Excel-NUMA: Toward Programmability, Simplicity, and High Performance

Zheng Zhang, **Marcelo Cintra**, and **Josep Torrellas**
Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801
zzhang,cintra,torrella@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

Center for Supercomputing Research and Development
Technical Report 1544
November 1996

Abstract

While hardware-coherent scalable shared-memory multiprocessors are relatively easy to program, they still require substantial programming effort to deliver high performance. Specifically, to minimize remote accesses, shared data must be allocated in memories local to the processors that will use it most, and the application working sets must be tuned for caches. It has been claimed that these activities are less necessary in hardware COMA machines like Flat-COMA because memory lines automatically migrate to the requesting nodes. Unfortunately, Flat-COMA is complex to design because its cache coherence protocol must ensure that last copies of lines are not lost. Consequently, we would like a machine as programmable as Flat-COMA, as simple as CC-NUMA, and that outperforms both. This paper presents an attempt at such an architecture: *Excel-NUMA* (EX-NUMA).

The idea is to exploit the fact that, after a local memory line is written by a processor, the memory storage that contained the line is unutilized. We use that storage to temporarily hold remote data displaced from the local caches. This enables automatic data movement like in Flat-COMA, enhancing programmability. The hardware required to manage the system is a simple module called *Remote Data Table (RDT)* that we add to a CC-NUMA. No changes to the global cache coherence protocol are required. Simulations of Splash2 applications show that EX-NUMA outperforms CC-NUMA and Flat-COMA in every single application. Furthermore, EX-NUMA gets close to the upper bound possible, namely elimination of all conflict misses.

1 Introduction

Hardware-coherent scalable shared-memory multiprocessors are hoped to deliver abundant computing power without surrendering much programmability. Manufacturers like Convex [3], Sequent [8], and Silicon Graphics [10] among others have recently commercialized systems of this type.

Some users of these systems, however, complain that, to achieve truly high performance, they still need to apply substantial programming effort. This is a result of the physical distribution of the memory system and the increasing cycle-count cost of remote memory accesses. Specifically, it is important to carefully lay out shared data structures in memory modules local to the processors that will use them most. Additionally, it is important to tune the application so that working sets fit in caches. These two techniques typically minimize remote memory accesses.

To offset the problem of remote accesses, most of the new machines now include a per-node *remote cache*. This is a very large DRAM L3 cache that caches data allocated in remote memory modules. We call these machines *NUMA-RC*, for CC-NUMA with remote caches. Unfortunately, workload sizes, particularly in the database domain, are growing so fast that their working sets often overflow even these caches. As a result, codes still need tuning to remove remote accesses.

It has been claimed that hardware COMA machines like Flat-COMA [11] may solve this programmability problem. In Flat-COMA, memory lines automatically migrate to the local memory of the processor that is using them. Consequently, it is argued that the initial placement of the data in memory is not very important and that tuning the working sets for caches is not as important as in NUMA-RC. This reduces the programming effort. In addition, Flat-COMA is considered to deliver high performance. This is because, by supporting migration at a memory-line grain size, it does not have any of the page-related overheads that a software incarnation of COMA like Simple-COMA may suffer. For example, Simple-COMA may suffer from internal page fragmentation that induces high page unmapping, copying and remapping overheads in the operating system.

Unfortunately, Flat-COMA is complicated to design. This is because its cache coherence protocol must ensure that last copies of lines are not lost [11]. This issue introduces corner cases and races in the protocol. In addition, recent work has shown that some types of data access patterns do not use the COMA memory as well as they use a plain remote cache [15]. As a result, the performance of Flat-COMA may not be as good as expected.

With this in mind, we would like an architecture that combines the advantages of both Flat-COMA and NUMA-RC. Ideally, the architecture should support fine-grain automatic data migration (and, therefore, be as programmable) as Flat-COMA, be nearly as simple as NUMA-RC, and outperform both of them by supporting all data access patterns well. In this paper, we present our proposal for such an architecture: *Excel-NUMA* (EX-NUMA).

The idea behind EX-NUMA is to exploit the fact that, after a local memory line is written by a processor, the memory storage that contained the line is unutilized. We can use that storage to temporarily hold remote data that is being displaced from the local cache hierarchy. This enables automatic data movement similar to Flat-COMA, enhancing programmability. This remote data in memory can be managed with a simple module called *Remote Data Table (RDT)* that we add to a NUMA-RC. No changes to the global cache coherence protocol are required. Simulations of Splash2 applications show that EX-NUMA outperforms NUMA-RC and Flat-COMA in every single application. Furthermore, EX-NUMA gets close to the upper bound possible, namely elimination of all conflict misses.

This paper is organized as follows: Section 2 presents a model for NUMA-RC and Flat-COMA; Section 3 presents EX-NUMA; Section 4 discusses how we evaluate it; Section 5 evaluates the base EX-NUMA; Section 6 examines extensions to the base EX-NUMA; and Section 7 considers related work.

2 A Model for NUMA-RC and Flat-COMA

To motivate the EX-NUMA architecture, we compare a NUMA-RC and a Flat-COMA such that they have the same primary and secondary caches, the same memory overhead (*MemOvhd*), and where it takes the same time to access the remote cache in NUMA-RC as the attraction memory in Flat-COMA. *MemOvhd* is the size of the remote cache in NUMA-RC or the extra memory given to Flat-COMA for data replication. We neglect the difference in tag space because of its small relative size. In the rest of the paper we refer to Flat-COMA as COMA.

Memory accesses can be *local*, *remote-cold*, *remote-coh*, or *remote-conf*. *Local* are those satisfied by the local memory hierarchy, namely caches, local attraction memory in COMA, and, in NUMA-RC, remote cache and local memory. The other accesses are *remote-cold*, *remote-coh*, or *remote-conf* depending on whether the miss is due to accessing a line for the processor's first time, data sharing, or overflow in the local memory hierarchy respectively. For a given application, COMA and NUMA-RC have the same total number of accesses, the same *remote-cold* accesses, and the same *remote-coh* accesses. The number of *remote-conf* accesses, however, varies between the two architectures. It changes at the expense of the number of *local* accesses.

Usually, *remote-conf* accesses are significant only if the size of the remote data in the working set of a thread is larger than the size of *MemOvhd*. Under such conditions, we want to understand whether COMA or NUMA-RC suffers more *remote-conf* accesses. To this end, we identify two major data access patterns, namely data replication (*Repl*) and migration (*Mig*). Replication occurs when a line is accessed in a read-mostly manner by several processors. Migration occurs when a line allocated remotely is accessed largely by one processor at a time. Migration data includes data that is largely accessed by one processor or that is accessed sequentially by several processors in the course of the application. It should not be confused with migratory data [6], which is a special type of migration data with frequently-changing locality. Migration data can be read-mostly (*MigR*) or read-write (*MigRW*).

The weight of the *Mig* data in the working set largely determines whether COMA has fewer *remote-conf* accesses than NUMA-RC. Ideally, the *MemOvhd* of a COMA only needs to fit the *Repl* data. This is because the *Mig* data will simply use up in one attraction memory the same space that it frees up in another one. The *MemOvhd* of a NUMA-RC, however, needs to house both the *Repl* and the *Mig* data. Consequently, in an ideal situation, we would start seeing *remote-conf* accesses only when:

$$\begin{array}{ll} COMA: & Repl > MemOvhd \\ NUMA-RC: & Repl + Mig > MemOvhd \end{array}$$

Therefore, if *Mig* dominates, COMA has fewer *remote-conf* accesses than NUMA-RC. This COMA ability to handle *Mig* data for free is the reason why COMA is considered more programmable. There is less need to lay out the data carefully in memory and, more arguably, to tune working sets for caches.

Real caches, however, suffer conflicts even with small working sets. Interestingly, for a fixed size of *MemOvhd*, NUMA-RC can utilize the *MemOvhd* memory better and, therefore, suffer fewer conflicts than COMA. This is a result of the limited associativity of attraction memories. Indeed, consider 4-way set-associative attraction memories and remote caches, and a *MemOvhd* of 25% of all memory. In COMA, each set in the attraction memory has one free entry (Figure 1). Consequently, the *MemOvhd* in COMA appears like a direct-mapped cache. However, the *MemOvhd* in the NUMA-RC is 4-way set-associative (Figure 1). As a result, NUMA-RC suffers fewer *remote-conf* accesses. This effect particularly affects the data structures with the highest memory appetite, namely *Repl* data. Overall, therefore, we conclude that, if *Mig* data dominates, COMA will likely have fewer *remote-conf* accesses than NUMA-RC. However, if *Repl* data dominates, the difference is likely to be small or NUMA-RC may have fewer *remote-conf* accesses than COMA.

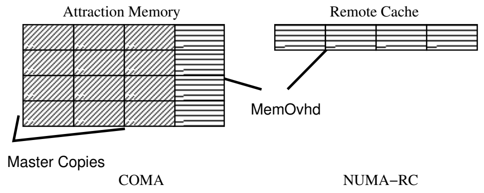

Figure 1: NUMA-RC can utilize the *MemOvhd* memory better than COMA.

Note, however, that the performance difference between COMA and NUMA-RC is not determined only by the relative number of *remote-conf* accesses. It is also affected by how much processor stall time is induced by the average *local*, *remote-cold*, *remote-conf*, and *remote-coh* accesses as well. These stall times tend to be larger in COMA than in NUMA-RC for several reasons. First, the average *local* access costs more in COMA because, in general, COMA tends to have more *local* accesses than NUMA-RC. These extra *local* accesses tend to come from hits in the slow attraction memory, thereby increasing the average *local* cost. Secondly, the average *remote-conf* access also costs more in COMA. This is because, while these accesses are always satisfied in two network hops in NUMA-RC, they may involve three hops in COMA due to conflict-induced line displacements from the home memory. Finally, the average *remote-coh* access also costs more in COMA. This is because, in COMA, the home is less likely to have an up-to-date copy of a shared line, and therefore, allow the transaction to finish in just two network hops. One cause is the fact that exclusive data displacement from a COMA attraction memory is less frequent than from the smaller NUMA-RC remote cache. Hence, the home in COMA is less frequently up to date. A second cause is that, in a read miss on data that is exclusive in some node, our protocol updates the home node's memory in NUMA-RC, but not in COMA. If the home gets updated, future read transactions can be satisfied in two network hops.

Aside from these performance issues, COMA and NUMA-RC have different design complexity. COMA is more complicated because its cache coherence protocol has to ensure that last copies of lines are not lost [11]. This issue introduces corner-cases and races in the protocol.

Overall, we would like an architecture that combines all the advantages of both COMA and NUMA-RC. The architecture should handle *Repl* data as NUMA-RC, handle *Mig* data (and, therefore, be as programmable) as COMA, issue low-stall memory accesses like NUMA-RC, and be nearly as simple as NUMA-RC. Our attempt at such an architecture is called *Excel-NUMA* (EX-NUMA) and is presented next.

3 The EX-NUMA Architecture

To describe EX-NUMA, we present the concept, its implementation, a cost-performance comparison to NUMA-RC and COMA and, finally, some advanced issues.

3.1 The Concept of EX-NUMA

When data is accessed under the *Mig* pattern, the memory of a NUMA machine is underutilized. Indeed, after the data migrates to the cache of the requesting processor, the copy left in memory is unused for a while. Furthermore, if the data is actually written, the copy in memory becomes stale and will definitely not be ever used. Any such stale line left in memory we call a *cell*. It includes lines written by the local as well as remote processors. The up-to-date copy of the line, resident in a cache, we call the *owner* of the cell. When the owner is written back, the cell is destroyed. The creation of cells is illustrated in Figures 2-(a) and 2-(b): Figure 2-(a) shows a 2-processor machine with 6 arrays allocated in memory, while Figure 2-(b) shows the system after processor *P1* wrote to *2* and *P2* wrote to *1*.

The idea behind EX-NUMA is that each node records the cells that exist in its memory and temporarily uses them to hold remote data that gets displaced from its cache. When the owner finally returns to the memory, the line stored in the cell is displaced and sent to its home. Figure 2-(c) illustrates the concept. In the figure, processor *P1* writes to *4*, bringing it to its cache. In doing so, it displaces *2*, saving it in the cells of its memory. *P2* does a similar operation for other arrays.

In EX-NUMA, *all* the data in a memory module can migrate to another memory module. This is shown in Figure 2-(d), where *P1* finally writes to *6* and *P2* to *5*. At that point, all the data originally allocated in *P1*'s node has migrated to *P2*'s node and vice-versa. The machine, therefore,

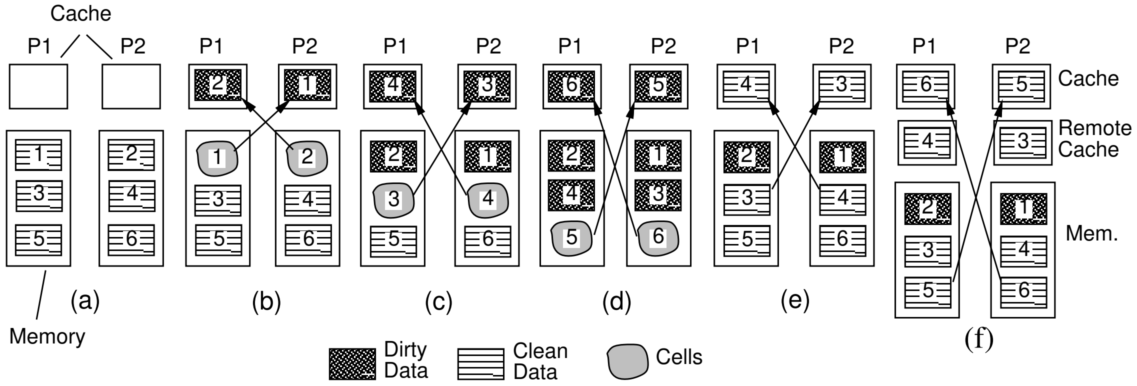

Figure 2: Basic operation of EX-NUMA. The figures correspond to a 2-processor EX-NUMA with arrays *1* to *6* allocated in memory.

behaves like a COMA for the *Mig* data in the figure. On the other hand, EX-NUMA performs no differently than a regular CC-NUMA machine for *Repl* data. Indeed, EX-NUMA can only replicate as much data as it fits in its caches. This is shown in Figure 2-(e), a continuation of Figure 2-(b) after processor *P1* reads *4* and *P2* reads *3*. However, if we add remote caches to an EX-NUMA, we will not decrease its ability to handle *Mig* data, and we will enable it to handle *Repl* data as well as NUMA-RC. For example, Figure 2-(f) shows Figure 2-(e) after *P1* reads *6* and *P2* reads *5*.

The ideal scenario presented, where cells are created and filled uniformly across nodes, shows that EX-NUMA can handle *MigRW* for free, without using any part of *MemOvhd*. Consequently, ideally, *remote-conf* accesses appear only when:

$$Repl + MigR > MemOvhd$$

MigR is, in fact, relatively uncommon [14]. Furthermore, we will see in Section 6 that it is easy to enhance the EX-NUMA design to handle *MigR* as well. Therefore, EX-NUMA can behave very much like COMA for *Mig* data.

Note, however, that there is a key difference in the way data migration occurs in COMA and EX-NUMA. In COMA, a line migrates directly, displacing the line that exists at the destination location. In EX-NUMA, instead, for a line to migrate, some other transaction must have created a cell in the destination location. While it would seem that the resulting migration ability of EX-NUMA is weaker than COMA, the evaluation will show that, in practice, there is not much difference.

3.2 The Implementation of EX-NUMA

The functionality described can be implemented with a module called *Remote Data Table (RDT)*. The RDT is closely coupled with the memory controller and the remote cache. It can be thought of as an extension of the tag array of the remote cache. The RDT contains the tag array of the remote cache and pointers to cells in the local memory. In a relatively simple baseline implementation, the RDT keeps the cell pointers in a DRAM table. In a more complicated, advanced implementation, the RDT is combined with the directory and the cell pointers come nearly for free because they reuse the directory storage. In this section, we focus only on the baseline RDT implementation. Figure 3-(a) shows how it fits in a possible organization of an EX-NUMA node.

The RDT records the location and state of all the cells present in the local memory. When a remote line is brought into the secondary cache in state non-exclusive, the RDT saves a copy

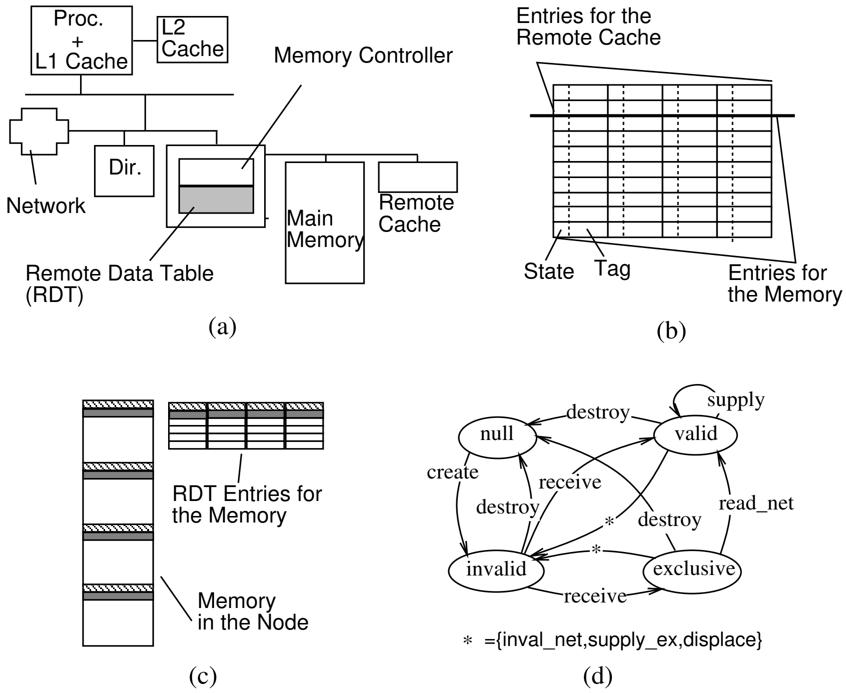

Figure 3: Baseline EX-NUMA: node organization (Chart (a)), 4-way set-associative RDT (Chart (b)), mapping between the RDT entries for the memory and memory locations (Chart (c)) and state diagram for RDT entry states (Chart (d)).

in one of the cells or in the remote cache. This is done because that line may be later displaced silently from the secondary cache. When a remote line in state exclusive is displaced from the local secondary cache, the RDT tries to save it in one of the cells or in the remote cache. Later, if the secondary cache requests it again, the RDT is checked and, if the line is still stored locally, it is supplied. If the request loads the line in state exclusive, the cell or remote cache entry is freed up as it is being read. Otherwise, the cell or remote cache entry keeps a copy of the line in case it is later displaced silently from the secondary cache.

The remote lines stored in the cells and remote cache are kept coherent as invalidation, read, and read with invalidation messages arrive from other nodes. In addition, at any time, the owner of a cell may return to its home. The owner may come from a remote node or from the secondary cache of the local node. The owner displaces the line that was stored in its cell. The cell is destroyed and, if the displaced line is dirty, it is sent to its home. Overall, it is important to note that all the tasks related to the RDT are local to the node and thus do not require global changes to the cache coherence protocol.

We treat cells and remote cache entries similarly because we do the bookkeeping for both of them in the same way (Figure 3-(b)). A fraction of the RDT's entries are the remote cache's tag array. There is one entry per line in the remote cache. The rest of the RDT entries are dedicated to the memory. They are used to identify cells. In the simplest design, there is one entry associated to each memory line. Each RDT entry contains an address tag and state fields. The state can be *invalid*, *valid*, *exclusive* or *null*. The latter is only possible in RDT entries dedicated to memory and means that the corresponding memory location does not have a cell. *Invalid* means that the location has an empty cell, while *valid* and *exclusive* mean that the location contains a remote line in one of these states.

To save a line in the remote cache, we select a RDT entry among those dedicated to the remote cache; to save it in a cell, we select a RDT entry among those dedicated to the memory. In either case, we use the index bits of the remote line's address to select the entry, and set the tag and state fields as usual. The only difference is that we will not be able to save the line in memory if the selected memory location does not have a cell. To reduce the likelihood of this event, the RDT is associative. For example, Figure 3-(c) shows a four-way set-associative RDT. In this case, there is still a one-to-one mapping between RDT entries and memory lines. This is shown in different shades in the figure. However, a given remote line can now be inserted in any one of the four entries of the set.

The finite state machine for RDT entry states is shown in Figure 3-(d). Initially, all RDT entries dedicated to memory are *null* while the others are *invalid*. After a write to a local line, the state of the corresponding RDT entry changes from *null* to *invalid* (*create* transition). When a remote line is brought into the secondary cache in state non-exclusive or a remote line in state exclusive is displaced from the secondary cache, the RDT tries to save it. The RDT identifies the appropriate memory-dedicated set and remote cache-dedicated set that can potentially take the line. It selects one entry from these two groups of entries using the following receiving algorithm: select, with decreasing priority, an *invalid*, *valid*, or *exclusive* entry. In the latter two cases, the line currently using the entry is displaced (*displace* transitions). By giving low priority to *exclusive*, we try to avoid sending an owner home, which could in turn destroy a cell. However, we could be unfairly thrashing within *Repl* and *MigR* data. Other, fairer receiving algorithms are studied in [14]. In any case, the entry selected is filled and set to *valid* or *exclusive* depending on the state of the incoming line (*receive* transitions).

When the secondary cache requests a remote line, if the RDT has a copy in a cell or in the remote cache, it supplies it. If the request loads the line in state exclusive, irrespective of the current state of the line, the cell or remote cache entry is set to *invalid* as it is read (*supply_ex* transitions). Otherwise, it is left in state *valid* (*supply* transition). Later, when an owner is written back, the state of the cell is set to *null* (*destroy* transitions). Finally, coherence messages coming from other nodes may invalidate, read, or read and invalidate data in cells or in the remote cache (*inval_net* and *read_net* transitions).

3.3 Cost-Performance Comparison to NUMA-RC and COMA

To assess the performance of EX-NUMA, we start with the model of Section 2. For a given application, NUMA-RC, COMA, and EX-NUMA all have the same number of total, *remote-cold*, and *remote-coh* accesses. Similarly, the number of combined *local* plus *remote-conf* accesses is the same in the three architectures. To determine the relative number of *remote-conf* accesses, we consider three extreme scenarios (Figure 4-(a)). First, if all the data is *Repl*, EX-NUMA behaves as well as the machine with the fewest number of *remote-conf* accesses, namely NUMA-RC, because it reverts to NUMA-RC. Second, if all the data is *MigRW*, EX-NUMA behaves nearly as well as the machine with the fewest number of *remote-conf* accesses, namely COMA, because it supports large-scale line-level data migration. Data migration in EX-NUMA, however, is trickier because it needs cells to get started (Section 3.1). On the other hand, *Mig* data can also benefit from the remote cache of EX-NUMA. Finally, if all the data is *MigR*, EX-NUMA reverts to NUMA-RC and, therefore, it is worse than COMA. However, we will see later that *MigR* data has little weight and can also be handled by EX-NUMA.

All these accesses have different latencies in different architectures. In general, EX-NUMA's latencies are higher than NUMA-RC's and lower than COMA's (Figure 4-(b)). To see why, consider the model of Section 2. First, it is likely that the number of accesses intercepted locally by EX-NUMA is about as high as the highest of COMA and NUMA-RC. Consequently, the latency of the average *local* access in EX-NUMA will be about as high as the highest of COMA and NUMA-RC.

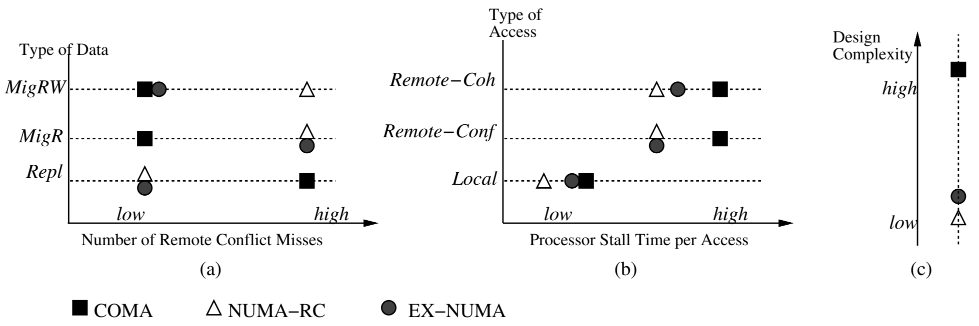

Figure 4: Qualitative comparison between NUMA-RC, COMA, and EX-NUMA: remote conflict misses (Chart (a)), processor stall time per access (Chart (b)), and design complexity (Chart (c)).

Secondly, the average *remote-conf* access costs more in COMA than in EX-NUMA or NUMA-RC. Indeed, *remote-conf* accesses are always satisfied in two network hops in EX-NUMA and NUMA-RC because the home node keeps a clean copy of the line. This is not true for COMA. Finally, the average *remote-coh* access in EX-NUMA costs more than in NUMA-RC and less than in COMA. It costs more than in NUMA-RC because exclusive lines are less likely to be displaced from a node and sent to their home in EX-NUMA. As a result, a *remote-coh* access is less likely to finish in just two network hops. However, *remote-coh* accesses cost less than in COMA because, in a read miss on data that is exclusive in some other processor, the protocol updates home in EX-NUMA but not in COMA. This reduces to two the number of network hops required in future read transactions.

The DRAM-based RDT does not slow down EX-NUMA relative to NUMA-RC. In NUMA-RC, after a secondary cache miss, the DRAM tags of the remote cache are accessed in parallel with the remote cache data. In EX-NUMA, we have doubled the associativity of the search. The RDT entries associated with the remote cache and the remote cache are accessed in parallel with the RDT entries associated with the memory and the memory. This system can be organized in a pipelined manner such that multiple requests can be pipelined, therefore reducing the occupancy of the RDT, memory, and remote cache.

Finally, we consider the cost of EX-NUMA. We consider two cost factors, namely the design complexity and the amount of DRAM memory required for the RDT. The design complexity of EX-NUMA is modest (Figure 4-(c)) because all the modifications required to produce an EX-NUMA out of a NUMA-RC are local. There are no changes to the global cache coherence protocol. Seen from outside of a node, the coherence transactions that go in and out of an EX-NUMA node are exactly the same as the ones in NUMA-RC. The race conditions in the cache coherence protocol of EX-NUMA are largely the same as NUMA-RC's [14]. This is in contrast to COMA, where new machine states and transactions are necessary to ensure that the last copy of a line is not lost.

To determine the DRAM memory required for the RDT to store the pointers to cells, consider a machine with P processors and A-way set-associativity for the RDT entries dedicated to memory. In this case, the tag size for RDT entries dedicated to memory is $log_2(P \times A)$ bits. With 64 processors and a set associativity of 4, the tag size is 8 bits. Adding 2 bits for the state field, we get 10 bits per RDT entry. If the size of the memory lines is 32, 64, or 128 bytes, the ratio between the size of the RDT dedicated to memory and the memory size is 3.9%, 1.9% and 1% respectively. Adding this much DRAM to the system is not very expensive.

Furthermore, this memory overhead can be further reduced if the RDT has fewer entries dedicated to memory than memory lines. For example, if we reduce the number of such RDT entries to one fourth, the overhead becomes about one quarter of the previous figures. In this case, however, since each RDT entry can only identify one cell for every four locations, we risk wasting cells

(Section 6.2).

Finally, in a more advanced RDT design that we discuss in the next section, we can combine the RDT and the directory. In that case, the directory storage is reused to store the RDT entries dedicated to memory. As a result, the overhead of the memory-dedicated RDT is negligible.

3.4 Advanced Implementation Issues

There are several advanced implementation issues to consider. The RDT is designed such that the entries dedicated to memory and to the remote cache work in the same way. It is possible to move the boundary that separates the two types of entries. For example, if the applications that we run do not need much physical memory, we can increase the size of the remote cache at the expense of the memory. This may be done by reprogramming the memory controller and the RDT controller. Alternatively, the remote cache can be expanded "logically" by simply marking up all the RDT entries that are pointing to lines in unused memory pages as if these lines all had cells. That will allow us to store remote data in all these unused memory pages.

A second design issue to consider is whether to have full exclusivity between the data in the secondary cache and the data in the remote cache and cells. The base design presented does not have full exclusivity. Instead, every time that a remote line is brought into the secondary cache *in state non-exclusive*, the RDT also saves a copy in one of the cells or in the remote cache. This is done because the line can be later displaced silently from the secondary cache. However, consider an ideal scenario where the secondary cache writes back even lines in state non-exclusive on displacement. While this scenario is unrealistic, it is interesting because we can then have full exclusivity between secondary cache, remote cache, and cells. The node holds on to more remote data. Of course, one problem of this design is the bandwidth consumed by the extra transfers of non-exclusive data. We evaluate this scenario in Section 5.

A third issue relates to our decision to design a RDT that minimizes the modifications to a NUMA-RC. If we are willing to redesign the directory to integrate it with the RDT, we can eliminate practically all of the memory requirements of the RDT. Specifically, the directory storage can be reused to store the tags and state of the RDT entries dedicated to memory. This is possible if the machine uses a pointer-based directory scheme with invalidations [1, 7]. In such schemes, after a memory line is written, the corresponding directory entry contains only one pointer, pointing to the owner processor. At that point, most of the bits in the directory entry are unused. Recall that, it is at this point that a cell is created. If the cell is filled, we can use these unused directory bits to store the RDT tag and state of the incoming line. More discussion can be found in [14].

4 Experimental Setup

To estimate the performance of the baseline EX-NUMA, we perform execution-driven simulations driven by parallel applications. In this section, we describe the architectures simulated, the simulation environment, and the applications run.

4.1 Architectures Simulated

We model a CC-NUMA, NUMA-RC, COMA, and EX-NUMA architecture, each with 32 nodes. We refer to the CC-NUMA as NUMA. Each node includes a processor, two levels of cache, a memory controller, and a portion of the global memory and directory (Figure 5). The size of the remote cache in NUMA-RC and EX-NUMA is equal to the size of the extra memory given to COMA. The difference in tag space is neglected due to its small size. In EX-NUMA, the RDT is 4-way set-associative and has as many entries dedicated to memory as there are memory lines.

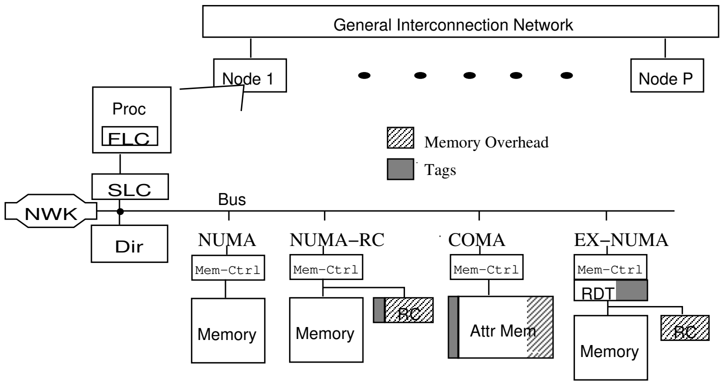

Figure 5: Architectures simulated.

In all the architectures, the processor runs at 200 MHz. The on-chip primary cache is direct-mapped and has 16-byte lines. The secondary cache, remote cache, and attraction memory are all 4-way set-associative and have 32-byte lines. Caches are kept coherent with a DASH-like protocol [7]. The memory bus is 64-bit wide, supports split transactions, and runs at 40 MHz. The memories, attraction memories, RDTs and remote caches are pipelined, with a latency of 12 clocks to get the first word of their lines. All machines have a general interconnect where the latency between two nodes is fixed at 100 cycles. With all these parameters, the resulting no-contention round-trip latency of loads satisfied by the secondary cache, local memory, and remote memory in 2- and 3-hops are 4, 39, 249, and 351 cycles respectively. The remote cache has the same latency as the local memory. The machines use the release memory consistency model. All the contention in the system is carefully modeled except in the network, where it is neglected.

The page allocation policy that we use for NUMA and NUMA-RC is first-touch after the parallel section starts. For COMA and EX-NUMA, since the allocation should not matter much, we use round-robin allocation. We have also applied round-robin to NUMA and NUMA-RC and, while the results are slightly different, they do not change qualitatively.

The memory hierarchy of the architectures is dimensioned for each application, so that the size of the working set is comparable to or larger than the per-processor *MemOvhd*. It is in such a situation that the different architectures exhibit their performance advantages or disadvantages. To dimension the memory hierarchy, we proceed as follows. For each application, we divide the size of its footprint by the number of processors. The resulting number, divided by 4, 16, and 64, is the size of the per-processor *MemOvhd*, secondary cache, and primary cache respectively. Except for *MemOvhd*, all sizes are rounded up to the closest power of two. The resulting *MemOvhd* is, therefore, 20% of the total memory. For comparison, we also study an EX-NUMA with only half of the *MemOvhd* (which corresponds to about 11% of the total memory), and no *MemOvhd* at all.

4.2 Simulation Environment and Applications

We use the Tangolite [5] tracing-simulation environment to run six Splash2 [13] applications. The applications are listed in Table 1, together with their data footprint size, the secondary cache used, and the major type of data in the working set. In *Barnes*, *Volrend* and *Water-NQ*, *Repl* dominates. However, in *Barnes* and *Volrend*, the *Repl* data is a tree structure that gets scattered around, while in *Water-NQ*, it is a regular linear array of molecules. Consequently, while the COMA *MemOvhd* will not be used well in *Barnes* and *Volrend*, it will be better utilized in *Water-NQ*. These three applications also have some *Mig* data. In *Ocean*, *Radix* and *FFT*, *Mig* dominates. In *Ocean*, the *Mig* data is of the *MigRW* type. *Radix* and *FFT* have weaker *Mig* behavior because processors change working sets frequently, creating transient periods. *Radix* tends to have *MigR* data, while *FFT* is dominated by *MigRW*. Note that a given application has *Repl* behavior irrespective of whether we use a first-touch or a round-robin page allocation policy. Some types of *Mig* behavior that appear with round-robin may become less important or disappear with first-touch allocation. Our classification of applications, however, is valid irrespective of the page allocation policy.

Table 1: Characteristics of the applications used.

Application	Short Description	Problem Size	Footprint(MB)	Sec. Cache (KB)	Data Type
Barnes	N-body simulation	16K bodies	3.8	8	*Repl*
Volrend	3-D volume rendering	Head	22.9	32	*Repl*
Water-NQ	N-body problem	512 molecules	7.0	8	*Repl*
Ocean	Current simulation	258×258 grid	15.5	32	*Mig*
Radix	Integer radix sort	256K keys	2.8	8	*Mig*
FFT	Complex 1-D 6-step FFT	64K points	3.7	8	*Mig*

5 Evaluating the Baseline EX-NUMA

To evaluate EX-NUMA, we first compare its performance to NUMA-RC and COMA. Then, we examine in detail some of its characteristics.

5.1 Overall Performance Comparison

We compare the performance of NUMA and three machines with 20% *MemOvhd*: NUMA-RC (*N-RC20*), COMA (*COMA20*), and EX-NUMA (*EX-20*). In addition, for reference purposes, we consider the ideal EX-NUMA environment where the secondary cache, remote cache, and cells are exclusive, and the secondary cache informs the RDT of all displacements, including those of non-exclusive lines. In this case, the local memory hierarchy holds the maximum amount of remote data. We examine this environment for 0%, 11%, and 20% *MemOvhd* (*eEX-0*, *eEX-11*, and *eEX-20* respectively, where *eEX* stands for exclusive EX-NUMA).

These seven architectures are compared in Figure 6. The first row of the figure shows the remote miss rate, which is the fraction of read accesses not satisfied in the local node. The second and third rows show the processor stall time due to memory accesses and the total execution time respectively. The three *Repl*-intensive applications are on the left, while the three *Mig*-intensive ones are on the right. The miss rates are broken down into the start-up (*Cold*), conflict (*Conf*), and coherence (*Coh*) components. The memory access stall time is divided into stall caused by local (*Local*), remote-cold (*RCold*), remote-conflict (*RConf*), and remote-coherence (*RCoh*) accesses. Finally, the execution time is divided into execution of instructions (*Busy*), stall due to memory accesses (*Mem*), and stall due to synchronization, write buffer overflow, and other factors (*Other*). The memory access stall and execution time data is normalized to the *NUMA* measurements for each application.

The *NUMA* bars in the execution time chart show that processors spend an average of over 60% of the time stalled on memory accesses. Furthermore, the memory access stall chart shows that a large fraction of this time is due to remote accesses caused by conflicts (*RConf*). Consequently, architectures that provide extra caching space like *N-RC20* and *COMA20* reduce the number of

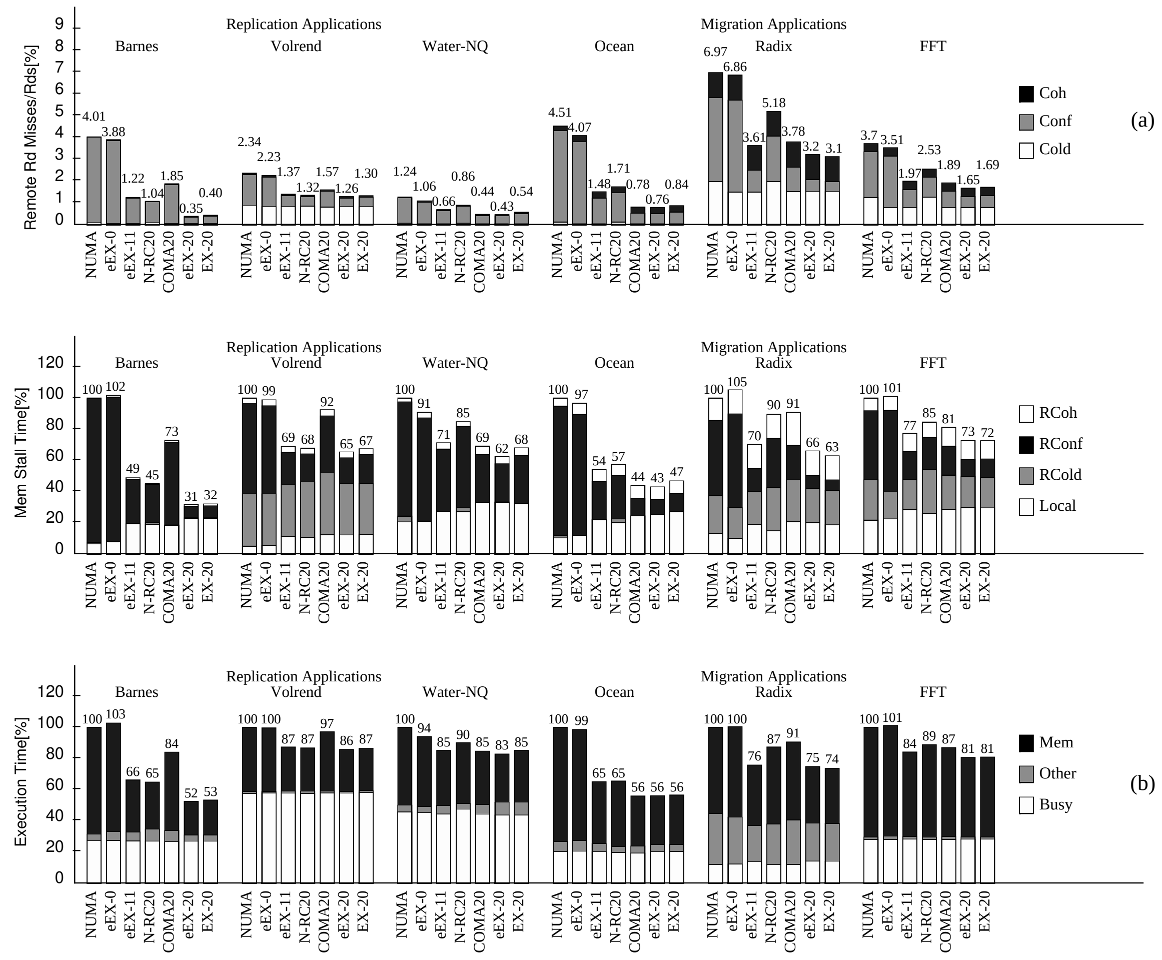

Figure 6: Remote miss rate, memory access stall time, and total execution time of the applications running on different architectures.

remote misses significantly (topmost chart). This results in average execution time reductions of 20% and 17% for *N-RC20* and *COMA20* respectively (bottommost chart).

The relative performance of *N-RC20* and *COMA20* depends on the dominant sharing pattern in the application. As expected, *N-RC20* tends to perform better than *COMA20* in *Repl*-intensive applications. It performs better than *COMA20* in *Barnes* and *Volrend* because the *Repl* data in these applications is irregular. However, the opposite occurs in *Water-NQ* because the *Repl* data is now regular. For *Mig*-intensive applications, *COMA20* eliminates more conflict misses than *N-RC20* (topmost chart). In terms of execution time, however, *COMA20* is clearly faster only in *Ocean*. *COMA20* has similar performance as *N-RC20* in *Radix* and *FFT* because the *Mig* behavior is weaker in these applications. Overall, averaging out all the applications, *COMA20* removes more remote misses than *N-RC20* but has a performance comparable to *N-RC20* because it has higher memory access latencies.

EX-20 runs as fast or (usually) faster than *N-RC20* and *COMA20* for all applications. Overall, *EX-20* runs an average of 12% and 10% faster than *COMA20* and *N-RC20* respectively. While these numbers may seem modest, they are important because they are close to the upper bound possible for these applications. Indeed, if we look at the topmost chart of Figure 6, there are few remaining *Conf* misses in *EX-20*. This has been accomplished across all types of applications and with a design that is much simpler than COMA.

Finally, we consider the ideal scenarios of *eEX-20*, *eEX-11*, and *eEX-0*. The fact that *eEX-20*'s performance is practically the same as *EX-20*'s indicates that we lose nothing by not supporting an unrealistic fully-exclusive system. In addition, the poor performance of *eEX-0* suggests that the lack of a remote cache strongly inhibits the process of exploiting cells. Finally, the surprisingly good performance of *eEX-11* points out the crucial role that even small remote caches have in enabling cell exploitation. We expand on this issue next.

5.2 Understanding EX-NUMA Performance

To understand the performance of EX-NUMA, we examine the behavior of cells in the four EX-NUMA environments considered. We examine the fully-exclusive environments because they represent ideal conditions. We start by counting the read accesses to non-local data that are not intercepted by the secondary caches. The topmost chart of Figure 7 breaks these accesses into those that are intercepted by the remote cache (*RC*), those intercepted by cells (*Cell*) and those that cause a remote miss. The latter are further divided into read misses due to conflicts (*Miss.Conf*) and the rest of the misses (*Miss.Rest*). Remote caches and cells can only eliminate *Miss.Conf*.

The figure shows that the cells in *eEX-0*, which do not have the assistance of a remote cache, do not eliminate many conflict misses. This is true even for *Mig*-intensive *Ocean*. With the addition of a remote cache, *eEX-11* and *eEX-20* eliminate many more conflict misses. We see the multiplicative effect of the remote cache: it intercepts requests and is a catalyst to create more cells that intercept requests as well. The only exception is *Repl*-intensive *Volrend*, where the *Cell* category is very small. Most of its *Mig* data is read-only and, therefore, unable to create cells. Comparing *eEX-20* to *EX-20*, we confirm that whether or not we use the ideal exclusivity makes little difference. Overall, the combination of remote caches and cells is very effective in removing conflict misses.

The central chart of the figure shows why cells do not eliminate all the conflict misses. The figure classifies what happens to lines with remote data that are displaced from the secondary caches. They can either be absorbed by cells or by the remote cache (or, in the case of non-exclusive lines in *EX-20*, displaced silently, but in this case they are already stored in cells). The height of the bars in the figure shows the fraction that are absorbed by cells. The bars are divided depending on what finally happens to the lines absorbed by cells: the line may eventually be re-used by the processor (*Hit*), invalidated by the protocol (*Inv*), displaced by the returning *owner* (*Repl*), displaced by another line (*Recv*), or simply sit there until the end of the program without being accessed (*Waste*).

The figure shows that, in *eEX-0*, only a small fraction of the displaced lines are stored in cells. As we add a remote cache, cells intercept a larger fraction of the lines. In *eEX-20*, about 50% of the lines are intercepted by cells. The remaining 50% still have to be intercepted by the remote cache. Of the lines intercepted by the cells, the fraction that are really used (*Hit*) depends on the size of the remote cache. It is small for *eEX-0*. Most of the time, the data in cells is replaced by owner lines (*Repl*) or by other lines displaced from the secondary cache (*Recv*). However, with large remote caches as in *eEX-20* and *EX-20*, an average of about 70% of the data in cells is used. This is good news. The only two applications where the fraction of *Hit* is not much larger than 50% are *Radix* and *FFT*. In these applications, *Mig* data moves from one processor to another frequently, reducing the chance of reusing the data in the cell. As a result, lines in cells are often displaced by other lines (*Recv*) or simply wasted (*Waste*). Overall, when the cells are not reused, it is largely a problem of lack of application locality.

Finally, to see how well memory is utilized, the bottommost chart of Figure 7 shows the state of the RDT entries dedicated to memory at the end of the program. A memory location can have

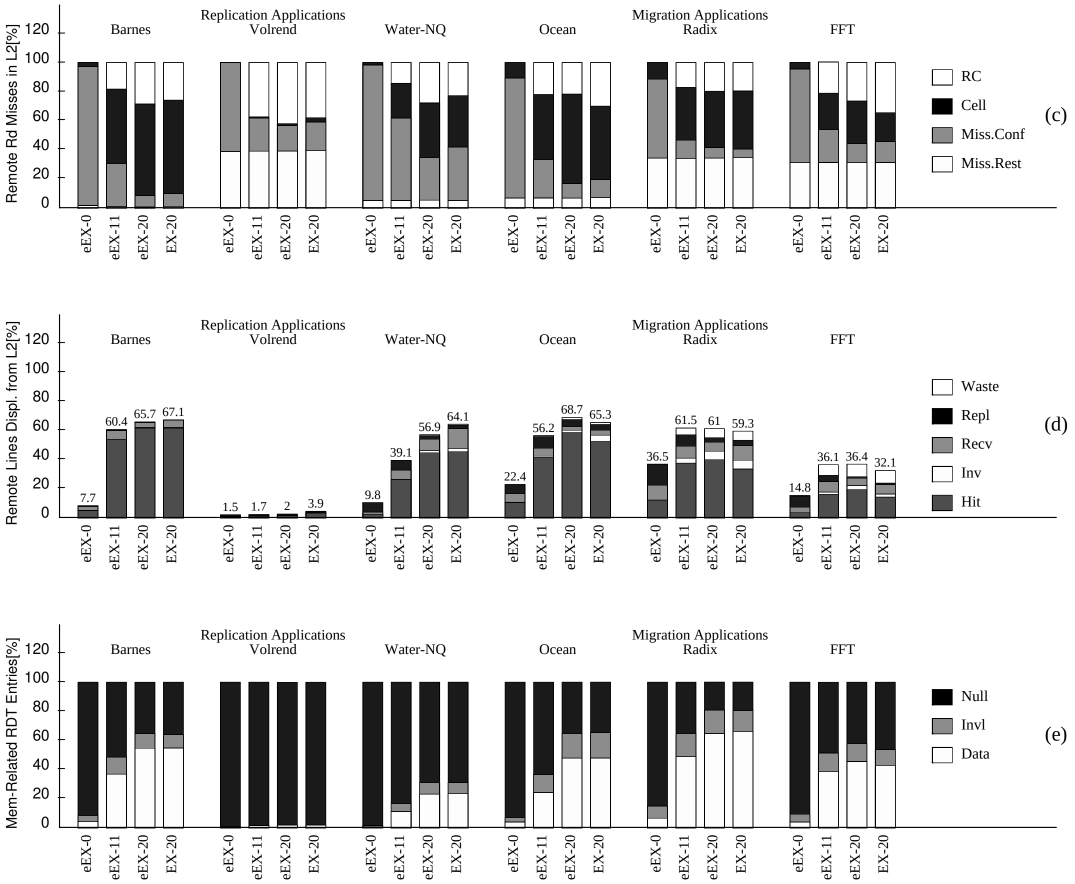

Figure 7: Behavior of cells in EX-NUMA. The topmost chart classifies the read accesses to non-local data that are not intercepted by the secondary caches. The central chart classifies what happens to the memory lines with remote data displaced from the secondary caches. Finally, the bottommost chart shows the state of the RDT entries dedicated to memory.

a cell with data (*Data*), an empty cell (*Invl*), or no cell (*Null*). While in *eEX-0* the number of cells is small, as we move to *eEX-20* and *EX-20*, the number increases significantly. This is a result of the multiplicative effect of the remote cache. For example, for the three *Mig*-intensive applications, about 2/3 of the memory locations have cells. There are, however, applications like *Volrend* where, for reasons explained before, few cells exist. Overall, in *EX-20*, an average of 50% of the memory locations have cells. Of those, about 80% contain data. These two figures show that EX-NUMA-style migration works.

It is important to point out that we have not seen instances of line write backs that trigger long sequences of block displacements from cells across the machine. This is because the presence of a remote cache, the associativity of the RDT, and a RDT receiving algorithm that favors replacements of non-exclusive lines, all help to minimize cell destruction. We find the performance of EX-NUMA very stable in this regard.

6 Improving the EX-NUMA Architecture

We now discuss several changes to improve the performance of EX-NUMA and reduce its overhead. To save space, our discussion is very brief. For details and quantitative results, see [14].

6.1 Improving EX-NUMA Performance

6.1.1 High RDT Associativity

Increasing the associativity of the RDT increases the probability of finding a cell able to accept the remote line. With high associativity, cells retain more data, which reduces the pressure on the remote cache and results in fewer conflict misses. To help support high associativity, we can use paged-mode DRAM and place all the memory lines that correspond to a given RDT set in the same DRAM page. The same can be done for the remote cache lines that belong to one set. With high associativity, however, we increase the busy time for the RDT. To avoid bottlenecks, we must rely on pipelining the RDT and memory access.

We have examined the execution time of the applications under RDT associativities of 2, 4, and 32 [14]. Recall that, in our baseline EX-NUMA, we have used a RDT associativity of 4. This means that, for each RDT access, we check 4 RDT entries dedicated to memory and 4 dedicated to the remote cache. Our results show that the largest gains occur while going from 2 to 4-way set-associative RDTs. On average, the applications speed up by 6%. Increasing the associativity from 4 to 32 has a large impact for some applications. For example, *Ocean* runs 11% faster. On average, the applications speed up by 5%. However, the design of such a fast, high-associative circuitry is challenging.

6.1.2 Relaxing the Home Location

A second approach to increase the effectiveness of cells is to minimize the chances that, when the *owner* line is written back, it displaces the line in its cell. Such an event has not been seen to lead to a long chain of cell destructions in our experiments. However, we propose a solution to cut short any chain of cell destruction. The idea is to treat the returning owner as if it were a remote line displaced from the secondary cache. As such, it is written into any location in its corresponding RDT sets (memory or remote cache) as determined by the receiving algorithm. If the place that the owner goes into is not its default location, we mark the new location as the new home of the owner line. This is called *home relaxation*. This action would likely terminate any chain of cell destruction very soon.

There are several ways of implementing this home relaxation. In a simple and not very efficient approach, each directory entry has one bit called *moved*. This bit indicates whether the line associated with this directory entry has changed its home line location. In addition, the state field of each RDT entry can take an extra state, namely *moved_home*. With this support, assume that an owner is written back and, since its RDT entry i is full, we have to store it in the empty cell of entry j in the same RDT set. The tag and state of RDT entry j are set to the incoming line's tag and to state *moved_home* respectively. In addition, the *moved* bit in the incoming line's directory is set. From now on, any request for the line finds the *moved* bit set in the directory. The request, while accessing the RDT, finds the correct tag and reads the line from entry j. Note that, after a line has changed its home, it may be written to again and, therefore, leave an empty cell in its new home. At that point, the *moved* bit is reset. The line is not logically "moved" any more. A more refined home relaxation scheme that involves swapping the contents of directory entries is described in [14].

We have evaluated the performance of this scheme. Compared to the baseline system, the new machine retains more cells. Overall, the average application is sped up by 3%.

6.1.3 Migrating Read-Only Data

To have more cells, we can create one every time that a home node receives a read request for a line that is currently not shared by any processor. The directory takes this request as a hint that the data is of *Mig* type and supplies it in state exclusive, therefore creating a cell. This optimization helps *MigR* data. In addition, it also helps *MigRW* data because cells are created as soon as the first read is issued. However, it hurts *Repl* data. In this case, the first read creates a cell, while a subsequent read by a second processor needs to fetch the data from the first reader processor. Overall, we find that these two effects tend to cancel each other out. While this optimization improves the performance of some applications, the average impact is small.

6.2 Reducing the Overhead of EX-NUMA

The second way to improve EX-NUMA is to reduce its overhead. One approach is to integrate the RDT and directory as discussed in Section 3.4 (see [14] for more details). This approach eliminates the overhead of memory-related RDT entries. A second approach is to use fewer RDT entries dedicated to memory. Recall from the bottommost chart of Fig 7 that an average of 50% of the memory locations contain cells in *EX-20*. We can, therefore, reduce the number of RDT entries. Each RDT entry can now record a cell from one of several memory locations by including a few more bits specifying which location it actually points to. This organization may help a line find a cell because each RDT entry maps to more memory locations. However, the downside is that the RDT loses the ability to record all cells. Only one cell per several memory locations can be recorded at any one time. To test the effect of the RDT size, we simulate *EX-20* with only 50% of the baseline RDT entries dedicated to memory. The results show that three applications, namely *Barnes*, *Water-NQ* and *Ocean*, are slowed down by 2-4%. On average, however, the six applications run only about 1% slower.

7 Related Work

Perhaps the closest work is R-NUMA, proposed by Falsafi and Wood [4] concurrently with EX-NUMA. While R-NUMA is also a hybrid machine, its approach is very different. R-NUMA is a hybrid between CC-NUMA and Simple-COMA, the software incarnation of COMA. As such, in R-NUMA, data migrates at the page granularity. Switching between CC-NUMA and Simple-COMA is performed at the page grain size, and requires operating system involvement via an interrupt. Finally, the interrupt is induced by run-time feedback from traffic-monitoring counters. EX-NUMA, instead, is a hardware-only approach that tries to incorporate into CC-NUMA the good points of Flat-COMA, the hardware incarnation of COMA. Consequently, data migration is supported at the cache line level. This fine-grain migration eliminates any concern about page fragmentation and can handle many objects of different behavior in the same page. Furthermore, there is a single (CC-NUMA) protocol under which all memory lines live, and there is no operating system involvement. No run-time feedback is necessary. Both R-NUMA and EX-NUMA can be implemented over existing CC-NUMA machines with simple hardware. Finally, the performance results in the two papers cannot be directly compared because the architecture assumptions are very different. While our paper reports quantitatively lower performance gains, we show that EX-NUMA gets close to the upper bound possible, namely elimination of all conflict misses.

Other related work is operating system-induced migration and replication of pages based on run-time feedback. Several authors have shown that this technique boosts the performance of CC-NUMA [2, 9, 12]. This technique may have some of the effects of EX-NUMA, although migration is only supported at page granularity and, in addition, it has operating system overhead. We are currently comparing its effectiveness to that of EX-NUMA.

8 Discussion and Concluding Remarks

This paper has presented EX-NUMA, an architecture that attempts to combine the good points of both Flat-COMA and NUMA-RC. The architecture supports fine-grain automatic data migration nearly as well (and, therefore, is nearly as programmable) as Flat-COMA, is not much more complex than NUMA-RC, and outperforms both of them by supporting the different data access patterns well. Furthermore, it gets close to the upper bound of eliminating all conflict misses.

An important characteristic of EX-NUMA is its ability to migrate data at the memory-line grain-size without the costly Flat-COMA hardware. In theory, migration in EX-NUMA may be harder than in Flat-COMA. This is because the space that a node has available to hold remote *Mig* (and *Repl*) data locally is determined, in part, by how much *Mig* data allocated locally is being accessed by other nodes. For the system to perform best, these two sets of *Mig* data should be balanced. One case where this happens is when the application is such that each processor works on roughly the same amount of *Mig* data, and the data layout is such that *Mig* data is laid out roughly uniformly across the different nodes. These conditions are largely satisfied under the applications and page allocation policies analyzed. Furthermore, when they are not perfectly satisfied, the large remote cache can absorb the fluctuations. Therefore, we find that EX-NUMA-style migration works well. It will be interesting to examine other applications like databases and applications much less tuned for caches than Splash2.

Overall, EX-NUMA represents a good cost-performance design point, as we consider complexity, performance and programmability. Indeed, NUMA-RC is the simplest one but it is the least programmable one because it does not handle *Mig* data well and, as a result, its performance suffers. Flat-COMA, on the other hand, is the most programmable one because it handles fine-grain *Mig* data, but it is the most complex one and its performance is low for *Repl* data. Finally, EX-NUMA has a modest complexity over NUMA-RC, a programmability close to Flat-COMA, and performs best for the different types of data.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung.
The MIT Alewife Machine: Architecture and Performance.
In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] D. Black, A. Gupta, and W. Weber.
Competitive Management of Distributed Shared Memory.
In *Proceedings of the 34rd IEEE Computer Society International Conference - COMPCON 89*, pages 184–190, February 1989.

[3] Convex Inc.
Convex Exemplar Description.
URL: http: //www.convex.com/prod_serv/exemplar/exemplar.html.

[4] B. Falsafi and D. Wood.
Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA.
In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[5] S. Goldschmidt.
Simulation of Multiprocessors: Accuracy and Performance.
Ph.D. Thesis, Stanford University, June 1993.

[6] A. Gupta and W. Weber.
Cache Invalidation Patterns in Shared-Memory Multiprocessors.
In *IEEE Transactions on Computers*, pages 794–811, July 1992.

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

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

[9] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott.
Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems.
In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

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

[11] 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.

[12] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum.
Operating System Support for Improving Data Locality on CC-NUMA Compute Servers.
In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[13] 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.

[14] Z. Zhang.
Architectural Alternatives to Reduce Remote Conflict Misses in Shared-Memory Multiprocessors.
Ph.D. Thesis, University of Illinois at Urbana-Champaign, October 1996.

[15] Z. Zhang and J. Torrellas.
Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA.
In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 272–281, February 1997.