in COMA appears like a direct mapped cache, while the *MemOvhd* in NUMA-RC is 4-way set-associative. It is true that the whole attraction memory of COMA is available to store the line. However, it is clear that the *MemOvhd* in NUMA-RC suffers fewer conflicts and can, therefore, be utilized better than the *MemOvhd* in COMA. This problem particularly affects the data with the highest memory appetite, namely *Repl* data.

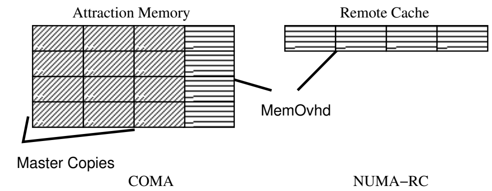

Figure 3: NUMA-RC can utilize the *MemOvhd* memory better than COMA.

The second effect that may hurt COMA performance is that, under a group of data mastership assignment and memory line replacement protocols in COMA, *MigR* data may have a behavior somewhat like *Repl* data. As a result, it performs sub-optimally under COMA. Indeed, consider a COMA data mastership assignment protocol such that, when a processor suffers a remote read miss, it loads the data in state shared non-master. The processor that supplies the data keeps the mastership. Further assume that the data is *MigR* and, therefore, will not be used by the supplier of the data anymore. In an environment with high memory pressure, memory line displacements will be frequent. Usually, when we need to displace a line from a set of a set-associative attraction memory, non-master lines are selected for displacement first. As a result, in the *MigR* scenario described, the copy in the receiver processor is more likely to be displaced in the future, while the useless copy in the supplier processor is more likely to hang on. We call this effect *False Replication*. False replication tends to increase the number of *remote-conf* misses in COMA over NUMA-RC. False replication of a given memory line can be largely eliminated with a protocol such that, when a processor misses on the line, it gets the master copy.

We can now go back to Figure 2 and divide the case where the working set is larger than *MemOvhd* into three extreme cases: all the data is *Repl*, all the data is *MigR*, and all the data is *MigRW*. Consider first the case where all the data is *Repl*. If the *MemOvhd* area of the attraction memory is used well, COMA has roughly the same *remote-conf* misses as NUMA-RC. Otherwise, COMA suffers more *remote-conf* misses than NUMA-RC.

If all the data is *MigRW*, COMA has the least number of misses. This is the most favorable scenario for COMA. However, the savings in misses depend on the type of migration behavior. If a processor accesses a data structure many times before it relinquishes it to another processor, COMA will work relatively better. We call this behavior *Steady MigRW*. The extreme example of this behavior is frequently-used private data. If, instead, a data structure changes owner processors very often, COMA will work less well. This is because COMA cannot eliminate the misses present in the transient phase after the data changes owners. We call this behavior *Transient MigRW*. Therefore, if the data is *Steady MigRW*, COMA saves many misses relative to NUMA-RC. Otherwise, the miss savings are smaller.

Finally, if all data is *MigR*, the situation is trickier. If the data mastership assignment algorithm is such that it causes false replication, this case is similar to the *Repl* case. Otherwise, it is similar to the *MigRW* case.

Overall, this model gives a qualitative comparison of the number of *remote-conf* misses in the three architectures. To address the issue of machine performance, we need to combine this information with the relative cost of memory accesses in the architectures as shown in Figure 1.

3 Experimental Setup

To compare the performance of the NUMA-RC and COMA organizations, we use execution-driven simulations of parallel applications. In the following, we first examine the architectures and the simulation system. Then, we examine the applications.

3.1 Architectures and Simulation System

We model three 32-processor architectures, namely a NUMA, a NUMA-RC, and a COMA. The architectures are shown in Figure 4. Each architecture includes, per node, two levels of cache, a memory controller, and a portion of the global memory and directory. The simulated processor runs at 200 MHz. While the primary cache is direct-mapped, on-chip, and has 16-byte lines, the secondary cache is 4-way set associative, off-chip, and has 32-byte lines. The bus between the primary and secondary caches is 32-bit wide, while the one between the secondary cache and the bus is 64-bit wide, has split transactions, and runs at 40 MHz. The memory modules are pipelined and have a latency of 12 processor clocks to fetch the first word of the 8-word memory lines. Both the remote cache in NUMA-RC and the attraction memory in COMA are 4-way set-associative and have 32-byte lines. The directory controller has an occupancy time of 3 processor clocks. The network is modeled as a general interconnect where the latency of one network hop in any transaction is a fixed 100 processor cycles. With all these parameters, the resulting no-contention round-trip latency of read requests satisfied at various levels of the memory hierarchy is shown in Table 1. In all three architectures, caches are maintained coherent in hardware via a full-vector write-invalidate protocol. The machine uses the release memory consistency model. All the contention in the system is carefully modeled except in the network, where a constant latency is assumed.

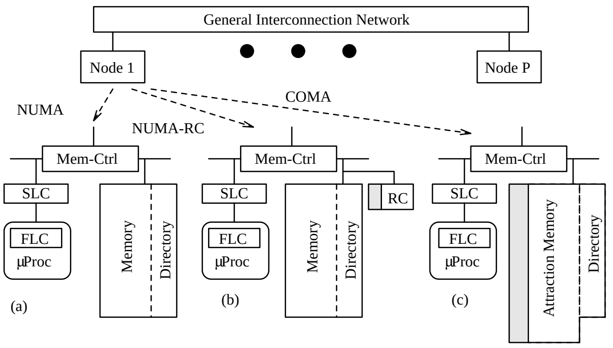

Figure 4: NUMA, NUMA-RC, and COMA architectures simulated. In the figure, FLC, SLC, and RC stand for first-level cache, second-level cache, and remote cache. The shaded boxes represent tags.

Table 1: No-contention round-trip latency of read accesses for the different architectures. The latency of local and remote accesses assume that the data is supplied from the memory, the remote cache, or attraction memory.

Round Trip Access	Latency (Cycles)		
	NUMA	NUMA-RC	COMA
Secondary Cache	4	4	4
Local	39	39	39
Remote (2-Hop)	249	264	264
Remote (3-Hop)	351	373	373

To dimension the memory hierarchy so that the working set of the applications often overflows the secondary caches, we proceed as follows. For each application, we measure the footprint of the data accessed by the application. We divide the size of such footprint by the number of processors and then (somewhat arbitrarily) by 16. The resulting size, rounded up to the nearest power of two, is the size that we usually use for the secondary caches for that particular application. The resulting secondary cache sizes are shown in Table 2. For most of the applications, the resulting sizes are slightly smaller than the working set. We used this approach instead of using one fixed cache size for all applications because we feel that using a fixed size would wipe out important differences among applications. Once the secondary cache size is set, the primary cache size is set to a quarter of it. Finally, for NUMA-RC and COMA, we use a range of values for *MemOvhd*. The values chosen for *MemOvhd* are 1/5, 2/6, 3/7, and 4/8 of the total memory (including *MemOvhd*). Expressed as percentages, these figures are 20%, 33%, 43%, and 50% of the total memory respectively.

The default algorithms that we use for data mastership assignment and memory replacement in COMA were designed for simplicity. However, they can cause false replication. When a processor suffers a read miss on a line, the request is

Table 2: Characteristics of the applications used in this study.

Application	Description	Data Size	Ftprt(MB)	SLC (KB)	Type
Barnes	Barnes-Hut N-body simulation	16K bodies	3.8	8	*Repl*
Water-NQ	Water molecule N-body simulation	512 molecules	7.0	8	*Repl*
Water-SP	Improved version of Water-NQ	512 molecules	2.8	8	*Repl*
Volrend	3-D volume rendering	Head	22.9	32	*Repl/MigR*
Radix	Integer radix sort	256K keys	2.8	8	*MigRW/MigR*
Ocean	Ocean eddy current simulation	258 × 258 grids	15.5	32	*MigRW*
FFT	Complex 1-D 6-step FFT	64K points	3.7	8	*MigRW*

forwarded to the node that keeps the master copy. However, to reduce the number of transitions, the requesting processor does not acquire the master state: the master state is left where it was. When we need space and, therefore, need to replace a memory line from a set in an attraction memory, we displace lines with the following priority, from higher to lower: invalid, shared non-master, and master. When a master line is displaced, it is sent to its home. The home first attempts to forward the mastership to one of the sharers of the line if there is one. If that fails, then the home is forced to make space for the line.

Finally, the evaluation is performed with execution-driven simulations using the TangoLite tracing environment [1]. We have developed a highly-efficient simulation package called Zibrasim, capable of simulating a number of shared-memory multiprocessor configurations.

3.2 Applications

We examine seven applications from the Splash2 suite [10] running on 32 simulated processors. We use the default problem size in all the applications. The main characteristics of the applications are listed in Table 2, including their footprint in memory, the size of the secondary cache used, and the major type of data in the working set according to our model of Section 2.

Three of the applications have working sets where the *Repl* sharing pattern dominates. These applications are *Barnes*, *Water-NQ*, and *Water-SP*. They are all N-body problems but each of them uses a different algorithm. The most important phase of the N-body problem is usually the inter-body force computation, in which a typical iteration involves picking up a given body and interacting it with many others in the system. Usually, those "many others" are *Repl* data, while the body being worked on is *MigRW*. Therefore, *Repl* dominates.

In *Barnes*, an octree is used to represent the space in three dimensions. The leaves of the tree contain the bodies, while the internal tree nodes are space cells which hold the quantity and position of the center-of-mass for all the bodies in their subtrees. Force computation for one body is accomplished by a partial traversal of the octree. Since the tree data is read by all processors, it is largely *Repl*, although the degree of sharing gradually decreases as we go to lower levels of the tree. Pointers to bodies and cells, and the bodies that a processor owns are *MigRW*. However, the *Repl* component dominates.

Water-NQ models the motion of water molecules in a cubical box. The N molecules are stored in a linear array, with each of the p processors owning N/p of them. During the inter-molecule force computation phase, each molecule interacts with $N/2$ other molecules. These $N/2$ molecules are of *Repl* type. Each processor's partition of molecules is *MigRW* data. Like in Barnes, the *MigRW* data in *Water-NQ* is not very important. Finally, *Water-SP* models the same problem as *Water-NQ* but uses a different algorithm. It imposes a 3-D grid of boxes on the problem domain. For a given molecule, a processor only needs to look at the neighboring boxes for molecules to interact with. This effectively makes the working set and the size of the *Repl* data much smaller than in *Water-NQ*.

The two applications that exhibit *MigRW* access patterns are *Ocean* and *FFT*. *Ocean* simulates a cuboidal ocean basin over many time-steps. The data structures are composed of many two-dimensional arrays. Each processor owns a fixed sub-grid partition of the arrays, and communication happens only at the sub-grid boundaries across time-steps. Since most of the shared data is largely per-processor private, this is the prototypical example of a *MigRW* application. The migration pattern of *Ocean* is *steady* as defined in Section 2.2.

In *FFT*, the complex data points of the FFT are organized as a matrix. Each processor gets a contiguous set of rows of the matrix. The algorithm involves several steps. Each step is a transpose-FFT pair. Although data in *FFT* is conceptually *MigRW*, the transposition introduces coherence misses when some rows are accessed by processor i, then processor j, and then i again across time steps. Overall, the migration pattern of *FFT* is rather *transient* as defined in Section 2.2.

The last type of sharing pattern that we consider is *MigR*. Two applications have some *MigR* data in their working set: *Radix* and *Volrend*. However, they also exhibit other, maybe even more dominant, patterns. We have put these applications in a different class to emphasize *MigR*.

Volrend is a visualization application rendering a 3-D volume using a ray casting technique. The volume is represented as a cube of voxels, and an octree is used to traverse the volume quickly. The parallelism is exploited across rays. The image plane is partitioned among the processors and is then further divided into fixed-sized image tiles. The tiles are the unit of task scheduling and stealing. The voxel data set is mostly *MigR*. However, the voxels at the edge of an image tile may be read by neighboring processors and, therefore, are of type *Repl*. The top levels of the octree are *Repl* data because they are read by all the processors, while the lower levels spread into different regions of the volume and gradually become *MigR*. Finally, although the size of the data in *Volrend* is large, the working sets are fairly small.

Radix performs an iterative radix sort on an array of keys. The data set of one processor is the set of keys of a statically assigned index range, and a histogram used to rank the keys. In each iteration, a given processor reads a new set of keys. The keys are thus *MigR* data within one iteration. However, the keys that a processor reads change across iterations. In addition, within one iteration, a processor will read the keys only twice. Consequently, the migration pattern of *Radix* is *transient* as defined in Section 2.2.

4 Evaluation

Using the experimental setup just described, we compare the performance of the NUMA, NUMA-RC, and COMA architectures. In our evaluation, we analyze the memory accesses of the applications according to the model described in Section 2. In the following, we group the applications into three groups depending on their sharing behavior as described in Table 2.

4.1 *Repl* Applications

To compare the NUMA, NUMA-RC, and COMA archi-

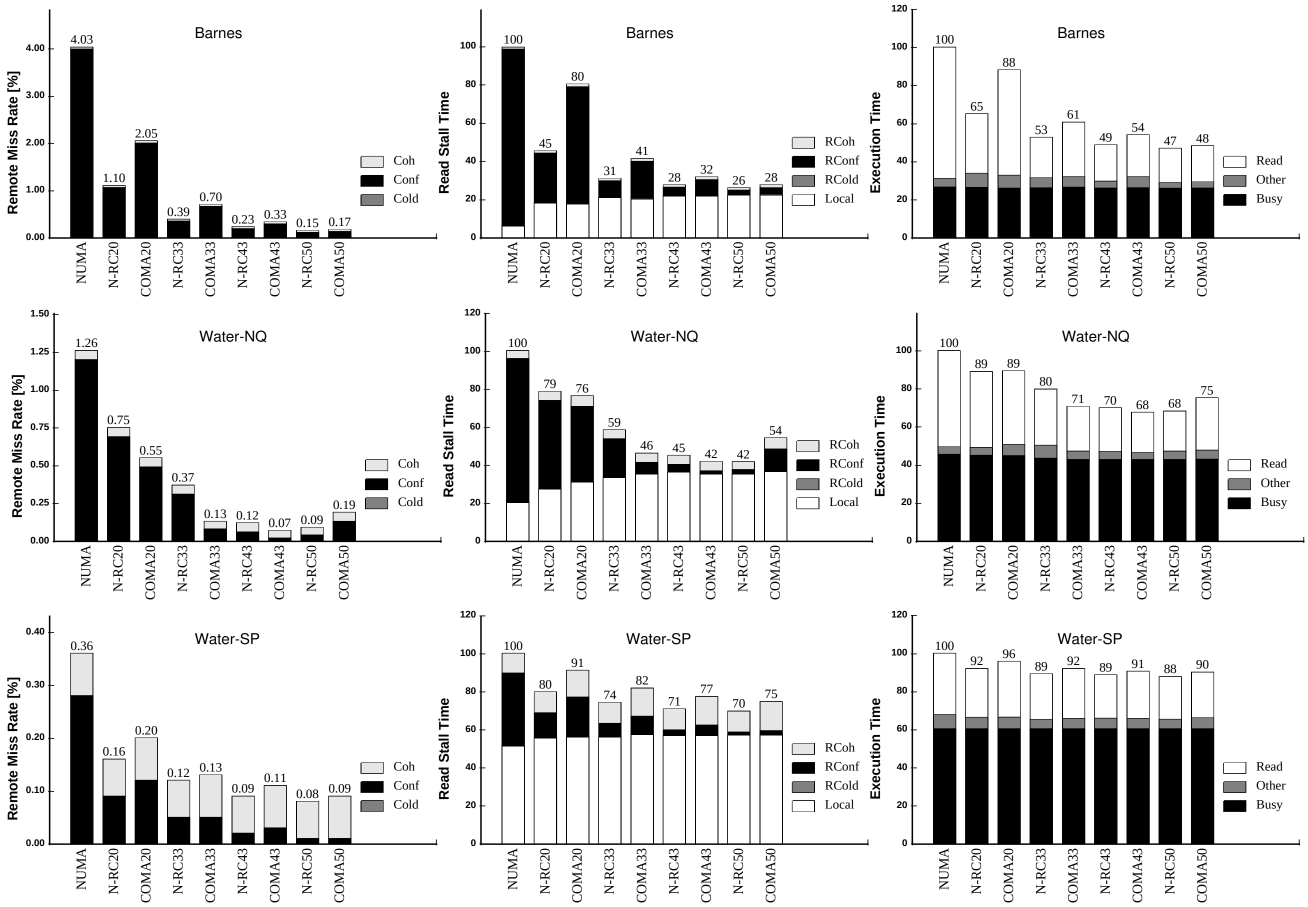

Figure 5: Remote miss rate, read stall time, and execution time for the *Repl* applications: *Barnes* (top row), *Water-NQ* (middle row), and *Water-SP* (bottom row).

tectures, we examine three metrics: the remote miss rate, the processor read stall time, and the total execution time. These metrics are shown in Figure 5 for *Barnes* (first row), *Water-NQ* (second row), and *Water-SP* (third row). Each chart shows 9 bars which correspond, from left to right, to NUMA and NUMA-RC interleaved with COMA for *MemOvhd* of 20%, 33%, 43%, and 50%. The bars in the read stall time and execution time charts are normalized to NUMA. In the remote miss rate chart, the misses are classified into start-up misses (*Cold*), conflict misses (*Conf*), and coherence misses (*Coh*). In the read stall time chart, the stall time is divided into stall caused by accesses satisfied by the local memory hierarchy (*Local*, see Section 2.1), remote misses due to the start-up effect (*RCold*), remote misses due to conflicts (*RConf*), and remote misses due to coherence (*RCoh*). Finally, in the execution time chart, the execution time is divided into execution of instructions (*Busy*), read access stall time (*Read*), and stall time due to synchronization and write buffer overflow (*Other*).

In our analysis, we focus first on the remote miss rate charts, which are the leftmost ones. In all cases, *NUMA* has a much higher remote miss rate than the other architectures. This is because the working set of the applications is larger than the secondary caches simulated. In *Barnes*, the working set is captured by the 33% *MemOvhd*, while in *Water-NQ* and *Water-SP*, the working set is captured by the 43% *MemOvhd*. This can be seen by comparing the different *N-RC* bars. Indeed, for *Barnes* and *Water-NQ*, the *N-RC* bars flatten out beyond *N-RC33* and *N-RC43*, respectively. For *Water-SP*, the miss rate is already very low for *N-RC20*.

The interesting comparison is between *N-RC20* and *COMA20* for *Barnes* and *Water-NQ*. This is where the working set is larger than *MemOvhd*. This scenario corresponds to the *Repl* subtree in Figure 2. However, these two applications have opposite behavior. For *Barnes*, *COMA20* has about 90% more remote conflict misses than *N-RC20*. This is because *Barnes* is a pointer intensive application and its *Repl* data, which is a tree, is scattered around. It is too irregular to use the *MemOvhd* of COMA well. Consequently, COMA suffers more remote conflict misses than NUMA-RC. As a result, the read stall time for *Barnes* in *COMA20* is about 80% larger than in *N-RC20* (central chart). Furthermore, *COMA20* runs about 35% slower than *N-RC20* (rightmost chart).

Water-NQ shows a different trend. The *Repl* data in *Water-NQ* are water molecules stored in a linear array. Consequently, *Water-NQ* uses the *MemOvhd* in COMA as well as it does in NUMA-RC. In addition, COMA handles the *MigRW* data of *Water-NQ* more effectively. As a result, COMA has fewer misses than NUMA-RC (leftmost chart). Unfortunately, as predicted in the memory access model of Section 2.1, COMA has a higher average stall time for every type of remote accesses. As a result, the reduction in read stall time and execution time is not as dramatic. In fact, *COMA20* performs about the same as *N-RC20*, and *COMA33* is only 11% faster than *N-RC33* (rightmost chart).

Water-SP shows similar patterns as *Barnes*. This is because its *Repl* data is also unstructured. However, *Water-SP* is less interesting than *Barnes* because the remote miss rate is very small in NUMA to start with (leftmost chart). Overall, as seen in the rightmost chart, the difference in execution time between COMA and NUMA-RC for any *MemOvhd* is small. In general, as *MemOvhd* increases, the working set of all the applications is captured more. Therefore, the difference in performance between COMA and NUMA-RC decreases.

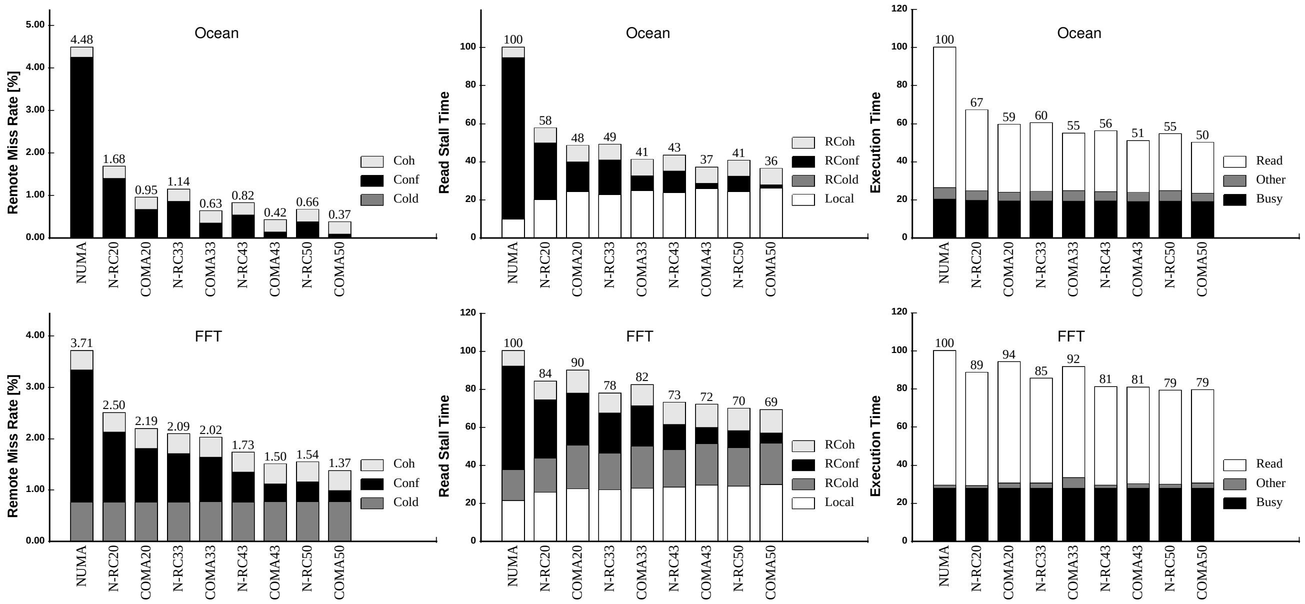

Figure 6: Remote miss rate, read stall time, and execution time for the *MigRW* applications: *Ocean* (top row) and *FFT* (bottom row).

4.2 *MigRW* Applications

Figure 6 shows the remote miss rate, read stall time, and execution time for *Ocean* and *FFT* under NUMA, NUMA-RC, and COMA. The charts are organized in the same way as those in Figure 5. For both applications, *NUMA* has a high miss rate. This is because the working sets of the applications are larger than the secondary caches. Furthermore, the *N-RC33* bars are smaller than the *N-RC20* bars and, therefore, the working set of the applications is larger than the 20% *MemOvhd*. This makes the comparison between NUMA and NUMA-RC for 20% *MemOvhd* interesting.

If we compare the remote conflict misses in *N-RC20* and *COMA20* we see that, in *Ocean*, COMA has about 50% fewer remote conflict misses. This is an application where COMA nicely exploits the *MigRW* nature of the data. In *FFT*, COMA has about 20% fewer remote conflict misses than NUMA-RC. This more moderate reduction is the result of the *transient* migration pattern of FFT. The impact of COMA on the remote miss rates is watered down by the large contribution of the cold and coherence misses.

For *Ocean*, the read stall time is smaller in *COMA20* than in *N-RC20* (central chart). However, the difference is not too large because, as indicated in Section 2.1, accesses in COMA are more costly on average. This effect is especially bad in *FFT* because there are many cold and coherence misses. Overall, with a 20% *MemOvhd*, COMA runs 12% faster than NUMA-RC for *Ocean*, but it is 6% slower for *FFT* (rightmost chart). Here we see that the quality of data migration, that is, whether the migration pattern is *steady* or not, makes a difference.

4.3 *MigR* Applications

Figure 7 shows the remote miss rate, read stall time, and execution time for *Radix* and *Volrend* under NUMA, NUMA-RC, and COMA. From the remote miss rate charts, we can induce the sizes of the working sets of the applications. For *Radix*, the working set does not fit with 20% *MemOvhd* but it fits with 33% *MemOvhd*. For *Volrend*, the working set already fits with 20% *MemOvhd*.

From these observations, we would expect that, for *MemOvhd*, the number of remote conflict misses in *Radix* is smaller in COMA than in NUMA-RC. For *Volrend*, COMA and NUMA-RC should have very similar numbers of remote conflict misses. The data behaves as expected in *Radix*. The miss reduction is small because the migration pattern is *transient*. However, the data for *Volrend* shows that *COMA20* has about twice as many remote conflict misses as *N-RC20*.

The main reason for these many remote conflict misses in *Volrend* is that the *MemOvhd* is used inefficiently in COMA. Note that, for a 20% *MemOvhd*, some of the sets of the 4-way set-associative COMA attraction memory are completely full and do not have any room for replication of data. This is bad for *Volrend*, which has much *Repl* data. We measured that, in *Volrend*, each attraction memory miss is followed by a chain of an average of 3.6 transfers of displaced memory lines between different attraction memories. Note that, as we move to higher *MemOvhd*, the number of conflict misses in COMA decreases dramatically.

A second, less important reason for this higher number of remote conflict misses is the false replication effect discussed in Section 2.2. Such effect causes some extra remote conflict misses in COMA, for both *Radix* and *Volrend*. We address this problem in Section 5.

The more numerous conflicts in COMA for *Volrend*, combined with the higher latency of misses in both *Radix* and *Volrend*, result in higher read stall times for *COMA20* than for *N-RC20* (central chart). Overall, as shown in the rightmost chart, *COMA20* runs 8% and 20% slower than *N-RC20* in *Radix* and *Volrend* respectively.

4.4 Summary

The previous analysis has resulted in several observations. The first one is that it is surprisingly hard for COMA to outperform NUMA-RC for the Splash2 applications. Indeed, only *Ocean* clearly and consistently runs faster on COMA than on NUMA-RC. The reason for this effect is that, for applications to run significantly faster on COMA, at least two conditions have to be met. The first one is that the size of the remote data in the working set be larger than *MemOvhd*. Only in that case can remote conflict misses cause a major performance problem. The second condition is that the dominant sharing pattern in the working set be *steady MigRW*. As it is, of the applications studied, only *Ocean* meets these conditions. On average, the applications studied run 9% slower on COMA than on NUMA-RC for 20% *MemOvhd*. Overall, considering

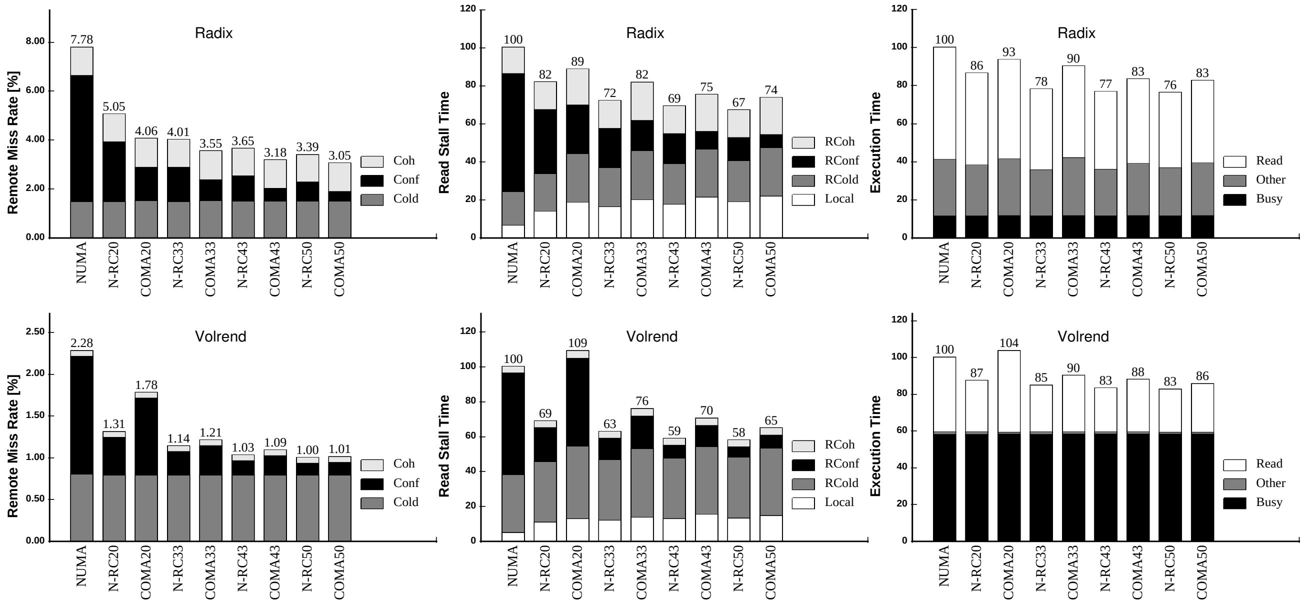

Figure 7: Remote miss rate, read stall time, and execution time for the *MigR* applications: *Radix* (top row) and *Volrend* (bottom row).

that NUMA-RC machines are simpler to design than COMA machines, NUMA-RC machines are more cost-effective than COMA machines for the applications studied. This finding may be less surprising if we realize that many of the Splash2 applications have been optimized for a cache-coherent NUMA machine.

A second observation is that COMA with a relatively small *MemOvhd* is susceptible to substantial attraction memory conflicts. The situation is especially bad when, due to small *MemOvhd*, some of the sets in the attraction memory start up the execution already completely full.

A third, obvious observation is that, in applications with large working sets relative to the secondary cache sizes, NUMA machines are not able to keep up with COMA or NUMA-RC. We expect that the sizes of the problems executed in multiprocessors will continue to increase, especially when more commercial workloads are executed in these machines.

Finally, we note that after many remote conflict misses are eliminated with NUMA-RC or COMA, it becomes very important to optimize the accesses that hit in the local memory hierarchy (*Local* in the read stall time charts). Indeed, this can be seen in Figure 5 for *Barnes*. While *COMA50* has about 20 times fewer remote misses than NUMA, its read stall time is only 3.5 times smaller than NUMA.

5 Improving COMA

Section 2.2 indicated that the naive data mastership assignment and memory line replacement protocol used for COMA in this and other previous papers causes false replication. In this section, we modify the protocol to eliminate false replication. In addition, we examine another protocol that attempts to minimize the number of remote misses.

The protocol that we have used in the previous sections was described in Section 3. In such a protocol, which we will call *Base*, when a processor suffers a remote read miss, it loads the requested data in state shared non-master. The processor that supplies the data keeps the mastership. This was done to simplify the protocol by keeping the number of mastership transfers to a minimum. In addition, when a memory line needs to be replaced from a set in an attraction memory, the algorithm chooses first an invalid line, then a shared non-master line and, finally, a master line. Clearly, master lines are less likely to be replaced soon. Therefore, in the example above, the processor that supplies the line may well keep the copy for a while. If the data in the line is of *MigR* type, false replication occurs.

To eliminate false replication, we use a protocol where remote read misses load the data in state shared master. This protocol we call *Optimized*. This protocol should have the additional advantage of reducing misses. This is because the principle of locality indicates that the processor that references a given shared line last is likely to be the one that references the line next. Therefore, if the processor that misses on the data gets the line in state master, the processor is more likely to keep the line in the attraction memory and, therefore, its next access to the line is more likely to be a cache hit.

A second weakness of the *Base* protocol is that when a processor suffers a remote read miss, it reads the data from the node that currently keeps the mastership, even if the home node has a non-master copy of the line. Recall that, to access the master node, the request has to interrogate the directory in the home node. The total transaction, therefore, involves three network hops. However, the home node could provide a copy of the line. The transaction would then involve only two network hops. In *Base*, we use the three-hop approach for simplicity. In *Optimized*, we use the two-hop transaction. Of course, if the home node does not have mastership of the line when it supplies the line, the requesting processor does not get the mastership.

While the *Optimized* protocol may minimize misses and network hops, both the *Optimized* and *Base* protocols may also create extra misses indirectly via replacements. When a master line needs to be replaced from an attraction memory and there are no sharers, the two protocols send it to the line's home node. This transfer may in turn force another replacement in the home node. One can imagine that a miss may create a chain of such replacements. The overall result is that the homes get polluted with data that they are not using and, consequently, suffer more misses in the future. To alleviate this problem, Joe and Hennessy proposed an algorithm that we call *Swapping* [4]. In the *Swapping* algorithm, when a processor suffers a remote read miss, the line is loaded in state shared master. If the load forces the processor to displace a master line and there are no sharers, the displaced line is sent to the supplying processor. The supplying processor

Table 3: The impact of several data mastership assignment and memory line replacement protocols on the remote miss rate, number of network hops per remote read and number of replacements per attraction memory miss. In the table, *Opt* stands for *Optimized* and *Swap* for *Swapping*.

Application	Remote Miss Rate(%)				# of Network Hops per Read				# of Replacements per AM Miss		
	N-RC20	*COMA20*			*N-RC20*	*COMA20*					
		Base	Opt.	Swap.		Base	Opt.	Swap.	Base	Opt.	Swap.
Barnes	1.10	2.05	1.86	1.37	2.00	2.48	2.43	2.77	4.27	2.79	1.38
Water-NQ	0.75	0.55	0.44	0.33	2.03	2.52	2.51	2.76	1.87	1.21	0.89
Water-SP	0.16	0.20	0.18	0.18	2.02	2.56	2.64	2.80	2.40	1.39	0.93
Volrend	1.31	1.78	1.58	1.98	2.04	2.48	2.47	2.85	3.39	1.84	1.03
Radix	5.05	4.06	3.78	3.56	2.06	2.59	2.66	2.83	1.49	1.01	0.74
Ocean	1.68	0.95	0.78	0.74	2.11	2.43	2.44	2.82	1.21	1.05	0.86
FFT	2.50	2.20	1.89	1.71	2.06	2.46	2.47	2.81	1.73	1.21	0.83
Average	1.79	1.68	1.50	1.41	2.05	2.50	2.52	2.81	2.34	1.50	0.95

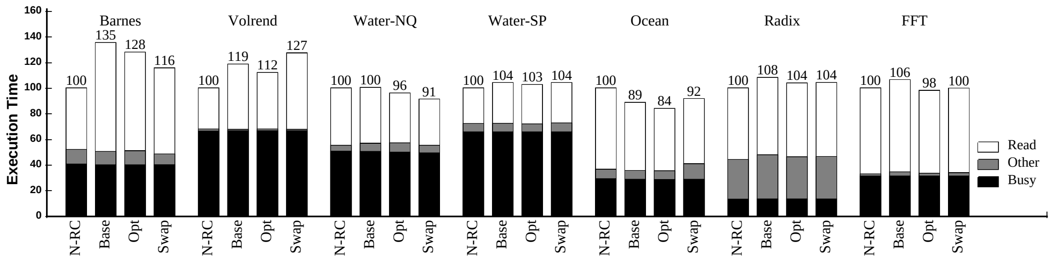

Figure 8: Execution time of the applications under NUMA-RC and COMA with three different protocols. All bars correspond to 20% *MemOvhd*.

will likely have space for a line because it has just supplied another line and given up the mastership of it.

With the *Swapping* protocol, no chains of replacements are possible and, therefore, attraction memories are not polluted. This should decrease the number of misses. Furthermore, as indicated before, since the mastership is transferred on each read miss, the number of misses is likely to be further reduced. One shortcoming of this algorithm, however, is that it does not handle *Repl* data well. Indeed, when a processor suffers a remote read miss and, as a result, makes a local copy of a remote line, the line in the supplying processor is often overwritten, therefore inhibiting replication. A second shortcoming is that the home node is not likely to get a copy of the line as often as in the *Optimized* or *Base* protocols. As a result, most of the remote transactions in *Swapping* are likely to involve three network hops and, therefore, have long latency. In our implementation, we include the second optimization described for the *Optimized* protocol: in a read miss, if the node that is home to the data has a non-master copy of the line, the transaction finishes in two hops. In this case, however, the line is loaded in state shared non-master and, if a master replacement is generated, it is directed to the home of the displaced line.

The *Base*, *Optimized*, and *Swapping* protocols are examined in Table 3. The table shows the remote miss rates, number of network hops per remote read access and number of replacements per attraction memory miss for each of the applications. In addition, the table shows data for a NUMA-RC machine as a reference. In all cases, *MemOvhd* is 20%. The column with the number of replacements per attraction memory miss in *Base* explains the high miss rates observed for some applications under COMA. The applications for which COMA has a higher miss rate than NUMA-RC (Columns 2 and 3) are those where severe replacements occur. For example, *Barnes* and *Volrend*, the applications with COMA miss rates higher than NUMA-RC, have 4.3 and 3.4 replacements per miss respectively. Therefore, we see that for small, badly-utilized *MemOvhd*, misses induced by replacements may degrade COMA performance.

If we focus on the *Optimized* protocol, we see that the remote miss rate is smaller than the *Base* one. This is consistent with our previous discussion. The average reduction is 11%. As expected, the number of replacements per attraction memory miss also decreases significantly, namely an average of 36%. The number of network hops per remote read access does not change much. This counter-intuitive result occurs because, as the number of replacements decreases, so does the chance that the home has the data. We note in passing that the number of network hops for NUMA-RC is very small. The reason is two-fold. First, the large working sets considered often displace data from the remote cache and send it to the home. As a result, when processors need data, they often find it at the home. In the COMA protocols, instead, data often gets displaced from its home node. Second, in NUMA-RC, when a processor suffers a read miss on a line that is dirty elsewhere, the home receives a copy of the line in state shared. Future misses on the line will get it from the home.

Focusing now on the *Swapping* protocol, we see that it decreases the number of replacements significantly. As a result, it generally removes more remote misses than the *Optimized* protocol. However, it does not do well for *Volrend*. The reason is that *Volrend* has much *Repl* data. For the other applications with *Repl* data like *Barnes*, *Water-NQ*, and *Water-SP*, however, *Swapping* does well. The favorable effects of the protocol overwhelm the unfavorable ones. The table also shows that *Swapping* induces a higher number of network hops per read than *Optimized*. As indicated above, this is expected because *Swapping* has many three-hop transactions.

To summarize, both the *Optimized* and the *Swapping* protocols induce fewer remote misses than *Base*. *Optimized* is better because it handles *Repl* data better and its transactions involve fewer network hops. To determine how these miss reductions translate into execution time, consider Figure 8. The figure shows the total execution time of the seven applications under the protocols listed in Table 3 with 20% *MemOvhd*. For each application, the execution time is normalized to that of NUMA-RC.

From the figure we see that, due to the lower miss rates of *Optimized* and *Swapping*, the new protocols are faster than *Base*. Given that they are not much more complicated than *Base*, they are more cost-effective than *Base*. *Base* is simply not enough. The *Optimized* protocol is the most cost-effective one of the two because it is faster. For *Volrend*, it eliminates the false replication induced by *Base*.

If we compare NUMA-RC to COMA with the *Optimized* protocol, we see that COMA is only an average of 4% slower. The main reason why COMA is still slower is that *Volrend* and *Barnes* still suffer many conflict misses under COMA. The cause is the lack of available replication space in some sets of the attraction memory in COMA. Overall, while the COMA results are better than the ones that we found in Section 4, a COMA organization is still less attractive than a NUMA-RC organization for these applications.

Finally, with the optimization of transferring mastership on read, false replication is removed. Consequently, the *MigR* branch of our model in Figure 2 can be merged with the *MigRW* branch. Therefore, there are only two cases, namely *Repl* and *Mig*. This leads to a simpler model with which to compare NUMA-RC and COMA. The applications that we selected to emphasize *MigR* can now be moved to other branches. For example, *Volrend* is now under *Repl* and *Radix* under *Mig*. This is the reason why we have changed the order of the applications in Figure 8.

6 Conclusions and Future Work

Future applications for scalable shared-memory multiprocessors may have large working sets that overflow the machines' secondary or tertiary caches. Two possible solutions to this problem are to add a *remote cache* (NUMA-RC), or to organize the machine as a COMA. This paper has addressed the question of which solution is best.

This paper makes two contributions. First, it presents a model of data sharing that gives insight into which machine should perform better for the same amount of total memory. We identify three key sharing patterns, namely replication, read-mostly migration, and read-write migration. Ideally, COMA needs to fit only the replication data in its extra memory space while the remote cache of NUMA-RC needs to house both the replication and the migration data.

The second contribution is that we found that it is hard for COMA to outperform NUMA-RC for Splash2-like applications. On average, these applications run 4% slower on COMA. This is due to two reasons. First, the *MemOvhd* has more associativity in NUMA-RC than in COMA and, therefore, can be utilized better by the working set in NUMA-RC. Second, COMA memory accesses are more expensive. Of course, our results are affected by the applications used, which have been optimized for a cache-coherent NUMA machine. Overall, since NUMA-RC is cheaper, NUMA-RC is more cost-effective for these applications.

The future work for this research evolves in several directions. First, we will examine whether NUMA-RC works just as well for other applications like commercial ones or applications written for a bus-based machine. Second, we are looking at whether the COMA hardware can separate the migration from the replication data and exploit the former without being harmed by the latter. Finally, we are addressing the issue of how to optimize the memory accesses that hit in the local memory hierarchy. In a NUMA-RC or COMA architecture, these accesses often account for one of the largest aggregate costs.

Acknowledgments

We thank the referees and members of the I-ACOMA group for their valuable feedback. Steven Woo and Phil Lacroute kindly helped us with the Splash2 applications. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] S. Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance*. PhD thesis, Stanford University, June 1993.

[2] E. Hagersten, A. Landin, and S. Haridi. DDM – A Cache-Only Memory Architecture. *IEEE Computer*, pages 44–54, September 1992.

[3] T. Joe. *COMA-F: A Non-Hierarchical Cache Only Memory Architecture*. PhD thesis, Stanford University, 1995.

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

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

[6] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishi. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages I1–I10, August 1995.

[7] Kendall Square Research. *KSR1 Technical Summary*. Waltham, MA, 1992.

[8] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 276–285, January 1995.

[9] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, May 1992.

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